When Speed Has a Price: Fast Information Extraction Using Approximate Algorithms

Gonçalo Simões
INESC-ID and Instituto Superior Técnico, Portugal
goncalo.simoes@ist.utl.pt

Helena Galhardas
INESC-ID and Instituto Superior Técnico, Portugal
helena.galhardas@ist.utl.pt

Luis Gravano
Columbia University
New York, USA
gravano@cs.columbia.edu

ABSTRACT

A wealth of information produced by individuals and organizations is expressed in natural language text. This is a problem since text lacks the explicit structure that is necessary to support rich querying and analysis. Information extraction systems are sophisticated software tools to discover structured information in natural language text. Unfortunately, information extraction is a challenging and time-consuming task. In this paper, we address the limitations of state-of-the-art systems for the optimization of information extraction programs, with the objective of producing efficient extraction executions. Our solution relies on exploiting a wide range of optimization opportunities. For efficiency, we consider a wide spectrum of execution plans, including approximate plans whose results differ in their precision and recall. Our optimizer accounts for these characteristics of the competing execution plans, and uses accurate predictors of their extraction time, recall, and precision. We demonstrate the efficiency and effectiveness of our optimizer through a large-scale experimental evaluation over real-world datasets and multiple extraction tasks and approaches.

1. INTRODUCTION

Information extraction (IE) systems are sophisticated software tools to discover structured information in natural language text. For example, we may train an IE system to extract instances of an *Occurs-In*(*NaturalDisaster, TemporalExpression*) relation. Such a system can extract a tuple <eruption of Grímsvötn, 22-25 May 2011> from the text excerpt "The eruption of Grímsvötn during 22-25 May 2011 brought back the memories of the eruptions of Eyjafjallajökull in 2010." By discovering and extracting such structured information from text, IE systems enable much richer querying and analysis of the structured information than would be possible over the natural language text where the information is expressed. For this reason, IE systems have attracted substantial academic and commercial interest over the last decade. Despite this surge in interest, IE remains a challenging, time-consuming task. In this paper, we focus on the optimization of the IE process, with the goal of producing efficient IE executions.

Proceedings of the VLDB Endowment, Vol. 6, No. 13

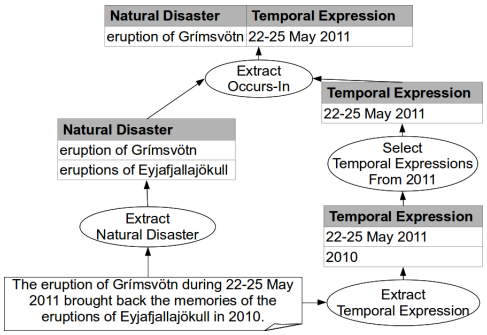

Figure 1: Extracting natural disasters from text.

IE programs aim at extracting structured information from natural language text. IE programs are generally composed of smaller *IE tasks* such as document retrieval, segmentation, entity extraction, normalization, co-reference resolution, and relationship extraction [14]. Figure 1 shows an IE program to extract *Occurs-In* tuples. It involves IE tasks for extracting entities (namely, natural disasters and temporal expressions), filtering temporal expressions not referring to 2011, and finally establishing *Occurs-In* relationships. Each IE task is associated with one *IE technique* that implements it. These techniques can be based on rules (e.g., dictionaries [2], patterns [23, 5]) or statistics (e.g., Conditional Random Fields (CRF) [15], Support Vector Machines (SVM) [6]).

A naive strategy to execute an IE program is to process all the available documents with some implementation of the IE tasks involved. This is usually a prohibitively time-consuming strategy because: (i) some IE techniques need to extract a large number of textual features (e.g., a CRF depends on features such as words and their lemmas); (ii) IE techniques may rely on complex text analysis methods (e.g., POS tagging, parsing) that are usually time-consuming [8]; and (iii) document collections often consist of a large number of documents, the vast majority of which are, generally, not relevant to a specific IE program. For example, a naive execution of the IE program at Figure 1 may take one minute or more to process just 100 news articles, so real-world collections with millions of documents would be unmanageable for such execution strategy. Furthermore, such inefficient strategy would be particularly problematic for important scenarios in which offline, once-and-for-all extraction is not feasible, including: (i) when a document collection (e.g., the

Web at large) is dynamic; (ii) when the extraction program itself is subject to tuning and adjustment, as is typically the case in operational settings; or (iii) when the extraction tasks are defined dynamically, as their need arises.

The optimization of IE programs is critical to make IE possible over real-world document collections. This challenging task has attracted substantial attention over the last few years. Notably, CIMPLE [19], SystemT [12], and SQoUT [8, 9, 10] introduce complementary approaches to optimize parts of the IE process. Specifically, CIMPLE [19] determines a fast execution order for the IE tasks of an IE program. By choosing a particular execution order, CIMPLE avoids the application of expensive IE techniques to parts of documents not containing relevant information. In turn, SystemT [12] determines the best algorithm to implement the IE technique associated with each IE task. SystemT can also choose the execution order of some IE tasks but the range of choices is not as wide as in CIMPLE. Despite these differences, CIMPLE and SystemT share two important limitations of their optimization approaches: (i) all the alternative execution plans for an IE program produce the same results; and (ii) all the input documents are used in the IE process. In contrast, SQoUT [8, 9, 10] relaxes these two constraints and focuses on determining which documents should be used during the extraction process [8]. Moreover, when several black-box IE systems are available, SQoUT can choose between them [9] and set parameters, if supported by the IE systems, to affect their efficiency and extraction results [10]. Hence, the execution alternatives that SQoUT considers might produce different extraction results. In fact, SQoUT considers plans with varying recall (i.e., the fraction of relevant information extracted) and precision (i.e., the fraction of the extracted information that is relevant). Then, SQoUT chooses the fastest execution alternative that produces results above given threshold values for recall and precision.

Despite the existing work, IE remains a time-consuming process. As discussed above, CIMPLE and SystemT rely on conservative optimization techniques that demand that all their alternative execution plans produce the same extraction results. Hence, these systems miss optimization opportunities from allowing for approximate extraction results, hopefully with only a modest loss in recall and precision. SQoUT considers execution plans that produce different extraction results. However, it cannot directly perform modifications to the extraction process, namely, the order of execution of IE tasks and the algorithms used to perform them. Therefore, even though the optimization techniques used by CIMPLE, SystemT, and SQoUT are complementary, they cannot be directly combined to holistically optimize the execution of an IE program while exploiting all optimization opportunities, as we propose in this paper.

This paper presents a holistic approach for IE optimization that addresses the above limitations of the state-of-the-art systems. Our optimization approach focuses on the following challenges: (i) to optimize all key aspects of the IE process collectively and in a coordinated manner, rather than focusing on individual subtasks in isolation; (ii) to accurately predict the execution time, recall, and precision for each IE execution plan; and (iii) to use these predictions to choose the best execution plan to execute a given IE program. Specifically, the main contributions of this paper are:

- A novel probabilistic prediction model to accurately determine the extraction time, recall, and precision of each IE execution plan (Section 3.2). This prediction model accounts for plans with approximate results (unlike CIMPLE and SystemT), so we can holistically optimize the entire IE execution plan (unlike CIMPLE, SystemT, and SQoUT).
- A solution to efficiently and effectively estimate the parameters of the predictors for an IE program over a specific document collection (Section 3.3). Unlike the method used by SQoUT to predict recall and precision, our estimation method is efficient even when there is a large number of plans and is accurate even if only a small percentage of documents produce extraction results.
- A solution to choose the fastest IE execution plan for user-specified constraints on the desired recall and precision (Section 3.4). Our method supports complex constraints that may involve more than simple threshold conditions on recall and precision.

We conducted a large-scale experimental evaluation of our proposed optimization approach against the state-of-the-art IE optimization systems (Sections 4 and 5). The results show that our optimizer significantly outperforms those IE optimization systems, while closely matching the user-specific constraints on recall and precision.

2. IE OPTIMIZATION: PRELIMINARIES

We now discuss background and terminology (Section 2.1), as well as the optimization techniques used by state-of-the-art IE systems and their limitations (Section 2.2). We also define our problem of focus for this paper (Section 2.3).

2.1 Background and Terminology

Natural language text often contains information that is structured by nature. *Information extraction (IE)* programs extract this structured information from natural language documents and output tuples, which can then be used by other applications. This structured information typically consists of: (i) *entities*, which correspond to references in the text to real-world objects with predefined types (e.g., person, location, natural disaster); and (ii) *relationships*, which correspond to mentions of semantic associations between multiple entities (e.g., *Occurs-In*, *Works-For*). Developing an IE program involves two main activities: (i) specifying the program and (ii) choosing the best strategy to execute it. Similarly to SQL query optimization, the separation between these two activities is the key for the automatic optimization of IE programs.

IE specification: We can specify an IE program as a composition of IE tasks such as document retrieval, segmentation, entity extraction (*EE*), relationship extraction (*RE*), normalization, and co-reference resolution [14]. More precisely, an *IE specification* corresponds to a directed acyclic graph. Figure 2(a) shows the specification of the IE program from Figure 1. The nodes of this graph are operators that either perform an IE task using a specific IE technique (e.g., EE_{CRF} performs the *Extract Natural Disaster* task with a CRF [15] and RE_{SVM} performs the *Extract Occurs-In* task with an SVM [6]) or impose conditions on the results of other operators (e.g., the $\sigma_{contains(Temporal\ Expression,\ \text{'2011'})}$ operator performs the *Select Temporal Expressions from 2011* task by determining whether the '2011' string is a substring of the input temporal expression). If two operators are connected with an edge, then the output tuples of the origin operator are provided as input to the target operator.

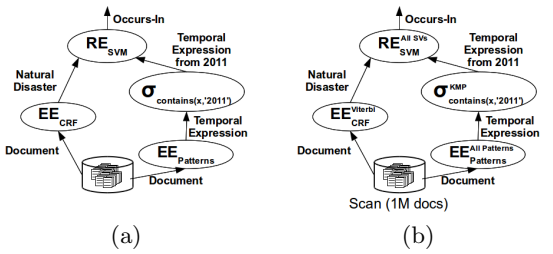

Figure 2: IE specification (a) and IE execution plan (b) for the Figure 1 example.

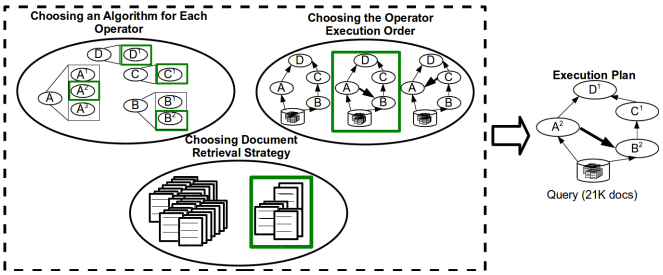

Figure 3: IE program implementation choices.

IE execution plans: An IE specification can be implemented by several execution plans. An *IE execution plan* is also a directed acyclic graph of operators, each of them annotated with the algorithm that executes it. Figure 2(b) shows an example of an IE execution plan for the IE specification of Figure 2(a). For example, the EE_{CRF} operator, which performs the *Extract Natural Disaster* task with a CRF, is executed with the Viterbi algorithm, which we denote as $EE_{CRF}^{Viterbi}$. The operator to *Select Temporal Expressions From 2011*, $\sigma_{contains(x,\text{'2011'})}$, is executed with the Knuth-Morris-Pratt (KMP) algorithm [11], and we denote it as $\sigma_{contains(x,\text{'2011'})}^{KMP}$.

We can choose from a wide variety of algorithms to execute each operator.[1] We can divide these algorithms into two major categories. An *exhaustive algorithm* follows exactly the semantics of the IE technique associated with the operator (e.g., the Viterbi algorithm [22] for the implementation of a CRF). An *approximate algorithm* uses simplifications to speed up the execution of the operator (e.g., Viterbi Beam Search [13] for the execution of a CRF). Approximate algorithms may not be able to produce all the tuples an exhaustive algorithm produces. Moreover, they may produce some additional and possibly erroneous tuples. The tuples produced by the operators can then be divided in two classes: the *correct tuples*, which are the ones produced by an exhaustive algorithm, and the *incorrect tuples*, which are the ones that are produced by an approximate algorithm but not by an exhaustive one.[2]

2.2 IE Optimization and State of the Art

The alternative IE execution plans for an IE program may differ on the following implementation choices (Figure 3): (i) the choice of the algorithm to execute each operator; (ii) the choice of the execution order of operators; and (iii) the choice of the document retrieval strategy. These choices are analogous to the decisions that an RDBMS makes when optimizing SQL queries, namely, the choice of join algorithms, the execution order of the operators, and the choice of access paths. We now describe these IE implementation choices and how state-of-the-art IE optimization systems use them.

[1]Our algorithms are black boxes that perform the tasks of specific operators. Thus, we can plug in new alternative algorithms for an operator as they become available.

[2]We do not require any human intervention to determine the correctness of a tuple. Instead, the correctness of a tuple is determined from the output of the exhaustive algorithm of the operator that produced the tuple. Exploring alternative definitions of tuple correctness, perhaps involving human input, is the subject of interesting future work.

Choosing an Algorithm for Each Operator: In general, the operators used to perform an IE task with a specific IE technique can be implemented with different algorithms. For example, the EE_{CRF} operator in Figure 2(a) can be implemented with the Viterbi algorithm or with the Viterbi Beam Search algorithm. The choice of the algorithm has a significant impact on the execution time of the IE program since the computational complexity of alternative algorithms may vary greatly. When we consider approximate algorithms to execute an operator, the difference between the complexity of alternative algorithms tends to be even greater since they use simplifications to speed up the execution of operators with a small impact on recall and precision.

Choosing the Operator Execution Order: Imposing an execution order to the operators may significantly speed up an IE program. In fact, by executing some operators first, it is possible to use their results to filter the input tuples provided to later operators, thus avoiding the application of expensive IE techniques to some parts of the documents. We consider two key techniques to impose an execution order to the operators. First, *pushing-down text properties* [19] aims at evaluating properties of the text imposed by operators as early as possible in the IE execution plan. Second, *narrowing the region of crawled text* [19, 16] (called scoping in [19]) uses the fact that some RE operators may impose conditions on the location, in the text, of the input entities that can be related. We can thus extract some entities first and use their location to narrow the document region for the extraction of the other entities. We can also extend these solutions and define execution orders based on approximations. For example, we can greedily prune the input of an operator based on the results of the previous ones.

Choosing the Document Retrieval Strategy: Typically, only a small subset of a document collection produces extraction results. Thus, we can speed up IE programs by choosing a retrieval strategy that tries to obtain only documents that contribute to the IE task at hand. Three examples of document retrieval strategies are: (i) scanning through the whole collection; (ii) using queries [1]; or (iii) using classifiers [3, 7] to find documents that are likely to produce extraction results. Retrieval strategies based on queries and classifiers are good at finding documents that contain correct extraction results, but they might miss some of these documents, unlike the exhaustive scanning strategy.

State-of-the-Art IE Optimization Systems: State-of-the-art IE systems exploit some, but not all, of the above implementation choices to optimize IE programs (Table 1). CIMPLE [19] uses a cost-based optimizer to choose the execution order of the operators, namely, pushing-down text properties and narrowing the region of crawled text. Sys-

	Exhaustive Plans Only		Exhaustive and Approximate Plans	
Choice	**CIMPLE [19]**	**SystemT [12, 16]**	**SQoUT [8, 9, 10]**	**This paper**
Algorithm	✗	Heuristic choice	✗	Holistic cost-based optimizer
Execution Order	Cost-based optimizer	Cost-based optimizer	✗	
Retrieval Strategy	✗	✗	Cost-based optimizer	

Table 1: Summary of state-of-the-art IE systems and the implementation choices that they make.

temT [16] also uses a cost-based optimizer to choose the execution order of some operators. SystemT's space of execution plans is not as vast as CIMPLE's (e.g., SystemT does not push down text properties; additionally, SystemT's solution for narrowing the region of crawled text is not as general as that of CIMPLE); unlike CIMPLE, though, SystemT chooses among different algorithms for operators. (SystemT uses heuristics for this choice.) Both CIMPLE and SystemT are conservative in that they guarantee that the results produced by all the alternative execution plans are the same. Thus, these systems miss promising optimization opportunities that speed up the execution of the IE program significantly with a minimum impact on recall and precision. In contrast, SQoUT [8, 9, 10] considers execution plans with varying values of recall and precision, by choosing the document retrieval strategy and a black-box IE system to execute the IE program, including setting parameters that affect efficiency, recall, and precision [10]. However, SQoUT cannot directly optimize the internals of the IE systems, namely, their algorithms for operators and the operator execution order. SQoUT uses a cost model that estimates the extraction time, recall, and precision of each IE execution plan. Then, SQoUT chooses the fastest plan with recall and precision above a user-provided threshold. In this paper, as hinted in Table 1, we address the limitations of the state-of-the-art IE optimization systems and exploit, collectively, all three optimization opportunities, namely, the choice of algorithms for operators, the choice of operator execution order, and the choice of document retrieval strategy. As we will see later, in Section 5, our approach produces holistic IE execution plans that are significantly faster than the ones produced by state-of-the-art IE optimization systems, with a minimum impact on recall and precision.

2.3 Problem Definition

The problem we address in this paper is to optimize the execution of IE programs by fully and collectively exploiting the implementation choices discussed in Section 2.2. Our approach considers a rich selection of execution plans that vary on the execution time and extraction results (i.e., leading to output with different recall and precision). There is usually a tension between execution time and output quality. In fact, execution plans that produce high-recall and high-precision output tend to do so at the expense of efficiency. Interestingly, the choice of a plan for an IE program is user-dependent: some users may be interested in high-recall and/or high-precision executions, for which they are willing to wait as long as needed; in contrast, other users may prefer to receive extraction results as fast as possible, even if their quality is less than perfect [10].

To express different user needs, we rely on *and-or trees*. The leaves of an and-or tree specify constraints on the desired recall and precision. Internal nodes represent conjunctions or disjunctions of constraints. The optimization process will then consider user-specified constraints on recall and precision as input, as shown in our problem definition:

PROBLEM 1**:** *Consider a relation to be extracted from a document collection D using an IE program P. We can execute P using alternative IE execution plans p_1, p_2,..., p_n. These IE execution plans vary in their choice of the exhaustive or approximate algorithms used to execute the operators, the execution order of the operators, and the document retrieval strategy. Additionally, let C be a tree with user-specified constraints on precision and recall. Then, our objective is to estimate if any of the p_1, p_2,..., p_n IE execution plans for the IE program P satisfies C over D and, if so, to select the plan that is expected to be the fastest among such plans.*

3. HOLISTIC IE OPTIMIZATION

We now describe our IE optimization approach. First, in Section 3.1, we provide a high-level description of the optimization process. Then, in the remaining subsections, we provide details about each key optimization step.

3.1 Optimization Process: Outline

As discussed above, we consider three major implementation choices for an IE program: (i) choosing an algorithm to execute each operator; (ii) choosing the execution order of operators; and (iii) choosing the document retrieval strategy. We argue that making these choices collectively has a significantly positive impact on the optimization process.

To see why, consider a simple IE program that extracts conference names and their dates. This program takes a substantial time to execute by processing all documents with exhaustive algorithms and no regard for the execution order of the operators. Current state-of-the-art IE systems could speed up this program somewhat by making independent implementation choices. For example, choosing a good execution order of the operators or retrieval strategy could cut the extraction time by more than half, while considering approximate algorithms for the operators might speed up the execution further. We can obtain even more gains by putting these individual implementation choices together. Unfortunately, the choices that we make individually may not necessarily result in the best solution when we combine them. For example, when using a selective document retrieval strategy, which focuses on just a few documents, we may be able to afford expensive and exhaustive algorithms for the IE operators and still execute the program fast, with a negligible penalty on recall or precision. Hence, we should not make these optimization choices in isolation, but rather as part of a holistic effort, as we argue in this paper.

To perform this holistic optimization, our approach receives three inputs (see Figure 4): (i) an IE specification; (ii) a collection of documents; and (iii) an and-or tree representing the user-specified extraction constraints (see the problem definition in Section 2.3). Then, our approach consists of four main steps. The first step is *Plan Enumeration*, which produces the alternative IE execution plans to consider in the optimization process. These plans differ in the implementation choices described in Section 2.2. To reduce the number of IE execution plans that we consider, for efficiency, we only explore an IE execution plan if new operators

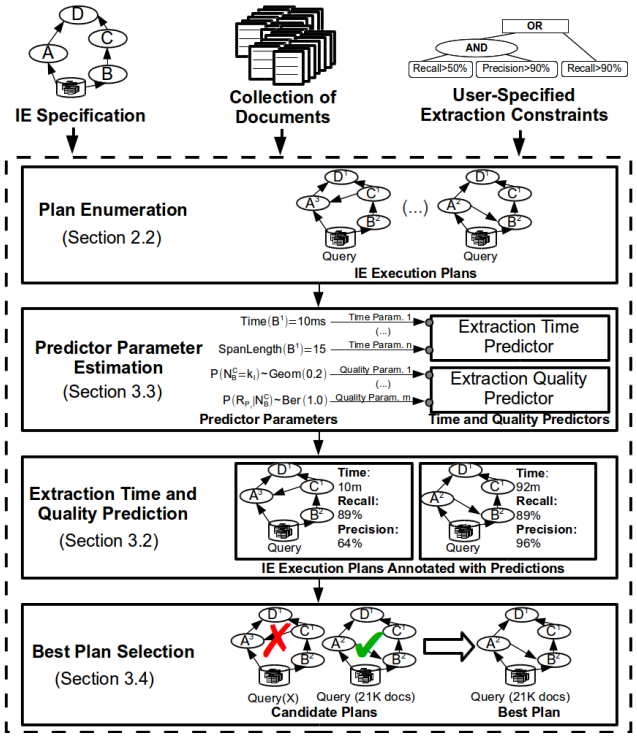

Figure 4: Holistic IE optimization process.

that check text properties and narrow the region of crawled text are executed as early as possible in the plan.

Next is *Predictor Parameter Estimation* (Section 3.3). To predict the extraction time, recall, and precision of the IE execution plans, we rely on parameters that depend on the IE specification and document collection. The objective of this step is to estimate these parameters. We divide this step into three phases: (i) *sampling*, to select the subset of input documents to use in the estimations; (ii) *extraction*, to retrieve information from the sample using the plans produced during plan enumeration; and (iii) *estimation*, to determine the optimizer parameters using the results extracted in (ii). The major challenge is the fact that the majority of the IE programs produce sparse results (i.e., most of the input documents do not contain tuples). In this case, the quality of the estimations tends to be quite low. Thus, we need to rely on clever solutions to sample the documents and to estimate the parameters, as we will discuss.

After estimating the parameters of the predictors, we use them in the *Extraction Time and Quality Prediction* step (Section 3.2), where we predict the extraction time and quality (i.e., recall and precision) of each IE execution plan. The number of tuples produced by a plan is difficult to anticipate because it largely depends on the dataset and the semantics of the IE program. For this reason, predicting recall and precision is challenging. Furthermore, we need to perform this step for a wide range of IE execution plans, which can be time consuming. We propose a novel approach to predict the recall and precision for each IE execution plan in an accurate and efficient way (Section 3.2.2).

Finally, the *Best Plan Selection* step (Section 3.4) receives as input the alternative IE execution plans, annotated with their corresponding predicted recall, precision, and extraction time per document. This step relies on the user-specified constraints on recall and precision (see Section 2.3). During this step, we need to: (i) find the IE execution plans that are able to fulfill the user-specified constraints; (ii) for these plans, determine the minimum number of documents that they must process in order to satisfy the constraints; and (iii) select the fastest IE execution plan based on the minimum number of documents that it needs to process. The key challenge is to effectively and efficiently use the information provided by the predictors to determine whether the plans can fulfill the user-specified constraints.

3.2 Extraction Time and Quality Prediction

To compare the IE execution plans produced during the Plan Enumeration step, we must predict their extraction time and quality, which we discuss next.

3.2.1 *Extraction Time Prediction*

To predict extraction time, we adapt CIMPLE's approach for this task [19]. Specifically, we predict the extraction time of an IE execution plan by summing the extraction time of all the operators in the plan. Thus, we need to determine the extraction time of each operator O when it is executed with a specific algorithm O^i. For this, our predictor relies on the following parameters: (i) $Time(O^i)$, the average extraction time of the algorithm O^i per input tuple; (ii) $SpanLength(O^i)$, the average input text span length of algorithm O^i; and (iii) the number of input tuples provided to the operator O. We can compute the first two parameters easily by using CIMPLE's prediction adapted to work on each algorithm rather than on the (coarser) operators.

The third parameter of the extraction time predictor is the number of input tuples of an operator. Our optimization approach needs to compute the number of correct and incorrect input and output tuples of each operator to predict the recall and precision of the operators (see Section 3.2.2). Thus, we can use these predictions directly to also estimate the extraction time. This is a key difference with CIMPLE's extraction time prediction, which relies on (coarser) output size estimates based on the selectivity of the operators (CIMPLE does not predict extraction quality because it can only produce exhaustive executions).

A second difference between our extraction time predictor and CIMPLE's is that the number of documents processed by our IE execution plans may vary from plan to plan. Since the extraction time grows linearly with the number of processed documents, we estimate the extraction time for a plan by predicting the extraction time per document of the plan and then linearly extrapolating that value for different numbers of documents.

3.2.2 *Extraction Quality Prediction*

Predicting the recall and precision values of each IE execution plan is significantly more challenging than predicting the extraction time, because these values highly depend on the IE specification and document collection. As discussed above, SQoUT predicts the precision and recall of its execution plans [8, 9, 10]. However, this prediction approach falls short for our holistic optimization approach. Specifically, SQoUT predicts the recall and precision for each document retrieval strategy, while treating the IE systems as black boxes. In contrast, we need to consider the impact of each internal operator in an IE program on precision and recall, which is a key contribution of our approach.

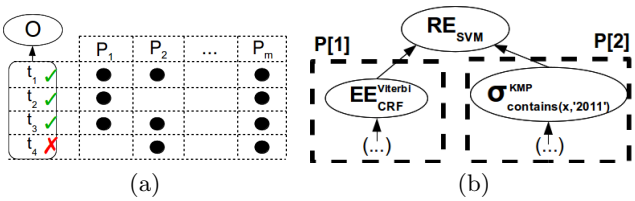

Figure 5: (a) Input tuples for an operator O; (b) operator RE_{SVM} and its input paths, $P[1]$ and $P[2]$.

Another contribution of our precision and recall predictor is its efficiency: we decompose the prediction of recall and precision for a full plan into several small tasks that determine the number of correct and incorrect tuples produced by each operator in the IE execution plan. Different IE execution plans may have some operators in common. So, when computing the number of correct and incorrect tuples at the operator level, we can reuse these results across different IE execution plans using a dynamic programming approach.

By definition, the values of precision and recall of an IE execution plan depend on the number of correct and incorrect tuples that it produces. (Recall is the fraction of correct tuples that the plan produces while precision is the fraction of produced tuples that are correct.) To predict the number of correct and incorrect tuples produced by each operator, we must estimate, in turn: (i) the number of input tuples of the operator and (ii) the number of output tuples produced by the operator when it receives a given input tuple. We now discuss how to compute these two estimates.

Operator Input Tuples: The set of input tuples of an operator O depends on the IE execution plan where it is included. There are several alternative sub-plans that may precede O in the execution plan. Each of these plans may produce very different output tuples that are then provided as input to O. Figure 5(a) shows an example of an operator O that, according to the sub-plan that precedes it (i.e., one of the execution plans $P_1, ..., P_m$), can receive a different set of input tuples. Some tuples, marked with a check sign in the figure, are correct tuples, and others, marked with a cross sign, are incorrect. If sub-plan P_2 precedes O, this operator will receive as input tuples t_1, t_3 and t_4, but not t_2. We denote the set of output tuples of each sub-plan P as T_P. We also denote the set of correct and incorrect output tuples of P as $Cor(T_P)$ and $Inc(T_P)$, respectively.

Each possible sub-plan P that precedes an operator O in the execution plan may be composed of multiple paths, denoted $P[1]$, $P[2]$, ..., $P[n]$. For example, in Figure 5(b), we show the operator RE_{SVM} from the motivating example and the two paths that provide its input tuples. We denote the path that ends with $EE^{Viterbi}_{CRF}$ as $P[1]$ and the path that ends with $\sigma^{KMP}_{contains(x,\text{'2011'})}$ as $P[2]$. The set of input tuples of O is, then, obtained by combining the individual tuples produced by each of the paths, $T_{P[1]}$, $T_{P[2]}$, ..., $T_{P[n]}$. However, the output tuples in T_P do not necessarily include all the tuples in the Cartesian product $T_{P[1]} \times T_{P[2]} \times ... \times T_{P[n]}$. In fact, IE programs usually process each document individually, so two tuples from different documents will never be combined and given as input to O.

To predict the number of correct input tuples of O, we compute, for each path $P[k]$, the average number of correct tuples generated by combining a correct output tuple of $P[k]$ with the output tuples of the other paths. We denote these values as $aCor(P[k])$. Using the values of $aCor(P[k])$, we can compute n different predictions (one per path) of the number of correct input tuples $|Cor(T_P)|$ given by $aCor(P[k]) \cdot \mathbb{E}[|Cor(T_{P[k]})|]$. While these predictions tend to be similar, they may differ. Thus, to make the final prediction as precise as possible, we need to compute the geometric mean of $aCor(P[k]) \cdot \mathbb{E}[|Cor(T_{P[k]})|]$. We rely on the geometric mean for two reasons. First, it tends to be close to the minimum value of the input estimates, which is the behavior we desire for our application. Second, when one of the individual path estimates corresponds to 0, the geometric mean is also 0. This behavior is particularly important as estimating 0 for an individual path indicates that it is likely that the corresponding operator will not receive any tuples. Thus, if we define $ACor(P)$ as the geometric mean of $aCor(P[k])$ for all paths in P, we can compute $\mathbb{E}[|Cor(T_P)|]$ with Equation 1, as follows:

$$\mathbb{E}[|Cor(T_P)|] = ACor(P) \cdot \sqrt[n]{\prod_{k=1}^{n} \mathbb{E}[|Cor(T_{P[k]})|]} \quad (1)$$

We predict the number of incorrect input tuples for O analogously, by adapting our approach to match the definition of incorrect tuples. Specifically, an input tuple is incorrect if at least one of the individual tuples that are combined to produce it is incorrect. Thus, the set of incorrect tuples given as input to O can be generated by any combination of tuples produced by the paths that precede O in the graph except those that only involve correct tuples. Like in the correct-tuple case, we rely on the parameters $aInc(P[k])$, which are the average number of incorrect tuples generated by combining an output tuple of $P[k]$ with the output tuples of the other paths. We define $AInc(P)$ as the geometric mean of $aInc(P[k])$ for all paths in P. Equation 2 shows how to predict $\mathbb{E}[|Inc(T_P)|]$ using these parameters:

$$\mathbb{E}[|Inc(T_P)|] = AInc(P) \cdot \sqrt[n]{\prod_{k=1}^{n} \mathbb{E}[|T_{P[k]}|] - \prod_{k=1}^{n} \mathbb{E}[|Cor(T_{P[k]})|]} \quad (2)$$

Operator Output Tuples: To estimate the number of output tuples of an operator when it receives an input tuple, we need to consider two factors: (i) the input tuple itself and (ii) the algorithm that executes the operator. The characteristics of the input tuple of an operator determine how much, or how little, the operator can extract. For example, consider an entity extraction operator that extracts natural disasters from sentences. Most sentences contain no references to natural disasters; a few sentences might contain one or two references; and very few sentences discuss more than two natural disasters. Therefore, when predicting the number of output tuples of an operator, we need to consider the distribution of input tuples and the number of tuples they are bound to produce.

To account for these distributions, we consider two random variables, $Cor(N_O)$ and $Inc(N_O)$, which correspond to the number of correct and incorrect tuples produced by an operator O when it receives an input tuple. The distribution of these random variables depends on: (i) the sub-plan that precedes O in the execution plan (e.g., some plans may be able to filter input sentences that are less likely to contain natural disasters, thus decreasing the probability that no tuple is produced when a tuple is given as input); and

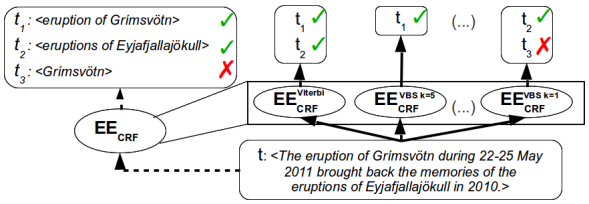

Figure 6: Tuples produced by alternative algorithms that execute operator EE_{CRF}.

(ii) the correctness of the input tuple (i.e., when an operator receives a correct input tuple it may produce correct or incorrect tuples with different probabilities). To account for these factors, we consider two Boolean random variables: (i) R_P, which indicates whether an input tuple of operator O is an output tuple of the sub-plan P; and (ii) C, which indicates whether the tuple given as input to O is correct.

The algorithm that executes the operator also influences the tuples that it produces. Some algorithms rely on approximations, so not all algorithms for a given operator produce the same set of tuples. Figure 6 shows the EE_{CRF} operator from the motivating example. This operator can be executed with multiple algorithms, including $EE_{CRF}^{Viterbi}$, which uses the Viterbi algorithm, or $EE_{CRF}^{VBS\ k=i}$, which uses the Viterbi Beam Search algorithm with parameter i. Some correct output tuples may be missing from the result of EE_{CRF} (e.g., $EE_{CRF}^{VBS\ k=5}$ produces t_1 but not t_2) and some incorrect tuples may be produced (e.g., $EE_{CRF}^{VBS\ k=1}$ produces t_3). If we assume that an operator O is bound to produce a set of tuples T_O when it receives an input tuple, then, by using a specific algorithm, O^i, the operator produces T_{O^i}, which is a subset of T_O. We can model the production of tuples in T_{O^i} as a set of Bernoulli trials over the elements of T_O. Thus, the number of correct and incorrect tuples produced by a specific algorithm O^i, $Cor(N_{O^i})$ and $Inc(N_{O^i})$, respectively, follow Binomial distributions parameterized with the probability that a tuple of T_O belongs to T_{O^i}.

We can use the previously defined random variables to predict the number of correct and incorrect tuples produced by an operator O when it receives a correct or incorrect input tuple: $\mathbb{E}[Cor(N_{O^i})|R_P, C]$, $\mathbb{E}[Inc(N_{O^i})|R_P, C]$, and $\mathbb{E}[Inc(N_{O^i})|R_P, \neg C]$.[3] Although these expected values are very different from each other, we can compute them analogously. Hence, we omit the information about the correctness of the tuples from now on. With this in mind, the general solution to compute these expected values is:

$$\mathbb{E}[N_{O^i}|R_P] = p_{O^i}(P)\frac{\sum_{k_l=0}^{\infty} k_l \cdot P(R_P|N_O = k_l) \cdot P(N_O = k_l)}{\sum_{k=0}^{\infty} P(R_P|N_O = k) \cdot P(N_O = k)} \quad (3)$$

where $p_{O^i}(P)$ is the parameter of the Binomial distribution that gives the probability that an output tuple of operator O is produced when using algorithm O^i. For more details, please refer to Appendix A.

Putting All Predictions Together: So far, we described how to determine: (i) $\mathbb{E}[|Cor(T_P)|]$ and $\mathbb{E}[|Inc(T_P)|]$, the predicted number of correct and incorrect input tuples received by an operator O when the sub-plan P precedes it in the execution plan; and (ii) $\mathbb{E}[Cor(N_{O^i})|R_P, C]$, $\mathbb{E}[Inc(N_{O^i})|R_P, C]$, and $\mathbb{E}[Inc(N_{O^i})|R_P, \neg C]$, the predicted number of output tuples that the operator is expected to produce when it receives a random tuple as input and is executed by algorithm O^i. However, we still need to understand how to put these values together to predict the total number of correct and incorrect tuples produced by each operator of the plan. All we need to do is to combine these values according to the correctness of the input and output tuples. Thus, we predict the number of correct and incorrect tuples produced by algorithm O^i when the sub-plan P precedes it in the execution plan, as follows:

$$\mathbb{E}[|Cor(T_{O^i}(T_P))|] = \mathbb{E}[|Cor(T_P)|] \cdot \mathbb{E}[Cor(N_{O^i})|R_P, C] \quad (4)$$

$$\begin{aligned}\mathbb{E}[|Inc(T_{O^i}(T_P))|] = &\mathbb{E}[|Cor(T_P)|] \cdot \mathbb{E}[Inc(N_{O^i})|R_P, C] \\ &+ \mathbb{E}[|Inc(T_P)|] \cdot \mathbb{E}[Inc(N_{O^i})|R_P, \neg C]\end{aligned} \quad (5)$$

We explain next how to compute the parameters required in Equations 4 and 5.

3.3 Predictor Parameter Estimation

The time and quality predictors presented in Section 3.2 rely on several parameters, namely: (i) $P(N_O = k_l)$, the probability distribution for the number of tuples produced by O when it receives a random input tuple; (ii) $P(R_P|N_O = k_l)$, the parameter of the Bernoulli distribution for the presence of an input tuple of operator O that produces k_l output tuples, when P precedes O in the IE execution plan; (iii) $p_{O^i}(P)$, the parameter of the Binomial distribution that gives the probability that an output tuple of an operator O is produced when using the algorithm O^i; (iv) $aCor(P[k])$ and $aInc(P[k])$, which correspond to the average number of correct and incorrect tuples provided to an operator O when combining output tuples of the path $P[k]$ with tuples produced by the other paths; and (v) $Time(O^i)$, the average extraction time per input tuple for each operator, and $SpanLength(O^i)$, which is the average text span length used in the prediction of the O^i execution time.

Determining the values of the predictor parameters is a challenging task that we perform in three phases: (i) sampling, (ii) extraction, and (iii) estimation. In the *sampling* phase, we find a sample of the input document collection. Most document collections for IE are sparse (i.e., only a few documents contain results). For these collections, it is likely that no document in the sample produces results, leading to low quality estimations. We apply stratified sampling [17] to increase the probability of including in the sample documents containing results. We partition the document collection according to the retrieval strategies that are able to retrieve the documents. We chose this partition method because retrieval strategies are strongly related to the probability that a document produces extraction results (e.g., a document retrieved by a query-based strategy is more likely to be useful than a document that only scan can find).

During the *extraction* phase, we provide the document sample as input to the IE execution plans in order to produce the tuples used to estimate the predictor parameters. Executing the entire set of IE execution plans with all the documents in the sample would be a time-consuming task. Since different IE execution plans have common operators that are executed with the same algorithms, we rely on dynamic programming to make sure that no operator is executed twice with the same algorithm and input tuple, even if it is present in several IE execution plans.

[3] Any tuple that results from an incorrect input is also incorrect. Thus, we do not need to compute $\mathbb{E}[Cor(N_{O^i})|R_P, \neg C]$ because its value is always 0.

In the *estimation* phase, we use the tuples produced during the extraction phase to estimate the parameters of the predictors. One possible estimation method would be maximum likelihood (ML) [21], which chooses the parameter values that maximize the probability of observing the sample data. Unfortunately, ML performs poorly on small and sparse samples. To account for such cases, we use maximum a-posteriori (MAP) estimation [21]. MAP assumes we have additional knowledge about the parameter to estimate, θ, in the form of a probability distribution $P(\theta)$, called the *prior distribution*. For the MAP estimations to be effective, we must properly model the prior distribution for our specific estimation. We rely on the conjugate priors of the probability distributions we use in our prediction model [21] and adapt their parameters to make them interpretable in the context of our estimation. These parameters are: (i) Str, the strength of the prior relative to the likelihood of the sample data; and (ii) E, the expected value of the probability distribution when we estimate its parameter only with information from the prior (i.e., without sample data). The value of E is dependent on the probability distribution, so we rely on two variants: (i) $E_{N_O} \in \mathbb{R}_0$, which we use for estimating $P(N_O = k_l)$; and (ii) $E_{Ber} \in [0, 1]$, which we use for estimating parameters of any Bernoulli distribution. Due to space constraints, we omit the details for computing the MAP estimations for each parameter of the predictors. For further details, please refer to [20].

Since we use stratified sampling, our document sample may not constitute a good representation of the document collection. (The sampling is biased to increase the probability of selecting documents containing tuples.) Thus, a predictor based directly on the observations of the document sample may be biased. To avoid this problem, we start by predicting the values of recall, precision, and extraction time for each sample stratum. Then, we compute the weighted sum of these values according to the proportion of each stratum in the entire input document collection. This approach produces an unbiased predictor [17].

With the above estimation methodology, we can find the parameters of the predictors of our optimizer for particular IE specifications and document collections. Thus, we can use the predictors to estimate the recall, precision, and extraction time of alternative IE execution plans and, in turn, choose the best plan, as discussed next.

3.4 Best Plan Selection

The last optimization step consists of choosing the fastest IE execution plan and the minimum number of documents it needs to process while satisfying the user-specified constraints. For this task, we must perform three steps: (i) determining the minimum number of documents that each IE execution plan must process to satisfy the constraints; (ii) predicting the extraction time of each IE execution plan when processing the number of documents determined in (i); and (iii) choosing the fastest IE execution plan.

In step (i), we execute the *findNumbOfDocsRange* algorithm (see Algorithm 1) for finding the range of the possible numbers of documents that can be processed by each IE execution plan P while still satisfying the constraints. This algorithm starts at the root of the and-or tree, denoted T. If the root is an *and* node (respectively, an *or* node), the result of the algorithm corresponds to the intersection (respectively, union) of the ranges of numbers we obtain from recursively invoking the algorithm giving T's child nodes as input. Finally, if the root is a leaf node that imposes a condition on the recall or precision of the IE execution plan, then, the result depends on the *getConditionRange* method. This method returns a set containing the numbers of documents for which the specified condition is satisfied.

Algorithm 1: *findNumbOfDocsRange*(P, E, T, n)

Input: IE execution plan P
Extraction time and quality predictions E
And-or tree T representing recall/precision constraints
Number of documents n in the collection
Output: Number of document range R satisfying the conditions of T when executing P

```
1  if isAndNode(root(T)) then
2      R = [0, n];
3      foreach T_c ∈ children(T) do
4          R = R ∩ findNumbOfDocsRange(P, E, T_c, n);
5  else if isOrNode(root(T)) then
6      R = ∅;
7      foreach T_c ∈ children(T) do
8          R = R ∪ findNumbOfDocsRange(P, E, T_c, n);
9  else
10     R = T.getConditionRange(P, E, n);
11 return R;
```

After executing the *findNumbOfDocsRange* algorithm, we need to determine the exact number of documents that each IE execution plan must process. First, we discard any IE execution plan for which *findNumbOfDocsRange* returns an empty range, since such plan could never satisfy the imposed constraints. Second, for the remaining plans, we choose the minimum number of documents that satisfies the constraints. This corresponds to the lowest integer in the range returned by *findNumbOfDocsRange* for each execution plan.

In step (ii), we predict the extraction time for each IE execution plan that processes the number of documents computed in step (i). We obtain this value by extrapolating the extraction time values provided by the extraction time predictor. For instance, if the predictor determines that an execution plan takes one second per document, then we assume it takes 500 seconds to process 500 documents. Finally, in step (iii) we choose the fastest IE execution plan.

4. EXPERIMENTAL SETTINGS

In this section, we describe the experimental settings for the evaluation of our IE optimization approach.

Datasets: We used two real-world document collections in our experiments (Table 2): (i) the *Homepages* dataset, which consists mostly of Web pages of Computer Science researchers; and (ii) the *NYT Annotated Corpus* [18], which consists of New York Times articles from 1987 to 2007. We divided both datasets into disjoint training, development, and test sets. We used the training sets to train retrieval strategies (e.g., Qxtract [1] or PRDualRank [5]) and IE techniques (e.g., SVM for relationship extraction [6]) that IE programs rely on. We used the development sets in initial experiments to select the values for the parameters of the optimizer and show experimental evidence about implementation decisions. Finally, we used the test sets to compare our optimizer with state-of-the-art IE systems using the parameters we selected with the development sets.

IE Programs: We designed four diverse IE programs that extract the following relationships (Table 3): (i) *ConferencesDates*, (ii) *Advises*, (iii) *DiseasesOutbreaks*, and (iv) *OrganizationAffiliation*. The first two programs extract in-

Dataset	Training	Development	Test
Homepages	3,301	3,146	3,248
NYT	97,258	671,457	1,086,944

Table 2: Characteristics of our datasets.

IE Programs	Semantics	Number of Plans
ConferencesDates	Conferences and their dates.	90
Advises	Students and their academic advisors.	2,160
DiseasesOutbreaks	Disease outbreaks and their time.	3,240
OrganizationAffiliation	Organizations and people that are affiliated with them.	324

Table 3: IE programs used in our experiments.

formation from the Homepages dataset while the others do so from the NYT dataset. These IE programs cover the most important families of IE techniques, and use them in workflows with different complexities (to see their IE specifications, please refer to [20]). The first three programs are rule-based, using dictionaries and regular expressions to extract entities, and using the distance between entities or the HTML structure to determine if the entities are related. *OrganizationAffiliation* is based on machine learning, relying on Hidden Markov Models [4] and automatically generated patterns [23] to extract entities, and on SVM [6] to extract relationships between the entities. *ConferencesDates* and *OrganizationAffiliation* are dense relationships that occur often in the datasets, while *Advises* and *DiseasesOutbreaks* are sparse relationships that occur rarely. To implement the operators of these IE programs, we relied on two off-the-shelf libraries: Lingpipe[4] and OpenNLP[5]. We also used two tools developed in our research groups to train machine learning models for IE, namely, E-txt2db[6] and REEL[7]. Finally, we implemented our query-based retrieval strategies using Lucene[8] as a search interface for the text collections.

Techniques for Comparison: We compare our approach against the following state-of-the-art IE systems:

CIMPLE is our implementation of CIMPLE, based on the description provided in [19]. The original CIMPLE does not support IE execution plans that produce different recall values. Thus, for fairness in the comparison, we added a parameter to control the percentage of processed documents. Given a desired recall value, our implementation of CIMPLE processes that same percentage of documents.

SystemT corresponds to the IBM implementation of SystemT [16, 12] in Java. Similarly to our implementation of CIMPLE, we added an extra parameter that lets a user impose recall constraints.

SQoUT is a modification of the original SQoUT[9] code to support specifications of IE programs as graphs of operators.

SQoUT-Boosted is a variation of the SQoUT baseline that not only optimizes the document retrieval strategy, but also chooses between the execution plans produced by CIMPLE and SystemT to perform the extraction. Thus, this version of SQoUT corresponds to a direct combination of the SQoUT retrieval strategies with the CIMPLE and SystemT optimization techniques, without our holistic optimization of all aspects of the process.

The second set of IE optimization techniques we compared consists of variants of our optimizer (Section 3):

Optimized-Ret considers only IE execution plans based on the choice of the document retrieval strategy.

Optimized-Alg considers only IE execution plans based on the choice of the algorithms for each operator.

Optimized-Order considers only IE execution plans based on the choice of the operator execution order.

Holistic-MAP corresponds to our optimization solution exactly as described in Section 3.

Holistic-ML uses maximum likelihood estimation to determine the optimizer parameters (Section 3.3).

We ran our experiments on a grid of 12 computers with a uniform configuration: Intel Core2 Quad (Q6600) processors, with 8 GB of RAM, and OS Linux openSUSE 11.4 "Celadon." We implemented all the techniques in Java and ran them on a Sun Java 1.6.0_26 64-bit virtual machine. To account for the randomness introduced by the document sampling process, we repeated each experiment 10 times over independent samples. We report the average value of the metrics over the 10 executions.

Parameter Settings: Our optimizer relies on two types of parameters: (i) the document sample size and (ii) the parameters of the prior distribution (i.e., Str, E_{N_O}, and E_{Ber}; see Section 3.3). We tested multiple combinations of these parameters over the development sets to understand their impact on the average estimation error of precision and recall, as well as on the optimization process itself (see [20] for details). In Section 5, we report on the impact of the sample size on prediction quality. Overall, we concluded that the optimizer decisions tend to become stable with a sample of 2,000 documents. However, to handle both large (e.g., our NYT dataset) and small collections (e.g., our Homepages dataset), we set the sample size as the minimum of 2,000 and 1% of the collection. Regarding the prior parameters, we concluded that there are particular values for these parameters that lead to good estimations. From the combinations that we tested, we chose: $Str = 0.05$, $E_{N_O} = 0.5$, and $E_{Ber} = 0.5$.

5. EXPERIMENTAL RESULTS

Sample Size and Prediction Quality: To understand the impact of the sample size on the quality of the recall and precision predictions, we executed the first three steps of the optimization process described in Figure 4 for the two large-scale programs from Section 4.[10] In each execution, we varied the sample size from 200 to 4,000 documents from the development set, with steps of 200 documents, and measured: (i) the time needed to execute the optimization process; and (ii) the predicted recall and precision for each IE execution plan. Figure 7 reports the optimization time and prediction errors, averaged over the alternative IE execution plans. For *OrganizationAffiliation*, the prediction error is low, below 2%, even for small sample sizes. For

[4]http://alias-i.com/lingpipe/
[5]http://opennlp.apache.org/
[6]http://web.ist.utl.pt/ist155840/etxt2db/
[7]http://reel.cs.columbia.edu/
[8]http://lucene.apache.org/core/
[9]http://sqout.stern.nyu.edu/

[10]We also performed a similar experiment for the small-scale programs; see [20] for details.

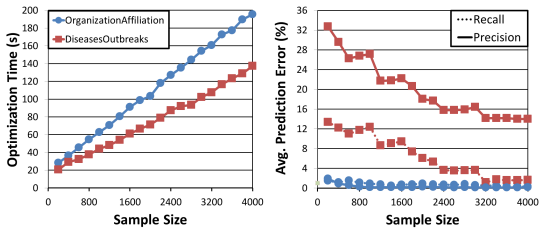

Figure 7: Optimization time, and recall and precision prediction error, as a function of sample size.

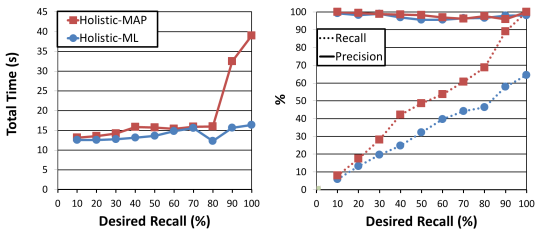

Figure 8: Total time, recall, and precision for *Advises*, for Holistic-MAP and Holistic-ML.

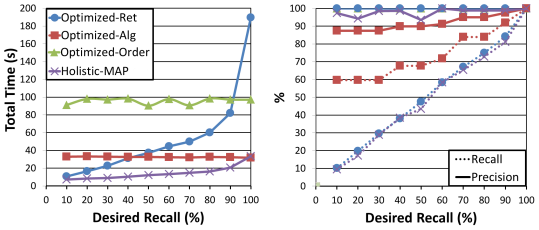

Figure 9: Total time, recall, and precision for *ConferencesDates*, for individual implementation choices as well as the full set of plans (Holistic-MAP).

DiseasesOutbreaks, the errors are higher than for *OrganizationAffiliation*[11]. *DiseasesOutbreaks* is a sparse relationship extracted from a large dataset: for small sample sizes, the sample is unlikely to contain any document that produces results, which, in turn, leads to predictions with lower quality than for *OrganizationAffiliation*. Nevertheless, our optimizer still identifies a fast IE execution plan for *DiseasesOutbreaks* with recall and precision close to the user-defined constraints. Finally, the optimization time for all the IE programs is approximately linear in the sample size. Fortunately, small samples are sufficient for our approach, as prediction errors stabilize after seeing a relatively small number of documents.

Comparison of ML and MAP-based Optimizers: In Section 3.3, we claimed that maximum likelihood (ML) estimation is not a good solution for our optimizer. Instead, we proposed the use of maximum a posteriori (MAP) estimators. We now report experimental evidence over the test sets for this claim. (From now on, all results are over the test sets.) We compare the extraction results of the IE execution plans chosen by Holistic-ML and Holistic-MAP. In each execution, we varied the value of the desired recall between 10% and 100%. For each optimization alternative, we measured the total CPU time for the optimization and extraction with the chosen plan, as well as the recall and precision of the results. Our experiments showed that the "sparse" IE programs (i.e., *Advises* and *DiseasesOutbreaks*) are most affected by the choice between MAP and ML. Thus, we present the results for *Advises* in Figure 8 (the results are similar for *DiseasesOutbreaks*). Despite being efficient in all cases, Holistic-ML is not able to fulfill the constraints for high recall values. Starting at 80%, the extraction times of Holistic-MAP and Holistic-ML are equivalent. However, for higher recall values, Holistic-MAP becomes slower since it changes the IE execution plan to be able to comply with the high recall demands. Holistic-ML keeps the same plan used for lower values of recall, just by processing more documents. Even though this plan ends up being significantly faster, it does not satisfy the high recall demands. Holistic-ML chose this plan because, during the Predictor Parameter Estimation step of the optimization process, all the documents of the sample that produced tuples could be retrieved by a query-based strategy. This led to the wrong conclusion that a query-based strategy was enough to obtain 100% recall, when, in fact, there were some tuples that only a scan-based strategy could retrieve.

Impact of Individual Implementation Choices: We now compare the optimizers that consider only the individual implementation choices presented in Section 2.2 (i.e., Optimized-Ret, Optimized-Alg, and Optimized-Order) with Holistic-MAP. Due to space constraints, we focus on *ConferencesDates* (see Figure 9) as its results show most of the characteristics that we observed for other programs. As expected, Holistic-MAP obtains the fastest plan in every case.[12] This plan obtains recall values that are close to the desired levels. Optimized-Ret is almost as fast as Holistic-MAP when the desired recall is low. However, as the target recall increases, Optimized-Ret selects plans that are slower than those from other approaches, because it relies on scan retrieval strategies without any additional optimization. Optimized-Alg and Optimized-Order consistently produce costly plans, because their plans process the entire document collection, even for low values of desired recall.

Impact of Precision Constraints: Our optimizer handles both recall and precision user constraints (see Section 2.3). So far, we have focused on recall. We now consider precision constraints in conjunction with recall constraints. Figure 10 shows the results of this experiment for the *OrganizationAffiliation* program where each curve represents Holistic-MAP with a specific precision constraint, as a function of the value for the recall constraint. As expected, the more demanding we are with the desired precision, the slower the resulting IE execution plan is. However, we do not need to significantly lower the desired precision value to obtain significant efficiency gains. For instance, if we request 99% precision, the resulting IE execution plan is

[11]There is generally no significant difference between the number of times the predictions are smaller or larger than the real recall and precision values. However, for small sample sizes, over 80% of the precision errors correspond to underestimations due to the smoothed predictions of MAP estimators (i.e., some plans produce no incorrect tuples yet their estimated precision is below 100%).

[12]When we consider optimization and extraction times combined, Holistic-MAP is also the fastest, with one exception: for *Advises*, Optimized-Ret is faster than Holistic-MAP for low recall values because the optimization time is shorter for Optimized-Ret and both extraction times are close to zero.

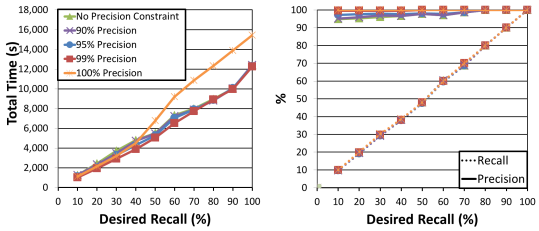

Figure 10: Total time, recall, and precision for *OrganizationAffiliation*, for Holistic-MAP, with different values of desired precision, and as a function of desired recall.

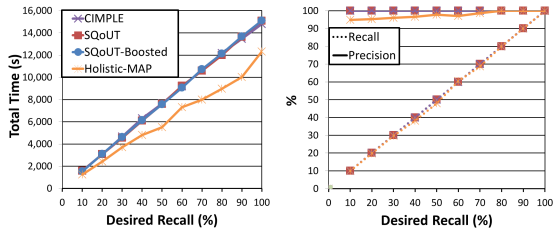

Figure 11: Total time, recall, and precision for *OrganizationAffiliation*, for CIMPLE, SQoUT, SQoUT-Boosted, and Holistic-MAP.

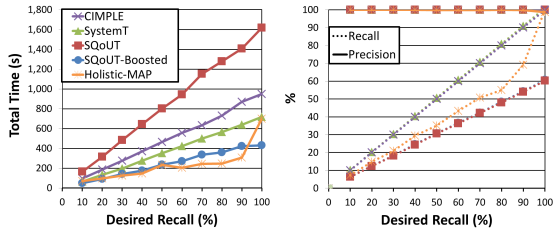

Figure 12: Total time, recall, and precision for *DiseasesOutbreaks*, for CIMPLE, SystemT, SQoUT, SQoUT-Boosted, and Holistic-MAP.

already much faster than when we request 100% precision: MAP estimators produce smoothed precision predictions for plans with approximate operators or imposing a particular execution order. Hence, the probability of these plans producing incorrect tuples will always be non-zero, which in turn makes their expected precision below 100%. Therefore, with a 100% precision constraint, Holistic-MAP discards these plans in favor of a slower plan that cannot produce incorrect tuples. Also, for any value of desired precision below 90%, the extraction time, recall, and precision results are the same as when we do not impose any precision constraint: when an operator produces too many incorrect tuples, the execution time of the operators that follow it in the IE execution plan tends to increase. Thus, the optimizer discards plans with low precision since they tend to be slow.

Comparison With the State-of-the-Art Techniques: We now compare our Holistic-MAP optimizer with the state-of-the-art approaches discussed in Section 4, as a function of target recall and without precision constraints.[13] For brevity, Figures 11 and 12 focus on the IE programs that run on the (larger) NYT dataset, namely, *OrganizationAffiliation* — as an example of a dense relationship — and *DiseasesOutbreaks* — as an example of a sparse relationship. (The results for the other relations are analogous; see [20].) Overall, Holistic-MAP is faster than CIMPLE and SystemT. For *DiseasesOutbreaks*, when the desired recall is below 90%, this difference is substantial, as Holistic-MAP can rely on query-based retrieval strategies to process fewer documents than CIMPLE and SystemT. As expected, CIMPLE and SystemT tend to obtain recall that is close to the desired levels: in our versions of CIMPLE and SystemT (see Section 4), the recall constraints only influence the percentage of documents that the IE execution plan uses, and these documents are chosen randomly. Interestingly, for dense relationships, Holistic-MAP is also able to closely match the desired recall, despite the fact that the Holistic-MAP IE execution plans may exploit approximations, for efficiency. In contrast, for a sparse relationship like *DiseasesOutbreaks*, Holistic-MAP does not match the recall constraints closely. However, as we increase the desired recall values, Holistic-MAP progressively corrects this behavior and, for example, obtains a perfect recall of 100% when we request it.

Our experiments also show the superiority of Holistic-MAP over the SQoUT baselines. To see why, note first that SQoUT is substantially slower than SQoUT-Boosted in most cases,[14] as expected: SQoUT-Boosted optimizes both the document retrieval strategy and the IE plan itself (as discussed in Section 4, SQoUT-Boosted is a strong baseline that optimizes all components of the IE process separately, combining state-of-the-art strategies but without our holistic approach); in contrast, SQoUT optimizes only the retrieval strategy. But SQoUT-Boosted has limitations relative to Holistic-MAP. For *DiseasesOutbreaks* (Figure 12), SQoUT-Boosted finds a fast execution plan but its recall is low: SQoUT-Boosted uses ML estimations, so it suffers from the limitations of ML discussed above. The results for *OrganizationAffiliation* (Figure 11) also reveal that SQoUT-Boosted is less efficient than Holistic-MAP for most values of desired recall: the queries on which SQoUT-Boosted relies for document retrieval, which are trained with automatic query generation [1, 5], could not select appropriate documents from this large dataset even for a 10% target recall. Thus, SQoUT-Boosted relies on an exhaustive scan retrieval strategy for every case, with a negative impact on extraction time. In contrast, Holistic-MAP exploits the semantics of the IE program and finds effective queries—based on resources from rule-based operators like dictionaries and patterns—that are better than those from automatic query generation. Furthermore, the *OrganizationAffiliation* results highlight the importance of our holistic optimization approach: for high target recall values, starting at 40%, Holistic-MAP holistically determines that it could match the recall constraints by using, for efficiency, either a query-based document retrieval strategy or approximate

[13] CIMPLE, SystemT, and the SQoUT baselines do not exploit optimization opportunities from using approximate algorithms or execution orders. Hence, their execution plans remain unchanged even in the presence of precision constraints. Our techniques adapt to the precision constraints. However, as shown above (Figure 10), the efficiency of our techniques is not substantially affected as long as the requested precision is below 90%.

[14] The only exception is *OrganizationAffiliation*, in which SQoUT and SQoUT-Boosted are equivalent.

algorithms for IE operators, but not both; using both would result in query plans not matching the recall constraints, a conclusion that could not be derived by optimizing each implementation choice independently. In summary, these experiments highlight the merits of our holistic optimization approach over the (strong) baseline approach of simply combining the existing optimization components from the state-of-the-art approaches, as embodied in SQoUT-Boosted.

6. RELATED WORK

In this paper, we presented an approach for the automatic optimization of IE programs. Our work is closest to CIMPLE [19], SystemT [12, 16], and SQoUT [8, 10]. Throughout this paper, we extensively compared our approach with these systems, both conceptually and experimentally. So, we do not repeat this discussion here. Instead, we briefly discuss document retrieval strategies for IE, since they can serve as an important optimization building block for IE programs. Specifically, early work on IE [3, 7] already relied on classifiers to filter unnecessary documents. Qxtract [1] and PRDualRank [5] automatically produce queries to find the promising documents for extraction. In our experiments, we used Qxtract and PRDualRank to enrich the set of alternatives for document retrieval that our approach and SQoUT use. As new approaches for document retrieval arise, they can easily be integrated in our optimization approach, making it possible to improve the results even further.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a holistic approach for IE optimization that addresses the limitations of the state-of-the-art systems. Our approach exploits, collectively, three optimization opportunities, namely, the choices of: (i) algorithms for IE operators; (ii) operator execution order; and (iii) document retrieval strategy. Our experiments show the gains of our approach in comparison to the state-of-the-art IE optimization systems, and against an optimization approach that combines the existing strategies without our holistic view. Our work can be extended in multiple directions. First, we consider using multi-objective optimization techniques instead of relying on user-imposed constraints on recall and precision. Second, we plan to adapt our optimization approach for modern parallel computation environments (e.g., Map-Reduce).

Acknowledgments: This work was supported by *Fundação para a Ciência e a Tecnologia*, under Project PEstOE/ EEI/LA0021/2011 and Ph.D. Grant SFRH/BD/61393/2009, as well as by European Project METANET4U, Number 270893. This material is also based upon work supported by the National Science Foundation under Grant IIS-08-11038.

8. REFERENCES

[1] E. Agichtein and L. Gravano. Querying text databases for efficient information extraction. In *ICDE*, pages 113–124, 2003.

[2] A. Baronchelli, E. Caglioti, V. Loreto, and E. Pizzi. Dictionary based methods for information extraction. *Physica A: Statistical Mechanics and its Applications*, 342:294–300, 2004.

[3] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB*, pages 172–183, 1998.

[4] A. Ekbal and S. Bandyopadhyay. A Hidden Markov Model based named entity recognition system: Bengali and Hindi as case studies. *Lecture Notes in Computer Science*, 4815:545–552, 2007.

[5] Y. Fang and K. C.-C. Chang. Searching patterns for relation extraction over the web: Rediscovering the pattern-relation duality. In *WSDM*, pages 825–834, 2011.

[6] C. Giuliano, A. Lavelli, and L. Romano. Exploiting shallow linguistic information for relation extraction from biomedical literature. In *EACL*, pages 3–7, 2006.

[7] R. Grishman, S. Huttunen, and R. Yangarber. Information extraction for enhanced access to disease outbreak reports. *Journal of Biomedical Informatics*, 35:236–246, 2002.

[8] 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*, pages 265–276, 2006.

[9] A. Jain, A. Doan, and L. Gravano. Optimizing SQL queries over text databases. In *ICDE*, pages 636–645, 2008.

[10] A. Jain and P. Ipeirotis. A quality-aware optimizer for information extraction. *ACM Transactions on Database Systems*, 34:5:1–5:48, 2009.

[11] D. Knuth, J. Morris, Jr., and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[12] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: A system for declarative information extraction. In *SIGMOD*, pages 7–13, 2009.

[13] B. Lowerre. The Harpy speech understanding system. In *Readings in speech recognition*. Morgan Kaufmann Publishers Inc., 1990.

[14] A. McCallum. Information Extraction: Distilling structured data from unstructured text. *ACM Queue*, 3:48–57, 2005.

[15] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *CONLL*, pages 188–191, 2003.

[16] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, pages 933–942, 2008.

[17] J. A. Rice. *Mathematical statistics and data analysis*. Duxbury Press, 2^{nd} edition, 1994.

[18] E. Sandhaus. The New York Times Annotated Corpus. In *Linguistic Data Consortium*, 2008.

[19] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.

[20] G. Simões, H. Galhardas, and L. Gravano. When speed has a price: Fast information extraction using approximate algorithms. Technical Report 3/2013, INESC-ID, June 2013. URL: `http://www.inesc-id.pt/ficheiros/publicacoes/8982.pdf`.

[21] H. W. Sorenson. *Parameter estimation: Principles and problems*. M. Dekker, 1980.

[22] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.

[23] C. Whitelaw, A. Kehlenbeck, N. Petrovic, and L. Ungar. Web-scale named entity recognition. In *CIKM*, pages 123–132, 2008.

A. PREDICTING OPERATOR OUTPUT

A key point in our optimization approach (Section 3.2.2) is the estimation of the number of correct and incorrect tuples produced by each algorithm O^i when it receives an input tuple: $\mathbb{E}[Cor(N_{O^i})|R_P, C]$, $\mathbb{E}[Inc(N_{O^i})|R_P, C]$, and $\mathbb{E}[Inc(N_{O^i})|R_P, \neg C]$. We now provide a general explanation of our estimation, which can be easily instantiated for $Cor(N_{O^i})$, $Inc(N_{O^i})$, C, and $\neg C$. Specifically, using the definition of expected value, we start with $\mathbb{E}[N_{O^i}|R_P] = \sum_{k_o=0}^{\infty} k_o \; P(N_{O^i} = k_o|R_P)$. Since the number of O^i output tuples depends on the distribution of tuples produced by the operator O, we can use marginalization to make our estimation dependent on N_O (i.e., $\mathbb{E}[N_{O^i}|R_P] = \sum_{k_o=0}^{\infty} k_o \sum_{k_l=k_o}^{\infty} P(N_{O^i} = k_o|N_O = k_l, R_P)P(N_O = k_l|R_P)$). Then, we algebraically change the order of the sums, $\mathbb{E}[N_{O^i}|R_P] = \sum_{k_l=0}^{\infty} P(N_O = k_l|R_P) \; \mathbb{E}[N_{O^i}|N_O = k_l, R_P]$. Since N_{O^i} follows a Binomial distribution with parameters k_l and $p_{O^i}(P)$, $\mathbb{E}[N_{O^i}|N_O = k_l, R_P] = k_l \cdot p_{O^i}(P)$. Finally, we use the Bayes rule, $P(N_O = k_l|R_P) = \frac{P(R_P|N_O=k_l)P(N_O=k_l)}{\sum_{k=0}^{\infty} P(R_P|N_O=k)P(N_O=k)}$, leading to: $\mathbb{E}[N_{O^i}|R_P] = p_{O^i}(P) \cdot \frac{\sum_{k_l=0}^{\infty} k_l \cdot P(R_P|N_O=k_l) \cdot P(N_O=k_l)}{\sum_{k=0}^{\infty} P(R_P|N_O=k) \cdot P(N_O=k)}$. For further details on the intermediate steps that lead to these equations, please refer to [20].