Incremental Information Extraction Using Relational Databases

Luis Tari, *Student Member*, *IEEE Computer Society*, Phan Huy Tu, Jörg Hakenberg, Yi Chen, *Member*, *IEEE Computer Society*, Tran Cao Son, Graciela Gonzalez, and Chitta Baral

Abstract—Information extraction systems are traditionally implemented as a pipeline of special-purpose processing modules targeting the extraction of a particular kind of information. A major drawback of such an approach is that whenever a new extraction goal emerges or a module is improved, extraction has to be reapplied from scratch to the entire text corpus even though only a small part of the corpus might be affected. In this paper, we describe a novel approach for information extraction in which extraction needs are expressed in the form of database queries, which are evaluated and optimized by database systems. Using database queries for information extraction enables generic extraction and minimizes reprocessing of data by performing incremental extraction to identify which part of the data is affected by the change of components or goals. Furthermore, our approach provides automated query generation components so that casual users do not have to learn the query language in order to perform extraction. To demonstrate the feasibility of our incremental extraction approach, we performed experiments to highlight two important aspects of an information extraction system: efficiency and quality of extraction results. Our experiments show that in the event of deployment of a new module, our incremental extraction approach reduces the processing time by 89.64 percent as compared to a traditional pipeline approach. By applying our methods to a corpus of 17 million biomedical abstracts, our experiments show that the query performance is efficient for real-time applications. Our experiments also revealed that our approach achieves high quality extraction results.

Index Terms—Text mining, query languages, information storage and retrieval.

1 Introduction

It is estimated that each year more than 600,000 articles are published in the biomedical literature, with close to 20 million publication entries being stored in the Medline database.[1] To uncover information from such a large corpus of documents, it is vital to address the information needs in an automated manner. The field of information extraction (IE) seeks to develop methods for fetching structured information from natural language text. Examples of structured information are the extraction of entities and relationships between entities.

IE is typically seen as a one-time process for the extraction of a particular kind of relationships of interest from a document collection. IE is usually deployed as a pipeline of special-purpose programs, which include sentence splitters, tokenizers, named entity recognizers, shallow or deep syntactic parsers, and extraction based on a collection of patterns. The high demand of IE in various domains results in the development of frameworks such as UIMA [1] and GATE [2], providing a way to perform extraction by defining workflows of components. This type of extraction frameworks is usually file based and the processed data can be utilized between components. In this traditional setting, relational databases are typically not involved in the extraction process, but are only used for storing the extracted relationships.

While file-based frameworks are suitable for one-time extraction, it is important to notice that there are cases when IE has to be performed repeatedly even on the same document collection. Consider a scenario where a named entity recognition component is deployed with an updated ontology or an improved model based on statistical learning. Typical extraction frameworks would require the reprocessing of the entire corpus with the improved entity recognition component as well as the other unchanged text processing components. Such reprocessing can be computationally intensive and should be minimized. For instance, a full processing for information extraction on 17 million Medline abstracts took more than 36 K hours of CPU time using a single-core CPU with 2-GHz and 2 GB of RAM.[2] Work by [4], [5] addresses the needs for efficient extraction of evolving text such as the frequent content updates of web documents but such approaches do not handle the issue of changed extraction components or goals over static text data.

In this paper, we propose a new paradigm for information extraction. In this extraction framework, intermediate

1. For Medline statistics, see http://www.nlm.nih.gov/bsd/licensee/2009_stats/2009_Totals.html.

- *L. Tari, P.H. Tu, J. Hakenberg, Y. Chen, G. Gonzalez, and C. Baral are with the Department of Computer Science and Engineering, Arizona State University, PO Box 878809, Tempe, AZ 85287-8809. E-mail: {luis.tari, jhakenbe, yi, Graciela.Gonzalez, chitta}@asu.edu, tuphanhuy@gmail.com.*
- *T.C. Son is with the Department of Computer Science, New Mexico State University, PO Box 30001, MSC CS Las Cruces, NM 88003. E-mail: tson@cs.nmsu.edu.*

Manuscript received 24 Oct. 2009; revised 29 Mar. 2010; accepted 11 June 2010; published online 26 Oct. 2010.
Recommended for acceptance by N. Bruno.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2009-10-0735.
Digital Object Identifier no. 10.1109/TKDE.2010.214.

2. Note that the Link Grammar parser [3] contributes to a large portion of the time spent in text processing.

output of each text processing component is stored so that only the improved component has to be deployed to the entire corpus. Extraction is then performed on both the previously processed data from the unchanged components as well as the updated data generated by the improved component. Performing such kind of *incremental extraction* can result in a tremendous reduction of processing time.

To realize this new information extraction framework, we propose to choose database management systems over file-based storage systems to address the dynamic extraction needs. Our proposed information extraction is composed of two phases:

- *Initial Phase.* We perform a one-time parse, entity recognition, and tagging (identifying individual entries as belonging to a class of interest) on the whole corpus based on the current knowledge. The generated syntactic parse trees and semantic entity tagging of the processed text is stored in a relational database, called *parse tree database* (PTDB).
- *Extraction Phase.* Extraction is then achieved by issuing database queries to PTDB. To express extraction patterns, we designed and implemented a query language called *parse tree query language* (PTQL) that is suitable for generic extraction. Note that in the event of a change to the extraction goals (e.g., the user becomes interested in new types of relations between entities) or a change to an extraction module (e.g., an improved component for named entity recognition becomes available), the responsible module is deployed for the entire text corpus and the processed data are populated into the PTDB. Queries are issued to identify the sentences with newly recognized mentions. Then extraction can be performed only on such affected sentences rather than the entire corpus. Thus, we achieve incremental extraction, which avoids the need to reprocess the entire collection of text unlike the file-based pipeline approaches.

  Using database queries instead of writing individual special-purpose programs, information extraction becomes generic for diverse applications and becomes easier for the user. However, writing such queries may still require much users' effort. To further reduce users' learning burden, we propose algorithms that can automatically generate PTQL queries from training data or a user's keyword queries.

We highlight the contributions of this paper.

- *Novel Database-Centric Framework for Information Extraction.* Unlike the traditional approaches, where IE is achieved by special-purpose programs and databases are only used for storing the extraction results, we propose to store intermediate text processing output in a database, *parse tree database*. This approach minimizes the need of reprocessing the entire collection of text in the presence of new extraction goals and deployment of improved processing components.
- *Query Language for Information Extraction.* Information extraction is expressed as queries on the parse tree database. As query languages such as XPath and XQuery are not suitable for extracting linguistic patterns [6], we designed and implemented a query language called *parse tree query language*, which allows a user to define extraction patterns on grammatical structures such as constituent trees and linkages. Since extraction is specified as queries, a user no longer needs to write and run special-purpose programs for each specific extraction goal.
- *Automated Query Generation.* Learning the query language and manually writing extraction queries could still be a time-consuming and labor-intensive process. Moreover, such an ad hoc approach is likely to cause unsatisfactory extraction quality. To further reduce a user's effort to perform information extraction, we design two algorithms to automatically generate extraction queries, in the presence and in the absence of training data, respectively.

The rest of the paper is organized as follows: we first present a short background on IE and Link Grammar parsing in Section 2. In Section 3, the system architecture of our extraction framework is discussed in details, including the PTDB, the query language PTQL, and evaluation of PTQL queries. We then describe the two query generation components in our framework in Section 4, which enable the generation of extraction queries from both labeled and unlabeled data. The query performance of our approach and the quality of the extracted results are presented in Section 5. We describe the related work and conclude in Sections 6 and 7.

2 Background

2.1 Information Extraction

IE has been an active research area that seeks techniques to uncover information from a large collection of text. Examples of common IE tasks include the identification of entities (such as protein names), extraction of relationships between entities (such as interactions between a pair of proteins) and extraction of entity attributes (such as coreference resolution that identifies variants of mentions corresponding to the same entity) from text. Readers are referred to [7] for a detailed survey of IE.

The examples and experiments used in our paper involve the use of grammatical structures for relationship extraction. Cooccurrences of entities is a typical method in relationship extraction, but often leads to imprecise results. Consider that our goal is to extract relations between drug and proteins from the following sentence:

> Quetiapine is metabolized by CYP3A4 and sertindole by CYP2D6. (PMID:10422890)

By utilizing our grammatical knowledge, a human reader can observe that ⟨CYP3A4, metabolise, quetiapine⟩ and ⟨CYP2D6, metabolise, sertindole⟩ are the only correct triplet relations for the above sentence. However, if we consider cooccurrences of entities as a criteria to extract relationships, incorrect relationships such as ⟨CYP3A4, metabolise, sertindole⟩ and ⟨CYP2D6, metabolise, quetiapine⟩ would also be extracted from the above sentence. This simple

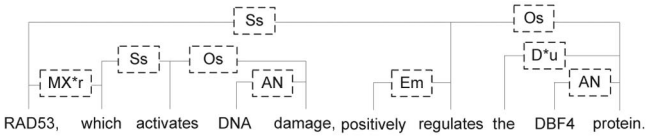


Fig. 1. Linkage of the sentence "RAD53, which activates DNA damage, positively regulates the DBF4 protein."

example highlights the need of grammatical knowledge in relationship extraction.

A typical IE setting involves a pipeline of text processing modules in order to perform relationship extraction. These include

- *Sentence splitting*: identifies sentences from a paragraph of text,
- *Tokenization*: identifies word tokens from sentences,
- *Named entity recognition*: identifies mentions of entity types of interest,
- *Syntactic parsing*: identifies grammatical structures of sentences,
- *Pattern matching*: obtains relationships based on a set of extraction patterns that utilize lexical, syntactic, and semantic features.

Extraction patterns are typically obtained through manually written patterns compiled by experts or automatically generated patterns based on training data. Different kinds of parsers, which include shallow and deep parsers, can be utilized in the pipeline. In our work, the Link Grammar parser [8] is utilized as part of our extraction approach. We describe the basic terminologies involved in Link Grammar in the next section.

2.2 Link Grammar

The Link Grammar parser is a dependency parser based on the Link Grammar theory [8]. Link Grammar consists of a set of words and linking requirements between words. A *sentence* of the language is defined as a sequence of words such that the links connecting the words satisfy the following properties: 1) the links do not cross, 2) the words form a connected graph, and 3) the links satisfy the linking requirements of each word in the sentence. The output of the parser, called a *linkage*, shows the dependencies between pairs of words in the sentence. Fig. 1 shows an example for the sentence "*RAD53, which activates DNA damage, positively regulates the DBF4 protein*" (PMID:10049915). The linkage contains several links, which include link `S` connecting the subject-noun `RAD53` to the transitive verb `regulates`, the `O` link connecting the transitive verb `regulates` to the direct object `DBF4` and the `MX*r` link connecting the relative pronoun `which` to the noun `RAD53`. For a complete description of links, we refer the reader to [3].

Besides producing linkages, the Link Grammar parser is also capable of outputting constituent trees. A *constituent tree* is a syntactic tree of a sentence with the nodes represented by part-of-speech tags and words of the sentences in the leaf nodes. For instance, the corresponding constituent tree for the above sentence is illustrated in Fig. 2. In the constituent tree, `S` stands for a sentence/clause, `SBAR` for a clause introduced by a subordinating conjunction, `WHNP` for a clause containing a relative pronoun, `NP` for a noun phrase, `VP` for a verb phrase, and `ADVP` for an adverb phrase. The leaf nodes of the constituent tree represent the words of the sentence and their part-of-speech tags. For words that are not recognizable by the parser, the tag `U` is given for such words for unknown part-of-speeches.

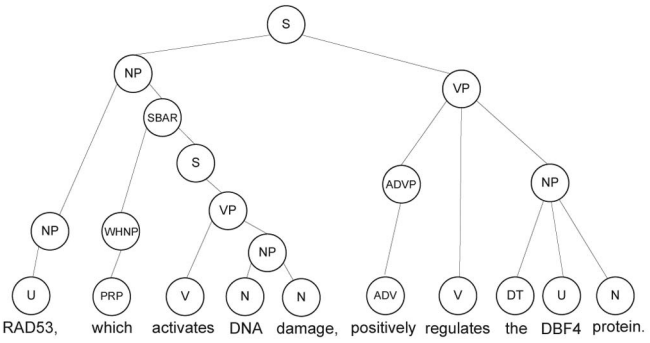


Fig. 2. Constituent tree of the sentence "RAD53, which activates DNA damage, positively regulates the DBF4 protein."

3 System

We first give an overview of our approach, and discuss each of the major components of our system in this section. Our approach is composed of two phases: *initial phase* for processing of text and *extraction phase* for using database queries to perform extraction. The *Text Processor* in the initial phase is responsible for corpus processing and storage of the processed information in the *Parse Tree Database* (PTDB). The extraction patterns over parse trees can be expressed in our proposed *parse tree query language*. In the extraction phase, the *PTQL query evaluator* takes a PTQL query and transforms it into keyword-based queries and SQL queries, which are evaluated by the underlying RDBMS and information retrieval (IR) engine. To speed up query evaluation, the *index builder* creates an inverted index for the indexing of sentences according to words and the corresponding entity types. Fig. 3 illustrates the system architecture of our approach.

Our approach provides two modes of generating PTQL queries for the purpose of information extraction: *training set driven query generation* and *pseudo-relevance feedback driven query generation*. To generate a set of patterns for information extraction using the training set driven approach, the *pattern generator* first automatically annotates an unlabeled

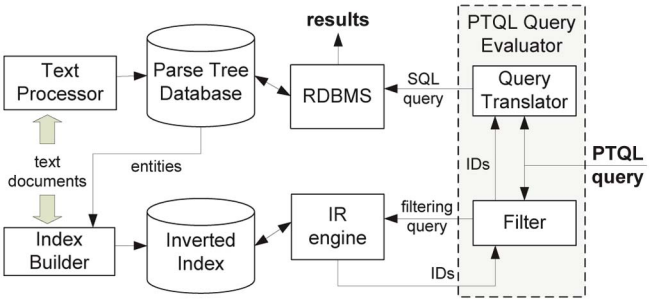


Fig. 3. System architecture of the PTQL framework. The framework includes the parse tree database for storing intermediate processed information and the query evaluator for the evaluation of PTQL queries through filtering and translation to SQL queries.

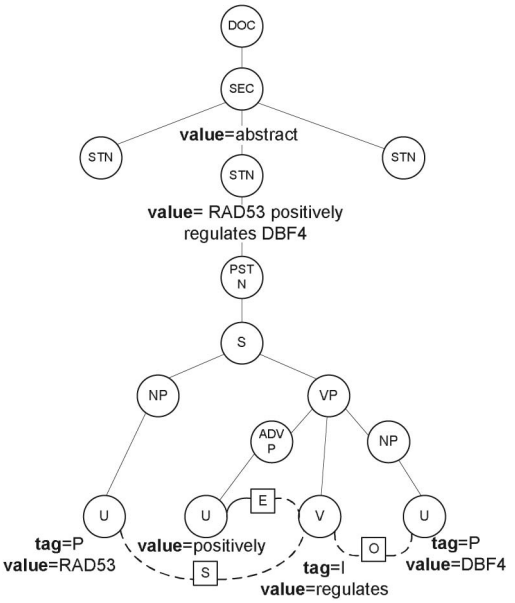


Fig. 4. An example of a parse tree for a document, which includes sections of the document, sentences, and the corresponding parse trees. The attribute *Tag* indicates the semantic type of a word, in which `P` stands for protein names and `I` for interaction words.

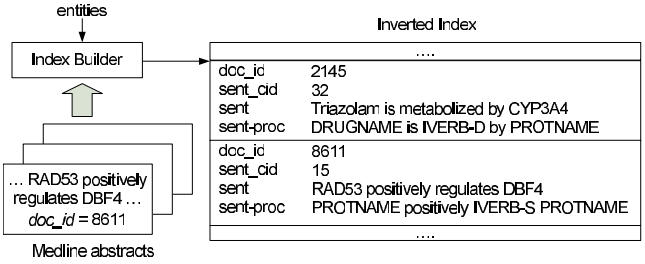


Fig. 5. An extended inverted index to handle queries that involve concepts rather than just instances.

document collection with information drawn from a problem-specific database. This step necessitates a method for precise recognition and normalization of protein mentions. From this labeled data, initial phrases referring to interactions are extracted. These phrases are then refined to compute consensus patterns and the resulting PTQL queries are generated by the *query generator*. However, training data are not always readily available for certain relationships due to the inherent cost of creating a training corpus. In that regards, our approach provides the pseudo-relevance feedback driven approach that takes keyword-based queries, and the *PTQL query generator* then finds common grammatical patterns among the top-k retrieved sentences to generate PTQL queries.

We first describe the parse tree database and the syntax of PTQL before we provide details of how PTQL queries are processed.

3.1 Parse Tree Database and Inverted Index

The *Text Processor* parses Medline abstracts with the Link Grammar parser [3], and identifies entities in the sentences using BANNER [9] to recognize gene/protein names and MetaMap [10] to recognize other entity types that include disease and drug names. Each document is represented as a hierarchical representation called the *parse tree of a document*, and the parse trees of all documents in the document collection constitute the *parse tree database*. The detailed schema and its description can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2010.214. A *parse tree* is composed of a constituent tree and a linkage. A *constituent tree* is a syntactic tree of a sentence with the nodes represented by part-of-speech tags and leafs corresponding to words in the sentence. A *linkage*, on the other hand, represents the syntactic dependencies (or links) between pairs of words in a sentence. Each node in the parse tree has labels and attributes capturing the document structure (such as title, sections, sentences), part-of-speech tags, and entity types of corresponding words.

Fig. 4 shows an example of a parse tree for a Medline abstract. The parse tree contains the root node labeled as `DOC` and each node represents an element in the document which can be a section (`SEC`), a sentence (`STN`), or a parse tree for a sentence (`PSTN`). A node labeled as `STN` may have more than one child labeled with `PSTN` to allow the storage of multiple parse trees. The node below the `PSTN` node indicates the start of the *parse tree*, which includes the constituent tree and linkage of the sentence. A solid line represents a parent-child relationship between two nodes in the constituent tree, whereas a dotted line represents a link between two words of the sentence. In the constituent tree, nodes `S`, `NP`, `VP`, and `ADVP` stand for a sentence, a noun phrase, a verb phrase, and an adverb phrase, respectively. The linkage contains three different links: the `S` link connects the subject-noun `RAD53` to the transitive verb `regulates`, the `O` link connects the transitive verb `regulates` to the direct object `DBF4` and the `E` link connects the verb-modifying adverb `positively` to the verb `regulates`. The square box on a dotted line indicates the link type between two words. Each leaf node in a parse tree has `value` and `tag` attributes. The `value` attribute stores the text representation of a node, while the `tag` attribute indicates the entity type of a leaf node. For instance, a protein is marked with a tag `P`, a drug name with a tag `D`, and an interaction word is marked with `I`.

Another essential component of our system architecture is an *inverted index* maintained by an IR engine such as Lucene.[3] This inverted index enables the efficient processing of PTQL queries, which will be discussed in details in a later section. As illustrated in Fig. 5, the *index builder* relies on the text preprocessor to recognize entities and replace the entities with identifiers in the sentences. Each sentence in the documents are indexed on its own so that each keyword-based filtering query retrieves a sentence rather than the entire document. Assuming that the *concepts* of interest are entities such as protein and drug names, and interaction verbs in the form of present tense and past participle. We use the identifiers `DRUGNAME`, `PROTNAME`, `IVERB-S`, and `IVERB-D` to represent these concepts, respectively. The index builder includes the original sentences in the inverted index, as well as sentences with entities replaced with identifiers. For instance, the sentence "*RAD53 positively regulates DBF4*" is indexed as `PROTNAME positively IVERB-S PROTNAME` under the field name `sent-proc` in the inverted index. The approach of indexing sentences with replaced identifiers is similar to

3. http://lucene.apache.org.

```
<query> ::= <pattern> ':' (<link cond>)? ':' (<proximity cond>)? ':' (<return exp>)?
<pattern> ::= <vert axis> <node exp> ('(' <var> ')')? ('{' <pattern list> '}')?
<node exp> ::= <node> ('[' <pred exp> ']')? (<or op> <node exp>)*
<node> ::= <id> | '?'
<vert axis> ::= '/' | '//'
<pattern list> ::= <pattern> (<horz axis> <pattern>)*
<horz axis> ::= '->' | '=>' | '<=>'
<pred exp> ::= <pred term> (<and op> <pred exp>)*
<pred term> ::= <field> ( (<op> <value>) | ( <IN op> '(' <value> (',' <value>)* ')' ) )
<link exp> ::= <link term> ( <bool op> <link exp> )*
<link term> ::= ( <var> <link type> <var> ) | '(' <link exp> ')'
<link type> ::= '!' <link_main_type> ( <link_subtype> )?
<proximity exp> ::= <proximity term> ( <bool op> <proximity exp> )*
<proximity term> ::= (<number>? ('[' | '{') <var> (<var>)+
                     (']' | '}') <number>?) | '(' <proximity exp> ')'
<return exp> ::= (DISTINCT)? <return term> (',' <return term>)*
<return term> ::= <var> '.' <field>
<op> ::= '=' | '<>' | 'like' | 'is' | 'is not'
<and op> ::= 'AND'      <or op> ::= 'OR'      <in op> ::= 'IN'
```

Fig. 6. PTQL Grammar.

[11], [12], [13]. Unlike [11], our approach requires no modification to the structure of the inverted index in order to process variabilized queries. For efficient access of the hierarchical structure and horizontal relations among nodes, we adopt the labeling scheme used in LPath [14], [6] and present the scheme in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2010.214.

3.2 Parse Tree Query Language

A fundamental design criteria for the query language is the ability of expressing linguistic patterns based on constituent trees. Standard XML query languages such as XPath [15] and XQuery [16] seem to be the ideal candidates for querying parse trees. However, the inability of expressing immediate-following siblings and immediate-preceding siblings in these standard XML query languages, as shown in [17], leads to the development of LPath [14], [6] as a query language for linguistic queries on constituent trees. An additional design criteria for the query language is the ability to express linguistic patterns based on dependency grammar, such as Link Grammar [8]. Links and link types can be useful in linguistic patterns, such as the type `MXsr` connects a relative pronoun to its corresponding noun. However, languages such as XQuery and LPath can only express ancestor-descendant and sibling relations between nodes. One of the novel features of our proposed query language PTQL is the ability to express links and link types between pairs of nodes, so that PTQL can be used to express linguistic patterns based on constituent trees and links, as well as link types.

We propose a high level extraction query language called PTQL. PTQL is an extension of the linguistic query language LPath [14], [6] that allows queries to be performed not only on the constituent trees but also the syntactic links between words on linkages. A PTQL query is made up of four components:

1. tree patterns,
2. link conditions,
3. proximity conditions, and
4. return expression.

A *tree pattern* describes the hierarchical structure and the horizontal order between the nodes of the parse tree. A *link condition* describes the linking requirements between nodes, while a *proximity condition* is to find words that are within a specified number of words. A *return expression* defines what to return. The EBNF grammar for PTQL is shown in Fig. 6. Before going into details of the definition of PTQL queries and its usage, we start with the basic element of PTQL queries called node expressions.

Definition 1. *A* node expression *is an expression of the form* `X[<pred exp>] (x)` *where* `X` *is a node name, e.g., a sentence (*`STN`*), a parse sentence (*`PSTN`*), a noun phrase (*`NP`*), or* `?` *(a node of any name),* `<pred exp>` *(predicate expression) is a boolean formula of expressions of the form* `<attribute> <op> <value>`*, and* `x` *is a variable name.*

Intuitively, `X[<pred exp>](x)` represents a node of type `X` that satisfies the condition specified by `<pred exp>` and this node is denoted by the variable `x`. When the predicate expression is empty (resp. variable binding is empty) we can omit the square brackets (resp. parenthesis). As an example, `N[tag='P'](x)` is a node expression describing a node labeled with `N` (noun) and tagged with `P` (i.e., a protein name) and this node is denoted by the variable `x`. A wildcard `'?'` denotes a node of any name. For instance, `?(y)` represents a node of any name and this node is denoted by the variable `y`.

To describe a tree pattern, we use two types of axis. A vertical axis `/` (parent-child relationship) or `//` (ancestor-descendant relationship) describes the hierarchical order of nodes in the parse tree. A horizontal axis `->` (immediate following) or `=>` (following) describes the horizontal order of nodes.[4] Formally, a tree pattern is defined recursively as follows:

Definition 2. *If* `e` *is a node expression then* `/e` *and* `//e` *are tree patterns. If* `e` *is a node expression and* $\mathtt{q}_1, \ldots, \mathtt{q}_n$ *are tree patterns then* $/\mathtt{e}\{\mathtt{q}_1 < \mathtt{ha}_1 > \cdots < \mathtt{ha}_{n-1} > \mathtt{q}_n\}$ *and* $//\mathtt{e}\{\mathtt{q}_1 < \mathtt{ha}_1 > \cdots < \mathtt{ha}_{n-1} > \mathtt{q}_n\}$ *are tree patterns, where* $\mathtt{ha}_i$ *can be* $->$, $=>$, *or* $<=>$.

4. A node `X` is said to (immediately) follow a node `Y` in a parse tree if the rightmost leaf of `X` (immediately) follows the leftmost leaf of `Y`.

TABLE 1
Examples of PTQL Queries

Relationship extraction (Q1):	Extract protein-protein interactions with the pattern <subject>-<verb>-<object>, where the subject and object correspond to protein names (tag='P') and the verb corresponds to "regulates"
PTQL:	//S{//?[tag='P'](i1)=>//V[value='regulates'](v)=> //?[tag='P'](i2)}: i1 !S v and v !O i2 :: i1.value, v.value, i2.value
Entity recognition (Q2):	Find protein mentions that precede the word "protein" within a noun phrase so that the return values of g are treated as protein names
PTQL:	//NP{/?(g)->/N[value='protein']} ::: g.value
Coreference resolution (Q3):	Find words that are referenced by the pronoun "which" so that the corresponding words are the return values of w
PTQL:	//S{//?(w)=>//PRP[value='which'](p)} : w !MX p :: w.value

The constituent tree and linkage given in Figs. 1 and 2 match any of these queries.

A parse tree matches a pattern /e (resp. //e) if one of the children (resp. descendants) of the root node satisfies the node expression e. A parse tree matches a pattern /e $\{q_1$<ha>$\ldots$<ha>$q_n\}$ (resp. //e $\{q_1$<ha$_1$>$\ldots$<ha$_{n-1}$>$q_n\}$) if there is a node X with children $Y_1, Y_2, \ldots, Y_n$ of X such that

1. X is a child (resp. descendant) of the root node of the parse tree,
2. X satisfies the node expression e,
3. each tree T_i with X being the root node and the subtree of the parse tree rooted at Y_i being the only subtree of T_i matches the pattern q_i, and
4. if $ha_i =$ '->' (resp. $ha_i =$ '=>') then Y_{i+1} immediately follows (resp. follows) Y_i. Y_{i+1} follows or precedes Y_i if $ha_i =$ '<=>'.

For instance, the parse tree in Fig. 4 matches the pattern //S{/NP−>/VP} as 1) the node labeled with S is a descendant of the root node of the parse tree, 2) the first noun phrase and the verb phrase are two children of that node, and 3) the verb phrase immediately follows the noun phrase. This tree also matches the pattern //S{//?[tag='P']=>//VP{/V->//?[tag='P']}}. However, it does not match the pattern //S{//NP->//V} as the verb does not immediately follow either the first noun phrase (there is an adverb in between) or the second noun phrase. A link condition is defined as follows.

Definition 3. *A* link term *is an expression of the form* x !<link> y, *where* x *and* y *are variable names and* <link> *is a link name in the linkage. A* link condition *is a boolean expression of link terms.*

For instance, x !S y is a link term representing the fact that the node denoted by x connects to the node denoted by y through an S link. Similarly, y !O z is a link term representing the fact that the node denoted by y connects to the node denoted by z through an O link.

Definition 4. *A* proximity term *is an expression of the form* $m\{x_1 \ldots x_k\}n$ *or* $m[x_1 \ldots x_k]n$, *where* $x_1, \ldots, x_k$ *are variable names and* m, n *are integers. A* proximity condition *is a boolean expression of proximity terms.*

We use an example to illustrate the definition of proximity terms. 1{x y}2 is a proximity term representing the fact that the nodes denoted by x, y are at least 1 node but not more than two nodes apart with respect to the sentence that contains words represented by x and y. In the case of 1[x y]2, an additional constraint is imposed such that words represented by x have to appear before words represented by y in the sentence.

Definition 5. *A* return expression *is a list of elements of the form* <var>. <attr> *separated by* ',', *possibly preceded by the keyword* DISTINCT, *where* <var> *is a variable name and* <attr> *is an attribute name.*

As an example, DISTINCT x.value, y.value, z.value is a return expression that returns the distinct value attributes of nodes denoted by variables x, y, z in the tree pattern. We now define the syntax of PTQL queries.

Definition 6. *A PTQL* query *is an expression of the form* <pattern> : <link cond> : <proximity cond> : <return exp> *where* <pattern> *is a tree pattern,* <link cond> *is a link condition,* <proximity cond> *is a proximity condition and* <return exp> *is a return expression.*

A parse tree matches a PTQL query if it matches the tree pattern of the query and the links between nodes satisfies the link condition of the query. The *return expression* of the query defines what information we want to return.

We illustrate the PTQL queries with examples, as shown in Table 1. For query Q1, the tree pattern //?[tag = 'P'](i1)=>//V[value='regulates'](v)=>//?[tag='P'](i2) represents that a protein name (denoted as i1) is followed by the verb "regulates" (denoted as v), which is followed by another protein name (denoted as i2), while the link condition i1 !S v and v !O i2 specifies that i1 has to be the subject of the sentence and v corresponds to the verb "regulates," and v and i2 has verb-object relation. This query returns the triplet ⟨RAD53, regulates, DBF4⟩. Query Q2 demonstrates how tree patterns can be used to identify protein mentions based on lexical clues using the word "protein" within a noun phrase. This query returns ⟨DBF4⟩ as a protein mention. Query Q3 utilizes the link MX to resolve coreferences, as the link MX connects a relative pronoun to its corresponding word. In this case, the relative pronoun "which" is connected to the word "RAD53" so that ⟨RAD53⟩ is returned.

3.3 Query Evaluation

Our approach for the evaluation of PTQL queries involves the use of IR engine as well as RDBMS. The role of the IR engine in query is to select sentences based on the lexical features defined in PTQL queries, and only the subset of sentences retrieved by the IR engine are considered for the evaluation of the conditions specified in the PTQL queries

by RDBMS. Unlike the filtering mechanism described in [18] that selects potentially relevant documents for extraction, our approach does not discard sentences that should otherwise be included for extraction. Using sample query Q1 as an example, the lexical features defined in the query imply that only sentences with at least one gene name together with the keyword "regulates" should be considered for extraction. We summarize the process of the evaluation of PTQL queries as follows.

1. Translate the PTQL query into a filtering query.
2. Use the filtering query to retrieve relevant documents D and the corresponding sentences S from the inverted index.
3. Translate the PTQL query into an SQL query and instantiate the query with document id $d \in D$ and sentence id $s \in S$.
4. Query PTDB using the SQL query generated in Step 3.
5. Return the results of the SQL query as the results of the PTQL query.

In step 2, the process of finding relevant sentences with respect to the given PTQL query requires the translation of the PTQL query into the corresponding filtering query. Here, we define the syntax of the *keyword-based filtering queries*, which adopts the syntax of Lucene.

A *query term* t for a filtering query is a string that can be preceded by the required operator +, as well as the term `<field>:`, where `<field>` is the name of a field. A *phrase* p is in the form "$t_1 \ldots t_n$," where $t_1, \ldots, t_n$ are query terms. p can be followed by a proximity operator in the form of `p~<number>`. A parenthesis expression is composed of query terms and phrases, enclosed by parentheses, and it can be preceded by the required operator. A *keyword-based filtering query* is a list of query terms, phrases, and parenthesis expressions. An PTQL query q is translated into a keyword-based filtering query using the following steps:

1. Generate query terms for each of the node expressions that are in the tree pattern of q.
2. Form phrases if consecutive node expressions are connected by "immediate following" horizontal axes (i.e., "->").
3. Form phrases followed by the proximity operator if the corresponding nodes are defined in the proximity condition of q.

The translation of a PTQL query q into a keyword-based filtering query involves the traversal of the parse tree of the PTQL query in preorder walk fashion. For each predicate term in the form of `<field>` = `<val>` (resp. `IN (val`$_1, \ldots,$ `val`$_k$`)`, we create the query term $+\langle field \rangle : \langle val \rangle$ (resp. $+\langle field \rangle$: $(\langle val_1 \rangle \ldots \langle val_k \rangle)$).[5] For example, the predicate term of a node expression `//N[tag='P' and value IN ('RAD53','DBF4')]` is `+sent: ("RAD53" "DBF4")+rep-sent:PROTNAME`. If the tree pattern of q contains `e`$_i$ `-> e`$_j$, where e_i and e_j are node expressions with predicate term in the form of `<field>` = `<val>`, then the phrase "$\langle val_i \rangle \langle val_j \rangle$" is formed for the query terms that represent e_i and e_j. If a proximity term in the proximity condition of q is in the form `[x`$_i$ `x`$_j$`]n`, then the phrase "$\langle val_i \rangle \langle val_j \rangle$"~$n$ is formed.

5. $+\langle field \rangle$: $(\langle val_1 \rangle \ldots \langle val_k \rangle)$ is the short form for $+ (\langle field \rangle : \langle val_1 \rangle \cdots \langle field \rangle : \langle val_k \rangle)$.

We use query Q1 in Table 1 to illustrate the translation of PTQL queries into keyword-based filtering. The following is the keyword-based filtering query for Q1:

```
+sent:regulates +rep_sent:PROTNAME
```

The translation of PTQL queries into SQL queries in step 3 adopts the approach used in LPath [6], [14] to translate the hierarchical representation and horizontal relations of PTQL queries into nested SQL queries. We further extend the translation of PTQL link conditions into SQL. The details of the PTQL-to-SQL translation is presented in Appendices C and D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2010.214. The translated SQL query is then instantiated with the sentences retrieved in step 2 by specifying the document (`Doc_ID`) and sentence IDs (`Sent_CID`) in the SQL queries. The corresponding SQL translation for query Q1 is as follows, in which table alias `C` and `T` refers to the table `Constituents` and `Linkages` that are used for storing the constituent trees and linkages of sentences, while table alias `B` refers to the table `Bioentities` that is used for storing the semantic information for each node in the parse trees:

```
SELECT v2,v3,v4
FROM (SELECT id id4,..,v v4,id1,..,v1
  FROM C,(SELECT id id3,..,v v3,id1,..,v1
    FROM C,(SELECT id id2,..,v v2,id1,..,v1
      FROM C,(SELECT pid pid1,..,v v1
        FROM C WHERE d>2 AND C.n='S') C2
      WHERE C.l>=C2.l1 AND C.r<=C2.r1
        AND C.d>C2.d1 AND (B.t='P' AND
        B.id=C.id)) C3
    WHERE C.l>=C3.l1 AND C.r<=C3.r1
     AND C.d>C3.d1 AND C.n='V'
     AND C.v='regulates'
     AND C.l>=C3.r2) C4
  WHERE C.l>=C4.l1 AND C.r<=C4.r1 AND
    C.d>C4.d1 AND (B.t='P' AND
      B.id=C.id) AND C.l>=C4.r3) T
WHERE ((T.id2,T.id3) IN (SELECT f_id,t_id
  FROM L WHERE TYPE='S')) AND ((T.id3,T.id4)
  IN (SELECT f_id,t_id FROM L WHERE TYPE='O'))
```

4 Query Generation

An important aspect of an IE system is its ability to extract high-quality results. In this section, we demonstrate the feasibility of our approach by first describing two approaches in generating PTQL queries automatically: *training set driven query generation* and *pseudo-relevance feedback driven query generation*. The first query generation approach takes advantage of manually annotated data to generate PTQL queries for the extraction of protein-protein interactions. As training data are not always available, we introduce the latter approach that identifies frequent linguistic structures to generate PTQL queries without the use of training data.

4.1 Training Set Driven Query Generation

We illustrate our approach with an application of protein-protein interaction extraction using a set of syntactic patterns that are expressed in PTQL queries. To generate a set of patterns for information extraction, the *annotator* component

1	P	interacts	with	the	P
2	P	binds	to		P
3	P	bound	to		P
	P	{i-verb}	{preposition}	{determiner}?	P

{determiner} := a, an, the, these, this, those, ...;
{preposition} := between, for, to, with, ...;
{i-verb} := binds, bound, interacts, interacted, ...;

Fig. 7. Multiple initial patterns (1-3) lead to the same general pattern, after words have been replaced with word categories and protein names with "P."

is applied to automatically annotate an unlabeled document collection with information drawn from a problem-specific database. This step necessitates a method for precise recognition and normalization of protein mentions. From this labeled data, the *pattern generator* identifies relevant phrases referring to interactions in order to generate patterns. These initial patterns are then used to compute consensus patterns through the *pattern generalization* component for protein-protein interactions (PPIs). PTQL queries are then formed by the *query generator* to perform extraction from the parse tree database.

As many sentences might contain coincidental mentions of proteins and not describe an interaction ("we study the proteins A, B, and C"), we reduce these initial candidate evidence by a number of refinement steps. As a first step, we manually compile a set of keywords that typically refer to protein-protein interactions ("binds," "association," "-mediated"). This results in a set of 123 verbs, 126 nouns, and eight adjectives, plus corresponding word forms. Such words have to appear between the two proteins under consideration, or precede/follow them in a short distance, which is parameterizable. We then reduce the full sentence to the snippet that likely conveys the information about an interaction; therefore, we may extract the shortest snippet that contains both proteins and an interaction-indicating word, or include additional words from the left and right of this snippet. Each snippet found is considered as a *relevant phrase*; it could be directly used to find similar (parts of) sentences in the test data set. The more the snippet extends to the left and right, the more precise the phrase will be. Shorter snippets, on the other hand, will typically increase the recall when using that phrase.

To increase the recall of the initial patterns, we generalize the patterns by substituting tokens belonging to certain word categories with a higher-level concept, such as gene/protein names, interaction verbs. It shows that many natural language sentences that describe certain events, like protein-protein interactions, exhibit a certain "behavior" that makes them similar to other such sentences. Consider Fig. 7, where the words "binds," "interacts," etc., can be generalized into `i-verb`, indicating that the words belong to the higher-level concept *interaction verbs*.

The generalized patterns are then translated into PTQL queries for extraction. For instance, the syntactic pattern `<P> {i-verb} {preposition} {determiner}? <P>`, where `<P>` corresponds to the matching of any protein names, is translated into

```
//S{//[tag='P'](i1)->//V[tag='I']->//
  IN[value
   in {'with','for',...}(w1)=>//
```

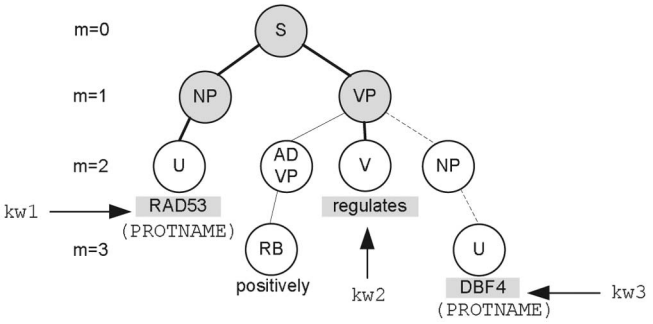


Fig. 8. An illustration of the mth level string encoding `//S/NP/kw1://S/VP/kw2://S/VP//kw3`, where $m = 1$, for the constituent tree of "RAD53 positively regulates DBF4" with respect to the query `PROTNAME and regulates and PROTNAME`. The shaded nodes in the constituent tree are used for the string encoding, and the node with label `S` is the lowest common ancestor node for the three query terms.

```
[tag='P'](i2)}
 :: 1[w1 i2]2 : i1.value, i2.value
```

The category `{preposition}` is represented by its set of predefined instances of prepositions in the PTQL query. The term `{determiner}?` indicates that a determiner is an optional match, and it is represented by the proximity condition `1[w1 i2]2` so that at most one word can be in between the words represented by the variables `w1` and `i2`. Notice that no link condition is used in the translation from syntatic patterns to PTQL queries.

4.2 Pseudo-Relevance Feedback Driven Query Generation

To make up for the lack of training data for the relationship of interest, we offer an alternative approach that is inspired by pseudo-relevance feedback in information retrieval. The idea is to automatically generate PTQL queries by considering the constituent trees of the top-k sentences retrieved with a boolean keyword-based query. The common grammatical patterns among the constituent trees of relevant sentences are utilized to form extraction patterns. Interaction extraction is then performed by using the PTQL queries translated from the generated extraction patterns.

A boolean keyword-based query q is composed of query terms $t_1 \ldots t_n$, where a query term t_i can be a keyword, or an identifier for an entity type, such as `PROTNAME` that represents any matches of protein names. With q, a ranked list of sentences S is retrieved and the constituent trees of the top-k sentences of S (denoted as S_k) are retrieved from PTDB. To find common grammatical patterns among the constituent trees of S_k, string encodings are generated for each of the sentence in S_k. A *0th level string encoding* records the labels of the lowest common ancestor lca of the query terms and the query terms themselves in a preorder tree traverse order. A mth level string encoding is defined as the string encoding that includes at most m descendants of lca on each of the paths connecting lca and a query term t_i. For instance, suppose $q =$ `PROTNAME and regulates and PROTNAME`, then the string

```
//S/NP/kw1://S/VP/kw2://S/VP//kw3
```

is a first level string encoding for the constituent tree in Fig. 2, in which `kw1`, `kw3` represent `PROTNAME` and `kw2` represents the keyword "regulates," and `S` is the lowest common ancestor node for the three terms. This is illustrated in Fig. 8.

TABLE 2
Examples of PTQL Queries Used in the Experiments

Query Q8:	`//S{/?[Tag='DRUG' AND Value='DRUGNAME'](kw2)=>/NP{/?[Value='metabolism'](kw1)}=> /VP{/VP{//?[Tag='GENE'](kw0)}}} ::: distinct kw0.value, kw1.value, kw2.value`
Query Q14:	`//NP{/NP{/PP{//?[Tag='GENE'](kw0)}}=>/PP{/NP{/?[Tag='DRUG' AND Value='DRUGNAME'](kw2) =>/?[Value='metabolism'](kw1)}}} ::: distinct kw0.value, kw1.value, kw2.value`

The word DRUGNAME corresponds to a drug name.

A mth level string encoding has a one-to-one translation to a PTQL query. Using the above string encoding as an example, the corresponding PTQL query is

```
//S{/NP{/?[tag='P']}=>/VP{/?[value=
    'regulates']=>//?[tag='P']}}}
```

With $m = 0$, the linguistic patterns in the relevant sentences are maximally generalized, potentially leading to a high recall with possible comprise to precision. With the increasing value of m, the patterns become more specific, potentially increase precision with possible compromise to recall. By forming the mth level string encodings, we can identify the similarity of the retrieved sentences based on their grammatical structures. We define that two sentences are *grammatically similar* if they have the same mth level string encoding. Grammatically similar sentences are grouped together to form a cluster. A PTQL query is then generated for each of the clusters of string encodings. The steps of generating PTQL queries can be outlined as follows.

Let C_m be a set of clusters with mth level string encodings. Given a boolean keyword-based query q and paramemter k,

1. Retrieve sentences using q from the inverted index and retrieve the constituent trees of the top-k sentences S_k from PTDB.
2. For each sentence in S_k extract the subtree that is rooted at the lca of all the query terms $t_1, \ldots, t_n$ with the query terms as leaf nodes from the constituent tree.
3. Generate mth level string encodings for each of the subtrees.
4. Sentences that are grammatically similar based on their mth level string encodings are grouped together to form clusters of common grammatical patterns C_m.
5. A PTQL query is generated for each common grammatical pattern C_m.

Interactions are extracted through the evaluation of the generated PTQL queries.

5 Experimental Evaluation

We first illustrate the performance of our approach in terms of query evaluation and the time savings achieved through incremental extraction. Then we evaluate the extraction performance for our two approaches in query generation.

5.1 Time Performance for PTQL

We performed experiments in finding the time performance of the evaluation of PTQL queries, as well as experiments to illustrate the amount of time saved in the event of change of an extraction goal and deployment of an improved module. All experiments were performed using a 2.2-GHz Intel Xeon QuadCore CPU running in Red Hat Linux. Only a single process was used to perform the experiments. The parse tree database is stored as a relational database managed by MySQL.

5.1.1 Query Evaluation

A set of 25 PTQL queries that involves in the extraction of drug-gene metabolic relations was applied to a large corpus of 17 million abstracts to measure the time performance for PTQL evaluation. Specifically, given a drug, the goal is to find which genes are involved in the metabolic relations with the drug. In our experiments, we specified a single drug ("1-drug"), a set of 5 drugs ("5-drugs") and a set of 10 drugs ("10-drugs") in each of the 25 PTQL queries. The keyword "metabolized" is involved in queries Q1-Q7, while the keywords "metabolism" and "substrate" are involved in queries Q8-Q20 and Q21-Q25. Table 2 shows some of the PTQL queries that were used in our experiments. Fig. 9 shows the average duration over different sets of drugs and runs for the PTQL evaluation. This figure shows that the query evaluation can be completed in the range of a second to 50 seconds for all answers to be returned. In the experiments, each query was evaluated with five different sets of drugs and repeated for five runs. The time performance indicates that our proposed framework is acceptable for real-time IE. We observed that queries specified with a larger set of drugs require a longer time to complete the evaluation. In addition, the evaluation on the set of queries with the keyword "metabolism" (Q8-Q20) generally takes longer than the other sets of queries to complete. This is due to the fact that a higher number of sentences matches with the keyword "metabolism." The complexity of the PTQL queries also plays a role in the amount of time it takes for evaluation. Comparing query Q8

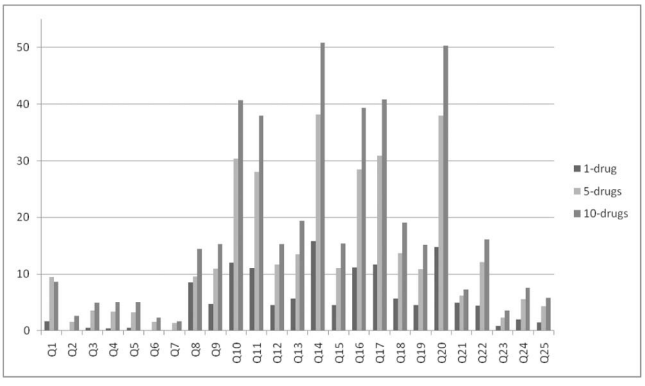


Fig. 9. Time performance in seconds for PTQL evaluation for a set of 25 queries that involve the extraction of drug-metabolic-gene relations.

with query Q14 as shown in Table 2, the SQL query translated from query Q14 requires more self-joins that its counterpart for query Q8.

5.1.2 Change of Modules

We performed experiments to show the time savings for our incremental extraction approach. The scenario behind our experiment is that the initial goal is to perform extraction for drug information from a text collection. The extraction goal is then changed into the extraction of drug-protein relations that requires the deployment of a gene named entity recognizing to identify gene mentions in the text collection.

To illustrate the amount of time savings, a collection of 13 K Medline abstracts was initially processed with the Link Grammar parser and a dictionary-based tagger for drug names. This process took about 62.38 hours. We then deployed a statistical-based tagger for gene names to process the corpus. With the pipeline approach, the whole process had to be started from scratch by running the Link Grammar parser, the drug name tagger and the newly deployed gene name tagger. This took another 64.8 hours to complete. With our approach, the intermediate processing data produced by the Link Grammar parser and the drug name tagger were populated into the parse tree database. The gene name tagger was then deployed to process the corpus and SQL insert statements were issued to update the parse tree database. This process took only 6.71 hours to complete. This experiment showed a tremendous decrease of 89.64 percent when a new module is deployed for text processing as compared to the pipeline approach. Such significant reduction of time is largely due to the fact that the Link Grammar parser is not utilized in the reprocessing, which can take an average of 7.6 seconds to parse a sentence when the timeout limit was set as 15 seconds for the parser. The reduction of time comes at the expense of disk space. The processed data for the 13 K Medline abstracts results in a disk usage of about 110 MB.

5.2 Extraction Performance for PTQL

We used two data sets to evaluate the performance of PTQL. For the evaluation of our training set driven query generation, we used the BioCreative 2 IPS test data [19] as a benchmark for relationship, in this case the extraction of protein-protein interactions. This data set consists of 358 full-text articles, which we transformed into 98,209 sentences. This set was reduced by us to include only sentences that contain at least one protein, resulting in 71,631 sentences. Another data set of 13,015 Medline abstracts from [20] that focus on drug-protein relations is used to evaluate the pseudo-relevance feedback driven query generation.

5.2.1 Training Set Driven Query Generation

For the training set driven query generation method, we used the Biocreative 2 IPS benchmark [19] for evaluation. The task is to find protein-protein interactions for which a text provides evidence for a physical interaction between the proteins. In addition, all proteins have to be mapped to corresponding identifiers in the UniProt database. We describe our approach for this additional task in [21]. To generate extraction patterns, the training data from the BioCreative 2 IPS corpus (740 full-text articles) was used.

TABLE 3
Performance of Various Approaches on the BioCreative 2 IPS Test Data

System	P	R	F
PTQL	83.6	58.6	64.2
* 74 manually created patterns [25], with NER from [21]	59.7	37.9	41.8
* Reported by [22]	39.1	29.7	28.5
Reported by [23]	37.0	32.7	30.4
Reported by [24]	25.2	23.3	24.2

*Mean precision, recall, and f-measure in percent. * are basically the same systems, except that their NER/EMN is replaced with our own to get comparable results. Only the results from the first two systems can be compared directly, as they assume "perfect" EMN.*

The generated patterns were then translated into PTQL queries, which were utilized to perform extraction from the testing data (358 full-text articles). The gold standard for both training and testing data were complied by a team of expert curators in the Biocreative challenge. The training set driven query generation method generated 11,208 extraction patterns. Our approach of using PTQL queries to express the generated patterns achieves a mean precision and recall of 83.6 and 58.6 percent with 64.2 percent as the f-measure. The extraction results are summarized in Table 3. Our results also show far better performance than the previously top-performing PPI systems [22], [23], [24]. To give a fair comparison, we performed another experiment that utilized the 74 manually curated patterns reported in [25] on their OpenDMAP system, but using the same gene normalization (EMN) and named entity recognition (NER) from [21]. We observed that our approach 111 still achieves significantly better results. The significant improvement over previous methods is largely contributed by the large number of generated extraction patterns that were generated by our training set driven query generation approach. Each of the generated patterns has a high precision but low recall, but the combined results of the 11,208 extraction queries contributes to a high overall recall.

5.2.2 Pseudo-Relevance Feedback Query Generation

We further evaluate the extraction performance for PTQL using the pseudo-relevance feedback driven query generation method. Our goal here is to extract drug-protein metabolic relations without the use of training data to generate queries. Specifically we created boolean keyword-based queries in the form of `DRUGNAME and <metabolic-word> and PROTNAME` to generate PTQL queries, where the identifiers `DRUGNAME` and `PROTNAME` correspond to matching of any mentions of drug and protein names, and `<metabolic-word>` is a class of words to indicate drug-protein metabolic relations that include the words "*metabolized*," "*metabolize*," "*metabolizes*," "*metabolised*," "*metabolise*," "*metabolises*," and "*metabolism.*"

The performance of the query generation using our pseudo-relevance feedback is compared with the cooccurrences method, which considers a drug-protein metabolic relation when drug and protein names together appear with one of the metabolic words in a sentence. To perform the comparison, we created a gold standard by analyzing all of the interactions that were extracted using the cooccurrences method. This results in a collection of 1,059 drug-protein

TABLE 4
Performance Comparison between Cooccurrences and the Pseudo-Relevance Feedback Query Generation Method (QueryGen) for Drug-Protein Metabolic Relations

System	Precision	Recall	F-measure
Cooccurrences	39.9	*100.0*	57.0
QueryGen (m=1)	53.5	64.3	58.4
QueryGen (m=2)	68.0	42.8	52.5
QueryGen (m=3)	83.3	31.0	45.2
QueryGen (m=4)	85.0	26.0	39.8

m is the maximum number of descendants to include in the mth level string encodings, using the top-k% (where $k = 60\%$) of the retrieved

TABLE 5
Comparison of Cooccurrences and Pseudo-Relevance Feedback (PTQL) on the Extraction of Drug-Protein Metabolic Relations Based on Number of True Positives (TP), Precision (P), Recall (R), and f-Measure (F)

	Cooccurrences				PTQL (m=1)			
	TP	P	R	F	TP	P	R	F
metabolized	344	55.0	100.0	71.0	243	82.2	68.6	74.8
metabolises	9	45.0	100.0	62.1	–	–	–	–
metabolised	82	51.9	100.0	68.3	43	79.6	52.4	63.2
metabolize	38	40.9	100.0	58.0	13	52.0	34.2	41.3
metabolism	546	32.0	100.0	48.4	355	41.0	65.0	50.3
metabolizes	47	72.3	100.0	83.9	26	83.9	55.3	66.7

m is the maximum number of descendants to include in the mth level string encodings, using the top 60 percent of the retrieved sentences to form clusters with at least three members.

metabolic relations[6] out of the 13,015 abstracts. Table 4 shows the performance of the query generation method compared to the cooccurrences method. While the f-measure shows that the query generation method lags behind due to the high recall achieved by cooccurrences, it is important to notice that precision is significantly higher than its cooccurrences counterpart. The results also shows the expected trade-off between precision and recall when m varies. With increasing m, which is the maximum number of descendants of LCA to include in the mth level string encodings, precision gains at the expense of recall. This is intuitive as the string encodings become more specific when more descendants are included. Table 5 shows the extraction performance of drug-protein metabolic relations with different keywords using our pseudo-relevance feedback driven query generation method. Extraction with the keyword "*metabolises*" does not yield any results with our query generation method, due to the small number of sentences retrieved by the initial keyword-based queries. The value of k is experimentally determined, and using top-60 percent works well for most of the cases we tested on.

6 Related Work

Information extraction has been an active research area over the years. The main focus has been on improving the accuracy of the extraction systems, and IE has been seen as an one-time execution process. Such paradigm is inadequate for real-world applications when IE is seen as long running processes. An example of a real-world application of IE is the extraction from evolving text [4], [5], such as the frequent update of the content of web documents. Hence, there is a need to minimize reprocessing of the text corpora. In our case, we assume the text corpora to be static. While new documents can be added to our text collection, the content of the existing documents are assumed not to be changed, which is the case for Medline abstracts. Our focus is on managing the processed data so that in the event of the deployment of an improved component or a new extraction goal, the affected subset of the text corpus can be easily identified. A recent special issue of SIGMOD Record [26] also highlights the need to develop IE frameworks that manage extraction tasks as long running processes. In the remaining of the section, we describe how our proposed extraction framework differs from traditional IE systems, rule-based IE systems and IE systems based on machine learning.

6. Omitting possible cross-sentence relations from the gold standard.

6.1 Traditional IE Approaches

Popular IE frameworks such as UIMA [1] and GATE [2] provide the ability of efficient integration of various NLP components for IE. Such frameworks are file based and they do not store the intermediate processing output of various components. Typical extraction systems such as QXtract [18] and Snowball [27] utilize RDBMS to store and query the extracted facts. Recent work on IE management systems rely on RDBMS for optimization of the execution of IE tasks. For systems such as Cimple [28] and SystemT [29], operations such as joins in RDBMS are performed over extracted facts that are stored in various database tables. Our proposed framework also follows traditional IE approaches in terms of first preprocessing the corpus and then performing extraction. However, our framework also manages the intermediate processing output such as the parse trees and semantic information using RDBMS. In the event of a deployment of an improved component or a change of extraction goals, our approach only requires the new module to be applied to the text collection. The intermediate processing data are then inserted into the parse tree database so that both the new and existing processing data can be utilized for extraction.

To address the high computational cost associated with extraction, document filtering is a common approach in which only the promising documents are considered for extraction [18], [30], [31]. These promising documents are selected based on a classifier that is trained for determining documents that are relevant for extraction. Such an approach can potentially miss out documents that should have been used for extraction. In our filtering approach, sentences are selected solely based on the lexical clues that are provided in a PTQL query. This filtering process utilizes the efficiency of IR engines so that a complete scan of the parse tree database is not needed without sacrificing any sentences that should have been used for extraction.

6.2 Rule-Based IE Approaches

Rule-based IE approaches have been proposed in [32], [33], [34], and [35]. The AQL query language proposed in the Avatar system [35] is capable of performing extraction for a variety of IE tasks that includes matching with regular

expression. Unlike PTQL, the language does not support the use of parse trees, which can be useful in IE tasks such as relationship extraction. Systems such as DIAL [33], TLM [34], KnowItNow [36] focus on relationship extraction based on their own query languages. However, these languages only support querying of data from shallow parsing, and they do not have the ability of extracting from rich grammatical structures such as parse trees. On the other hand, declarative languages are used in systems such as Xlog [37], Cimple [28] and SystemT [29]. Operations such as joins in RDBMS are performed over extracted facts that are stored in various database tables. Rules are then applied to integrate different types of extracted facts. However, these rules are not capable of querying parse trees.

The use of parse trees to define patterns for extraction has been proposed in [38], and kernels for relationship extraction in [39]. However, none of these work focus on the issue of storing and reutilizing parse trees. Our work is closest to the IE management system MEDIE [32] stores parse trees in a database and allows extraction over parse trees with a query language. The XML-like query language proposed is based on another kind of dependency grammar called head-driven phrase structure (HPSG). Unlike PTQL, link types cannot be expressed with this query language. In addition, MEDIE only provides simple query generation component that translates subject-verb-object extraction queries into its own query languages. This limits the utility of the system, as there can be a learning curve to learn the query language before users can perform their own extraction.

6.3 Machine Learning Approaches for IE

Our proposed approach provides mechanisms to generate extraction queries from both labeled and unlabeled data. Query generation is critical so that casual users can specify their information needs without learning the query language. Learning extraction patterns from training data has been proposed previously [18], [27], [40], [41]. However, training data are not always readily available and annotation of training data are both labor-intensive and time-consuming. To compensate the need of training data for extraction, [42], [43] proposed a new paradigm of automated discovery of relations through self-supervised learning. The core idea is to identify dependency structures from sentences that are retrieved by a user's query. The identified grammatical structures are generalized to form extraction patterns. However, the process of finding patterns can be expensive as any subtrees in the dependency structures can be considered as patterns. To reduce the complexity, only structures with verbs as predicates and a restricted number of nodes are considered in the process of identifying patterns. The techniques in [42], [43] require certain predefined linguistic heuristics for the generation of extraction patterns. Our query generation method does not make any assumptions for linguistic heuristics.

7 Discussion and Future Work

In this section, we discuss the main contributions of our work as well as their limitations.

- *Extraction framework*. Existing extraction frameworks do not provide the capabilities of managing intermediate processed data such as parse trees and semantic information. This leads to the need of reprocessing of the entire text collection, which can be computationally expensive. On the other hand, by storing the intermediate processed data as in our novel framework, introducing new knowledge can be issued with simple SQL insert statements on top of the processed data. With the use of parse trees, our framework is most suitable for performing extraction on text corpus written in natural sentences such as the biomedical literature. As indicated in our experiments, our increment extraction approach saves much more time compared to performing extraction by first processing each sentence one-at-a-time with linguistic parsers and then other components. This comes at the cost of overheads such as the storage of the parse trees and the semantic information, which takes up 1.5 TB of space for 17 million abstracts for the parse tree database. In the case when the parser fails to generate parse tree for a sentence, our system generates a "replacement parse tree" that has the node `STN` as the root with the words in the sentence as the children of the root node. This allows PTQL queries to be applied to sentences that are incomplete or casually written, which can appear frequently in web documents. Features such as horizontal axis and proximity conditions can be most useful for performing extraction on replacement parse trees.
- *Parse tree query language*. One of the main contributions of our work is PTQL that enables information extraction over parse trees. While our current focus is per-sentence extraction, it is important to notice that the query language itself is capable of defining patterns across multiple sentences. By storing documents in the form of parse trees, in which the node `DOC` is represented as the root of the document and the sentences represented by the nodes `STN` as the descendants. As shown in the sample queries illustrated in Table 1, PTQL has the ability to perform a variety of information extraction tasks by taking advantage of parse trees unlike other query languages. Currently, PTQL lacks the support of common features such as regular expression as frequently used by entity extraction task. PTQL also does not provide the ability to compute statistics across multiple extraction such as taking redundancy into account for boosting the confidence of an extracted fact.

For future work, we will extend the support of other parsers by providing wrappers of other dependency parsers and scheme, such as Pro3Gres and the Stanford Dependency scheme, so that they can be stored in PTDB and queried using PTQL. We will expand the capabilities of PTQL, such as the support of regular expression and the utilization of redundancy to compute confidence of the extracted information.

REFERENCES

[1] D. Ferrucci and A. Lally, "UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment," *Natural Language Eng.,* vol. 10, nos. 3/4, pp. 327-348, 2004.

[2] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications," *Proc. 40th Ann. Meeting of the ACL,* 2002.

[3] D. Grinberg, J. Lafferty, and D. Sleator, "A Robust Parsing Algorithm for Link Grammars," Technical Report CMU-CS-TR-95-125, Carnegie Mellon Univ. 1995.

[4] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan, "Efficient Information Extraction over Evolving Text Data," *Proc IEEE 24th Int'l Conf. Data Eng. (ICDE '08),* pp. 943-952, 2008.

[5] F. Chen, B. Gao, A. Doan, J. Yang, and R. Ramakrishnan, "Optimizing Complex Extraction Programs over Evolving Text Data," *Proc 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09),* pp. 321-334, 2009.

[6] S. Bird et al., "Designing and Evaluating an XPath Dialect for Linguistic Queries," *Proc 22nd Int'l Conf. Data Eng. (ICDE '06),* 2006.

[7] S. Sarawagi, "Information Extraction," *Foundations and Trends in Databases,* vol. 1, no. 3, pp. 261-377, 2008.

[8] D.D. Sleator and D. Temperley, "Parsing English with a Link Grammar," *Proc Third Int'l Workshop Parsing Technologies,* 1993.

[9] R. Leaman and G. Gonzalez, "BANNER: An Executable Survey of Advances in Biomedical Named Entity Recognition," *Proc. Pacific Symp. Biocomputing,* pp. 652-663, 2008.

[10] A.R. Aronson, "Effective Mapping of Biomedical Text to the UMLS Metathesaurus: The MetaMap Program," *Proc. AMIA Symp.,* p. 17, 2001.

[11] M.J. Cafarella and O. Etzioni, "A Search Engine for Natural Language Applications," *Proc. 14th Int'l Conf. World Wide Web (WWW '05),* 2005.

[12] T. Cheng and K. Chang, "Entity Search Engine: Towards Agile Best-Effort Information Integration over the Web," *Proc. Conf. Innovative Data Systems Research (CIDR),* 2007.

[13] H. Bast and I. Weber, "The CompleteSearch Engine: Interactive, Efficient, and Towards IR& DB Integration," *Proc Conf. Innovative Data Systems Research (CIDR),* 2007.

[14] S. Bird, Y. Chen, S.B. Davidson, H. Lee, and Y. Zheng, "Extending XPath to Support Linguistic Queries," *Proc. Workshop Programming Language Technologies for XML (PLAN-X),* 2005.

[15] J. Clark and S. DeRose, "XML Path Language (XPath)," http://www.w3.org/TR/xpath, Nov. 1999.

[16] "XQuery 1.0: An XML Query Language," http://www.w3.org/XML/Query, June 2001.

[17] C. Lai, "A Formal Framework for Linguistic Tree Query," Master's thesis, Dept. of Computer Science and Software Eng., Univ. of Melbourne, 2005.

[18] E. Agichtein and L. Gravano, "Querying Text Databases for Efficient Information Extraction," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 113-124, 2003.

[19] M. Krallinger, F. Leitner, and A. Valencia, "Assessment of the Second Biocreative PPI Task: Automatic Extraction of Protein-Protein Interactions," *Proc. Second BioCreative Challenge Evaluation Workshop,* 2007.

[20] J.T. Chang and R.B. Altman, "Extracting and Characterizing Gene-Drug Relationships from the Literature," *Pharmacogenetics,* vol. 14, no. 9, pp. 577-586, Sept. 2004.

[21] J. Hakenberg, C. Plake, R. Leaman, M. Schroeder, and G. Gonzalez, "Inter-Species Normalization of Gene Mentions with GNAT," *Proc. European Conf. Computational Biology (ECCB '08),* 2008.

[22] W. Baumgartner, Z. Lu, H. Johnson, J. Caporaso, J. Paquette, E. White, O. Medvedeva, K. Cohen, and L. Hunter, "An Integrated Approach to Concept Recognition in Biomedical Text," *Proc. Second BioCreative Challenge,* 2006.

[23] M. Huang, S. Ding, H. Wang, and X. Zhu, "Mining Physical Protein-Protein Interactions by Exploiting Abundant Features," *Proc. Second BioCreative Challenge,* pp. 237-245, 2007.

[24] J. Hakenberg, C. Plake, L. Royer, H. Strobelt, U. Leser, and M. Schroeder, "Gene Mention Normalization and Interaction Extraction with Context Models and Sentence Motifs," *Genome Biology,* vol. 9, Suppl 2, p. S14, 2008.

[25] L. Hunter, Z. Lu, J. Firby, W. Baumgartner, H. Johnson, P. Ogren, and K.B. Cohen, "OpenDMAP: An Open Source, Ontology-Driven Concept Analysis Engine, with Applications to Capturing Knowledge Regarding Protein Transport, Protein Interactions and Celltype-Specific Gene Expression," *BMC Bioinformatics,* vol. 9, article no. 78, 2008.

[26] A. Doan, L. Gravano, R. Ramakrishnan, and S. Vaithyanathan, "Introduction to the Special Issue on Managing Information Extraction," *ACM SIGMOD Record,* vol. 37, no. 4, p. 5, 2008.

[27] E. Agichtein and L. Gravano, "Snowball: Extracting Relations from Large Plain-Text Collections," *Proc. Fifth ACM Conf. Digital Libraries,* pp. 85-94, 2000.

[28] A. Doan, J.F. Naughton, R. Ramakrishnan, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong, "Information Extraction Challenges in Managing Unstructured Data," *ACM SIGMOD Record,* vol. 37, no. 4, pp. 14-20, 2008.

[29] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, "SystemT: A System for Declarative Information Extraction," *ACM SIGMOD Record,* vol. 37, no. 4, pp. 7-13, 2009.

[30] P.G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano, "Towards a Query Optimizer for Text-Centric Tasks," *ACM Trans. Database Systems,* vol. 32, no. 4, p. 21, 2007.

[31] A. Jain, A. Doan, and L. Gravano, "Optimizing SQL Queries over Text Databases," *Proc. IEEE 24th Int'l Conf. Data Eng. (ICDE '08),* pp. 636-645, 2008.

[32] Y. Miyao, T. Ohta, K. Masuda, Y. Tsuruoka, K. Yoshida, T. Ninomiya, and J. Tsujii, "Semantic Retrieval for the Accurate Identification of Relational Concepts in Massive Textbases," *Proc. 21st Int'l Conf. Computational Linguistics and the 44th Ann. Meeting of the Assoc. for Computational Linguistics (ACL '06),* pp. 1017-1024, 2006.

[33] R. Feldman, Y. Regev, E. Hurvitz, and M. Finkelstein-Landau, "Mining the Biomedical Literature Using Semantic Analysis and Natural Language Processing Techniques," *Information Technology in Drug Discovery Today,* vol. 1, no. 2, pp. 69-80, 2003.

[34] J.D. Martin, "Fast and Furious Text Mining," *IEEE Data Eng. Bull.,* vol. 28, no. 4, pp. 11-20, 2005.

[35] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan, "An Algebraic Approach to Rule-Based Information Extraction," *Proc IEEE 24th Int'l Conf. Data Eng. (ICDE '08),* 2008.

[36] M. Cafarella, D. Downey, S. Soderland, and O. Etzioni, "Knowitnow: Fast, Scalable Information Extraction from the Web," *Proc. Conf. Human Language Technology and Empirical Methods in Natural Language Processing (HLT '05),* pp. 563-570, 2005.

[37] W. Shen, A. Doan, J.F. Naughton, and R. Raghu, "Declarative Information Extraction Using Datalog with Embedded Extraction Predicates," *Proc 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 1033-1044, 2007.

[38] K. Fundel, R. Kuffner, and R. Zimmer, "RelEx-Relation Extraction Using Dependency Parse Trees," *Bioinformatics,* vol. 23, no. 3, pp. 365-371, 2007.

[39] S. Kim, J. Yoon, and J. Yang, "Kernel Approaches for Genic Interaction Extraction," *Bioinformatics,* vol. 24, no. 1, p. 118, 2008.

[40] F. Suchanek, G. Ifrim, and G. Weikum, "LEILA: Learning to Extract Information by Linguistic Analysis," *Proc. ACL Workshop Ontology Learning and Population,* pp. 18-25, 2006.

[41] F. Peng and A. Mccallum, "Accurate Information Extraction from Research Papers Using Conditional Random Fields," *Proc. Human Language Technology Conf. and North Am. Chapter of the Assoc. for Computational Linguistics (HLT-NAACL),* pp. 329-336, 2004.

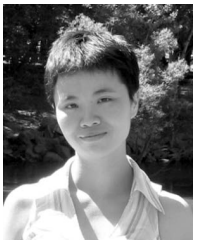
[42] S. Sekine, "On-Demand Information Extraction," *Proc. COLING/ACL Poster Session,* pp. 731-738, 2006.

[43] M. Banko, M.J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni, "Open Information Extraction from the Web," *Proc. Joint Conf. Artificial Intelligence (IJCAI),* 2007.

Luis Tari received the PhD degree in computer science from Arizona State University. He is currently a postdoctoral research fellow in text mining at Hoffmann-La Roche. His research interests include information extraction, information retrieval, and bioinformatics. He is a student member of the IEEE Computer Society.

Phan Huy Tu received the PhD degree in computer science from New Mexico State University. He is currently at Microsoft. His research interests include information extraction, information retrieval, reasoning, planning, knowledge representation, and logic programming.

Jörg Hakenberg received the PhD degree in computer science from Humboldt-Universität zu Berlin. He is currently a research associate at Arizona State University. His research interests include text mining and natural language processing for biomedical applications.



Yi Chen received the PhD degree in computer science from University of Pennsylvania. She is currently an assistant professor of computer science at Arizona State University. Her research interests include keyword search on structured and semistructured data, workflows, social network, information extraction and integration, and data streams. She is a member of the IEEE Computer Society.

Tran Cao Son received the PhD degree in computer science from New Mexico State University. He is currently an associate professor of computer science at New Mexico State University. His research interests include planning, knowledge representation and reasoning, logic programming, autonomous agents, and web agents.

Graciela Gonzalez received the PhD degree in computer science from the University of Texas at El Paso. She is currently an assistant professor of biomedical informatics at Arizona State University. Her research interests include biomedical informatics, text mining, multimedia databases, and human-computer interaction.

Chitta Baral received the PhD degree in computer science from University of Maryland. He is currently a professor of computer science at Arizona State University. His research interests include knowledge representation and reasoning, logic programming, bioinformatics, autonomous agents, and reasoning about actions.