Optimization Issues in Inverted Index-based Entity Annotation

Ganesh Ramakrishnan
IBM India Research Lab
New Delhi, INDIA
ganramkr@in.ibm.com

Sachindra Joshi
IBM India Research Lab
New Delhi, INDIA
jsachind@in.ibm.com

Sanjeet Khaitan*
InfoSpace Inc.
Bangalore, INDIA
sanjeet.khaitan@infospace.com

Sreeram Balakrishnan
IBM Software Group,
San Jose, CA, United States
sreevb@us.ibm.com

ABSTRACT

Entity annotation is emerging as a key enabling requirement for search based on deeper semantics: for example, a search on 'John's address', that returns matches to all entities annotated as an address that co-occur with 'John'. A dominant paradigm adopted by rule-based named entity annotators is to annotate a document at a time. The complexity of this approach varies linearly with the number of documents and the cost for annotating each document, which could be prohibiting for large document corpora. A recently proposed alternative paradigm for rule-based entity annotation [16], operates on the inverted index of a document collection and achieves an order of magnitude speed-up over the document-based counterpart. In addition the index based approach permits collection level optimization of the order of index operations required for the annotation process. It is this aspect that is explored in this paper. We develop a polynomial time algorithm that, based on estimated cost, can optimally select between different logically equivalent evaluation plans for a given rule. Additionally, we prove that this problem becomes NP-hard when the optimization has to be performed over multiple rules and provide effective heuristics for handling this case. Our empirical evaluations show a speed-up factor upto 2 over the baseline system without optimizations.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous

Keywords

Inverted index, evaluation plan, cost estimate

1. INTRODUCTION

Developments in semantic search technology [11, 14, 6] have motivated the need for efficient and scalable entity annotation techniques that can be deployed on a large scale. Entity annotation involves associating one of several well-defined labels with token sequences in unstructured documents. Example labels are 'person name', 'organization', 'place', 'date', *etc.* All rule based systems use a set of rules for the annotation purpose. A rule for entity annotation is a pair consisting of a *pattern* and a *type*. A *pattern* defines the sequence of tokens that need to be identified and the corresponding *type* denotes the type for the token sequence. In this paper, we only consider patterns that are regular expressions over tokens and the properties of tokens such as (i) which dictionaries they belong to, (ii) their part of speech and (iii) orthographic properties. Most current rule-based techniques [13, 1, 15] for this task operate at the document level, where each rule is evaluated against one document at a time. The computational complexity of this approach varies linearly with the number of documents and the cost for annotating each document, which could be prohibiting for large document corpora. Another drawback of this approach is that the entire document collection needs to be reprocessed for every minor modification made in the set of rules. An example modification is the addition of rules having common subexpressions with already evaluated rules.

Recently, [16] proposed a framework for annotating a document collection with typed entities by working solely on its inverted index. The framework yielded an order of magnitude speedup over a state-of-the-art document-based annotator [10]. The improvement achieved was even more pronounced when annotations had to be produced for a set of rules that were incrementally added. We will refer to this framework as *IndexAnnot*.

They use an inverted index that stores a posting list corresponding to each unique token in the document collection. A posting list contains all the position information for the token. They further define a set of operations on posting lists. The $consint$ operation is one of the example operations used in *IndexAnnot* framework. The $consint$ operation is a binary operation that takes two posting lists L_1 and L_2 corresponding to two tokens t_1 and t_2 as arguments. It returns a new posting list such that each entry points to a token sequence which consists of two consecutive tokens t_1 and t_2. The cost of a consint operation is dependent on the length and the nature of the posting lists given as its arguments.

The *IndexAnnot* framework translates the rules for annotation into a sequence of operations on the inverted index [23]. These sequence of operations are also known as evaluation plans. The associative and commutative property of index operations, define an equiva-
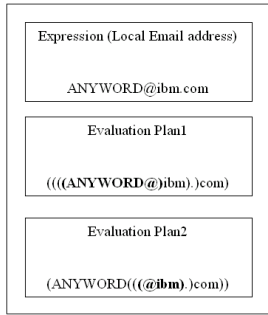
Figure 1: An example rule and two equivalent evaluation plans

lence class of evaluation plans. These evaluation plans are guaranteed to yield the same results, however may vary substantially in their cost of evaluation. As an example, consider the rule given in Figure 1. This rule annotates all the email ids from "ibm.com", as "local emails". The figure also present two equivalent evaluation plans for computing it. The first evaluation plan may be costly as it first determines an intermediate posting list corresponding to *consint* of 'ANYWORD' and '@'. This may result into a much larger intermediate posting list than the posting list achieved by performing a *consint* operation on '@' and 'ibm', as done in the second evaluation plan in the Figure 1. This is analogous to matrix chain multiplication where different orderings of a chain yield the same results but with differing cost [8].

The focus of this paper is to develop methods for estimating the cost of the different evaluation plans and develop algorithms for finding the optimal plan. Based on the nature of index operations, we first define an equivalence class for an evaluation plan. Since, we need to determine the minimum cost evaluation plan without actually evaluating any evaluation plan, we also develop cost estimates techniques for the evaluation plans. We then present a polynomial time algorithm based on dynamic programming, that yields the minimum cost evaluation plan from the equivalence class. We further consider the problem of jointly optimizing the evaluation plans of a collection of rules, where the rules may have common sub-expressions. In this case, the overall optimization is complicated by the fact that the order in which the sub-parts are evaluated also affects the overall execution time. We prove that finding the optimal evaluation plan for this case is an NP hard problem, and develop effective heuristics for handling this situation.

In the database community, several methods have been proposed to optimize query evaluation plans [18]. These methods primarily aim at determining an ordering of join operations that has the minimum cost estimate. Several methods have also been proposed for the problem of multi-query optimization [20, 17]. The join operation is commutative as well as associative, making the query optimization problem NP-Hard. On the other hand, the index operations that are used for entity annotations do not have the commutative property. This is because the order of tokens plays a crucial role in determining annotations. This distinction reduces the search space significantly and makes it possible for us to design a polynomial-time algorithm for finding the optimal evaluation plan. Note that the approach(es) described in this paper are applicable in any plan optimization setting that involves a sequence of operations on postings lists such that the operator is associative but not commutative. The specific associative and non-commutative operator considered in this paper is *consint* (discussed in subsequent sections).

There is a large body of prior art [19, 5, 21] on optimization strategies for evaluating multi-term search queries. However, these efforts focus on efficiently and accurately retrieving the top k documents using some scoring function instead of retrieving all the matching documents and sorting them on the scoring function (which can be very time consuming). Typically, these techniques tradeoff some drop in their precision/recall for high gains in efficiency. Each multi-term query can be viewed as a simple annotation rule. Consider an example regular grammar of the form '$< myname > \rightarrow$ John Smith' that annotates every occurrence of 'John Smith' with the label 'myname'. This rule is equivalent to the phrase query "John Smith". For such simple cases our problem reduces to the phrase query evaluation problem (assuming ranking of the results is not required). However, our algorithm is designed to handle the much more general case of rules that have the expressive power of regular grammars.

There is a body of literature [3, 22, 4, 2] that discusses modifications to the inverted-index structure to support fast evaluation of specific query classes. In prior work, nextword indexes [3, 22] were proposed as a way of supporting phrase queries and phrase browsing. In a nextword index, for each index term or firstword, there is a list of the words or nextwords that follow that term, together with the documents and word positions at which the firstword and nextword occur as a pair. Bahle *et al.* [4] try to overcome two disadvantages of the nextword index, *viz.*, its size (which is typically around half that of the indexed collection) and inefficiency (since nextwords must be processed linearly and compared to a standard inverted index for rare firstwords). They propose evaluation of phrase queries through a combination of an inverted index on rare words and a form of nextword index on common words. Baeza-Yates *et. al.* [2] propose an efficient searching of regular expressions on pre-processed text that runs in logarithmic expected time in the size of the text for a wide class of regular expressions. They use patricia tree as a logical model for the index. While the above mentioned techniques create special data structures to efficiently evaluate queries from specific classes, in [16] we restrict ourselves to the use of off-the-shelf, general purpose index structures, such as that provided by Lucene [12].

The rest of this paper is organized as follows: in Section 2, we recap the basics of the index-based annotation approach. We also introduce the set of index operators required for evaluating the rules as well as define the concept of the AND/OR Tree for representing the evaluation plan. In Section 3, we formulate the optimization problem and present a dynamic programming algorithm for optimally ordering the nodes of the AND/OR Tree . We also present methods for estimating cost of each of the index operators used in the AND/OR Tree . In Section 4 we delve into the issues raised by multiple rules that share common sub-parts. We prove this is an NP hard problem and develop several heuristic algorithms for picking the best evaluation plan. Finally, in Section 5, we present our results that show an overall improvement up to a factor of 2 and Section 6 presents our conclusions.

2. BACKGROUND: ENTITY ANNOTATION USING INVERTED INDEX OPERATIONS

Figure 2 shows the process for entity annotation presented in [16]. A given document collection $\mathcal{D}$ is tokenized and segmented into sentences. The tokens are stored in an inverted index I [23]. The inverted index I has an ordered list $\mathcal{U}$ of the unique tokens u_1, u_2, $..u_W$ that occur in the collection, where W is the number of tokens in I. Additionally, for each unique token u_i, I has a postings list
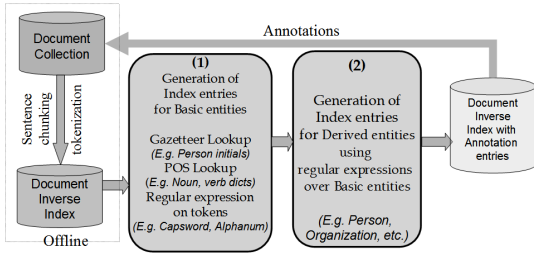
Figure 2: Overview of the entity annotation process referred to in this paper

$L(u_i) = < l_1, l_2, \ldots l_{cnt(u_i)} >$ of locations in $\mathcal{D}$ at which u_i occurs. $cnt(u_i)$ is the length of $L(u_i)$. Each entry l_k, in the postings list $L(u_i)$, has four fields: (1) a document identifier, $l_k.did$, (2) a sentence identifier, $l_k.sid$, (3) the begin position of the particular occurrence of u_i, $l_k.first$ and (4) the end position of the same occurrence of u_i, $l_k.last$. The entries in a posting list are sorted first by did, then by sid, followed by $first$ and finally by $last$.

The input grammar used in [16] is the same as that used for named entity annotations in GATE [10]. The GATE architecture for text engineering uses the Java Annotations Pattern Engine (JAPE) [9] for its information extraction task. JAPE is a pattern matching language. [16] supports two classes of properties for tokens that are required by grammars such as JAPE: (1) orthographic properties such as an uppercase character followed by lower case characters, and (2) dictionaries (gazetteers) to which a token or a token sequence belongsd. Examples of dictionaries are 'location' and 'person name'. The set of tokens along with entity types specified by either of these two properties are referred to as *Basic Entities*. The instances of basic entities specified by orthographic properties must be single tokens. However, instances of basic entities specified using dictionary containment properties can be token sequences.

2.1 Generation of posting lists for basic and derived entities

The module (1) of the system shown in Figure 2, identifies postings lists for each basic entity type. These postings lists are entered as index entries in I for the corresponding types. For example, if the input rules require tokens/token sequences that satisfy *Capsword* or *Location Dictionary* properties, a postings list is created for each of these basic types. Constructing the postings list for a basic entity type with some orthographic property is a fairly straightforward task; the postings lists of tokens satisfying the orthographic properties are merged (while retaining the sorted order of each postings list). The mechanism for generating the postings list of basic entities with gazetteer properties was developed in [16]. A rule for NE annotation may require a token to satisfy multiple properties such as *Location Dictionary* as well as *Capsword*. The posting list for tokens that satisfy multiple properties are determined by performing an operation $parallelint(L, L')$ over the posting lists of the corresponding basic entities. The $parallelint(L, L')$ operation returns a posting list such that each entry in the returned list occurs in both L as well as L'.

The module (2) of the system shown in Figure 2 identifies instances of each annotation type, by performing index-based operations on the posting lists of *basic entity* types and other tokens. The different operation types and the sequence of their application will be dis-

cussed in the remainder of this section. Every intermediate posting list generated by module (2) has a field Opt. When any expression in the JAPE grammar is optional (or has the '*' or '?' operators associated), the value of Opt for the corresponding posting list is set to 'true'.

2.2 Operations on Postings List

Two operations on postings lists were defined in [16].

1. $merge(L_1, L_2, \ldots, L_n)$: Returns a posting list such that each entry in the returned list occurs in at least one of L_1 or L_2 .... or L_n. If any of L_1 or L_2 .... or L_n has the Opt field set to true, the resultant posting list also has its Opt field set to true. This operation is associative as well as commutative. Using these properties, the $merge$ operation can be performed in a manner similar to the merge-sort algorithm. The cost of performing $merge(L_1, L_2, \ldots, L_n)$ operation is as follows:

$$\underbrace{cost}_{merge} = O(\log(n) \sum_{i=1}^{n} |L_i|) \qquad (1)$$

2. $consint(L, L')$: Returns a postings list such that each entry in the returned list points to a token sequence which consists of two consecutive subsequences @sa and @sb within the same sentence, such that, L has an entry for @sa and L' has an entry for @sb. If either of L or L' has its Opt field set to true, the resultant posting list is merged with the corresponding optional posing list. If both L and L' have their Opt fields set to true, the resultant posting list is merged with L and L' and the Opt field of the result is set to true.

   We will denote $consint(L, L')$ by $L * L'$.

There are several methods for computing $consint(L, L')$ depending on the relative size of L and L'. If they are roughly equal in size, a simple linear pass through L and L', analogous to a merge, can be performed. If there is a significant difference in sizes, a more efficient modified binary search algorithm can be implemented. The details are shown in Figure 3.

$consint(L, L')$
Let M elements of L be $l_1 \cdots l_M$
Let N elements of L' be $l'_1 \cdots l'_N$
if $M < N$ **then**
    set $j = 1$
    for $i = 1$ to M **do**
        set $k = 1$, keep doubling k until
        $l'_j.first \leq l_i.last < l'_{j+k}.first$
        binary search the L' in the interval $j \cdots k$
        to determine the value of p such that
        $l'_p.first \leq l_i.last < l'_{p+1}.first$
        if $l'_p.first = l_i.last$ a match exists, copy to output
        set $j = p + 1$
    end for
else
    Same as above except l and l' are reversed
end if

Figure 3: The modified binary search algorithm for consint

The cost of performing $consint(L, L')$ is as follows:

$$\underbrace{cost}_{consint} = \begin{cases} min\big(\Lambda(L, L'), \Lambda(L', L), |L| + |L'|\big) \text{ if } L.Opt = f, L'.Opt = f \\ min\big(\Lambda(L', L), |L| + |L'|\big) \text{ if } L.Opt = f, L'.Opt = t \\ min\big(\Lambda(L, L'), |L| + |L'|\big) \text{ if } L.Opt = t, L'.Opt = f \\ \big(|L| + |L'|\big) \text{ if } L.Opt = t, L'.Opt = t \end{cases}$$

(2)

where t and f stand for $true$ and $false$ respectively and $\Lambda(X, Y) = 2|X|(\log_2(|Y|/|X|) + 1)$.

2.3 Inverted Index-based Annotation using a

Left-Deep AND/OR Tree

Each annotation pattern, which is a regular expression over basic entities is first translated into a Left-Deep AND/OR Tree [16]. A Left-Deep AND/OR Tree specifies a sequence of $consint$ and $merge$ operations on posting lists to obtain a postings list for the annotation. Associated with each node in the tree is a regular expression and a postings list that points to all the matches for the node's regular expression in the document collection. There are two node types: AND nodes where the output list is computed from the consecutive intersection ($consint$) of the postings lists of two children nodes and OR nodes where the output list is formed by merging the posting lists of all the children nodes. Additionally, each node has two binary properties: Opt and selfLoop. The first property is set if the regular expression being matched is of the form 'R?', where '?' denotes that the regular expression R is optional. The second property is set if the regular expression is of the form 'R+', where '+' is the Kleen operator denoting one or more occurrences. For the case of 'R*', both properties are set.

We formally define an AND/OR Tree as follows:

DEFINITION 1. *An AND/OR Tree is a directed tree $T = (\mathcal{V}, \mathcal{E})$ such that*

1. *Each vertex $v \in \mathcal{V}$ has the following attributes:*

   - *type: Type of a vertex is one of the following: (1)AND, (2)OR and (3) LEAF.*
   - *Opt: A boolean indicating whether the subtree rooted at v is optional.*
   - *selfLoop: A boolean indicating whether the subtree rooted at v can repeat more than once.*
   - *signature: A string that denotes the regular expression associated with v.*

2. *Every AND vertex v has exactly two children vertices.*

If the right child of every AND node is a a leaf an OR node or an AND node with a self-loop, T is called a Left-Deep AND/OR Tree.

The Left-Deep AND/OR Tree is recursively built by scanning the regular expression from left to right and identifying every sub-regular expression for which a sub-tree can be built. Details of the algorithm that builds the Left-Deep AND/OR Tree from a regular expression are provided in [16].

Figure 4 shows an example regular expression and the corresponding Left-Deep AND/OR Tree ; AND nodes are shown as circles
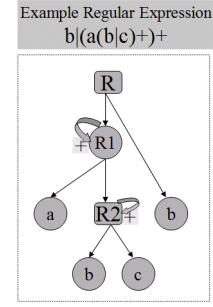

Example Regular Expression
b|(a(b|c)+)+

Figure 4: An example regular expression and the corresponding Left-Deep AND/OR Tree

whereas OR nodes are shown as square boxes. Nodes having Opt and selfLoop properties are labeled with +, ∗ or ?. The edges in the tree represent dependency between computing nodes. The main regular expression is at the root node of the tree. The leaf nodes correspond to basic entities and words.

Figure 5 outlines the algorithm for computing the postings list of a regular expression by operating bottom-up on the Left-Deep AND/OR Tree .

```
for Each node v in the reverse topological sorting of G_R do
    if v.nodetype == AND then
        Let v1 and v2 be the children of v
        L(v) = consint(L(v1), L(v2))
    else if v.type == OR then
        L(v) = merge(L(v.child1), · · · , L(v.children))
    end if
    if v.selfLoop == 1 then
        L(v) = consint(L(v), +)
    end if
    if v.Opt == 1 then
        L(v).Opt = 1
    end if
end for
```

Figure 5: The algorithm for computing postings list of a regular expression R using the inverted index I and the corresponding Left-Deep AND/OR Tree G_R

3. OPTIMIZING AND/OR Trees
3.1 Equivalence of AND/OR Trees

An AND/OR Tree defines a plan for computing the posting list for a regular expression on an inverted index I. More specifically, the tree specifies a sequence of $consint$ and $merge$ operations that should be performed on posting lists using the algorithm outlined in Figure 5. The overall time for computing the posting list at the root is proportional to the number of accesses made to the posting lists involved.

DEFINITION 2. *Given a sequence of operations S on posting lists from an index I, the $cost(S, I)$ denotes the number of accesses made to the posting lists while evaluating S. For an AND/OR Tree G, $cost(G, I)$ is the cost of the sequence of operations performed on posting lists using the algorithm outlined in Figure 5.*

We make the following important observations on the $consint$ and $merge$ operations:
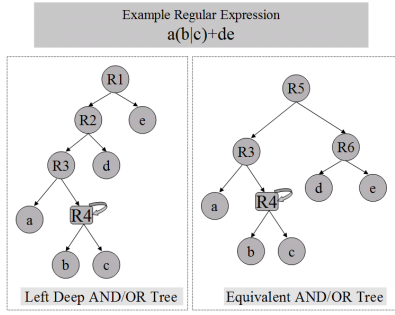
Figure 6: An example regular expression, the corresponding Left-Deep AND/OR Tree and an equivalent AND/OR Tree

1. The $consint(L, L')$ operator defined in Section 2.2 is associative *i.e.*, $(L_1 * L_2) * L_3 = L_1 * (L_2 * L_3)$. However, *consint* is not commutative. Thus, we cannot permute the order of *consint* operations without changing the result. By virtue of the associative property of *consint*, a sequence of *consint* operations $L_1 * L_2 * \ldots * L_k$ can be computed by parenthesizing the sequence in several equivalent ways. As an example, the parenthesization corresponding to a left-most evaluation first is $(\ldots (((L_1 * L_2) * L_3) \ldots) * L_k$. Each parenthesization of a chain of *consint* operations leads to a different *cost*, depending on the sequence and the nature of intermediate posting lists.

2. The *merge* operator is commutative and associative, *i.e.*, $merge(L_1, L_2, \ldots, L_n) = merge(\Pi(L_1, L_2, \ldots, L_n))$, where $\Pi(L_1, L_2, \ldots, L_n)$ is any permutation of the posting lists. The number of possible permutations for a list of length n is $n!$. The *merge* operates on several posting lists simultaneously. The *cost* of evaluating $merge(L_1, L_2, \ldots, L_n)$ is independent of the order of the posting lists being merged (*c.f.*, Section 2.2).

The $cost(G, I)$ is sensitive to the ordering of *consint* operations in G and does not depend on the ordering of the posting lists involved in the *merge* operations. Based on this observation, we define a relation Θ between two AND/OR Trees G_1 and G_2.

DEFINITION 3. *We say that $G_1 \Theta G_2$ iff G_1 and G_2 are either the same or differ only in the parenthesization of consint chains in G_1 and G_2.*

The relation Θ is *symmetric*, *reflexive*, and *transitive* by definition. Let us denote the equivalence class induced by Θ on an AND/OR Tree G by $\mathcal{G}_\Theta$. By virtue of the associativity property of the *consint* operation, evaluation of any $G' \in \mathcal{G}_\Theta$ will yield the same result. Figure 6 shows an example regular expression, the corresponding Left-Deep AND/OR Tree and an equivalent AND/OR Tree .

The value of $cost(G, I)$ cannot be obtained without executing the algorithm in Figure 5. In practice, however, we would require to compare two plans for their costs before they are evaluated. We can estimate the value of $cost(G, I)$ using cost estimates of *consint* and *merge* based on length of posting lists at leaves and length estimates of intermediate posting lists. Let $\kappa(G, I, \rho)$ denote an
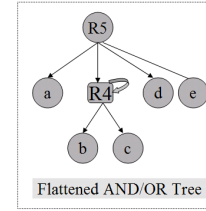

Figure 7: The Canonical-Flattened AND/OR Tree corresponding to the Left-Deep AND/OR Tree in Figure 6.

estimated cost of evaluating G on I, using some property ρ. ρ is a function of G and I that estimates some property of the posting list that results from the evaluation of G on I. As an example, $\rho(G, I)$ could be the estimated length of the posting list that results from evaluating G on I. $\kappa(G, I, \rho)$ gives an estimate of the cost of evaluating G on I, using the property function ρ. We next define the problem of determining an optimal plan $G_{opt} \in \mathcal{G}_\Theta$ for G, given a cost estimation model κ, a property ρ and the index I. We refer to this problem as OptPlan .

3.2 Problem Formulation

The plan specified by the Left-Deep AND/OR Tree G obtained using the algorithm outlined in [16] need not be optimal. There are many equivalent plans in the equivalence set $\mathcal{G}_\Theta$ for G, that could potentially have lower cost estimates than G.

DEFINITION 4. *Let G be an AND/OR Tree and I an inverted index. Let $\kappa(G, I, \rho)$ denote the cost of evaluating G on I using κ and a property ρ of G and I. We define OptPlan as the problem of determining $G_{opt} \in \mathcal{G}_\Theta$ that minimizes the estimated cost, $\kappa(G_{opt}, I, \rho)$. That is, $G_{opt} = \underbrace{argmin}_{G \in \mathcal{G}_\Theta} [\kappa(G, I, \rho)]$.*

Since $cost(G, I)$ is only sensitive to the ordering of *consint* operations in G and the length of intermediate posting lists, we will only consider cost estimates $\kappa(G, I, \rho)$ that are determined by the *consint* operations. We next define the Canonical-Flattened AND/OR Tree that represents every sequence of nodes operated by consecutive *consint* as children of a common AND node. Thus, the Canonical-Flattened AND/OR Tree represents a *consint* chain without the parentheses information.

DEFINITION 5. *A Canonical-Flattened AND/OR Tree is an AND/OR Tree that allows any number of children for every AND node. Every child of an AND node must hav one of the following properties (1) type is OR (2) type is LEAF and (3) type is AND and atleast one of Opt or isSelfLoop options is set. Figure 7 shows the Canonical-Flattened AND/OR Tree corresponding to the Left-Deep AND/OR Tree in Figure 6.*

In Figure 8, we give the algorithm for converting any Left-Deep AND/OR Tree G_{LD} into the corresponding Canonical-Flattened AND/OR Tree G_F. The algorithm is invoked as $flattenTree(G_{LD}, eList)$, where $eList$ is an empty list. The algorithm does a depth first search traversal on G_{LD} and in the course, chains together every sequence of nodes that are operated on by consecutive *consint* operations as sibling nodes of a common AND node in the resultant

G_F. Note that a Canonical-Flattened AND/OR Tree could also be directly generated from the original regular expression.

```
flattenTree(G, currentList)
if (G.type == OR) || (G.type == LEAF) || Opt || isSelfLoop
then
    currentList.add(G)
end if
if (G.type == OR) || Opt || isSelfLoop then
    for Each G' ∈ G.childList do
        Create a new List, newList
        flattenTree(G', newList)
        G'.childList = newList
    end for
end if
if (G.type == AND) then
    for Each G' ∈ G.childList do
        flattenTree(G', currentList)
    end for
end if
if G is a ROOT node then
    G_F.childList = currentList
end if
```

Figure 8: The algorithm for converting a Left-Deep AND/OR Tree G_{LD} into the corresponding Canonical-Flattened AND/OR Tree G_F.

The OptPlan problem on G_{LD} reduces to the following problem of OptConsintChain at each AND node of G_F.

DEFINITION 6. *Let $\Gamma = L_1 * L_2 * \ldots * L_n, n \geq 3$ be a chain of consint operations on posting lists and let $L_{1,n}$ denote the resultant posting list. Let P_Γ be any parenthesization of Γ. Thus, P_Γ determines a sequence in which consint is applied on the postings lists in Γ. Let ρ be a property of Γ such as the estimated length of the resultant posting list. The OptConsintChain problem is to determine a parenthesization P_Γ of the list Γ such that the cost estimate $\kappa_{consint}(P_\Gamma, I, \rho)$ is minimized.*

3.3 A dynamic programming algorithm

The number of parenthesizations for a sequence of n consint operations is the Catalan number [8] $\frac{1}{n+1}\binom{2n}{n}$, which is exponential in n. We could go through each possible parenthesization (brute force), but this would require time $\Omega(4^n/n^{(3/2)})$, which is very slow and impractical for large n.

We can, however, apply dynamic programming to this problem. Let $\Gamma_{i,j}$ denote the *consint* chain between the ith and the jth posting lists. Let $MinC[i, j]$ be the minimum cost for computing $\Gamma_{i,j}$. The key observations that form the basis of the algorithm are:

1. The outermost parentheses partition the chain of *consints* between i and j at some $k, i \leq k < j$.

2. The optimal parenthesization order has optimal ordering on either side of k.

A recurrence for this problem is stated in Equation 3.3.

$$MinC[i, j] = \underbrace{min}_{i \leq k \leq j-1} \{MinC[i, k] + MinC[k + 1, j] \\ + \kappa_{consint}(G^{consint}_{\Gamma_{i,k}, \Gamma_{k+1,j}}, I, \rho)\}$$
$$MinC[i, i] = 0$$

where $G^{consint}_{\Gamma_{i,k}, \Gamma_{k+1,j}}$ denotes a graph which has a *consint* node as a root node that has two children corresponding to $\Gamma_{i,k}$ and $\Gamma_{k+1,j}$. The function $\kappa_{consint}(\ldots)$ returns a cost estimate for the graph $G^{consint}_{\Gamma_{i,k}, \Gamma_{k+1,j}}$ using the estimates of property ρ for the posting list $\Gamma_{i,k}$ and $\Gamma_{k+1,j}$. As updates are made to $MinC$, using values already computed, we need to

Note, that we need to determine the property values for these lists.

There are only $\binom{n}{2}$ substrings between 1 and n. Thus, it requires only $\Theta(n^2)$ space to store the optimal cost for each of them. We represent all the possibilities in a triangle matrix. We also store the value of k in another triangle matrix $MinSplit$ to reconstruct the optimal parenthesization. The diagonal moves up to the right as the computation progresses. Figure 9 outlines our algorithm for computing the minimum cost parenthesization of the *consint*-chain Γ. The runtime for this algorithm is $O(n^3)$.

```
Procedure OptConsintChain (Γ)
for i = 1 to n do
    MinC[i, i] = 0
end for
for diag = 1 to n − 1 do
    for i = 1 to n − diag do
        j = i + diag
```
$$MinC[i, j] = \underbrace{min}_{i \leq k \leq j-1} [MinC[i, k] + MinC[k + 1, j] \\ + \kappa_{consint}(G^{consint}_{\Gamma_{i,k}, \Gamma_{k+1,j}}, I, \rho)]$$

$$MinSplit[i, j] = \underbrace{argmin}_{i \leq k \leq j-1} [MinC[i, k] + MinC[k + 1, j] \\ + \kappa_{consint}(G^{consint}_{\Gamma_{i,k}, \Gamma_{k+1,j}}, I, \rho)]$$
```
        Let s = MinSplit[i, j]
        ρ[i, j] = updateρ(ρ[i, s], ρ[s + 1, j])
    end for
end for
return MinSplit
```

Figure 9: The algorithm for computing the minimum cost parenthesization of a $consint$-chain Γ.

The method $update\rho(\rho([i, k], \rho[k + 1, j])$ obtains a ρ value for the posting list of $\Gamma_{i,j}$ based on the ρ values for $\Gamma_{i,k}$ and $\Gamma_{k+1,j}$. Examples of ρ will be discussed in Section 3.4. In particular, Equation 4 gives the possible update equations for $update\rho(\ldots)$.

3.3.1 Algorithm for OptPlan
We now outline the algorithm that produces an AND/OR Tree G_{opt} with the minimum cost estimate from a given Canonical-Flattened AND/OR Tree G_F. The algorithm, outlined in Figure 10, performs a depth-first traversal of G_F and for every AND node G,

the *consint*-chain formed by the children of G is optimally parenthesized using the OptConsintChain algorithm ((*c.f.* Figure 9). For every other node, OptPlan is recursively invoked.

Procedure **OptPlan** (G_F)
Let G be the root node of G_F
for Each $G' \in G.childList$ **do**
  Call OptPlan on G'
end for
if $(G.type == AND)$ **then**
  Form a *consint*-chain Γ comprising the children of G
  Optimize Γ using OptConsintChain to get an AND/OR Tree at G
end if

Figure 10: The algorithm for computing the minimum cost AND/OR Tree G from a given Canonical-Flattened AND/OR Tree G_F.

3.4 Cost Estimates using Properties of Posting Lists

In this section, we define some cost estimation models. As mentioned in Section 3.1, we need to assess the cost of a plan even before it is evaluated. This is done using the property ρ of the resultant list. In our implementation, we use the estimated length of the resultant list as the ρ. The ρ estimate for the posting list L_{basic} of any word or basic entity is the actual length of the posting list, *i.e.*, $\rho(L_{basic}) = |L_{basic}|$. The ρ estimate of the resultant list for a merge operation can be computed using the following equation:

$$\rho(G_{L_1,L_2,...,L_n}^{merge}, I) = \sum_{i=1}^{n} \rho(L_i) \qquad (3)$$

This approximation is an upper-bound on the length of the list resulting from a *merge* operations.

To determine the estimated length of the resultant list for a *consint* operation, we use one of the following estimates of ρ in the update function $update\rho(.)$:

$$\rho_{min}(G_{L_1,L_2}^{consint}, I) = min(\rho(L_1), \rho(L_2))$$
$$\rho_{am}(G_{L_1,L_2}^{consint}, I) = \frac{(\rho(L_1) + \rho(L_2))}{2}$$
$$\rho_{gm}(G_{L_1,L_2}^{consint}, I) = \sqrt{\rho(L_1) * \rho(L_2)}$$
$$\rho_{hm}(G_{L_1,L_2}^{consint}, I) = \frac{2 * \rho(L_1) * \rho(L_2)}{\rho(L_1) + \rho(L_2)} \qquad (4)$$

Here, ρ_{min}, ρ_{am}, ρ_{gm} and ρ_{hm} are length estimates obtained by using the minimum, arithmetic mean, geometric mean and harmonic mean respectively. The ρ_{min} estimate is not an upper-bound on the length of the resultant list from a *consint* operation. This is because of the following reason. Given a regular expression, there can be multiple entries in its posting list that have the same *first* or *last* values. As an example, consider the posting list for $CAPSWORD+$ which contains an entry for every consecutive sequence of capitalized words of length 1 or more; given a sequence of n consecutive capitalized words in the document collection, there will be atleast n entries in the posting list that have the same *start* value (similarly for *end*). We therefore consider the other ρ estimates to account for the length of both posting lists.

We obtain the cost estimate $\kappa_{consint}(G, I, \rho)$ based on length of posting lists at leaf nodes in G and length estimates of intermediate posting lists. This is achieved by choosing the length estimates as given in the above equations. The value of $\kappa_{consint}(G_{L_1,L_2}^{consint}, I, \rho)$ is obtained using the following equation.

$$\kappa_{consint} = \begin{cases} min(\Lambda(L, L'), \Lambda(L', L), |L| + |L'|) & \text{if } L.Opt = f, L'.Opt = f \\ min(\Lambda(L', L), |L| + |L'|) & \text{if } L.Opt = f, L'.Opt = t \\ min(\Lambda(L, L'), |L| + |L'|) & \text{if } L.Opt = t, L'.Opt = f \\ (|L| + |L'|) & \text{if } L.Opt = t, L'.Opt = t \end{cases}$$

$$(5)$$

where, $\Lambda(X, Y) = 2|X|(\log_2(|Y|/|X|) + 1)$.

4. OPTIMIZING MULTIPLE AND/OR Trees

Multiple AND/OR Trees might have common sub-sequences of evaluations. Figure 11 shows two Canonical-Flattened AND/OR Trees , G_1 and G_2 with nodes $R2$ and $R5$ that are effectively the same. Given this fact, G_1 and G_2 also share a sub-sequence of *consint* operations, namely, $L(a) * L(R5) * L(d)$. The application of OptPlan (*c.f.* Section 3.3.1) on G_1 or G_2 ignores this common sub-sequence information. In this section, we address the problem of optimizing multiple AND/OR Trees simultaneously.
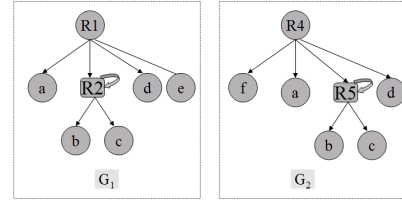

Figure 11: Two AND/OR Trees G_1 and G_2 having common sub-expressions.

In Section 3.2 we reduced the OptPlan problem to the OptConsintChain problem. Thus we can pose the problem of simultaneously optimizing multiple AND/OR Trees to the following problem of simultaneously optimizing multiple *consint*-chains.

DEFINITION 7. *Let $\mathcal{S}$ be a set, such that each $\Gamma \in \mathcal{S}$ is a chain of consint operations on posting lists and is of the form $\Gamma = L_1 * L_2 * \ldots * L_n, n \geq 3$. We define MPlanOpt as the problem of determining the parenthesization P_Γ of every $\Gamma \in \mathcal{S}$ so that the cost estimate $\sum_{\Gamma \in \mathcal{S}} \kappa(P_\Gamma, I, \rho)$ is minimized, where the cost for computing the posting list of any sub-expression β of P_Γ is considered only once.*

4.1 NP-Hardness

In this section, we prove that the MPlanOpt problem is NP-Hard.

THEOREM 1. *The MPlanOpt problem is* NP-*Hard.*

Instead of proving the Theorem 1, we prove the hardness result for the following simpler problem, RuleEvalWithCache.

DEFINITION 8. *Let, $\Gamma = L_1 * L_2 * \ldots * L_n$ be a consint chain. The problem of* RuleEvalWithCache *is to determine a parenthesization P_Γ of the chain Γ such that the cost estimate $costEst(P_\Gamma, I)$ is minimized, where the cost for computing the posting list of any sub-expression β of P_Γ is considered only once.*

The problem RuleEvalWithCache aims at evaluating a consint chain while caching the posting list of every intermediate sub-expression so that the cost of computing its posting list is accounted for only once. If any sub-expression is repeated in Γ, the cached result of it can be re-used. As an example consider the consint chain $\Gamma = C*D*X*Y*C*D$. The subexpression $C*D$ occurs twice in the chain Γ. The parenthesization $((C*D)*((X*Y)*(C*D)))$ of Γ, is one of several possible parenthesizations that can gain from the caching of posting list of intermediate sub-expressions.

THEOREM 2. *The* RuleEvalWithCache *problem is NP hard.*

PROOF. We give a reduction from the *smallest grammar* problem (SmallGrammar), a well-known NP-Hard problem [7]. The SmallGrammar problem can be stated as follows: Given a string σ, what is the smallest context free grammar that generates exactly σ. The size of the grammar is defined to be the total number of symbols on the right sides of all production rules. For example, the smallest context free grammar generating the string: $\sigma =$ <u>a rose is a rose is a rose</u> is as follows: (1) $S \rightarrow BBA$, (2)$A \rightarrow$ <u>a rose</u> and (3) $B \rightarrow A$ <u>is</u>.

There is a simple reduction from the SmallGrammar problem to RuleEvalWithCache. We omit details of the proof owing to space constraints. □

4.2 Heuristics for solving MPlanOpt

Since MPlanOpt is an NP-hard problem, we resort to heuristics for solving this problem. All of our heuristics are based on one of the following two techniques:

(1) *Caching during plan optimization:* Utilizing common subexpressions across AND/OR Trees during plan optimization.

(2) *Caching during plan evaluation:* Storing in a hash map, the posting list for each evaluated node in an AND/OR Tree , using the node signature as a key.

Based on these two techniques, we developed following 3 heuristics.

(1) **IOCache** : Order the AND/OR Trees randomly and optimize each plan individually. Evaluate rules in the same order and store in a hash map, the posting list of every evaluated node, using the corresponding node signature as the key. This heuristic uses only the second technique listed above.

(2) **COCache** : Order the AND/OR Trees randomly and optimize plans in that order. After optimizing a plan using the OptConsintChain algorithm, the signature of every node in the optimized plan is cached. While optimizing subsequent plans, the cost estimate κ for a sub-expression $\Gamma[i, j]$ is set to 0, if the sub-expression is already present in the cache.

(3) **CSECache**: Store the signature of each node of every AND/OR Tree against its frequency. The cost estimate κ of any sub-expression is obtained by dividing the estimate obtained in Section 3.4 by the above frequency.

5. EXPERIMENTS
5.1 Experimental setup

In this section, we present empirical comparison of performance of the optimized evaluation plans under various strategies, against the naive evaluation plan. The experiments were performed on three data sets, *viz.*

(1) A combination of Reuters-21578 data set[1] and the 20 Newsgroups data set[2]. The data set consists of 16331 non-blank documents after stripping off the header information. The size of this data set was 93 MB.

(2) A subset of the TREC wt10G data set comprising of 167943 documents from directories WTX001 to WTX011 and from directories WTX025 to WTX033. We consider only the 'title' and 'body' fields of each document in our experiments. The size of this data set was 1 GB.

(3) The enron email data set[3]. The data consists of 267978 documents after elimination of header information and discarding the resultant blank documents. The size of this data set was 1.2 GB.

Our entire implementation is in java. The experiments were performed on a dual 3.2GHz Xeon server with 4 GB RAM. The indexing was performed using Lucene [12]. Prior to indexing, the sentence segmentation and tokenization of each data set was performed using in-house Java versions of standard tools[4].

Our rule specification is in JAPE [9]. JAPE is a version of CPSL (Common Pattern Specification Language). JAPE provides finite state transduction over annotations based on regular expressions. The JAPE grammar requires information from two main resources: (i) a tokenizer and (ii) a gazetteer.

We experimented with two manually crafted set of rules: (1) 12 rules for 'Organization' and (2) 15 rules for 'Date'.

In the following subsection, we present results that we achieve by optimizing individual plans, while in Section 5.3, we present our results for the multi-plan optimization problem.

5.2 Individual Plan Optimization

In our first experiment, we compare the performance of the individual plan optimization algorithm *OptPlan* (*c.f* Section 3), abbreviated as IO, using different properties ρ_{min}, ρ_{gm}, ρ_{hm} and ρ_{am} against the baseline evaluation plan of the Left-Deep AND/OR Tree . The comparsion is performed in terms of the actual number of accesses (*cost*) that are made to the posting list entries during all the consint operations that are peformed at the time of evaluation. The experiments were performed for the 'organization' and 'date' rule sets.

Figure 12 compares the cost of *OptPlan* using the different properties against the baseline on the reuters+20NG, trec and the enron data sets. We observe that *OptPlan* performs consistently and substantially better than the baseline on all data sets, for both the rulesets and for all properties except ρ_{am}. The reason for the anomalous behavior of ρ_{am} is as follows. The length of the posting list resulting from $consint(L_1, L_2)$ is generally more biased toward the length of the shorter of the two lists L_1 and L_2. The *harmonic mean* and *geometric mean* of $|L_1|$ and $|L_2|$ are also more biased

[1] http://www.daviddlewis.com/resources/testcollections/reuters21578/

[2] http://people.csail.mit.edu/jrennie/20Newsgroups/

[3] http://www.cs.cmu.edu/~enron/

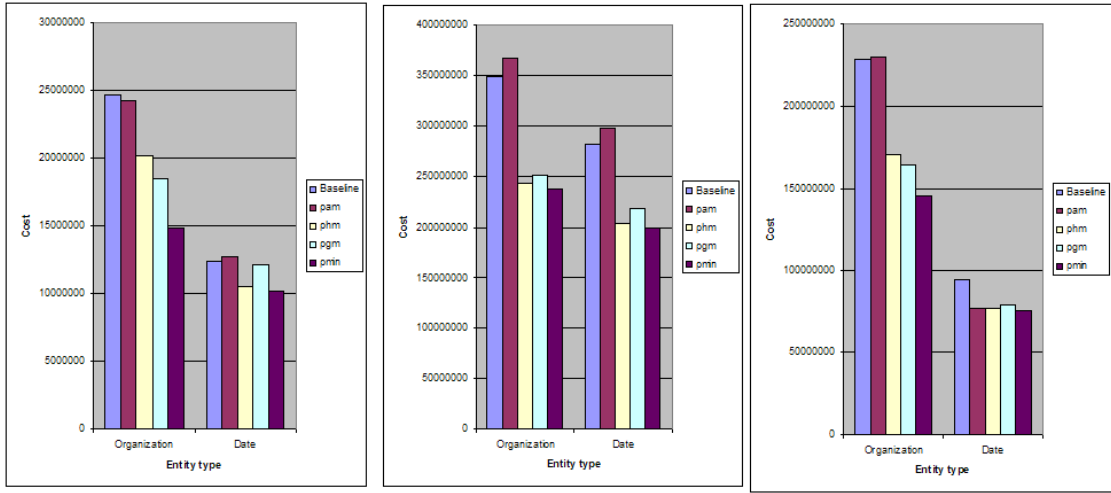[4] http://l2r.cs.uiuc.edu/~cogcomp/tools.php

Figure 12: Comparison of cost of the *OptPlan* algorithm against the baseline using different properties on the reuters, trec and enron data set from left to right.
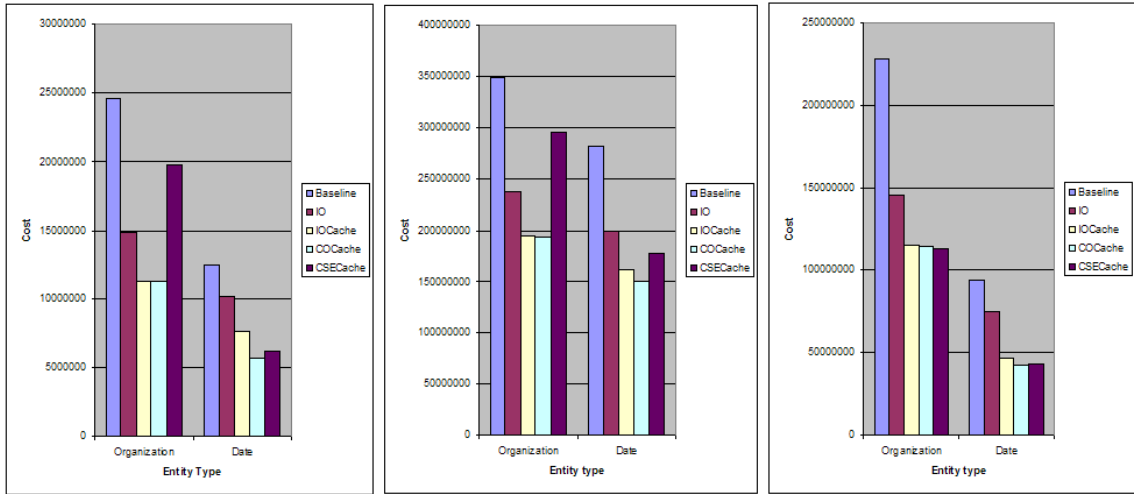

Figure 13: Comparison of cost of the *IOCache*, *COCache* and the *CSECache* heuristics against that of the baseline and the *OptPlan* algorithms on the reuters, trec and enron data set from left to right.

toward the length of the shorter of the two lists, while their *minimum* is determined only by the shorter list. Hence, these are good estimates of the length of the posting list resulting from *consint*. On the other hand, the *arithmatic mean* is equally biased by both the numbers and hence is not a good estimate of the length of the resultant list.

Figure 14 shows the performance gain we obtain on some individual rules for 'organization' and 'date' types on the reuters+20NG data set using ρ_{min}. The figure illustrates that the proposed algorithm is consistently better than the baseline across a variety of rules.

5.3 Multiple Plan Optimization

In our second experiment, we compare the performance of the heuristics that optimize multiple plans (*c.f* Section 4) against the baseline.

The measure used for comparison was the same *cost* as described

above. For these experiments, we chose ρ_{min} as the property , given its consistent and superior performance over the other properties .

Figure 13 compares the cost of the heuristics *IOCache*, *COCache* and *CSECache* against OptPlan and the baseline, on the reuters+20NG, trec and the enron data sets. We observe that *IOCache* shows a consistent speedup over *OptPlan* (which is always better than the baseline) and *COCache* performs consistently better than or as well as *IOCache*. However, this consistency is not observed in *CSECache*; this heuristic subsidizes the cost of computing sub-expressions common across plans, even before the optimal plans are computed and hence may not always perform well in practice.

6. CONCLUSION

In this paper we investigated the issues concerning optimization of evaluation plans using the inverted index. We developed a dynamic programming method that achieves an optimal evaluation plan for a
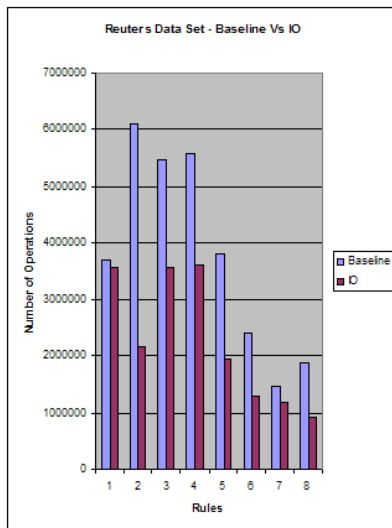
Figure 14: Comparison of cost of evaluation of plans for some sample 'organization' and 'date' rules, before and after optimization.

given rule in polynomial time. We further investigated the problem of multi-rule optimization and proved that the problem is NP hard. Several heuristic approaches for optimizing multiple rules were developed and implemented. Our experiments show a speed-up factor upto 2 over the baseline.

In our future work, we intend to develop optimization plans that also search in the equivalence class of evaluation plan including the distributive property of merge operation over *consint* operation. Better length estimates can be looked for. Including *consint* operation with gaps will further increase the evaluation plan search space, inviting the scope for even better results. Incorporation of such an operation can be made.

7. REFERENCES

[1] D. Appelt, J. Hobbs, J. Bear, D. Israel, M. Kameyama, D. Martin, K. Myers, and M. Tyson. Sri intnl fastus system: Muc-6 test results and analysis. In *MUC6 '95: Proc. of the 6th conf. on Message understanding*, 1995.

[2] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of ACM*, 43(6), 1996.

[3] D. Bahle, H. Williams, and J. Zobel. Eighth symposium on string processing and information retrieval. In *Proceedings of Australasian Database Conference*, 2001.

[4] D. Bahle, H. E. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. In *Proceedings of SIGIR*, 2002.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of CIKM*, 2003.

[6] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *Proceedings of the 15th international conference on World Wide Web*, 2006.

[7] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and abhi shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of STOC*, 2002.

[8] T. H. Cormen, C. E. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[9] H. Cunningham. Jape – a java annotation patterns engine, 1999.

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

[11] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien. Semtag and seeker: bootstrapping the semantic web via automated semantic annotation. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 2003.

[12] B. Goetz. The Lucene search engine: Powerful, flexible, and free. http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html, 2000.

[13] R. Grishman. Information extraction: Techniques and challenges. In *SCIE '97: Intnl. summer School on Information Extraction*, 1997.

[14] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar semantic search: a database approach to information retrieval. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006.

[15] D. Maynard, V. Tablan, C. Ursu, H. Cunningham, and Y. Wilks. Named entity recognition from diverse text types. In *Recent Advances in Natural Language Processing Conf.*, 2001.

[16] G. Ramakrishnan, S. Balakrishnan, and S. Joshi. Entity annotation based on inverse index operations. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 492–500, Sydney, Australia, July 2006. Association for Computational Linguistics.

[17] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[18] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[19] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proceedings of SIGIR*, 2005.

[20] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient-views. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 319–330, 1998.

[21] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.

[22] H. Williams, J. Zobel, and P. Anderson. Whatś next? index structures for efficient phrase querying. In *Proceedings of Australasian Database Conference*, pages 141–152, 1999.

[23] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.