Scalable Ad-hoc Entity Extraction from Text Collections

Sanjay Agrawal Kaushik Chakrabarti Surajit Chaudhuri Venkatesh Ganti
Microsoft Research
{sagrawal,kaushik,surajitc,vganti}@microsoft.com

ABSTRACT

Supporting entity extraction from large document collections is important for enabling a variety of important data analysis tasks. In this paper, we introduce the "ad-hoc" entity extraction task where entities of interest are constrained to be from a list of entities that is specific to the task. In such scenarios, traditional entity extraction techniques that process *all* the documents for each ad-hoc entity extraction task can be significantly expensive. In this paper, we propose an efficient approach that leverages the inverted index on the documents to identify the subset of documents relevant to the task and processes only those documents. We demonstrate the efficiency of our techniques on real datasets.

1. INTRODUCTION

The task of extracting names of people, products, locations and other such named entities is important for several applications. Enterprise search and ad-hoc business analytics and reporting applications over document collections can significantly benefit from entity extraction technology. Traditionally, the goal of entity extraction is to identify *all* named entities occurring in the input documents. For example, given a document $d2$ in Figure 1, a product entity extractor may identify the occurrence of the entity "Sony Vaio FS740" starting at word position 2. Leading approaches primarily rely on machine learning (ML) and natural language processing (NLP) techniques in order to identify various types of entities in documents (e.g., people names, locations, products) [4]. Sometimes a set of entities is provided as input to aid the entity extraction process but the entities extracted from the documents are not limited to this input set.

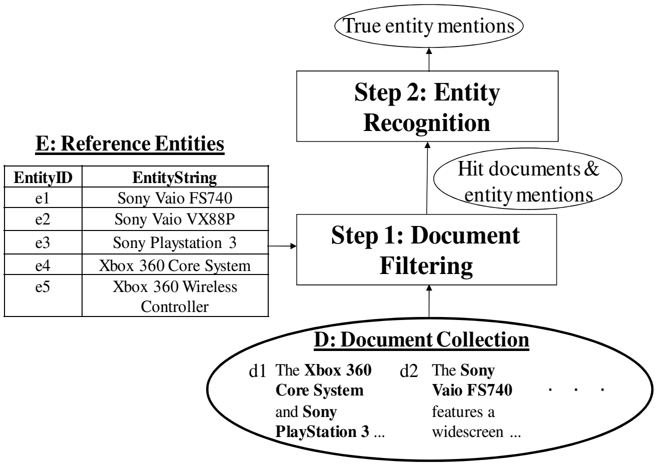
In this paper, we focus on a subclass of entity extraction scenarios with unique characteristics. These extraction tasks enable rich data mining and analysis over document collections. We now discuss a few such tasks.

1. Consider an analyst mining the news articles that appeared in the past one year for popular handheld devices. The list of interesting devices may be explicitly given or obtained via a query over a different database. The analyst needs to extract all occurrences of handheld devices in the given list from the news articles in order to perform the comparative analysis of handheld devices. The same news document repository may also be mined (perhaps, by a different analyst) for articles on a set of celebrities or on a set of movies.
2. Consider an analyst mining a product review repository for reviews pertaining to a set of electronic products obtained from CNET.com or from PriceGrabber.com. Once again all occurrences of the selected products in the documents must be extracted in order to enable the required analysis. The same review repository may also be mined for reviews on a set of kitchen appliances or gadgets.
3. Consider a researcher mining a medical documents database for a set of articles on bacteria, viruses, and parasites related to mosquito-borne diseases. The list of bacteria, viruses, and parasites may be obtained from the drug adminstration or an environmental agency. The same medical document database may be mined by another researcher looking for reports on variations of Flu viruses.

Each of the above entity extraction tasks exhibit the following set of unique characteristics.

- *Dictionary-constrained extraction*: The entities to be identified in a document repository is constrained to be from a given list of entities. This is unlike classical entity extraction schemes where the entity list is used as an aid, but not necessarily to restrict the entities extracted from documents.
- *Ad-hoc entity sets*: The set of entities to be extracted from the documents is not predefined, much like filter conditions in traditional database queries. Hence, identifying the relevant documents at the time of insertion of a document into the repository is not possible. Further, as seen in many practical scenarios (say, movies in news articles), these entity sets can often be large.
- *Approximate match*: To robustly support the entity identification tasks such as in the examples above, even if the entire entity string is not mentioned in the document, identifying sub-strings containing a meaningful subset of tokens of the original entity, e.g., "Vaio

VLDB '08, August 24-30, 2008, Auckland, New Zealand


True entity mentions
Step 2: Entity Recognition
Hit documents & entity mentions
Step 1: Document Filtering
E: Reference Entities
D: Document Collection
d1 The **Xbox 360 Core System** and **Sony PlayStation 3** ...
d2 The **Sony Vaio FS740** features a widescreen ...
. . .

EntityID	EntityString
e1	Sony Vaio FS740
e2	Sony Vaio VX88P
e3	Sony Playstation 3
e4	Xbox 360 Core System
e5	Xbox 360 Wireless Controller

Figure 1: Ad-hoc Entity Extraction

FS740" instead of "Sony Vaio FS740" is desired.[1] Therefore, given a list E of entities, matching against variations obtained through limited subsetting of the tokens is very important in practice.

In this paper, we define the *ad-hoc entity extraction* task as one that takes a collection of documents D and an ad-hoc *reference* set E of entities, and extracts all entities in E whenever they occur in the given documents. Note that the goal is to extract only the entities in E and we leverage this unique characteristic in our paper. We initially focus on exact match between entity strings and their occurrences in documents, and discuss the issue of approximate match in Section 4. Figure 1 illustrates the ad-hoc entity extraction scenario, and the two steps enabling it. The *document filtering* step identifies all documents that contain any entity string in E along with entity strings they contain. We refer to these documents as *"hit documents"*. Then, the *entity recognition* step analyzes the entity strings in the hit documents and their document contexts (using ML and NLP techniques) in order to ensure that the entity string in the document is actually a reference to the entity and not a generic phrase in the language. For example, the distinction between the movie "60 seconds" versus a phrase "60 seconds" (in reference to time) is important while extracting a set of movies. We apply known techniques for the entity recognition step [10, 15]. *In this paper, we focus on the first step of document filtering for entity extraction.* We now consider two simple approaches for ad-hoc entity extraction.

Document Scan Approach: A basic approach, as illustrated in Figure 2, is to scan each document and check whether or not the document contains any reference entity string. We can apply standard string matching techniques [3] for *entity string matching*. Only the documents that contain one or more reference entity strings are passed to the entity recognition step. In an ad-hoc entity extraction scenario, users can *dynamically* provide new or updated reference entity sets. Hence, unlike traditional entity extraction, we cannot piggy back on the processing of documents while building the inverted index. Consequently, we pay the stiff performance penalty (for each ad-hoc entity extraction task) of scanning all the documents and processing them to identify the hit documents for the given task.

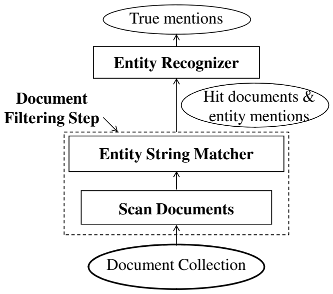


Figure 2: Document Scan Approach

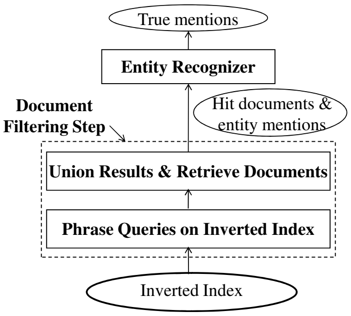

True mentions
Entity Recognizer
Document Filtering Step
Hit documents & entity mentions
Union Results & Retrieve Documents
Phrase Queries on Inverted Index
Inverted Index

Figure 3: Phrase Query Approach

Phrase Query Approach: Inverted index exists on document collections to enable efficient keyword search. Instead of scanning the documents, we can use the inverted index to identify the hit documents. A straightforward approach to do so is illustrated in Figure 3. We issue one phrase query per entity in the reference set against the inverted index and union the results to obtain the identifiers of all the hit documents. For example, for the reference entity set in Figure 1, we issue 5 phrase queries "Sony Vaio FS740", "Sony Vaio VX88P", "Sony PlayStation 3", "XBox 360 Core System" and "XBox 360 Wireless Controller" and union the results to get all the hit documents. We then retrieve the documents and pass it to the entity recognition step. However, we observe that in many domains, there are sets of tokens that are shared among the different entities. For example, "Sony Vaio VX88P" shares a number of tokens with "Sony Vaio FS740". Furthermore, to handle approximate matches, we need to generate variations of the original entities and add them to the reference set. The variations of an entity tend to have a high degree of overlap among themselves as well as with the original entity.[2] For example, the entity "Sony Vaio FS740" may produce variants "Vaio FS740" and "Sony FS740", all of which share a number of tokens with the original entity. The phrase query approach ends up scanning the document lists of these overlapping tokens multiple times and intersecting them multiple times, and is hence expensive (as illustrated in Section 6).

Our Approach: In this paper, we present efficient document filtering techniques for ad-hoc entity extraction tasks. Like the phrase query approach, we also exploit the inverted index. However, our key insight is to reduce the cost of querying against the inverted index by exploiting the overlap of tokens among the set of entities. Our main contributions are summarized as follows.

1. We propose a novel document filtering approach that judiciously chooses a set of DNF (disjunctive normal form) queries and executes them against the inverted index. This approach exploits the overlap of

[1] In this paper, we refer to words as tokens.

[2] This is the only feasible strategy to handle approximate match if we use the phrase query approach. We study more efficient strategies in the context of our approach in Section 4. These strategies also introduce a high degree of overlap of tokens in the reference set.

tokens among the entities and is significantly more efficient compared to the phrase query approach described above. Unlike the phrase query approach that returns the exact set of hit documents, the DNF queries will return a superset of the hit documents. These documents need to be processed subsequently using the entity string matcher to filter out the false positives. While the document scan and phrase query approaches lie at two ends of the spectrum with a high document processing cost (but no query cost) and a high query cost (but no document processing cost) respectively, our DNF query approach tries to carefully trade-off between the two costs in order to reduce the overall execution cost. We propose a cost model to quantify the overall execution cost and develop heuristic algorithms to pick the DNF queries that minimize the overall execution cost. Furthermore, if the inverted index engine exposes a relational view of the inverted index which supports efficient join with a potentially large set of tokens, we can further improve the performance of document filtering techniques.

2. We develop techniques for identifying approximate matches between entity strings in the reference set and their mentions in documents based on popular notions of similarity such as weighted Jaccard similarity [5, 10].

3. We evaluate our techniques over real datasets and show that they significantly improve the efficiency of ad-hoc entity extraction tasks in most cases. We also provide cost estimates for choosing among the candidate techniques.

The remainder of the paper is organized as follows. In Section 2, we formally define the document filtering for ad-hoc entity extraction problem and discuss our architecture. In Section 3, we present the cost model and the algorithm. In Section 4, we extend our solution to enable approximate matches between entity strings and their mentions in documents. Section 5 discusses some extensions. In Section 6, we present a thorough experimental study on real datasets. In Section 7, we review relevant related work and conclude in Section 8.

2. PROBLEM DEFINITION

In this paper, we focus on developing efficient techniques for the document filtering step, which we now formalize. Let g_E be an *entity string matcher* which takes a document d and returns the *mentions* in d of all entities from E. We say a document d mentions an entity $e \in E$ at position *pos* iff the contiguous sub-sequence of tokens in d at position *pos* is identical to the sequence of tokens in entity e and is defined by the triplet (d, e, pos). We denote the output of g_E on d by $g_E(d)$. An example instantiation for g_E is the trie-based Aho-Corasick technique [3], which builds a trie over all entities in E. The trie is used to significantly reduce the number of comparisons between subsequences of words in an input document d and those in the entity set E.

DEFINITION 1. **(Document Filtering for Ad-hoc Entity Extraction)** *Given a set D of documents and the set E of entities, the task of* document filtering *for ad-hoc entity extraction is to return the set $\{(d, g_E(d)) : d \in D \text{ and } g_E(d) \neq \phi\}$.*

Entities	Covering Tokens	Covering Token Pairs
Sony Vaio FS740	Vaio	Sony, Vaio
Sony Vaio VX88P		
Sony Playstation 3	Playstation	Sony, Playstation
XBox 360 Core System	XBox	XBox, 360
XBox 360 Wireless Controller		

Table 1: Entity Cover

EXAMPLE 1. Consider the reference entity set $E = \{e1, e2, e3, e4, e5\}$ and the document collection $D = \{d1, d2\}$ in Figure 1. Document $d1$ mentions the entities $e4$ and $e3$ at word positions 2 and 7 respectively while document $d2$ mentions entity $e1$ at word position 2. The result of document filtering with respect to E will include the document $d1$ along with mentions $(d1, e4, 2)$ and $(d1, e3, 7)$ and document $d2$ along with mention $(d2, e1, 2)$.

2.1 Inverted Index APIs

We assume that the following queries are efficiently supported over an inverted index by a full text search engine.

2.1.1 DNF Formula over Set of Token Sets

Consider a set $\{T_1, \ldots, T_K\}$ of token sets. We use small letters such as t to denote an individual token and capital letters such as T to denote a set of tokens (also referred to as "token set"). The DNF formula $dnf(\{T_1, \ldots, T_K\})$ over the set $\{T_1, \ldots, T_K\}$ of token sets is a disjunction of K conjuncts where the i^{th} conjunct is formed by *and*-ing the tokens in token set T_i. Formally, let $and(T_i)$ denote the *and query* over the tokens $t_i^1, \cdots, t_i^{|T_i|}$ in token set T_i, i.e., $and(T_i) = (t_i^1 \texttt{ and } \ldots \texttt{ and } t_i^{|T_i|})$. Then, $dnf(\{T_1, \ldots, T_K\}) = (and(T_1) \texttt{ or } \ldots \texttt{ or } and(T_K))$. Note that the above formula when executed on the inverted index returns the result $\bigcup_{i=1}^{K} \bigcap_{t \in T_i} D(t)$. where $D(t)$ denotes the set of identifiers (referred to as docids) of all documents $d \in D$ that contain the token t. We refer to the above result as the *docid set* of the DNF formula. For example, the dnf formula for the set of token sets {{Sony, Vaio}, {XBox, 360}} is (('Sony' `and` 'Vaio') `or` ('XBox' `and` '360')).

2.1.2 Or Query over Entity Phrases

We first consider a phrase query for a single entity. The phrase query $phrQ(e)$ for an entity e containing the sequence of tokens $[t_1, \ldots, t_{|e|}]$ is the query "$t_1 \ldots t_{|e|}$". The *or query over entity phrases* $orPhrQ(e_1, \ldots, e_K)$ for a set $\{e_1, \ldots, e_K\}$ of entities is the query $(phrQ(e_1) \texttt{ or } \ldots \texttt{ or } phrQ(e_K))$. Note that the above query returns the identifiers of the documents that contain one or more phrases from the above set. For example, or query of phrases for the entities {Sony Vaio FS740, Sony Vaio VX88P} is ("Sony Vaio FS740" `or` "Sony Vaio VX88P").

For the phrase query approach discussed in Section 1, instead of executing separate phrase queries for each entity, it would be better to execute a single or-query of all the entity phrases as this would avoid incurring the initialization cost for each query. For example, for the reference entity set in Figure 1, we issue the query ("Sony Vaio FS740" `or` "Sony Vaio VX88P" `or` "Sony PlayStation 3" `or` "XBox 360 Core System" `or` "XBox 360 Wireless Controller"). Note that such an or-query would have $|E|$ terms which can be in the order of thousands. However, typical inverted index engines support queries containing only a small number, order of ten to hundred, terms. In such a case, we *batch* the execution by partitioning the large or-query into multiple smaller or-queries such that each smaller or-query has the maximum number of allowed terms. We write the result of each smaller or-query into a temporary relation and then union the result. The above batching strategy also applies to the DNF query approach as discussed in detail in Section 3.

2.2 Our Approach

Our main insight is to exploit the overlap of tokens among entities in order to reduce the cost of querying against the inverted index. If a set of entities share a set of tokens T, we reduce the query cost by executing the simpler query $and(T)$ instead of the complex phrase query (or queries) corresponding to the entities that share T. Note that since a document that mentions an entity e also contains *every* subset of the set of tokens in e, the query $and(T)$ will return a *superset* of the docids returned by the corresponding phrase queries. This superset property guarantees correctness, i.e., the filter will not miss any hit documents. The false positives can be filtered out by processing the returned documents using the entity string matcher.

For example, consider the set of reference entities shown in Table 1.[3] The token 'Vaio' is shared by the first two entities; we can execute the query ('Vaio') instead of the phrase queries "Sony Vaio FS740" and "Sony Vaio VX88P". Similarly, we can execute the query ('Playstation') instead of the phrase query "Sony PlayStation 3" and the query ('Xbox') instead of the phrase queries "XBox 360 Core System" and "XBox 360 Wireless Controller". The benefit is that the overall cost of executing these three single token queries is much less compared to the five phrase queries. On the down side, the single token queries might return a much larger set of documents which need to be processed by the entity string matcher. This high document processing cost might offset the savings in the query cost. In that case, we might consider issuing more constrained queries to reduce the document processing cost. For example, since the first two entities share the token pair {Sony, Vaio}, we can execute the query ('Sony' and 'Vaio') instead of those two phrase queries. Similarly, we can issue the query ('Sony' and 'Playstation') instead of the third phrase query and the query ('Xbox' and '360') instead of the fourth and the fifth phrase queries. Executing these is more expensive than the single token queries presented above because we now need to perform pairwise intersections of the docid sets of the individual tokens. However, the document processing cost may be much lower since the resulting superset may be much smaller than in the single token case.

We therefore need to examine the tradeoff between issuing fewer and more efficiently executable queries against the inverted index versus that of processing more documents using entity string matcher before sending them to the entity recognizer. Consider a token set T. Let $E(T)$ denote the set of all entities in E whose token sets are a superset of the token set T (also referred to as the entities *covered* by T). Let $D(T)$ denote the intersection $\cap_{t \in T} D(t)$. We refer to $D(T)$ as the *docid set* of the token set T. Let $D(e)$ denote the set of docids of the documents that mentions the entity e. We refer to $D(e)$ as the *docid set* of the entity e. Note that $D(e)$ corresponds to the result of phrase query $phrQ(e)$. Intuitively, a token set T is a good candidate if (i) $|E(T)|$ is large and (ii) $|D(T)| - |\bigcup_{e \in E(T)} D(e)|$ is small. (i) implies that we can cover a large number of phrase queries with a single and-query and (ii) implies that the number of extra documents (false positives) to be processed by the entity string matcher is low.

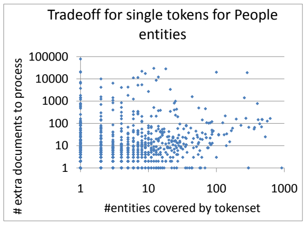


Figure 4: People: Coverage versus False Positives

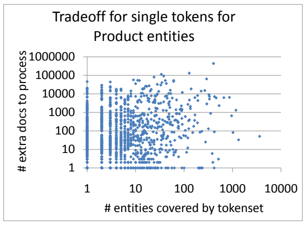


Figure 5: Products: Coverage versus False Positives

We now illustrate empirically that there does exist several such token sets that we can pick to reduce the overall cost of document filtering. We use a real news dataset consisting of about one million documents and real-life people and product entity sets. Figure 4 is a scatter plot where each point represents a randomly chosen token t occurring in people names. The X-value $|E(t)|$ represents the number of entities t covers while the Y-value represents the number of extra documents (false positives) to process if t is picked. Points with high X values and low Y values are desirable; that is, these tokens (or token sets) can replace many phrase queries and still not be processing a very large number of documents. Observe the clustering of points at low Y values. This is promising since there exists a large number of tokens that occur in many entities while the number of additional documents we need to process by choosing them is small. Figure 5 plots a similar distribution for a randomly chosen set of consumer and electronic products. Similar conclusions can be drawn here as well.

Observe that our approach exploits the overlap of tokens among multiple entities. Whenever we find token sets which only cause a small number of additional documents to be processed while covering a large number of entities (i.e., reducing a large number of phrase queries), we are able to significantly improve efficiency. In Section 3, we present the algorithm to pick such token sets.

2.3 Architecture

We now describe the architecture, which is illustrated in Figure 6. The input consists of (i) a reference set E of entities, (ii) a set D of documents, and (iii) an inverted index II (full text index) built over D. The output is the set of true mentions of entities in E in D. As discussed earlier, we consider a two step approach. The output of the first Document Filtering step is the set of hit documents along with the entity strings they mention. However, the mentions may not actually be referring to an entity in E (as illustrated by the "60 seconds" example in Section 1). The second Entity Recognition step applies an ML-based or NLP-based

[3] This example shows the overlap among the original entities. In reality, the overlap can be much higher due to presence of the variants generated for approximate match.

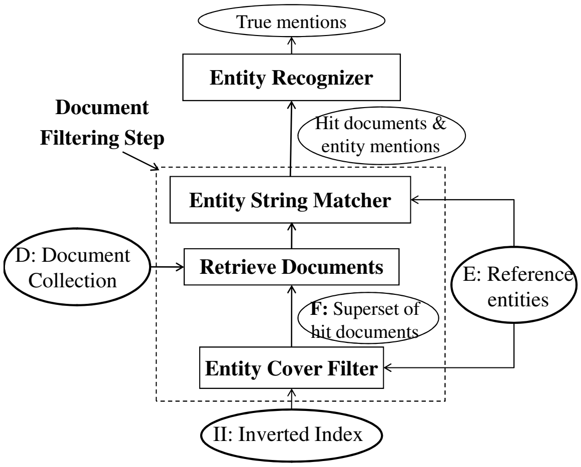


Figure 6: Entity Cover Approach

entity recognizer to detect the true entity mentions. In this paper, we focus on developing efficient techniques for the first step. We now briefly discuss the main components in the *document filtering* step.

The document filtering step has three components. The first component *Entity Cover Filter* takes as input the reference set E of entities and the inverted index II, and outputs the set F of filtered document identifiers. F is a superset of the hit documents. The second *document retrieval* component retrieves each document whose docid is in F. The third *entity string matching* component processes each retrieved document using the entity string matcher g_E (which is implemented using the Aho-Corasick algorithm) to filter out the false positives.[4] The third component's output is the result of document filtering for ad-hoc entity extraction problem over D and E. We apply known techniques for the second and third components in the Document Filtering step. In the next two sections, we describe efficient techniques for the first step (Entity Cover Filter component).

3. ENTITY COVER FILTER

We now describe techniques for the *Entity Cover Filter* component in Figure 6. Consider the DNF formula $dnf(\{Tokens(e_1), \ldots, Tokens(e_{|E|})\})$ formed from all entities in $E = \{e_1, \ldots, e_{|E|}\}$ where $Tokens(e_i)$ denotes the set of tokens in entity e_i. The docid set of this formula is the superset of the hit documents and is hence a valid output of the entity cover filter. But, as discussed earlier, there are a large number of other DNF formulas which can also be used to obtain a superset. Each formula has a different querying cost and returns a different superset (i.e., has different entity string matching cost). Our goal is to find the DNF formula with the lowest overall cost. We first formalize the class of formulas whose result sets are guaranteed to contain all the hit documents.

DEFINITION 2. **(Entity Cover (EC))** *We say that a token set T* covers *an entity e if $T \subseteq Tokens(e)$. A set EC of token sets is an* entity cover *for a set E of entities if for every entity $e \in E$ there exists a token set $T \in EC$ which covers e.*

Given an entity cover EC, it may be used to identify a superset of hit documents. The intuition is to prepare the DNF formula $dnf(EC)$ over all the token sets in EC. If $\{T_1, \ldots, T_K\}$ are the token sets in EC, we prepare the formula $dnf(EC) = (and(T_1)$ or $\ldots$ or $and(T_K))$ and execute it against the inverted index in a batched fashion.

Observation 1 (Correctness): Given an entity cover EC for the set E of entities, the docid set of $dnf(EC)$ includes identifiers of all the hit documents.

EXAMPLE 2. Consider the reference entity set E in Table 1. The set of token sets in the third column of Table 1 is an entity cover for E. The dnf formula for the above entity cover is ((‘Sony’ `and` ‘Vaio’) `or` (‘Sony’ `and` ‘Playstation’) `or` (‘Xbox’ `and` ‘360’)). The above query when executed against the inverted index produces a superset of hit documents.

3.1 Cost Model

We now define our cost model, which forms the basis for picking the “best” entity cover to solve the document filtering problem. The cost has two main parts:

- *DNF Query Cost*: the cost of executing the DNF formula against the inverted index. As discussed above, we only consider DNF formulas over tokens to query the inverted index.
- *Entity String Matching Cost*: the cost of processing the documents in the docid set of the DNF formula using entity string matcher (g_E) to filter out the false positives.

DNF Query Cost: We model the cost of a DNF query as the total cost of scanning the document lists of each token in each conjunct of the DNF query plus a fixed initialization cost.

Full text search engines typically execute a DNF query $dnf(\{T_1, \ldots, T_K\})$ by executing each conjunct $and(T_i)$ and then union-ing the results. Executing a conjunct $and(T_i)$ involves scanning the document lists corresponding to the tokens in the conjunct and merging them. We model the cost of execution of a conjunct to be proportional to the sum of sizes of document lists of the tokens in the conjunct, i.e., the cost of executing $and(T_i)$ is $(\sum_{t \in Tokens(T_i)} |D(t)|) \cdot C_{idx}$ where C_{idx} is a constant obtained by calibration experiments. The union cost is insignificant compared to the other costs and is hence omitted from the cost formula.

Recall that usually inverted indexes allow only a small number of literals in the DNF formula. We model this restriction by allowing only a small the number of conjuncts, say at most B, to be present in each DNF query. We therefore partition a large DNF formula with K conjuncts into multiple smaller DNF sub-formulas with at most B conjuncts each. We execute the sub-formulas independently against the inverted index and union the results. We assume that each DNF query issued against an inverted index has a fixed initialization cost C_{ini}. Thus, the execution cost of the entire DNF formula consisting of K conjuncts corresponding to token sets $T_1, \ldots, T_K$ is the sum of execution costs of the $\lceil \frac{K}{B} \rceil$ smaller DNF queries:

$$\left(\sum_{i=1}^{K} \sum_{t \in Tokens(T_i)} |D(t)| \right) \cdot C_{idx} + \lceil \frac{K}{B} \rceil \cdot C_{ini}$$

[4] Observe that the entity string matching and entity recognition steps can be performed at the same time to avoid parsing the relevant documents twice. We separated them in order to crisply define the document filtering problem we address in this paper.

Example 3. Consider the dnf formula ((‘Sony’ and ‘Vaio’) or (‘Sony’ and ‘Playstation’) or (‘Xbox’ and ‘360’)) from Example 2. Suppose the number of documents containing the tokens ‘Sony’, ‘Vaio’, ‘Playstation’, ’Xbox’ and ‘360’ are 1000, 500, 600, 800 and 400 respectively. Suppose $B = 2$. Suppose C_{idx} and C_{ini} are 0.1 and 10 respectively. The DNF query cost of the above formula is $(1000 + 500 + 1000 + 600 + 800 + 400) \cdot 0.1 + \lceil \frac{3}{2} \rceil \cdot 10 = 450$.

Entity String Matching Cost: Processing the documents in the docid set of the DNF formula involves (i) retrieving the documents and (ii) applying the entity string matcher on each of the retrieved documents. The document processing cost is proportional to the size of docid set of the DNF formula, i.e., $|\bigcup_i D(T_i)|$. Estimating $|\bigcup_i D(T_i)|$ accurately (using the inclusion-exclusion principle) requires maintaining not only the cardinalities of docid sets of candidate token sets but also those of intersections of docid sets of pairs of candidate token sets. Since the number of pairs of candidate token sets can be very large, this involves maintaining a very large amount of statistics. Hence, we ignore the overlap among docid sets of individual conjuncts and model it as $\sum_i |D(T_i)|$. Although this approximation would cause this particular component of the cost to be over estimated, our experiments show that even this approximate model leads to the choice of DNF formulas that significantly reduce the overall cost. The entity string matching cost is modeled as $\sum_i |D(T_i)| \cdot C_{doc}$ where C_{doc} is the average cost of retrieving a document and applying g_E on it. C_{doc}, like C_{idx} and C_{ini}, is also obtained via system calibration.

So, the overall cost of document filtering using a given DNF formula $f_{dnf} = dnf(T_1, \ldots, T_K)$ is the sum of the DNF query processing and the entity string matching costs:

$$cost(f_{\text{dnf}}) = \overbrace{\left(\sum_{i=1}^{K} \sum_{t \in Tokens(T_i)} |D(t)|\right) \cdot C_{idx} + \left\lceil \frac{K}{B} \right\rceil \cdot C_{ini}}^{\text{DNF Query Cost}} + \underbrace{\left(\sum_i |D(T_i)|\right) \cdot C_{doc}}_{\text{Entity String Matching Cost}} \quad (1)$$

3.2 Min-Cost Entity Cover

Our goal is to choose the entity cover that has the minimum cost. Formally, we can state the problem as follows.

Definition 3. **(Min-Cost Entity Cover Problem)** *Given a set E of entities and a set D of documents, find the entity cover EC^* over E such that for any other entity cover EC of E, $cost(dnf(EC^*)) \leq cost(dnf(EC))$*

The min-cost entity cover problem is NP-hard in the number of tokens in E, i.e., $|\bigcup_{e \in E} Tokens(e)|$. The reduction follows from the weighted set cover problem.

Theorem 1. *For given sets E of entities and D of documents, the problem of identifying the min-cost entity cover is NP-hard.*

3.3 Greedy Heuristic

We now discuss the greedy heuristic to solve the min-cost entity cover problem. This is motivated by the greedy heuristic for the weighted set cover problem. We empirically show in Section 6 that the greedy heuristic results in a low cost entity cover.

Consider the DNF formula $dnf(E) = dnf(Tokens(e_1), \ldots, Tokens(e_{|E|}))$. Our approach is to iteratively pick the best candidate token set T_{best} and replace all conjuncts in $dnf(E)$ corresponding to the entities that T_{best} covers with $and(T_{best})$. We continue this as long as it reduces the cost of the DNF formula. We start with a set of candidate token sets $\mathcal{T}_{cands}$ from which we pick the token sets. We discuss the generation of candidate token sets in Section 3.3.1.

The best token set at any stage of the algorithm is the one that results in the maximum reduction in cost of the current DNF formula. We refer to this reduction of cost as the *benefit* of the token set. In the i^{th} iteration, let $Cov_i(E)$ be the set of entities already covered by one of the previously chosen token sets. Given a candidate token set T to be picked in the current iteration, let $E_i(T) = E(T) - Cov_i(E)$ denote the set of entities covered by T, but not by any previously chosen token set. The benefit *Benefit(T)* of T is the reduction in cost of the current DNF formula by replacing the conjuncts corresponding to the entities in $E_i(T)$ with the conjunct $and(T)$:

$$\left(\sum_{e \in E_i(T)} \sum_{t \in Tokens(e)} |D(t)| - \sum_{t \in Tokens(T)} |D(t)|\right) \cdot C_{idx} + \left(\frac{|E_i(T)| - 1}{B}\right) \cdot C_{ini} - \left(|D(T)| - |\bigcup_{e \in E_i(T)} D(e)|\right) \cdot C_{doc} \quad (2)$$

Note that benefit computation requires cardinalities of docid sets of all the candidate tokensets as well the of all the entities. In Section 3.3.1, we describe sampling based estimators for these cardinalities. As before, due to the hardness of estimating $|\bigcup_{e \in E_i(T)} D(e)|$ accurately, we model it as $\sum_{e \in E_i(T)} |D(e)|$.

The pseudocode for the greedy algorithm is shown in Figure 7; it takes E and $\mathcal{T}_{cands}$ as input. Let *TokenSet(e)* denote the set of token sets in $\mathcal{T}_{cands}$ which cover the entity e. At any point in the algorithm, let *Uncovered* denote the set of entities that are not yet covered by any chosen token set in the entity cover EC, and *Covered* be the set covered by at least one token set in EC.

Initialization: Recognize that, in each iteration, we need to (i) pick the best token set and (ii) update the benefits of the token sets impacted by this choice. To support (i) efficiently, we build a priority queue over the candidate token sets. We compute the *initial* benefit of each candidate in $\mathcal{T}_{cands}$ using Equation 2. We insert them into the priority queue with the benefit as the priority. For (ii), we need to lookup the set of entities $E(T)$ covered by a given token set T and the set $TokenSet(e)$ of token sets that cover a given entity e. While there are several choices for data structures to make this lookup efficient, we use in-memory hash tables.

Iteration: In the i^{th} iteration, the greedy algorithm picks the token set T_{best} with the maximum benefit from the priority queue, adds it to the entity cover EC and then updates the benefit of the impacted token sets in the priority queue. The impacted token sets are the ones covering some entity in $E_i(T_{best})$, i.e., $\bigcup_{e \in E_i(T_{best})} TokenSet(e)$. We can obtain $E_i(T_{best})$ by intersecting the set *Uncovered* with $E(T_{best})$. We obtain $E(T_{best})$ and $TokenSet(e)$ by looking up the hash tables containing the associations. For any entity $e \in E_i(T_{best})$, we reduce the benefit of the impacted to-

```
GreedyEntityCover ($E$, $\mathcal{T}_{cands}$)
  Build priority queue PQ over $\mathcal{T}_{cands}$ by benefit
  UnCovered = $TokenSets(E)$, Covered = {}
  $EC$ = {}, LazyUpdates = {}
  while (Uncovered not empty)
     if (PQ is empty) break
     else $T \leftarrow$ PQ.pop()
     while (T $\in$ LazyUpdates)
        Remove $T$ from LazyUpdates & update Benefit(T)
        if Benefit(T)>0 push $T$ in PQ
        $T \leftarrow$ PQ.pop()
     $EC = EC \cup \{T\}$
     Add token sets covering any entity in
     $E(T)$ − Covered to LazyUpdates
     Uncovered = Uncovered − $E(T)$
     Covered = Covered $\cup$ $E(T)$
  return (EC, Phrase queries for uncovered entities)
```

Figure 7: Greedy min-cost Entity Cover Algorithm

ken set $T \in TokenSet(e)$ by $|D(e)| \cdot C_{doc} + (\sum_{t \in e} D(t)) \cdot C_{idx} + \frac{C_{ini}}{B}$. This follows directly from Equation 2; this is the incremental update to the benefit of T if we update $E_i(T)$ by removing e from it. Note that if an impacted token set covers multiple entities in $E_i(T_{best})$, the above reduction will occur multiple times, once for each entity it covers. If the benefit of an impacted token set becomes negative, it is not considered in subsequent iterations. We continue to iterate until we have either covered all entities or there are no remaining token sets with positive benefit. The latter might happen when there are few uncovered entities remaining and none of the remaining token sets cover enough entities among these uncovered ones to result in any reduction in overall cost. In that case, we issue phrase queries for the uncovered entities in addition to the DNF query $dnf(EC)$.

Lazy Updates: A straightforward approach that would reduce the benefit of impacted token sets in the priority queue as soon as T_{best} is picked would cause many wasteful updates as those updated values will probably not be used. We lazily update benefits of impacted token sets as follows. Instead of updating all impacted tokens in the priority queue, we add all impacted token sets to a "LazyUpdates" hash table. Whenever we pick the token set with the highest benefit from the priority queue, we check whether it is in the LazyUpdates hash table. If it is, we update its benefit and insert it into the priority queue, and then pop again from the priority queue. Since most of the impacted tokens never surface to the top of the priority queue, we reduce a significant number of unnecessary updates to the priority queue.

3.3.1 Candidate Tokensets & Statistics

We now discuss the generation of candidate token sets and the estimation of cardinalities of their docid sets.

In general, any subset of tokens of any entity $e \in E$ is a candidate token set. We restrict the space of candidate token sets to subsets of cardinality $\leq m$. In our experiments, we chose $m = 3$.

Recall that computing the benefit of a token set T requires us to estimate $|D(T)|$ and $|D(e)|$ for entities in $E(T)$. We take a sampling based approach. We draw a small uniform random sample (fraction p) of documents. For each candidate token set, we count the number of documents in the sample that contains the token set. We do the same for each entity string in E. We scale these counts by a factor $\frac{1}{p}$ to obtain the estimates. For any token set or entity which is not observed even once in the sample, we assign the estimates to be $\frac{0.5}{p}$. This is based on the *smoothing* technique commonly applied in Statistics [2] when counts are unknown because the sample may be too small. We compute the initial benefit for each candidate token set based on the above estimates. We only add candidate token sets with positive initial benefit to $\mathcal{T}_{cands}$.

Entities	Set of Signatures
Sony Vaio FS740	{Vaio, FS740}, {Sony, FS740}
Sony Vaio VX88P	{VX88P}
Sony Playstation 3	{Playstation, 3}
Microsoft XBox 360 Core System	{XBox, 360}
Microsoft XBox 360 Wireless Controller	{Xbox, Wireless, Controller}

Table 2: Entities and their corresponding signatures

4. APPROXIMATE MATCH

In the previous sections, we focused on identifying mentions in documents that exactly match with some entity string in the reference set. However, an entity in the reference set, say "Microsoft Xbox 360 Core system", may be mentioned in documents under other representations: "Microsoft Xbox 360" and "Xbox 360". It is important to recognize such approximate mentions in order to improve the accuracy of entity extraction [10]. Note that the set of tokens corresponding to an approximate mention (e.g., "Microsoft Xbox 360") in a document has high overlap with the set of tokens in the matching entity. We now extend our techniques to identify sub-strings (i.e., subsequences of tokens) in documents that have considerable overlap with some entity string in the reference set. In this paper, we focus on token based similarity and do not consider closeness due to edit or spelling errors, i.e., if Microsoft is incorrectly spelt as "microsft" we do not attempt to match these tokens. The reason is that inverted indexes over document collections only index documents on individual tokens, and hence do not support docid retrieval for sub-tokens. We now present a technique based on signatures that enables approximate match.

The problem of approximate string matching based on string similarity functions such as Jaccard similarity and edit distance has received significant interest (e.g., [16, 5]). These approaches efficiently identify pairs of strings which are closer than a specified similarity threshold. Their general approach is to generate a set of *"signatures"* for every string such that two strings have high similarity *only if* they match exactly on at least one of these signatures. These signatures are sets of tokens, usually a subset of those in the original string. Table 2 illustrates an example reference set and the signatures—according to a signature scheme called WtEnum [5] which we use in this paper. For example, the entity string "Sony Vaio 740" has two signatures, {Vaio, FS740} and {Sony, FS740}, while the entity string "Sony Vaio VX88P" has one signature {VX88P}.

WtEnum Signature Scheme: We illustrate the WtEnum signature scheme using the weighted token overlap similarity function which is fairly general and allows us to model other known similarity functions such as Jaccard similarity. The details of the WtEnum signature scheme can be found in [5].

Similarity: Let $w(t)$ denote the weight of a token t, and $w(T)$ denote the sum of weights of all tokens in T. The (unweighted) *token overlap similarity (ov)* between two strings s_1 and s_2 equals $|Tokens(s_1) \cap Tokens(s_2)|$, the size of the intersection between their token sets. The *weighted overlap similarity (wov)* between s_1 and s_2 is $w(Tokens(s1) \cap Tokens(s2))$.
Signatures: Let *sig* be a signature generator for the weighted overlap similarity function and θ be a threshold. Given a string s, $sig_\theta(s)$ returns a set of signatures such that if for two strings s_1 and s_2, $wov(s_1, s_2) > \theta$ then $sig_\theta(s_1) \cap sig_\theta(s_2)$ is non-empty. For example, assuming all tokens have weight 1, sig_2("Sony Vaio FS740") = {{Vaio, FS740}, {Sony, FS740}, {Sony, Vaio}}.
Identifying Approximate Mentions Using Signatures: We say a sub-string s in a document is an *approximate mention* of an entity string s' iff $wov(s, s') > \theta$. A sub-string in a document is an approximate mention of an entity string *only if* it contains all the tokens of at least one of the signatures of the entity [5]. For example, a substring is an approximate mention of the entity "Sony Vaio FS740" if and only if it either contains both 'Vaio' and 'FS740' or both 'Sony' and 'FS740'.

Recall the architecture shown in Figure 6. We can identify the filtered set F of docIds as follows. We build a *signature reference set* by taking each signature of each entity in the reference set, concatenating the tokens in the signature to form a string and adding it to the signature reference set. Once the signature reference set is built, we apply the techniques described in Section 3 against this signature reference set instead of on the entity reference set, i.e., execute a DNF formula that covers the signature reference set. Note that we concatenate the tokens in each signature to generate a string for each signature because our filtering techniques take as input a reference set of (tokenizable) strings. However, the order in which the tokens are concatenated does not matter as the DNF formula treats each reference string as a set of tokens and disregards the order of the tokens in it. For example, we can simply concatenate the set of tokens in each signature in $sig_\theta(e)$ in the order they appear in e with whitespace between tokens. We use $sig_\theta(e)$ to not only refer to the set of signatures of e but also the set of concatenated strings. For example, sig_2("Sony Vaio FS740") also denotes the set {"Vaio FS740", "Sony FS740", "Sony Vaio"} of strings generated from the signatures.

Definition 4. **(Signature Reference Set)** *Given a reference set E of entities, the signature reference set $sig_\theta(E) = \cup_{e \in E} sig_\theta(e)$ is the reference set obtained by adding the concatenated signatures for each entity in E.*

Note that we cannot use the phrase query approach here since the tokens in the concatenated signatures may not occur contiguously and in the same order in the documents containing the approximate mentions. For example, if we issue the phrase query "XBox wireless controller" corresponding to the signature {XBox, wireless, controller}, we will miss documents mentioning "Microsoft XBox 360 Wireless Controller". We can only issue intersection queries involving all tokens in a signature. Note that our algorithm executes phrase queries for uncovered entities; in the approximate match case, we execute intersection queries instead of the phrase queries.

Our experiments show that the number of strings in the signature reference set is not significantly larger than the number of reference entities. In our experiments, it is 2-3 times the number of entities in the reference set. Further, all signatures added to the reference set contain only a subset of tokens from the original entity. So, no new token sets are added to the signature reference set. At the same time, the overlap among tokens across the strings is significantly increased. Therefore, we anticipate that the number of documents in the result of the $dnf(EC)$ formula and the cost of executing the DNF query does not change significantly. In Section 6, we demonstrate that our approach (i) does not result in the signature set exploding significantly, and (ii) is very efficient.

After the entity cover filter identifies documents that may contain approximate mentions of reference entities, we need to apply a *approximate string matcher* which can identify such approximate mentions. Given a document d, this approximate string matcher returns all approximate mentions (according to the weighted overlap similarity and threshold θ) of entities in E from d. We denote the output by $g_E^{wov_\theta}(d)$. Such document processing procedures have been described in [8, 7], which we use in this paper.

Definition 5. *(Document Filtering with Approximate Match) Given a set D of documents and the set E of entities, the goal of document filtering with approximate match is to return $\{d : d \in D$ and $g_E^{wov_\theta}(d) \neq \phi\}$.*
Observation 2 (Correctness): Given a set D of documents and the set E of entities, let $D' \subseteq D$ be the filtered set of docIds returned by the entity-cover filtering techniques with respect to D and the signature reference set $sig_\theta(E)$. Then, $\{d : d \in D$ and $g_E^{wov_\theta}(d) \neq \phi\} = \{d : d \in D'$ and $g_E^{wov_\theta}(d) \neq \phi\}$.

In summary, our algorithm first generates a signature reference set and then applies the entity cover filter we described in Section 3 on that set. We then process each filtered document using $g_E^{wov_\theta}$.

5. EXTENSIONS

Choice of Document Filtering Strategy: We discussed three choices for the document filtering problem: (i) Phrase query approach, (ii) Entity Cover approach, and (iii) Document Scan approach. We can choose the best document filtering strategy using our cost model.
Phrase Query vs. Entity Cover: Note that our approach gracefully degenerates to the phrase query approach when we cannot find token sets that reduce the cost of the original DNF formula. In our algorithm (Figure 7), each entity in the set *Uncovered* will generate a phrase query. This situation arises when (i) the overlap of token sets among entities is not significant or (ii) the overlapping token sets are too expensive to use in that there are too many false positives.
Document Scan vs. Entity Cover: With respect to the choice between the document scan approach and our DNF formula approach, we use our cost model to decide between the two. If the estimated cost of our approach is higher than that for the document scan approach, one can choose a document scan. Figure 14 in Section 6 illustrates our predictions based on the cost model.

Exploiting Relational View of Inverted Index: As mentioned earlier, most inverted index engines only allow queries with a small number of tokens. But, the DNF formulas that we execute may have a large number of tokens. As discussed earlier, we can batch these queries in order to execute them against a typical inverted index engine. However, some of the inverted index engines (e.g., Microsoft SQL Server 2008 Integrated Full Text Search) expose a relational view of the inverted index. For instance, the relational view over the inverted index may be $II - RelationalView[Token, DocId, Pos]$. If this view supports efficient join with a potentially large set of tokens, we can execute a union of docid sets of a large number of single tokens efficiently (referred to as bulk union). For example, the following SQL query would get us the union of docid lists of a set of single tokens in a $TokenTable[Token]$.

```
SELECT distinct R.DocId
FROM II-RelationalView R, TokenTable S
WHERE R.Token = S.Token
```

We can exploit the above view by taking the single tokens in the entity cover and issuing the above join query. The remaining token sets in the entity cover are executed as batched DNF queries. In Section 6, we demonstrate empirically that our entity cover contains a large number of single tokens and hence the above modification can result in significant benefits.

6. EXPERIMENTAL EVALUATION

We now present the results of an extensive empirical study to evaluate the techniques described in this paper. The major findings of our study can be summarized as follows:

- **Effectiveness of entity cover approach**: The entity cover approach for document filtering is effective in improving the efficiency of ad-hoc entity extraction. In most cases, it is about 2 times faster than the phrase query approach and 1-2 orders of magnitude faster than the document scan approach.
- **Further improvement using relational view**: Exploiting the relational view further improves the performance of entity cover approach.
- **Ability to exploit overlap**: The entity cover approach is effective in exploiting overlap in tokens among entities: higher the degree of overlap, lower the cost of the entity cover approach.
- **Effectiveness of entity cover for approximate match**: The entity cover approach is effective in the case of approximate match as well and significantly outperforms the intersection query as well as document scan approaches.

All experiments reported in this section were conducted on an AMD x64 machine with two 1.99GHz AMD Opteron processors and 8GB RAM, running Windows 2003 Server (Enterprise x64 edition).

6.1 Experimental Methodology

Datasets and Pre-processing: We consider two document collections—the news articles collection and the web pages collection. The news collection consists of 955,571 news articles that appeared on the MSNBC news portal between 2003 and 2005. The web page collection consists of 26.9 million web pages obtained by crawling a small part of the web. We store the document collections as text columns in two separate tables in Microsoft SQL Server 2005 and build separate inverted indexes on them. To enable the bulk union API, we explicitly build the inverted index relation II[TokenId, DocId, Pos] and TokenTable[Token, TokenId] by using custom tools. We consider two reference entity sets: a product entity set consisting of 304,940 product names obtained from MSN Shopping Product catalog and a person entity set consisting of 2.04 million person names extracted from Wikipedia, IMDB, Encarta and DBLP data. We implement the entity string matcher g_E using the Aho Corasick pattern matching algorithm [3].

Implementation of Various Approaches and Cost Computation: We have implemented 4 approaches: document scan, phrase query, entity cover described in Section 3 and entity cover using the relational view described in Section 5.

Document Scan: The document scan approach simply runs the entity string matcher g_E on every document and outputs the docids of those documents $d \in D$ for which $g_E(d) \neq \emptyset$. The overall cost is the cost of running g_E on all documents.

Phrase Query: We partition the given set E of entities into batches of size B. For each batch $\{e_1, e_2, ..., e_{|B|}\}$ of entities, we issue the following phrase query:

```
SELECT DocId FROM DocTable
WHERE CONTAINS(DocText, 'phrQ(e_1) OR ... OR phrQ(e_|B|)')
```

where $phrQ(e_i)$ denotes the phrase corresponding to entity e_i. We write the results of each batch into a temporary relation and then take the union (by running select distinct query on the temporary relation). The overall cost is simply the cost of running the above query for all batches. Since the phrase query approach returns only hit documents, there is no additional entity string matching cost.

Entity Cover: We first construct the entity cover for the given set E of entities using the algorithm described in Section 3. We partition the tokensets in the cover into batches of size B. For each batch $\{T_1, T_2, ..., T_{|B|}\}$ of tokensets, we issue the following DNF query:

```
SELECT DocId FROM DocTable
WHERE CONTAINS(DocText, '(and(T_1)) OR ... OR
(and(T_|B|))')
```

We write the results of each batch into a temporary relation and then take the union. The overall cost is the sum of the costs of constructing the entity cover, that of running the above query for all batches and that of running the entity string matcher on the resulting documents.

Entity Cover using Relational View: We first construct the entity cover for the given set E of entities. We partition the tokensets in the entity cover into two parts: single token and multi-token tokensets. For the first part, we insert the single tokens into a single column temporary relation $TokenTable$ and obtain the union of docids lists by issuing the SQL query in Section 5. For the second part, we follow the batched execution discussed above. Finally, we take the union of the two results.

Approximate Match: For approximate match, we build the signature reference set using the WtEnum signature scheme [5]. We construct the entity cover on the signatures and issue batched DNF queries as discussed above.

To measure the query times accurately, we always start with a clean database cache and buffers. We use $B = 50$ as that was the maximum size of the DNF query typically allowed.

6.2 System Calibration

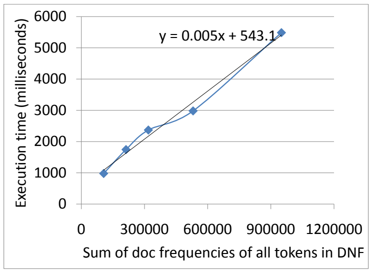


Figure 8: System Calibration for news dataset

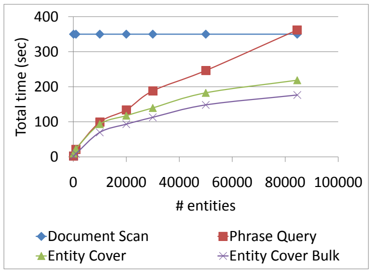


Figure 9: Execution times of product name extraction task on news data

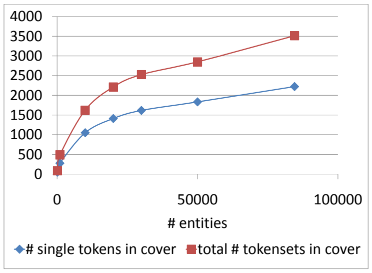


Figure 10: Number of single tokens in entity cover

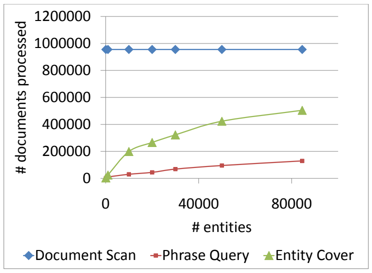


Figure 11: Number of documents processed for product name extraction on news data

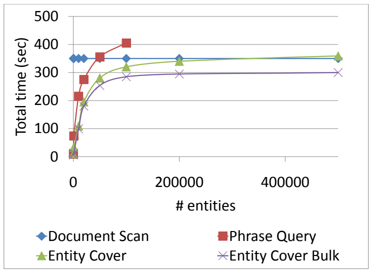


Figure 12: Execution times of people name extraction task on news data

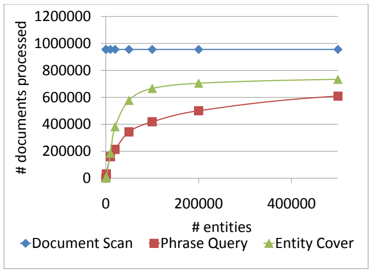


Figure 13: Number of documents processed for people name extraction on news data

Our entity cover technique relies on 3 system constants: (1) the average cost C_{doc} of running entity string matcher on a document, (2) the average cost C_{idx} of scanning an element in a document list in the inverted index and (3) the initialization cost C_{ini} of a executing a DNF query on the inverted index. We calibrate the system for each document collection and entity set to obtain the values of these constants. To obtain C_{idx} and C_{ini}, we execute DNF queries of various sizes on the inverted index and plot the overall execution time of each query against the sum $\sum |D(t)|$ of document frequencies of each token t in the DNF formula as shown in Figure 8. We then fit the above data to the best straight line using least-squares fitting. The slope of this line is C_{idx} and the y-intercept is C_{ini}. Figure 8 shows the result of the calibration for product entities over news data where C_{idx} turns out to be 0.005 milliseconds and C_{ini} to be 543.1 milliseconds. We obtain C_{doc} by processing a small set of documents and dividing the overall time by the number of documents processed. C_{doc} is 0.35 ms for the news documents and 3.52 ms for the web data. This is consistent with the fact that the documents in the web collection are on average 10 times larger compared to those in the news collection.

6.3 Experimental Results

Comparison with document scan and phrase query approach: Figure 9 shows the overall execution times of the 3 approaches for the product name extraction task on the news collection for various sizes of the reference entity set. The entity cover approach is about 2 times faster than the phrase query approach. This is because the size of the DNF query executed for entity cover is 1-2 orders of magnitude smaller than that for phrase query as shown in Figure 10. For example, the DNF query for entity cover of 84000 entities contains 3513 tokensets (out of which 2222 are single tokens) while the phrase query contains all the 84000 entities. Since a large fraction of the tokensets are single tokens, we can exploit the faster bulk union API for the single token part of the query. This results in further improvement of performance of the entity cover approach as shown in Figure 9. The entity cover approach also outperforms the document scan by 1-2 orders of magnitude. This is because the entity cover approach processes much fewer documents compared to the latter as shown in Figure 11. For example, entity cover processes 5 times fewer documents than document scan approach for entity set size 10K. In summary, while document scan incurs a high document processing cost and phrase query incurs a high query cost, the entity cover achieves a good tradeoff between the two costs and hence performs better that those approaches.

We also executed the people extraction task on the news data collection. Figures 12 and 13 show the overall execution times and number of documents processed by the 3 approaches. The entity cover approach again outperforms the the phrase query approach. When compared to document scan, the entity cover approach is faster for entity sets of size below 350K. For very large entity sets (> 350K), document scan is faster than entity cover approach. This is because the entity cover approach ends ups processing about 70% of the documents for these entity sizes. Hence, the savings in document processing is too low to offset the query cost at such sizes of the entity set. This is a consequence

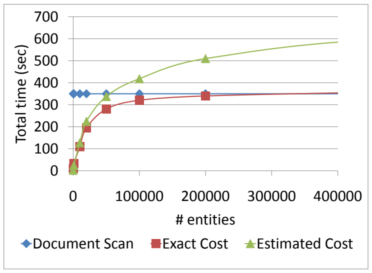


Figure 14: Cost estimation

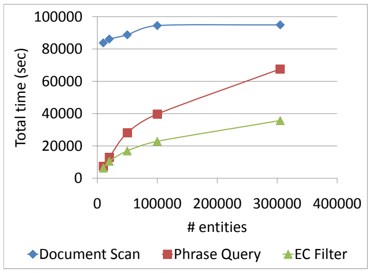


Figure 15: Execution times of product name extraction task on web data

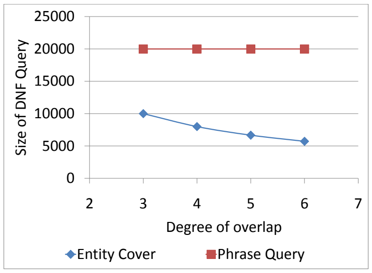


Figure 16: Size of DNF query for different degrees of overlap

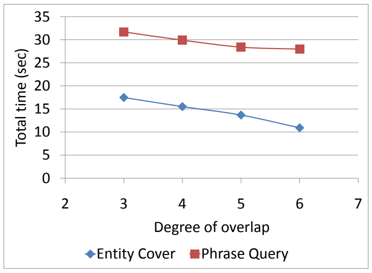


Figure 17: Execution cost for different degrees of overlap

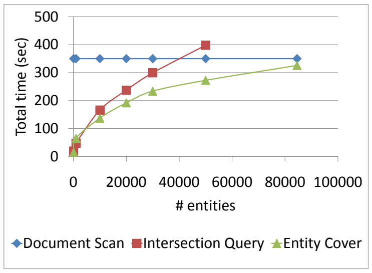


Figure 18: Execution times of approximate match on news data

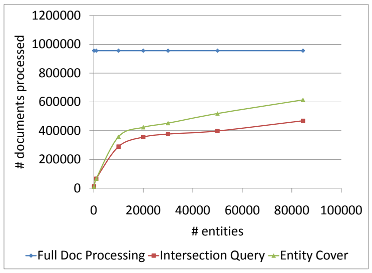


Figure 19: Number of document processed for approximate match on news data

of the fact that news documents are quite dense in terms of people names. For example, more that 50% of the documents mention entities from the people reference set. This sets an upper bound on the filtering factor and hence restricts the benefit of the filtering approaches. In such cases, we can use the cost model to estimate the costs of the two approaches and choose the one with the lower estimated cost. Figure 14 shows the estimated as well as true costs of the two approaches. For document scan, the estimated cost ($C_{doc} * |D|$ where D is the document collection) is typically almost identical to the true cost. For entity cover approach, the estimated cost is quite accurate for smaller entity sizes but tends to overestimate for larger document sizes. This is due to estimation of union size $|\cup_i D(T_i)$ by $|\sum_i D(T_i)$ as discussed in Section 3.1. This gap can be reduced by using better estimators of union size.

Figure 15 shows the overall execution times of the 3 approaches for the product name extraction task on the much larger web page collection. The entity cover approach is about 2 times faster than the phrase query approach for this dataset as well. The entity cover approach also outperforms document scan by 1-2 orders of magnitude as the former processes much fewer documents compared to the latter. The gap between document scan and the other approaches is wider in this dataset compared to the news dataset due to the higher cost of processing a web page (C_{doc} is 3.52 ms for web pages vs 0.35 ms for news documents).

Sensitivity of entity cover approach to overlap: In order to test the ability of entity cover approach to exploit overlap among entities, we need entity sets with varying degrees of overlap. We synthetically generate such entity sets as follows. We first generate entities of a fixed length l with no overlap by picking l distinct tokens from the documents and concatenating them. We then generate m variants of each such entity e by picking subsets of tokens from $Tokens(e)$ having Jaccard similarity with e above a certain threshold θ. We vary the degree of overlap among entities by varying m. We use $l = 6$ and $\theta = 0.8$ for this experiment. We fixed the size of entity set comprising of all the variants to 20000. Figure 16 shows the size of the DNF query executed by entity cover and phrase query approaches for various values of m. As the overlap increases, the DNF query for the entity cover becomes smaller as the entity cover approach picks token sets that cover more entities. On other other hand, the size of the phrase query is always $|E|$=20000. Figure 17 shows the execution cost of the two approaches for varying degrees of overlap. The cost of entity cover approach decreases with increasing overlap due to the reduced query cost. The cost of the phrase query approach is relatively flatter but also shows a small decrease with increasing overlap due to caching effects.

Effectiveness of entity cover approach for approximate match: Figure 18 shows the execution times of entity cover, intersection query and document scan approaches for approximate match. Note that we cannot use phrase queries for this approach as discussed in Section 4. Instead, we use intersection queries involving all the tokens in each signature. The entity cover approach significantly outperforms the intersection query and document scan approaches in this scenario as well. Figure 19 shows the number of documents processed by each of the approaches. The number of documents scanned by the entity cover approach is higher than in the exact match case but is still 2-3 times fewer compared to the document scan approach.

7. RELATED WORK

Entity extraction has been an area of extensive research. Commercial software is available from companies such as Verity, Inxight (which target enterprise document analytics) to assist the extraction of entities such as people and products from documents (e.g., [4]). There are two basic approaches to this problem, namely, linguistic grammar-based and machine learning (ML)-based approaches [4, 12, 15, 18]. While both approaches depend on lexicons containing common first and last names for people, common brand names for products, etc., they typically do not constrain the extracted entities to a (potentially large) reference sets of entities. The FASTUS system [14] focuses on a dictionary constrained scenario. However, they adopt a document scan approach and do not try to exploit the inverted index.

Cohen and Sarawagi showed that exploiting reference entity sets can improve the accuracy of entity extraction [10]. They incorporate similarity (according to a string similarity function) of a substring of a document to a reference entity as a feature in the entity classifier. Recently, Chandel et al. proposed an algorithm for sharing string similarity computation across overlapping substrings and integrating it deeply with a specific entity classification algorithm [8]. Since the reference entity set is used as an additional feature of the classifier and not to constrain the extracted entities, they still need to process all the documents.

An alternative approach to avoid processing irrelevant documents would be to very efficiently classify whether or not each document is relevant based on its content [17]. That is, whether a document is likely to have any entity in the target set. However, most of these techniques still need to process the content of a document for classification thus defeating the purpose of avoiding the documents to be processed for entity string matching in the first place. Second, these classification techniques can still make several errors. They may misclassify documents as not relevant and hence may miss several documents. In any case, they may still be used in conjunction with the techniques we developed in this paper, say, by further applying these classification filters on the filtered documents we return.

Another proposal that avoids processing all documents is to use rule-extraction algorithms to derive queries from the classifier that recognizes the entities (e.g., [1, 13]). Subsequently, these queries are issued against the database or a search engine to retrieve promising documents. Etzioni et al. also propose to query search engines for extracting information from the web [6, 11]. None of these techniques exploit the overlap among the queries, nor do they consider "hybrid" strategies that try to trade-off between query and document processing costs. Their techniques if adapted to our setting would correspond to the phrase query approach. [13] proposes a cost model to choose between the document scan and querying strategies but does not consider hybrid strategies. Furthermore, their cost model assumes a fixed cost for all queries which is inadequate for our scenario.

Factorization of complex boolean formulas has been explored in the context of query processing (e.g., [9]). These approaches focus on exact rewriting, i.e., the results are identical. In contrast, our approaches in this paper focus on deriving DNF formulas which obtain a superset of results obtained by using the original formula.

8. CONCLUSIONS

In this paper, we considered the problem of ad-hoc entity extraction from indexed document collections. Our main observation is that in many scenarios, there exists a significant overlap of tokens among entities. We exploit this observation to develop techniques to efficiently identify a set of documents which need to be processed for entity extraction. Through an extensive empirical evaluation using real datasets, we demonstrated that our techniques result in significant improvements over prior approaches.

9. REFERENCES

[1] E. Agichtein and L. Gravano. Querying text databases for efficient information extraction. In *Proceedings of ICDE Conference*, 2003.

[2] A. Agresti. *An introduction to categorical data analysis*. Wiley, 2007.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, June 1975.

[4] D. E. Appelt and D. Israel. Introduction to Information Extraction Technology. *IJCAI-99 Tutorial*, 1999.

[5] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of VLDB*, Seoul, South Korea, September 2006.

[6] M. J. Cafarella and O. Etzioni. A Search Engine for Natural Language Applications. In *WWW Conference*, 2005.

[7] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *Proceedings of ACM SIGMOD*, 2008.

[8] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *Proceedings of ICDE*, Los Alamitos, 2006.

[9] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *Proceedings of ACM SIGMOD*, June 2003.

[10] W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *In Proceedings of ACM SIGKDD*, 2004.

[11] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Web-scale information extraction in KnowItAll. In *In Proceedings of WWW*, 2004.

[12] R. Grishman. Information Extraction: Techniques and Challenges. In *SCIE*, 1997.

[13] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To Search or to Crawl?: towards a Query Optimizer for Text-Centric Tasks. In *SIGMOD*, pages 265–276, 2006.

[14] H. Jerry, R. Douglas, E. Appelt, J. Bear, D. Israel, M. Kameyama, M. Stickel, and M. Tyson. Fastus: A cascaded finite-state transducer for extracting information from natural-language text, 1996.

[15] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: probabilistic models for segmenting and labeling sequence data. In *Proceedings of ICML*, 2001.

[16] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of ACM SIGMOD*, pages 743–754, 2004.

[17] F. Sebastiani. Machine Learning in Automated Text Categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[18] I. Witten, A. Moffat, and T. Bell. *Managing gigabytes: Compressing and indexing documents and images*. Morgan Kaufmann, 1999.

Reference Entities	$\|D(e)\|$
Sony Vaio FS740	100
Sony Vaio VX88P	50
Sony Playstation 3	150

Table 3: Reference entities and their docid set sizes

Token sets	$\|D(T)\|$	Initial Benefit	Benefit after iteration 1
Sony	1000	185	-585
Vaio	500	-80	-80
FS740	100	150	150
VX88P	50	150	150
Playstation	600	150	-
3	5000	-4690	-
Sony, Vaio	150	170	170
Vaio, FS740	100	100	100
Sony, FS740	100	50	50
Vaio, VX88P	50	100	100
Sony, VX88P	50	50	50
Sony, Playstation	300	350	-
Playstation, 3	200	-390	-
Sony, 3	600	50	-

Table 4: Token sets and benefit computation

APPENDIX

A. EXAMPLE ILLUSTRATING GREEDY ALGORITHM

Consider the reference entities in Table 3. Suppose we only consider token sets of sizes 1 and 2. The candidate token sets and their $|D(T)|$ are shown in Table 4. Suppose $B = 2$. Suppose C_{idx}, C_{ini} and C_{doc} are 0.1, 10 and 1 respectively. The initial benefits are shown in the third column of Table 3. For example, the initial benefit of the token set {'Sony'} is computed as follows. $E_i('Sony')$ is the set of all three entities. The $\sum_{t \in Tokens(e)} |D(t)|$ for these entities are 1600, 1550 and 6600 respectively ($|D(t)|$ for the tokens are shown in Table 4). So, the initial benefit of the token set Sony is ((1600+1550+6600)-1000)*0.1 + $\frac{(3-1)}{2}$*10 - (1000-300)*1 = 185.

In the first iteration, we pick the tokenset {Sony, Playstation} as that has the maximum benefit. $E_i(T)$ for the above tokenset consists of only "Sony PlayStation3", so the impacted tokensets are {Sony}, {Playstation}, {3}, {Sony, Playstation}, {Playstation, 3} and {Sony, 3}. For example, we reduce the benefit of {Sony} by 150*1 + 6600*0.1 + 5 = 815 to -630. The updated benefits are shown in the fourth column in Table 4. Token sets with empty $E_i(T)$ are shown with '-' and are not considered in subsequent iterations. In the second iteration, we pick the tokenset {Sony, Vaio} as that has the maximum benefit. At this point, the algorithm terminates since all entities are covered. So, the entity cover produced is {{Sony, Playstation}, {Sony, Vaio}}.

B. MEMORY REQUIREMENT AND COMPLEXITY ANALYSIS OF GREEDY ALGORITHM

Memory Requirement: The algorithm maintains four data structures in memory: the priority queue that contains at most $|\mathcal{T}_{cands}|$ token sets (the identifier and the benefit value of each tokenset), the LazyUpdates hash table that contains at most $|\mathcal{T}_{cands}|$ token sets (the identifier and the benefit value of each tokenset) and the two hash tables that store the tokenset to $E(T)$ associations and the entity to $TokenSet(e)$ associations respectively (identifier pairs only). Each of the latter two hash tables store $\sum_{e \in E} |TokenSet(e)|$ associations. Since $\sum_{e \in E} |TokenSet(e)| >> |\mathcal{T}_{cands}|$, the overall memory requirement is $\sum_{e \in E} |TokenSet(e)|$. If we only consider token sets up to size m, the memory requirement comes to $\sum_{e \in E}(|Tokens(e)| + \binom{|Tokens(e)|}{2} + \ldots + \binom{|Tokens(e)|}{m})$. In our experiments, we use $m = 3$.

Time Complexity: Initially, the algorithm inserts $\mathcal{T}_{cands}$ token sets and their benefits into the priority queue. Assuming a Fibonacci heap implementation of the priority queue, inserts are $O(1)$ operations. The algorithm performs worst-case $|EC| \cdot \mathcal{T}_{cands}$ updates to the LazyUpdates hash table which are also O(1) operations. The algorithm performs at least $|EC|$ pop operations on the priority queue which are $O(\log n)$ amortized time, resulting in complexity $|EC| \cdot O(\log |\mathcal{T}_{cands}|)$. In the worst case, the number of pop operations is $|EC| \cdot \mathcal{T}_{cands}$. However, in practice, the time complexity of the algorithm is tends to be $|EC| \cdot \mathcal{T}_{cands}$.