Mining Document Collections to Facilitate Accurate Approximate Entity Matching

Surajit Chaudhuri Venkatesh Ganti Dong Xin

Microsoft Research
Redmond, WA 98052
{surajitc, vganti, dongxin}@microsoft.com

ABSTRACT

Many entity extraction techniques leverage large reference entity tables to identify entities in documents. Often, an entity is referenced in document collections differently from that in the reference entity tables. Therefore, we study the problem of determining whether or not a substring "approximately" matches with a reference entity. Similarity measures which exploit the correlation between candidate substrings and reference entities across a large number of documents are known to be more robust than traditional stand alone string-based similarity functions. However, such an approach has significant efficiency challenges. In this paper, we adopt a new architecture and propose new techniques to address these efficiency challenges. We mine document collections and expand a given reference entity table with variations of each of its entities. Thus, the problem of approximately matching an input string against reference entities reduces to that of exact match against the expanded reference table, which can be implemented efficiently. In an extensive experimental evaluation, we demonstrate the accuracy and scalability of our techniques.

1. INTRODUCTION

Entity extraction is important for several applications and hence received significant attention in the literature [13, 1, 18, 16]. A typical application is the *product analytics and reporting* system which periodically obtains many review articles (*e.g.*, feeds from review websites), identifies mentions of reference product names in those articles, and analyzes user sentiment of products. Such reports are very relevant for comparison shopping web sites which provide consumers with reviews, consumer sentiment and offers for each product. Typically, these systems maintain a reference list of products (*e.g.*, products from certain manufacturers or categories), and require effective and efficient identification of mentions of reference product names in the review articles.

Another application is to identify entities in queries submitted to search engines. If the search engine identifies the entity and matches it with that in the reference table, then richer information about the entity can be directly surfaced "in-line" to the user's query.

Both the above applications exploit the existence of extensive *reference entity databases*. The core *entity matching* task they rely upon is to efficiently check whether or not a *candidate substring* (a substring from a review article, or a search query) matches an entity member in a reference table.

In many realistic scenarios, say for extracting product names, expecting that a substring in a review article matches exactly with an entry in a reference table is very limiting. For example, consider the set of product entities in Table 1. In many documents, users may just refer to e_1 by writing "Canon XTI" and to e_3 by writing "Vaio F150". Insisting that substrings in review articles match exactly with entity names in the reference table may likely cause these product mentions to be missed. Therefore, it is important to consider approximate matches between candidate strings and entity names in a reference table [13, 9].

ID	Entity Name
e_1	Canon EOS Digital Rebel XTI SLR Camera
e_2	Lenovo ThinkPad X61 Notebook
e_3	Sony Vaio F150 Laptop

Table 1: Example Entities

Many previous approaches for enabling approximate match rely on string similarity functions which measure similarity by considering the information only from the input candidate string and the target entity string that it could match with.[1] For example, the (unweighted) Jaccard similarity function comparing a candidate string "Canon XTI" and the entity name of e_1 "Canon EOS Digital Rebel XTI SLR Camera" would observe that two out of seven distinct tokens (using a typical white space delimited tokenizer) are common between the two strings and thus measures similarity to be quite low at $\frac{2}{7}$. However, from the common knowledge, we all know that "Canon XTI" does indeed refer to e_1. This is because "Canon XTI" is strongly correlated to other tokens in e_1. The string similarity does not capture such correlation between tokens.

In order to overcome the above limitation, many techniques (such as those based on co-occurrence or associa-

[1]These similarity functions also use token weights, say IDF weights, which may in turn depend on token frequencies in a corpus or a reference table.

VLDB '09, August 24-28, 2009, Lyon, France

tion) in natural language processing measure the correlation, say mutual information, between a variation and an entity across a large collection of documents [21, 20, 14]. For instance, many documents which contain the tokens "Canon XTI" also mention the remaining tokens in e_1. This provides a strong evidence that "Canon XTI" matches with e_1. However, computing the correlation measure between a document substring and an entity in the reference table requires us to scan the entire corpus. Therefore, it is not at all feasible to measure correlation at the time of matching document substrings with entities.

In this paper, we therefore adopt an alternative approach to identify *variations* of entities in reference tables [12]. We develop techniques to mine variations from the document collections (e.g., a large collection of Wikipedia pages or web corpus). This allows us to expand a given reference entity table with variations of each of the entities in the reference table. Thus, the problem of approximately matching an input string against reference entities reduces to that of exact match against the expanded reference table, which can be implemented efficiently [2]. Observe that the identified variations can also be used for approximately matching records in the context of data cleaning [6].

For an entity e, we focus on identifying variations which consist of a subset of the tokens in e. There are two key justifications for focusing on this subclass [12]. First, the reference entity tables are provided by authoritative sources; hence, each entity name contains not only the most important tokens required to identify an entity exactly but may also contain redundant tokens which are not required for identifying the entity. Therefore, it is sufficient to isolate the identifying subset of tokens for each entity as a variation. Second, in the entity extraction application, documents are mainly drawn from the web such as blogs, forums, reviews, search queries, where it is often observed that users like to represent a long entity name by a subset of tokens (*e.g.*, 1-3 keywords). In this paper, we call the set of tokens of an entity e, which refers to e, as an *identifying token set* (or, *IDTokenSet*) of e.

For our approach, it is natural to ask two questions. First, by expanding the reference table with variations, are we likely to add a large number of IDTokenSets? This question has significance to the scalability of exact match against the expanded reference table. Secondly, how do we efficiently generate these variations? For the first question, we observe that although the number of token subsets of entities is large, the number of variations that *appear contiguously in some documents* is fairly small. For instance, the average number of IDTokenSets over $70TB$ of web pages per product entity is less than 3. Therefore, matching against the expanded reference table is going to be very efficient.

For the second question, we propose an architecture which scales to a very large number of documents and reference entities. Our architecture consists of three phases as shown in Figure 1. The first *candidate IDTokenSets generation* phase identifies (τ, e) pairs, where τ is a candidate IDTokenSet and e is an entity. The second *per-doc score computation* phase measures the correlation between τ and the entity e it might identify across each document containing τ. The third *score aggregation and thresholding* phase aggregates the correlation between τ and e across all documents. Finally, we output all candidates whose aggregated correlation w.r.t. their target entities is higher than a given threshold.

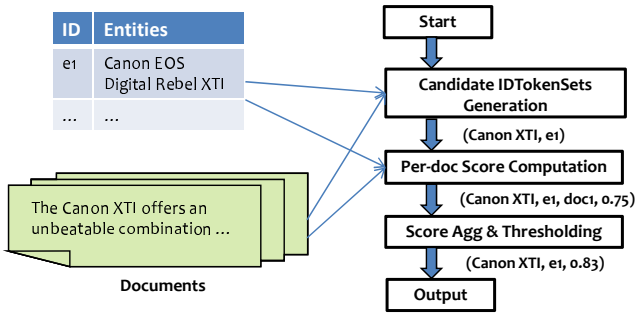

Figure 1: Architectural Overview

The candidate IDTokenSets generation phase is the most computationally challenging problem. It essentially involves a *Subsequence-Subset Join* problem, where the goal is to identify all sub-strings in documents whose token sets are a subset of some entity in the reference table. Adaptations of current approaches for addressing this problem are prohibitively expensive because of the following reasons. First, checking whether or not the token set of a document substring is a subset of an entity (denoted *subset-check*) in a large reference entity set typically is not cheap [10]. Second, there are a large number of documents, and each substring in a document is a possible candidate for IDTokenSets. Therefore, a straight-forward approach would perform a huge number of subset checks, and hence will not be efficient.

In this paper, we develop efficient techniques to solve the *Subsequence-Subset Join* problem. Our main insight is to "efficiently and effectively shrink" documents by identifying relevant substrings in each document so that we are now only required to process small documents to identify candidate IDTokenSets. Our main idea here is to develop a compact in-memory filter and a very efficient algorithm leveraging this filter to quickly and accurately (with no false negatives) prune out a large number of document substrings. While processing the surviving shrinked documents, we observe that many token sets occur commonly across multiple document substrings. We develop an optimal algorithm which exploits this commonality across the entire document collection to reduce (by orders of magnitude) the number of subset checks. Thus, we are able to process a large collection of documents to efficiently identify candidate IDTokenSets. In the score computation and aggregation phases, we adapt existing techniques. As illustrated by our experiments in Section 5, our techniques result in significant improvements (by orders of magnitude) over baseline techniques.

We also demonstrate that our architecture and techniques can be easily extended to a map-reduce infrastructure [15]. These extensions enable us to handle very large document collections such as a snapshot of the web. In fact, many of our experiments (reported in Section 5) are performed over a large subset (of size 70TB) of web documents in a map-reduce infrastructure [8].

The remainder of the paper is organized as follows. In Section 2, we define the IDTokenSets problem. In Section 3, we describe our techniques for solving the subsequence-subset join problem, and in Section 4, we discuss the complete architecture for solving the IDTokenSet problem. In Section 5, we present a thorough experimental evaluation. In Section 6, we discuss related work. We conclude in Section 7.

2. PROBLEM DEFINITION

We first define the notation used in the paper, and then formalize the problem of detecting IDTokenSets. We then outline the architecture to generate IDTokenSets.

Let $\mathcal{E}$ denote the set of entities in a reference table. For each $e \in \mathcal{E}$, let $Tok(e)$ denote the set of tokens in e. For simplicity, we loosely use e to denote $Tok(e)$. We say that an entity e *contains* a token set τ if $\tau \subseteq Tok(e)$. The *frequency* of a token set τ with respect to $\mathcal{E}$ is the number of entities in $\mathcal{E}$ which contain τ. We use the notation τ_e to denote a subset of tokens of e. That is, $\tau_e \subseteq e$.

Intuitively, if a subset τ_e identifies e, then (i) τ almost uniquely identifies e in $\mathcal{E}$ and (ii) a large fraction, say θ, of documents mentioning τ_e should contain the remaining tokens in $e - \tau_e$.

In the ideal case if τ_e identifies e in $\mathcal{E}$, then e is the only entity which contains τ_e. Otherwise, it may be ambiguous as to which entity in $\mathcal{E}$ it refers to. However, typical reference sets may contain a very "close" set of entities. For example, product reference tables often contain many different entities such as "apple ipod nano", "apple ipod nano pink", "apple ipod nano black", etc. In these cases, an application might want to recognize "ipod nano" as an IDTokenSet for "apple ipod nano". To enable such applications, we relax the constraint that only e contain τ_e to allow a few entities in $\mathcal{E}$ containing τ_e. Informally, a token subset is a *discriminating token set* (*i.e.*, DisTokenSet) of an entity with respect to $\mathcal{E}$ if it is contained by very few entities in $\mathcal{E}$.

DEFINITION 1. **(DisTokenSet)** *Given a non-negative integer K, we say that τ_e is a* K-DisTokenSet *of $e \in \mathcal{E}$ if τ_e's frequency with respect to $\mathcal{E}$ is at most K, and τ_e is not a member of a set of given stop token sets. Since K is always implicit, we loosely use DisTokenSet to denote a K-DisTokenSet.*

Consider the correlation between τ_e and e in documents. In the ideal case if τ_e identifies e, a large fraction of documents mention tokens in $e - \tau_e$ next to the mention of τ_e. However, requiring adjacent mentions may be too constraining. Hence, we relax this notion in two ways. First, we want to parameterize the context window size p within which we expect to observe all tokens in $e - \tau_e$. Second, it may be good enough to find a significant fraction of tokens in $e - \tau_e$ within a small window of the mention. We first define the notion of a document *mentioning* a token subset, and that of *context window*. The similar notions are also defined in [12]. The difference between this paper and [12] is discussed in Section 7.

DEFINITION 2. *Let d be a document and τ_e be a set of tokens. We say that d* mentions *τ_e if there exists a substring s of d such that $Tok(s) = \tau_e$.*

DEFINITION 3. *(**p-window context**) Let $M = \{m\}$ be a set of mentions of τ_e in a document $d = t_1, \ldots, t_n$. For each mention m, let $c(m, p)$ be the sequence of tokens by including (at most) p tokens before and after m. The p-window context of τ_e in d is $C(\tau_e, d, p) = \bigcup_{m \in M} c(m, p)$.*

EXAMPLE 1. *The documents d_2 and d_3 in Table 2 mention the subset {ThinkPad, X61}. The 1-window context of "Thinkpad" in document d_2 of Table 2 is {Lenovo, ThinkPad, X61} and that of "Lenovo" is {Lenovo, ThinkPad}.*

ID	Document
d_1	Lenovo ThinkPad X Series...X61 Specs...
d_2	Lenovo ThinkPad X61 is...business notebook...
d_3	Lenovo ThinkPad X61 notebook review...

Table 2: A set of documents

We now define the stricter notion of evidence g_1 of a document referring to an entity e, where all tokens in $e - \tau_e$ are required to be present in the p-window context of τ_e.

$$g_1(\tau_e, e, d) = \begin{cases} 1 & if\ e \subseteq C(\tau_e, d, p) \\ 0 & otherwise \end{cases} \quad (1)$$

We now define the relaxed notion of evidence g_2 of a document referring to an entity e, which is quantified by the fraction of tokens in $e - \tau_e$ that are present in the p-window context of τ_e.

$$g_2(\tau_e, e, d) = \frac{\sum_{t \in C(\tau_e, d, p) \cap e} w(t)}{\sum_{t \in e} w(t)} \quad (2)$$

where $w(t)$ is the weight (*e.g.*, IDF weight) of the token t.

We are now ready to define the aggregated *correlation* between τ_e and e with respect to a document set $\mathcal{D}$. Informally, the aggregated correlation is the aggregated evidence that τ_e refers to e from all documents mentioning τ_e.

DEFINITION 4. **(Correlation)** *Given e, τ_e, a set of documents $\mathcal{D}$, we define the aggregated correlation $corr(\tau_e, e, \mathcal{D})$ as follows.*

$$corr(\tau_e, e, \mathcal{D}) = \frac{\sum_{d \in \mathcal{D}, d\ mentions\ \tau_e} g(\tau_e, e, d)}{|\{d | d \in \mathcal{D}, d\ mentions\ \tau_e\}|}$$

EXAMPLE 2. *In Table 2, each document completely contains {X61}. In order to validate whether $\tau_{e_2} = \{X61\}$ is an IDTokenSet of e_2, we compute $g_1(\tau_{e_2}, e_2, d_1) = 0$, $g_1(\tau_{e_2}, e_2, d_2) = 1$, $g_1(\tau_{e_2}, e_2, d_3) = 1$. Thus, $corr(\tau_{e_2}, e_2, \mathcal{D}) = \frac{2}{3} = 0.667$. Assume the token weights[2] are Lenovo(8.4), ThinkPad(9.8), X61(8.3) and Notebook(6.8). Using g_2, we have $corr(\tau_{e_2}, e_2, \mathcal{D}) = \frac{2.708}{3} = 0.903$.*

Observe that several measures, notably that of mutual information, have been considered for quantifying similarity between words across documents [20, 14]. We adopt the above measures (g_1 and g_2) instead of the popular mutual information measure because our scenario is inherently "asymmetric" (*i.e.*, τ_e is a subset of e). A detailed discussion can be found in Section 6.

The goal of the IDTokenSets problem is to generate all IDTokenSets of all entities in a reference set with respect to a document collection $\mathcal{D}$.

DEFINITION 5. *(**IDTokenSets Problem**) Given a reference entity set $\mathcal{E}$, a set of documents $\mathcal{D}$, a non-negative integer K, and the correlation threshold θ, the IDTokenSets problem is to identify the set $S_{\mathcal{E}}$ of all IDTokenSets such that for each $\tau_e \in S_{\mathcal{E}}$, τ_e is a K-DisTokenSet of e with respect to $\mathcal{E}$ and $corr(\tau_e, e, \mathcal{D}) \geq \theta$.*

[2]These are IDF weights computed from a collection of $70TB$ web documents

2.1 Architecture

We outline the complete architecture to generate IDTokenSets. As shown in Figure 1, our architecture consists of three phases. The first *candidate IDTokenSet generation* (**CIG**) phase generates the (candidate IDTokenSet, entityID) pairs. Specifically, a candidate IDTokenSet is a substring of some documents, and also a DisTokenSet of the entity. The second *per-document score computation* (**PSC**) phase computes the per-document correlation (using either g_1 or g_2) between each candidate IDTokenSet and the entity it may identify for each of the documents. The third *score aggregation and thresholding* (**SAT**) phase aggregates these scores, and outputs the candidate IDTokenSets whose score is above the given threshold. We summarize each phase and its output format in Table 3.

Phases	Output Format
CIG	(candidate IDTokenSet, entityID)
PSC	(candidate IDTokenSet, entityID, docID, score)
SAT	(IDTokenSet, entityID, score)

Table 3: Three Phases to Compute IDTokenSets

3. SUBSEQUENCE-SUBSET JOIN

As we discussed in Section 1, the most challenging phase in our architecture is the first phase of generating (candidate IDTokenSet, entity) pairs. It is essentially a *Subsequence-Subset Join* problem. In this section, we discuss our solutions for this problem. The complete procedure for generating IDTokenSets will be discussed in the next section.

3.1 The Baseline Solutions

We first discuss the limitations of baseline solutions before describing our techniques. The baseline solution to the *Subsequence-Subset Join* problem consists of two operations: *substring extraction* and *subset check*. The former extracts substrings documents, and the latter checks whether or not the extracted substring is a DisTokenSet of some entities.

Substring extraction: Suppose the maximum entity length (*i.e.*, the number of tokens) in the reference table is S. Given a document d, all substrings with length up to S are possible candidates to be subsets of some reference entities. Therefore, there will be roughly $|d| \times S$ (without considering the boundary effect) substrings to be subset-checked.

Subset check: We now describe two baseline techniques to subset-check each document substring. The first approach is to pre-compute all DisTokenSets of reference entities, and index (DisTokenSet, entity) pairs. Given a substring, we check whether it is in the index. This technique would enable us to perform subset checks efficiently. However, the number of DisTokenSets can be extremely large and the amount of main memory required to keep them all is excessive. Therefore, we have to keep all DisTokenSets on disk. This approach will make $|d| \times S$ disk accesses per document, and is clearly not efficient.

An alternative approach is to build an inverted index over the reference entities, by constructing a sorted list of entity identifiers (called *idlists*) for each distinct token, and then to identify whether or not a document substring matches with a DisTokenSet, by intersecting idlists [10]. In this approach, the number of disk accesses will be reduced to $|d|$. However, this approach would require intersection of idlists for all tokens in each document substring. Since the number of substring is very large, this approach is also inefficient.

3.2 Overview of Our Solution

We exploit two observations. First, we propose to very efficiently filter irrelevant document substrings. Second, we batch subset-checks. Here, we give an overview of our techniques. The detailed techniques are discussed in the remainder of this section.

Filtering document substrings: The straight-forward substring extraction would generate $|d| \times S$ candidate substrings. Note that many such document substrings are not DisTokenSets, or even not subsets of some reference entities. Therefore, it is important to quickly prune substrings which cannot be a DisTokenSet. In order to achieve this, we propose an efficient in-memory filtering structure and an algorithm based on the notion of *core token sets*. The properties of our filtering mechanism are as follows.

1. Linear Time Document Processing: We often have to process millions of documents to identify DisTokenSets. One observation from our experiments (Table 7) is that efficient document processing is critical, because even with our optimizations, it constitutes a significant portion of the overall time for identifying IDTokenSets. The computational complexity of our filter is $C \cdot |d|$ in processing a document d, where C is a small positive constant;

2. High Selectivity: The cost of subset-check (*e.g.*, the number of disk accesses) is proportional to the number of document substrings surviving the filter. Therefore, we aim to significantly reduce the number, as well as their length, of surviving document substrings;

3. Compactness: Our filter is compact and resides in main memory.

Batching subset checks: We exploit the following characteristics to develop an "optimal" subset-checking algorithm.

1. Candidate substrings across documents contain many common token sets; thus, checking these common token sets once across all documents results in significant savings. For example, a token set "canon XTI" may occur across many substrings; because we separate the substring extraction from subset-check, we perform subset-check of a token set once across all of these candidate substrings;

2. We also "order" the subset-checks for each of these token sets so that we can reuse partial computation across different token sets. For example, if "canon XTI" and "canon XTI digital" are both valid token sets in hit sequences we may use the result of "canon XTI" for "canon XTI digital", *provided* we perform the subset-check in the right order;

3. Since we cache the intermediate results, we need to discard results that are not required for future subset-checks. We determine this efficiently.

We leverage the suffix tree data structure to optimize the order. Our algorithm is *optimal* in that for any distinct token set, we perform the subset-check exactly once.

3.3 Filtering Document Substrings

To distinguish surviving document substrings from those pruned by the filter, we introduce the notion of a *hit sequence*. A document substring is a hit sequence if it contains one or more contiguous token sets which are DisTokenSets. By identifying hit sequences, document substrings which do not contain any hit sequences can be pruned. Note all substrings which encompass the DisTokenSet are valid hit sequences. However, shorter hit sequences are desirable since the cost of the subset-check is directly proportional to the number of substrings of a hit sequence. Therefore, our goal is to efficiently generate hit sequences whose lengths are as small as possible. In the rest of this subsection, we first define the filter structure, and then show how to use the filter to efficiently identify hit sequences.

3.3.1 CT Filter

The naive approach is to directly compare each document substring with a DisTokenSet. However, this approach raises two issues. (i) There may be too many DisTokenSets to be kept in memory. (ii) We need to look up all substrings with length up to the number of tokens in the largest DisTokenSet. As we show in Section 5, this look up cost is not trivial. On the other hand, we observe that many DisTokenSets such as "canon eos rebel xti digital camera" could be fairly long. While their subsets such as "rebel xti" are short and selective enough to generate hit sequences. To distinguish such short selective subsets from DisTokenSet, we define the *Core Token Sets* as follows.

DEFINITION 6. **(Core Token Sets)** *Let $\mathcal{E}$ be the set of reference entities, $f(\tau)$ the frequency of a token set τ w.r.t. $\mathcal{E}$, L a positive integer, and K' a positive integer. Let K be the constant which determines whether or not a token set is a K-discriminating token set.*

1. *If $K' \geq K$, a token set τ is a* core token set *if (i) $|\tau| = L$ or $f(\tau) \leq K'$; and (ii) there exists at least one subset[3] $\tau' \subset \tau$, such that $|\tau'| = |\tau| - 1$ and $f(\tau') > K'$. Let $\mathcal{C}_{K'}$ be the set of such core token sets.*

2. *If $K' < K$, then the set of core token sets is $\mathcal{C}_{\mathcal{K}} \cup \{\tau | \tau \subset \tau' \in \mathcal{C}_{\mathcal{K}}$ and $f(\tau) \leq K\}$.*

EXAMPLE 3. *Figure 2 demonstrates core token sets of an entity "sony viao F150 laptop", by setting $K' = 5$ and $L = 2$. The numbers at the right of the subsets are frequency. Subsets in bold font are core token sets.*

Intuitively, the core token sets are a collection of "frontier" token sets whose *subsets* are small (contain less than L tokens) and frequent (occur in more than K' entities). Also, a token set τ is not a core token set if all of its immediate subsets (with one less token, i.e., with $|\tau| - 1$ tokens) are core token sets (Condition 1.(ii) in Definition 6). This property is important for *linear time document processing*. Specifically, given any DisTokenSet, if we progressively remove one arbitrary token at a time, we will obtain a core token set. Therefore, any DisTokenSet, no matter how the tokens are ordered, can be broken down into a contiguous sequence (possibly overlapping) of core token sets. For instance, a DisTokenSet "vaio sony F150" can be broken into two core token sets "vaio sony" and "sony F150". This property enables efficient generation of hit sequences because we only need to check consecutive token sets, with size up to L, in the documents. Therefore, the complexity of processing a document to generate hit sequences is linear in the number of tokens in the document. Therefore, the lookup complexity is $L|d|$, for a document d.

The core token sets are defined with respect to two constraints (Condition 1.(i) in Definition 6): (i) the maximum number of tokens in a core token set L, or (ii) the maximum frequency of a core token set K'. L controls the *compactness* of the set of core token sets, and K' controls the *selectivity* of the set of core token sets. Because the distribution of tokens among reference entities and that among documents are often very similar, low frequency token sets enable us to effectively prune document substrings.

Observe that the core token sets are different from the set of *minimal* DisTokenSets, where a DisTokenSet τ is *minimal* if *none* of its subsets is also a DisTokenSet. If we use minimal DisTokenSets, the algorithm for processing a document now becomes significantly more expensive. Suppose the minimal DisTokenSets are {vaio, F150} and {F150, laptop}. A document subsequence "vaio sony F150" is broken into "vaio sony" and "sony F150", and none of which matches with a minimal DisTokenSet. In fact, in order to identify "vaio sony F150" as a valid hit sequence, the algorithm needs to look up combinations of tokens which are not contiguous. Suppose the maximal length of entities is S, keeping minimal discriminative token sets requires $2^S|d|$ lookups for a document d.

Procedure for Identifying Core Token Sets: The core token sets can be efficiently identified as follows. Observe that for each entity, all its token subsets constitute a lattice. Therefore, we make a depth-first traversal on this lattice structure from the bottom root node ϕ. At any subset τ, if $|\tau| = L$ or $f(\tau) \leq K'$, we identify τ as a core token set and traverse back. $f(\tau)$ can be pre-computed using one of the frequent item-set mining algorithms [17].

CT Filter: Let $\mathcal{C}$ be the set of all core token sets. We define $\mathcal{I} = \{\tau | \tau \subset \tau' \in \mathcal{C}\} - \mathcal{C}$. That is, $\mathcal{I}$ contains all "internal token sets". We design a *CT Filter* which consists of two hash tables: one maintaining all subsets in $\mathcal{C}$, and the other maintaining all subsets in $\mathcal{I}$. In general, the size of $\mathcal{I}$ is smaller than that of $\mathcal{C}$ since $\mathcal{C}$ is the boundary closure of $\mathcal{I}$.

Given a token set τ the *CT Filter* returns one of three values: *internal (I), core (C), missing (M)*. If τ matches an internal token set in $\mathcal{I}$, the *CT Filter* returns the value I; if τ matches a core token set in $\mathcal{C}$, the *CT Filter* returns the value C. Otherwise, the filter returns the value M, indicating that τ is missing in $\mathcal{C} \cup \mathcal{I}$.

3.3.2 Generating Hit Sequences for a Document

As shown above, the hit sequences can be consecutively split into core token sets. As the maximal size of core token sets is L, a brute-force solution is to perform up to L lookups against the *CT Filter* at each document token. Observe that if a token set has status M, then all its supersets must have status M. We can leverage this property to reduce the number of lookups against the *CT Filter*.

We iterate over an input document. In the first iteration, we check subsequences with one token. In each subsequent iteration we increment the token set size by one. We lookup

[3]We assume $|\phi| = 0$, and $f(\phi) = |\mathcal{E}|$

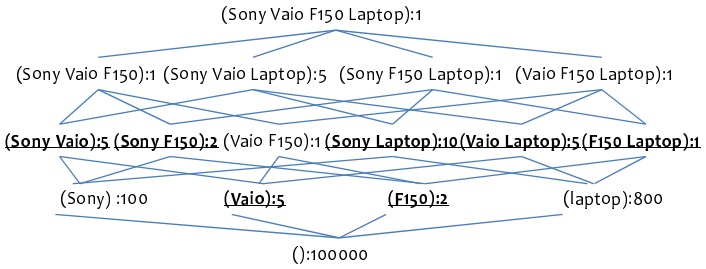

Figure 2: Core Token Sets

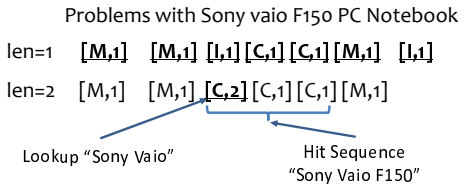

Figure 3: Hit Sequence Generation

the token sets corresponding to a contiguous document subsequence against the *CT Filter* and mark them with status C, I, M returned by *CT Filter*. Whenever all token sets corresponding to contiguous subsequences are either marked with M or C, we stop. All contiguous token sets marked with C form a hit sequence.

Algorithm 1 Hit Sequence Generation

Input: A document d, *CT Filter* F

```
1: Create an array s, such that |s| = |d|;
2: Let s_i = [Lookup(F, d_{i,1}), 1];//return I, C or M
3: for j = 2 to L
4:     for i = 1 to |d| − j + 1
5:       if (s_i = I and s_{i+1} ≠ M)
6:         s_i = [Lookup(F, d_{i,j}), j];
7:       else if (s_i = C and s_{i+1} = I);
8:         s_i = [Lookup(F, d_{i,j}), j];
9:       else if (s_i = I and s_{i+1} = M)
10:        s_i = [M, j];
11:return
```

```
LookUp(F, token_set)
//F maintains two hash tables I and C
10: if (I.ContainsKey(token_set) = true) return I;
11: if (C.ContainsKey(token_set) = true) return C;
12: return M;
```

The pseudo code is shown in Algorithm 1. Given a document d, we first tokenize d to a sequence of tokens. Let d_i be the i^{th} token of d. Let $d_{i,j}$ denote the contiguous token set $\{d_i, d_{i+1}, \ldots, d_{i+j-1}\}$. We maintain an array s of [*status, iteration number*] pairs of length $|d|$ to record the status of contiguous token sets in d. For example, if $s_i = [C, j]$, then the token set $d_{i,j}$ matches a core token set, and if $s_i = [I, j]$, $d_{i,j}$ matches an internal token set.

For clarity of exposition in the following description, we ignore the special cases arising at the boundaries of d. The algorithm iterates over the token sequence of d at most L times. In the first iteration, we lookup all individual tokens d_i (the same as $d_{i,1}$) against the *CT Filter*. We initialize the array s with $s_i = [X, 1]$ where X is the value returned by the *CT Filter*. In the j^{th} iteration (where $j > 1$), we lookup the token sets $d_{i,j}$ against the *CT Filter* if the status values (s_i, s_{i+1}) are marked with any of the following three status value combinations: (C, I), (I, C), or (I, I). We record the values returned by *CT Filter* in s: $s_i = [X, j]$, where X is the value returned by looking up $d_{i,j}$ against the *CT Filter*.

We stop when (i) none of the contiguous token set pairs is marked with pairs (C, I), (I, C), or (I, I), or (ii) if the number of iterations has reached L. This is because all core token sets contain at most L tokens. The complexity is $O(L|d|)$, but we anticipate that in each subsequent iteration, the number of subsequences processed decreases significantly. We then extract contiguous tokens marked C as hit sequences. An example of hit sequence generation is shown in Example 4. Lemma 1 claims the correctness of the algorithm.

Example 4. *Suppose the input document is "Problems with Sony vaio F150 PC Notebook". We first lookup all individual tokens against the* CT Filter*. The lookup results are shown on the row with Len = 1 (Figure 3). We identify that "sony" and "notebook" have status* $[I, 1]$*. Since "notebook" is at the end of the document, we do not further look it up against the* CT Filter*. In the second scan, we lookup the token set* $\{sony, vaio\}$*, and the* CT Filter *returns* $[C, 2]$*. At this stage, none of the status values is I, and the algorithm halts. Further, it finds three consecutive* $[C, j]$*s, and returns "sony vaio F150" as a hit sequence. Observe that all status in bold font are returned by filter lookup.*

Lemma 1. *(***Correctness***) Given a document d, Algorithm 1 generates all hit-sequences that may contain discriminative token sets and performs at most* $O(L|d|)$ *lookups against the* CT Filter*.*

3.4 Batching Subset Checks

After hit sequences are generated from documents, we want to identify contiguous token sets from a hit sequence that can be a DisTokenSet of some reference entity. The output consists of (candidate IDTokenSet, entityID) tuples. As discussed in Section 3.2, the goal is to share computation across all hit sequences. In the following, we first discuss how to perform subset-check for a single token set, and then present techniques for batched subset-check.

3.4.1 CT Index

Consider a hit sequence "sony vaio F150". We need to consider contiguous token subsets {"sony", "vaio", "F150", "sony vaio", "vaio F150", "sony vaio F150"}, and check whether or not each of these is a DisTokenSet of a reference entity, and also determine the entity identifiers which a DisTokenSet may identify. A popular technique we adapt for this task builds an *inverted index* over all reference entities (Section 3.1). For example, if the idlist for the token "sony" is {1, 5, 10} and that for "vaio" is {1} then the intersection of these two lists contains the identifier 1, which contains both tokens.

We generalize this approach to exploit the core token sets from the *CT Filter* and build a *CT Index*. We maintain idlists for each core token set in the *CT Index*. Those idlists

are usually much smaller than those for individual tokens. Recall that each hit sequence is a contiguous sequence of core token sets. For a single hit sequence, we process it by intersecting the idlists of core token sets.

3.4.2 Suffix Tree based Scheduling

Although the sizes of idlists from the *CT Index* are much smaller than those in the standard inverted index, merging idlists repeatedly can still be very expensive. We now describe our algorithm which minimizes the idlist intersections. We order the intersections of idlists of core token sets so that they can be reused across hit sequences. Our main idea is to build a *suffix tree* over all hit sequences (Example 5). Note that in our implementation, a hit sequence is actually a contiguous set of core token sets. For simplicity, we demonstrate it using tokens.

EXAMPLE 5. *Figure 4 shows a suffix tree built upon a hit sequence "sony vaio F150". Each node maps to a subsequence. For instance, n_2 maps to a subsequence "sony vaio". All non-root nodes n have a suffix link (dashed link) to another node, which corresponds to the largest suffix subsequence of n. We maintain in each node an intersected idlist corresponding to the related subsequence.*

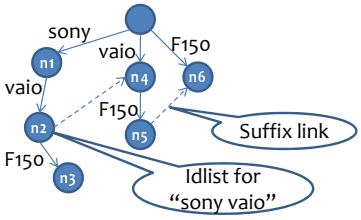

Figure 4: Example Suffix Tree

The algorithm for performing idlist intersections traverses the suffix tree, and computes the idlist for each node. Observe that each node in the suffix tree corresponds to a distinct subsequence among all hit sequences. The suffix tree guarantees that the inverted idlists of all distinct subsequences of input hit sequences will be computed exactly once. If required, the idlists in the nodes would be reused thus avoiding any redundant intersections. We also identify and exploit two optimization opportunities: (i) In order to efficiently compute the intersected idlist for a node, we leverage the suffix links. Specifically, to compute the idlist for a node n, we will intersect the idlists from n_p, n's *parent node*, and n_s, n's *suffix node* (the node pointed by n's suffix link). Note the procedure is recursive such that if the idlist on either n_p or n_s has not been computed yet, we will recursively check their parent node and suffix node. (ii) If an idlist will not be used for intersection in the future, we will delete the idlist from the memory, and thus save the space for future usage. An example scheduling of idlist intersection is shown below.

EXAMPLE 6. *In Figure 4, suppose we start with the node n_1. The algorithm first retrieves idlist of "sony" from* CT Index. *We then traverse to node n_2. n_2's parent node is n_1, whose idlist has been computed. n_2's suffix node is n_4, whose idlist has not been computed. The algorithm thus processes n_4 first, and retrieves idlist for "vaio" from* CT Index. *Now both n_2 and n_4 are ready, and the algorithm computes the idlist for n_2 by intersecting the idlists from n_2 and n_4. Now n_1 is neither a parent node nor a suffix node for any remaining nodes. We release n_1 as well as its idlist from memory since n_1 will not be referenced in the future. The algorithm continues its traversal to n_3, and so on. At any stage, we output a subsequence as a valid candidate IDTokenSet if the length of its idlist is no larger than K.*

Algorithm 2 Candidate IDTokenSet Generation

Input: A set of hit sequence: H, *CT Index*: IND

```
1: Build a suffix tree for all h ∈ H;
2: Initialize a stack Stack = φ;
3: while (the node traversal not complete)
4:     if (Stack = φ)
5:         Stack.push(n= next node in the traversal);
6:     else
7:         n = Stack.pop();
8:         if (n_p's idlist has not been not computed);
9:             Stack.push(n_p);
10:        else if (n_s's idlist has not been not computed);
11:            Stack.push(n_s);
12:        else;
13:            Compute idlist for n, pop n from Stack;
14:            Output n as IDTokenSet if |n's idlist| ≤ K;
15:            Delete n_p if it is not referenced in future;
16:            Delete n_s if it is not referenced in future;
17: return
```

Note that each idlist of a core token set is retrieved from disk at most once. For each node n (*i.e.*, a subsequence from hit sequences), the idlist of n is computed only once. Furthermore, it is computed from the two largest subsequences of n. We formalize this observation in the following lemma.

LEMMA 2. *Algorithm 2 accesses the idlist of each core token set at most once, and it also computes the intersected idlist for all relevant subsequences at most once. Hence, it is optimal.*

If there is an excessive number of hit sequences, we adopt a partitioning approach, which divides documents into chunks. We discuss the details in Section 4.3.

4. THE COMPLETE ARCHITECTURE

We now discuss the complete algorithm. We first describe the second and the third phases, and then discuss various extensions such as how to handle large intermediate results, and other alternative implementations.

4.1 Per-Document Score Computation

The input to this phase consists of (candidate IDTokenSet, entityID) tuples, and our goal is to generate the (candidate IDTokenSet, entityID, docID, score) tuples. Note that we use either g_1 or g_2 to generate the score values, both of which require access to the document text and the entity strings. Therefore, we scan documents again.

We maintain two hash tables in memory. First, we insert all candidate IDTokenSets and their corresponding entityIDs in a hash table H_1 (or, a TRIE structure [3]) to

enable efficient lookup. Second, we build a hash table H_2 mapping entity identifiers to entity strings. We now scan the document set again. For each document, we check whether each substring matches with a candidate IDTokenSet using H_1. If it does, we retrieve the corresponding entities from H_2, and compute the correlation score (g_1 or g_2) for (candidate IDTokenSet, entityID).

We believe that both H_1 and H_2 fit in memory. We observe that although the total number of DisTokenSets is larger, the number of distinct candidate IDTokenSets (*i.e.*, DisTokenSets which appear contiguously in some documents) is often limited. In our experiments, we observe that the number of distinct candidate IDTokenSets for over $70TB$ of web documents is no more than 5 times the number of entities. Therefore, for most practical scenarios, H_1 fitting in memory is not an issue. However, in case there is an excessive number of candidate IDTokenSets, we adopt the partitioning approach (See Section 4.3). In the scenarios where the number of entities is so large that we cannot put H_2 in memory, we partition the entities into groups and repeat the procedure for each group.

4.2 Score Aggregation and Thresholding

The input to this phase is the set of (candidate IDTokenSet, entityID, docID, score) tuples. The output is the set of (IDTokenSet, entityID) whose aggregated score is greater than a threshold. This is equivalent to grouping the input by (candidate IDTokenSet, entityID) and aggregating the scores within each group. Therefore, we apply standard aggregation techniques (sort grouping). Given the threshold, if an IDTokenSet maps to multiple entities, we choose the entity with the highest score (motivated by the "ipod nano" example in Section 2).

4.3 Handling Large Intermediate Results

Here we discuss a document partition framework to handle really large intermediate results (*i.e.*, hit sequences or candidate IDTokenSets). The intuition is that the large intermediate results are generated from large collection of documents. If we can partition the entire document collection into smaller subsets, and process each smaller subset separately, the size of the intermediate results can be controlled. We adopt an implicit partitioning scheme to avoid scanning the entire document collection and writing smaller subsets into different files.

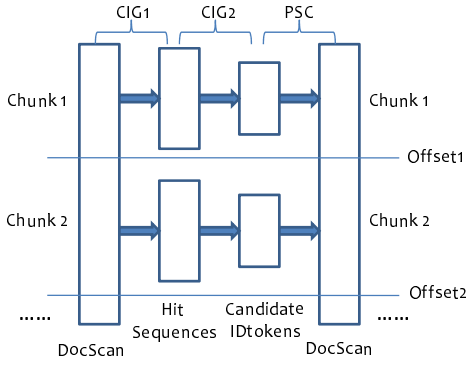

Figure 5: Document Partition

We further divide the first *CIG* phase into *CIG1* and *CIG2*, corresponding to hit sequence generation (Section 3.3) and batched subset check (Section 3.4), respectively. The partitioning scheme is illustrated in Figure 5, and it involves *CIG1*, *CIG2* and *PSC*. During the first document scan in *CIG1*, we keep track of the number of hit sequences generated so far. As soon as the size of generated hit sequence exceeds a threshold, the *CIG2* phase generates candidate IDTokenSets for these hit sequences, and outputs candidate IDTokenSets to a file. The hit sequences are discarded. The threshold can be configured so that the *CIG2* can be performed in memory. After we output candidate IDTokenSets to the file, we insert an *offset*, which is the number of documents scanned so far, to the same file.

Intuitively, the *offsets* create implicit partitions among the output of *CIG2*, and sync those partitions with the document partitions. We assume the *PSC* phase scans the documents in the same order as that in *CIG1*. The *PSC* phase basically conducts a merge join. It sequentially loads the candidate IDTokenSets partitions from the output of *CIG2*, and scans the corresponding document partitions identified by the *offsets*. After the score computation on the current partition is finished, it moves to the next partition.

4.4 Alternative Implementations

One DocScan Approach: We note that the *CIG* and *PSC* phases can be combined together, so that we directly check whether or not each contiguous token set of a hit sequence is a DisTokenSet of some reference entity, and output (candidate IDTokenSet, entityID, docID, score) tuples. Since this approach only scans documents once, we refer to it as *One DocScan* approach. In our original architecture (Section 2.1), we separate the two phases, and scan documents twice. Thus, it is a *Two DocScan* approach.

The one DocScan implementation leverages the fact that when the number of entities is not large, the *CT Index* can be loaded into memory. In this case, we can run *CIG* on b documents, where b is the batch size determined by the space budget. We cache all b documents in the memory. After the candidate IDTokenSets are generated, we run the *PSC* using the cached documents. We then output the (candidate IDTokenSet, entityID, docID, score) tuples, discard the cached documents, and read next b documents. The *score aggregation and thresholding* phase remains unchanged.

Observe that the one DocScan approach splits hit sequences into many small batches, and conducts subset-check on each of them. Therefore, it may not be able to fully leverage the benefit of batched subset check. There is a trade-off between the cost saved from another document scan and the cost paid for multiple subset-check. In Section 5, we show that there is a switching point between the one DocScan and two DocScan approaches, as we vary the number of entities.

Map-Reduce Framework: We can also adopt our proposed method in the *map-reduce* infrastructure [15]. Map-reduce is a software framework that supports parallel computations over large data sets on clusters of computers. This framework consists of a "map" step and a "reduce" step. In the map step, the master node chops the original problem up into smaller sub-problems, and distributes those to multiple worker nodes. The worker node processes that smaller problem, and passes the answer to the reduce step. In the reduce step, the master node combines the answers to all the sub-problems and gets the answer to the original problem.

Our techniques, either one DocScan or two DocScan, perfectly fit into the map-reduce framework. In the map step,

we chop the entire document set into smaller collections, and run the first two phases (*e.g.*, *CIG* and *PSC*) over each small document collection on worker nodes. In the reduce step, we aggregate (candidate IDTokenSet, entityID, docID, score) tuples and run the *SAT* phase. Using the map-reduce infrastructure, we are able to process large number of documents (multiple terabytes). Observe that the map step is similar to the document partition framework discussed in Section 4.3. The difference is that the map step does it in parallel, while document partition does it sequentially.

5. PERFORMANCE STUDY

We use real data sets for the experiments. The reference entity database is a collection of $1,000,000$ product names (*e.g.*, consumer and electronics, bicycles, shoes, *etc*). The number of tokens in the product names varies from 1 to 10, with 4.7 tokens on average. To examine the quality of IDTokenSets, we generate IDTokenSets over $70TB$ web documents, using a map-reduce infrastructure. Since we do not have control over the number of machines assigned to a job, we are not able to precisely measure the algorithm's performance over the map-reduce infrastructure. Therefore, to study the computational performance of our proposed algorithms, we also conduct experiments on a single PC with $1M$ Wikipedia pages.

Entity	IDTokenSets
canon eos 1d mark ii	canon mark ii eos 1d
	canon mark ii eos
	eos 1d mark ii
	mark ii 1d
	1d ii canon eos
	1d ii canon

Table 4: Example IDTokenSets

We examine the quality of the IDTokenSets, as well as the scalability of our proposed techniques. The major findings of our study can be summarized as follows:

1. **High quality IDTokenSets**: the document-based measure performs significantly better than a representative string-based similarity measure in determining IDTokenSets. To begin with, we show some example IDTokenSets generated by our framework in Table 4.

2. **Scalable IDTokenSets generation**: our algorithms efficiently generate IDTokenSets from large collections of entities and documents. Specifically:
   (a) *CT Filter* and *CT Index* improve the baseline by orders of magnitudes.
   (b) Candidate IDTokenSet generation using suffix tree is significantly faster than the baseline.
   (c) Two DocScan is scalable to a very large number of entities.

5.1 Quality of IDTokenSets

We first describe the experimental setup, and then compare the precision-recall with respect to different similarity measures. We also examine the effect by using different window sizes in determining the context, and by allowing gaps in extracting candidate IDTokenSets.

5.1.1 Experimental Setup

To examine the quality of the IDTokenSets, we compare our proposed document-based measures with a representative string-based similarity measure, weighted Jaccard similarity(In fact, many other string similarity measures, *e.g.*, edit distance, can be rewritten by Jaccard similarity [9].). It is defined as follows.

$$WJS(\tau_e, e) = \frac{\sum_{t \in \tau_e \cap e} w(t)}{\sum_{t \in \tau_e \cup e} w(t)} = \frac{\sum_{t \in \tau_e} w(t)}{\sum_{t \in e} w(t)}$$

where $w(t)$ is the weight (*e.g.*, IDF weight) of the token t.

The IDTokenSets are generated from a large collection of web documents ($70TB$), and the computation is performed on clusters of computers which support map-reduce framework. The entire IDTokenSets computation takes around 5 hours. By default, we set $p = \infty$ in determining p-window context (Definition 3).

To evaluate the quality of the generated IDTokenSets, we configure the experiment as follows. We first extract 800 digital camera product names from our reference entity table. We then extract 463 queries, which are all related to digital cameras, from the web search query log. We ask domain experts to label the data using the following rule: a query is mapped to a digital camera name if it mentions the digital camera, which is in our list of 800 names; otherwise, the query is considered NOMATCH. Some sample labeled data is shown in Table 5. Note "cyber shot" is considered NOMATCH because there are multiple sony cameras in the cyber shot product line, and the phrase "cyber shot" does not match with a unique digital camera. Among 463 queries, there are 316 queries mapped to a digital camera name.

Query	Matched Entity
nikon l18 reviews	nikon coolpix l18
sony cyber shot w80	sony dsc w80
cyber shot	NOMATCH

Table 5: Example Query and Entity Pairs.

5.1.2 Comparison with String Similarity

We denote **Corr1** for the document-based measure with g_1 (Equation 1), **Corr2** for the document-based measure with g_2 (Equation 2), and **Jaccard** for the string-based weighted Jaccard Similarity. If a query contains a substring that matches with an IDTokenSet, we predict the corresponding entity as the match. For *Jaccard*, if a query contains a substring that matches with an entity (string similarity is higher than the threshold), we predict the corresponding entity as the match. We vary the threshold from 0.5 to 0.9) for all three methods. The precision-recall curves are plotted in Figure 6.

We observe that the document-based measures are significantly better than the string-based measure. Among two variants of document-based measures, *Corr1* has high precision and *Corr2* has higher recall, for each given threshold. This is because *Corr1* is a stricter measure since it requires all tokens (in the original entity) to be present in the documents, and *Corr2* uses a relaxed formulation.

We note that these correlation and similarity measures can feed into a machine learning model to potentially obtain more accurate results.

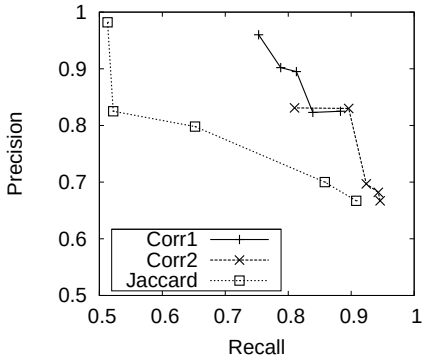

Figure 6: Precision-Recall w.r.t. Measures

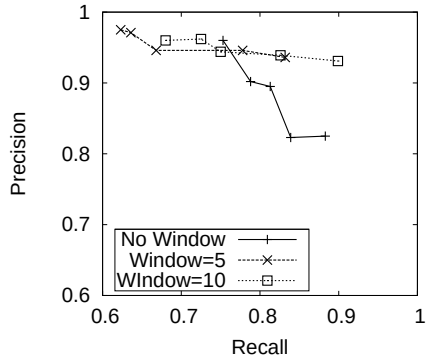

Figure 7: Precision-Recall w.r.t. Window Size

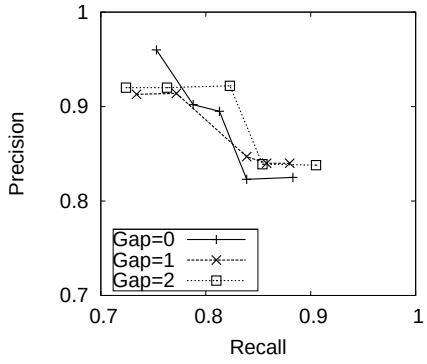

Figure 8: Precision-Recall w.r.t. Gap Size

5.1.3 *Effect of Window Size*

Here we examine the effect of window size in the p-window context (Definition 3). We vary the value of p from 5, 10 to ∞. For $p = \infty$, we use the threshold 0.5 to 0.9, since it is a relatively relaxed configuration. For $p = 5$ and $p = 10$, we use the threshold 0.1 to 0.5. The precision-recall curve for *Corr1* is shown in Figure 7.

We observe that when we set a high threshold, $p = \infty$ has a higher recall than that of $p = 5$ and $p = 10$. When we set a relatively low threshold, $p = \infty$ may generate more false IDTokenSets, and its precision is lower than that of $p = 5$ and $p = 10$. The method is not overly sensitive to the window size, as we can see that the performance of $p = 5$ and $p = 10$ are very close.

5.1.4 *Effect of Gapped Mentions*

In this paper, we discuss techniques which require IDTokenSets to be contiguous document substrings. One possible extension is to allow gaps in the IDTokenSets when they are mentioned in documents. For instance, if we set $gap = 1$ (*i.e.*, allow one token insertion between any two consecutive tokens in IDTokenSets), we will identify "dsc w80 sony" as a candidate IDTokenSet form "dsc w80 from sony".

We now report experiments based on adaptions of our techniques to handle gapped mentions. By varying the gap size from 0 to 2, the precision-recall curves of *Corr1* are shown in Figure 8. We observe that the precision-recall for different gap sizes are very similar. This is because we compute IDTokenSets across a large collection of documents. An IDTokenSet may be mentioned with gap in some documents. but in a majority of documents, it is mentioned contiguously. As a result, setting $gap = 0$ is still able to identify it. Therefore, the gapped mention does not yield additional benefit over the contiguous model ($gap = 0$).

5.2 Scalability

We now report the computational performance of our techniques. We use $1M$ Wikipedia pages (average 499 tokens per page) on a single PC with 2.4GHz Intel Core 2 Duo and 4GB RAM. We use Microsoft SQL Server 2005 to store *CT Index*.

We first examine the computational cost of each phase in our architecture, and then compare our approach with baseline solutions. In Section 3.1, we describe two baseline approaches. The first approach that enumerates all possible DisTokenSets is clearly not scalable because the number of DisTokenSets per entity is exponential in the number of tokens in the entity. It not only requires excessive space, but also performs much more disk accesses (see Section 3.1 for the detailed description). Therefore, we only compare with the inverted index based approach [10]. Observe that in our proposed framework, $L = 1$ is actually an improved version of the inverted index approach because it is integrated with substring filtering and batched subset-check. However, even comparing with this improved approach, our method is two orders of magnitude faster (Figure 9).

5.2.1 *Overall Performance*

We set $K = 5$ for DisTokenSet. In order to examine the computational and space complexity of the *CT Filter* and *CT Index*, we first vary K' and L values on the $1M$ product entities. The number of core token sets, the average length of idlists per core token set and the execution time to compute the core token sets are reported in Table 6.

K'	L	Number	Avg IDList Size	Exec. Time (s)
10	4	7.9M	1.4	486.5
10	3	5.9M	1.5	371.2
10	2	2.2M	1.9	156.3
10	1	248K	19.1	6.8

Table 6: Core Token Set Generation

We observe that $K' = 10$ and $L = 2$ achieves a good trade-off between costs of hit sequence generation and candidate IDTokenSet generation in that L is not large, and the average length of idlists is reasonably small. Therefore, we set $K' = 10$ and $L = 2$ in the following experiments. In Section 5.2.2, we have a detailed comparison of different parameter configurations. It takes 156 seconds to build the *CT Filter* and the *CT Index*.

We first report performance of the two DocScan approach. We fix the number of entities to be $1M$, and vary the number of documents from $1k$ to $1M$. The execution time is shown in Table 7. We observe that *CIG1* (Section 4.3) and *PSC* (Section 2.1) are the two most expensive phases since they involve document scan. Although the *CT Index* resides on the disk, *CIG2* is still quite efficient. This is because we use *CT Filter* to significantly reduce the hit sequences and exploit suffix tree to share computation in subset-check. We then fix the number of documents to be $1M$, and vary the number of entities from $1K$ to $1M$. The results are shown

#Document	1K	10K	100K	1M
CIG1	23.7	101.2	733.7	5327.9
CIG2	5.2	32.1	124.1	351.4
PSC	13.1	63.1	545.2	3803.1
SAT	0.1	1.1	14.0	100.4
Total	42.1	197.5	1417.0	9582.8

Table 7: Execution Time w.r.t. #Document

#Entity	1K	10K	100K	1M
CIG1	678.4	977.9	2087.8	5327.9
CIG2	0.5	2.7	18.2	351.4
PSC	740.3	836.5	1017.8	3803.1
SAT	0.1	0.6	6.1	100.4
Total	1419.3	1817.7	3129.9	9582.8

Table 8: Execution Time w.r.t. #Entity

in Table 8. Overall, our method is scalable to very large number of documents and entities.

5.2.2 Effect of Core Token Sets

We show the core token set is a key component to make the entire computation very efficient. We vary L from 1 to 4. Recall that $L = 1$ is actually an improved version of the traditional inverted index based approach. Therefore, we denote *InvertedIndex++* for $L = 1$. Figure 9 shows the accumulated execution time of *CIG1* and *CIG2*. Using *InvertedIndex++* over the $1M$ entity and 100 documents generates $8,237$ hit sequences, and each of which has more than 20 tokens on average. While using $L = 2$ ($L = 4$), the same data set generates 711 (615) hit sequences, each of which has 8.7 (8.5) tokens on average. Furthermore, the idlists of *InvertedIndex++* are much larger than the other two alternatives. Overall, *InvertedIndex++* is much slower.

Comparing with $L = 2$ and $L = 4$, $L = 2$ is slightly faster on the *CIG1* phase, but slightly slower on the *CIG2* phase. We observe that although the worst case complexity of *CIG1* is proportional to $L|d|$, Algorithm 1 conducts much less lookups since many token sets returns C (*e.g.*, core) or M (*e.g.*, miss) status in early iterations (See Section 3.3.1). We also observe that our proposed approach is not sensitive to L when $L \geq 2$.

5.2.3 Effect of Batched Subset Check

To demonstrate effectiveness of batched subset-check, we compare it with a baseline approach that performs subset-check per hit sequence, using progressive merge [10].

The results are reported in Figure 10. We observe that the batched subset-check is very effective in sharing the computation across different hit sequences. It reduces the execution time by 2 orders of magnitudes. Without suffix tree optimization, the computation time of *CIG2* will be significantly larger, and will be even larger than that of *CIG1*.

5.2.4 Two DocScan vs. One DocScan

In some scenarios where the reference entity list is small, and the *CT Index* also fits in memory. In those cases, one can combine *CIG1*, *CIG2* and *PSC* into one phase, and the documents are only scanned once.

In Figure 11, we show the execution time of both one DocScan and two DocScan approaches, by fixing the number of documents to be $1M$. As expected, when the number of entities is small, one DocScan approach is faster because it saves another scan of documents. When the number of entities reaches $1M$, the two DocScan approach wins. This is due to the limitation of the one DocScan approach: we need to cache documents so that we can compute the scores for candidate IDTokenSets on-the-fly. However, we are not able to cache many hit sequences in *CIG2*. This impacts the overall execution time. Figure 11 shows a clear trend that the two DocScan approach will perform better than the one DocScan approach beyond the $1M$ point.

6. RELATED WORK

As discussed earlier in Section 1, string similarity functions have been used for identifying substrings from documents that approximately match with some dictionary entry (*e.g.*, [9]), and for identifying pairs of matching records (*e.g.*, [10, 11]).

The idea of filtering document subsequences was explored earlier. [9] considers the problem of identifying document substrings whose string similarity with an entity is above a given threshold. They derive signatures based on the threshold and design filters based on signatures. In this paper, we want to identify substrings of documents that are DisTokenSets of some entities. There is no threshold over the string similarity between the document substring and the entity. Hence, the signature schemes do not apply.

In the context of data cleaning, techniques for leveraging pairs of matching and non-matching records labeled explicitly by users have been developed to identify synonyms which help improve the string similarity between input records [7, 22]. These techniques primarily rely on "unaligned" token set pairs (i.e., token sets which are present in one record but not its matching record) across example pairs. Hence, these techniques cannot discover the important class of subset synonyms (which contain a subset of tokens of the original entity) we consider in this paper.

In this paper, we focus on the problem of matching document substrings with entity names in a reference table. A related but orthogonal task is to further analyze the document substring and its document context using machine learning and natural language parsing techniques in order to (i) ensure that the document substring in the document is actually a reference to the entity and not a generic phrase in the language [13, 19], and to (ii) resolve ambiguity in entity references if multiple individual entity instances share the same name [5, 18].

Statistical approaches measuring co-occurrence (or association) for detecting synonyms have been studied in natural language literature [21]. Several measures, notably that of mutual information, have been considered for quantifying distributional similarity between words [20, 14]. As mentioned earlier, we adopt the measures (g_1 and g_2) instead of the popular mutual information measure because our scenario is inherently "asymmetric". Our target is to identify subset synonyms. Hence, the count of documents which contain the candidate IDTokenSet is always higher than that containing the original entity. Therefore, we adopt a measure which is derived from Jaccard containment. If required, our approaches can be adapted to use mutual information. As discussed earlier, measuring the distributional similarity for a large number of entities across a large collection of documents is expensive.

Domain-specific rule-based approaches have been proposed for generating variations of named entities [4]. But, it is quite hard to hand-craft the domain-specific rules for gener-

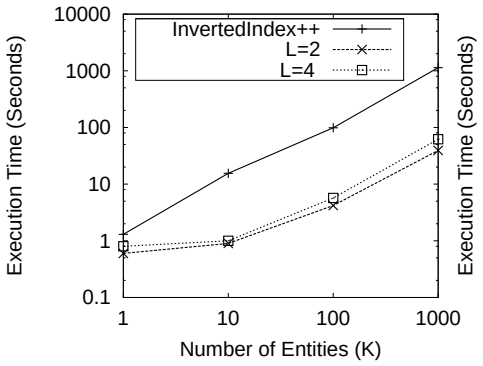

Figure 9: Core Token Sets

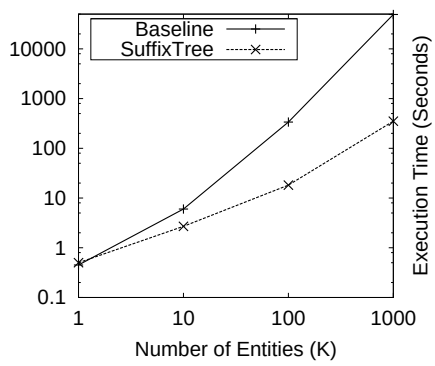

Figure 10: SuffixTree vs. Baseline

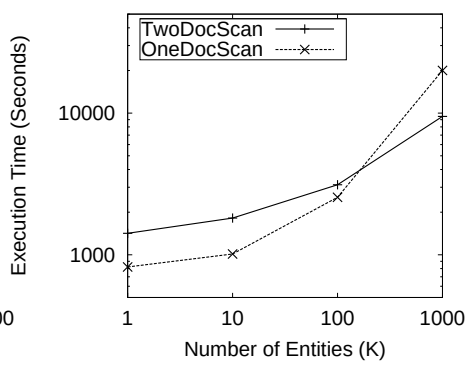

Figure 11: Two DocScan vs. One DocScan

ating synonyms. Therefore, it is hard to robustly and with a high recall generalize this approach to a significant number of domains. In contrast, we focus in this paper on the problem of efficiently identifying variations which have a high correlation measure with any entity in the reference table.

Turney [23] introduced an unsupervised learning algorithm that exploits a web search engine for recognizing synonyms. In a similar vein relying on web search engines, we explored optimization techniques to minimize the number of web search queries [12]. However, the web search based techniques issue a very large number of web search queries, at least one or often more than one per entity in the reference table. These techniques are often limited by: (i) the number of results returned by search engines, and (ii) the capacity to issue a large number of web searches. Hence, they may not scale to a large number of entities.

7. CONCLUSIONS

In this paper, we considered the problem of expanding a reference table of entities with IDTokenSets obtained by mining large document collections. We considered measures for quantifying the correlation between a candidate IDTokenSet and an entity over a set of documents, and developed very efficient techniques to identify IDTokenSets. We demonstrated that our architecture and the techniques can be implemented efficiently in a map-reduce infrastructure thus gaining the ability to mine web-scale document collections. Using large real databases and document collections, we demonstrated the efficiency and scalability of our techniques for identifying IDTokenSets and the accuracy of the results.

8. REFERENCES

[1] S. Agrawal, K. Chakrabarti, S. Chaudhuri, and V. Ganti. Scalable ad-hoc entity extraction from text collections. In *VLDB*, 2008.

[2] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 18, 1975.

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

[4] R. Ananthanarayanan, V. Chenthamarakshan, P. Deshpande, and R. Krishnapuram. Rule based synonyms for entity extraction from noisy text. In *AND '08*, pages 31–38. ACM, 2008.

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

[6] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

[7] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. In *VLDB*, 2009.

[8] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[9] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, pages 805–818, 2008.

[10] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[12] S. Chaudhuri, V. Ganti, and D. Xin. Exploiting web search to generate synonyms for entities. In *Proc. 2009 Int. World Wide Web Conf. (WWW'09)*, 2009.

[13] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *KDD*, pages 89–98, 2004.

[14] I. Dagan, S. Marcus, and S. Markovitch. Contextual word similarity and estimation from sparse data. In *In ACL*, pages 164–171, 1993.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[16] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, pages 85–96, 2005.

[17] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, Dallas, TX, May 2000.

[18] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

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

[20] D. Lin. Automatic retrieval and clustering of similar words. In *COLING-ACL*, pages 768–774, 1998.

[21] C. Manning and H. Schuütze. Foundations of statistical natural language processing. In *The MIT Press*, 1999.

[22] M. Michelson and C. A. Knoblock. Mining heterogeneous transformations for record linkage. In *IIWeb*, pages 68–73. AAAI Press, 2007.

[23] P. D. Turney. Mining the web for synonyms: Pmi-ir versus lsa on toefl. *CoRR*, cs.LG/0212033, 2002.