Building a Scalable Database-Driven Reverse Dictionary

Ryan Shaw, *Member*, *IEEE*, Anindya Datta, *Member*, *IEEE*,
Debra VanderMeer, *Member*, *IEEE*, and Kaushik Dutta, *Member*, *IEEE*

Abstract—In this paper, we describe the design and implementation of a reverse dictionary. Unlike a traditional forward dictionary, which maps from words to their definitions, a reverse dictionary takes a user input phrase describing the desired concept, and returns a set of candidate words that satisfy the input phrase. This work has significant application not only for the general public, particularly those who work closely with words, but also in the general field of conceptual search. We present a set of algorithms and the results of a set of experiments showing the retrieval accuracy of our methods and the runtime response time performance of our implementation. Our experimental results show that our approach can provide significant improvements in performance scale without sacrificing the quality of the result. Our experiments comparing the quality of our approach to that of currently available reverse dictionaries show that of our approach can provide significantly higher quality over either of the other currently available implementations.

Index Terms—Dictionaries, thesauruses, search process, web-based services

1 Introduction and Related Work

In this paper, we report work on creating an online *reverse dictionary* (RD). As opposed to a regular (forward) dictionary that maps words to their definitions, a RD performs the converse mapping, i.e., given a phrase describing the desired concept, it provides words whose definitions match the entered definition phrase. For example, suppose a forward dictionary informs the user that the meaning of the word "spelunking" is "exploring caves." A reverse dictionary, on the other hand, offers the user an opportunity to enter the phrase "check out natural caves" as input, and expect to receive the word "spelunking" (and possibly other words with similar meanings) as output.

Effectively, the RD addresses the "word is on the tip of my tongue, but I can't quite remember it" problem. A particular category of people afflicted heavily by this problem are writers, including students, professional writers, scientists, marketing and advertisement professionals, teachers, the list goes on. In fact, for most people with a certain level of education, the problem is often not lacking knowledge of the meaning of a word, but, rather, being unable to recall the appropriate word on demand. The RD addresses this widespread problem.

The RD problem description is quite simple: given one or more forward dictionaries, how can we construct a reverse dictionary, given the following two constraints?

First, the user input is unlikely to exactly match (indeed, might differ widely from) the definition of a word in the forward dictionary. For example, a user may enter the phrase "to waste resources on unimportant things" when looking for a concept such as "fritter," whose dictionary definition might be "spend frivolously and unwisely"—which is conceptually similar, but does not contain any of the same words as the user input.

Second, the response efficiency needs to be similar to that of forward dictionary online lookups, i.e., the RD needs to be usable online. According to a recent Forrester study, end users become impatient if a website takes longer than 4-5 seconds to respond to a request [8].

Effectively, the problem is as follows: upon receiving a search *concept*, the RD consults the forward dictionary at its disposal and selects those words whose definitions are similar to this concept. These words then form the output of this RD lookup. The problem reduces to a *concept similarity* problem (CSP), which, in principle, has been addressed in a variety of fields, such as psychology, linguistics, and computer science. Specifically in computer science, concept similarity has been addressed by both Information Retrieval (IR) researchers [18], [21] as well as Database researchers [32]. Yet, it turns out that the RD concept similarity problem possesses significantly different characteristics from the concept similarity work reported in the literature.

The CSP is a well-known hard problem [18] which has been addressed in a number of ways with a limited degree of success. The *real-time, online concept similarity identification* problem we need to tackle is different from what extant CSP work addresses. In effect, one of the core contributions of this work is the development and implementation of a practical and scalable (i.e., capable of supporting online interactive applications) concept similarity measurement system. Specifically, the two problems have key differences that make direct use of existing results infeasible.

- *R. Shaw is with Google, Inc., CA 94043. E-mail: shawrc@google.com.*
- *A. Datta and K. Dutta are with the Department of Information Systems, School of Computing, National University of Singapore, Singapore 117417. E-mail: datta@comp.nus.edu.sg, duttak@nus.edu.sg.*
- *D. VanderMeer is with the Decision Sciences and Information Systems Department, College of Business, Florida International University, Miami, FL 33199. E-mail: vanderd@fiu.edu.*

Manuscript received 15 June 2010; revised 5 Sept. 2011; accepted 4 Oct. 2011; published online 19 Oct. 2011.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-06-0332. Digital Object Identifier no. 10.1109/TKDE.2011.225.

The first problem considers the unit of analysis. Estimating the semantic similarity of concepts is an important problem, well studied in the literature. Results of such studies are reported in a variety of fields, including psychology [41], natural language processing [31], information retrieval [18], [21] language modeling [17], [28], and database systems [32]. Virtually all attempts to study the similarity of concepts *model concepts as single words*. Work in text classification for instance, surveyed in detail in [35], attempts to cluster documents as similar to one another if they contain co-occurring words (not phrases or sentences). Current word sense disambiguation approaches, where researchers seek to identify the contextual meaning of a polysemous word (i.e., a word with multiple meanings) based on nearby words in a sentence where the target word appears, still consider a single word at a time. This single-word focus is well recognized in the literature [21], [15]—in [21], for instance, the authors remark: *"Measures of semantic similarity have been traditionally defined between words or concepts, and much less between text segments containing two or more words."* For a RD, semantic similarities must be computed between multiword phrases.

Some work in the literature does address multiword similarity. One area of such work [9], [23] addresses the problem of finding the similarity of multiword phrases across a set of documents in Wikipedia. However, there is a further problem (our second problem) in applying these methods directly, because the methods proposed in these works assume that the documents contain sufficient contextual information (at least 50-100 words) for similarity matching, which fits within the traditional notion of "short documents" in IR research [3], [10]. In another area of related work, i.e., passage retrieval [34], [19], [4], the concept of interest is modeled as a multiword input query. Work in this area attempts to identify relevant passages in large-text documents.

In contrast, in the reverse dictionary scenario, the "documents" considered for similarity are very short dictionary definitions (often consisting of fewer than ten words), which contain very little contextual information. The lack of contextual information in the RD case adds to the difficulty of addressing this problem space.

Finally, we consider the online responsiveness and scaling problems. A key requirement for our RD system is that its performance allows online interaction with users. Current semantic similarity measurement schemes are highly computationally intensive, making online scaling difficult [15]. Consider for instance, vectorization—perhaps the most effective technique applied to similarity measurement. In this technique, concepts are represented as vectors in a feature (or keyword) space. The two most common methods to achieve this, *latent semantic indexing* (LSI) [11] and *principal component analysis* (PCA) [16], both analyze the keywords of documents in a corpus to identify the *dominant concepts* in the document. Subsequently these dominant concepts are represented as vectors in the keyword space and are used as the basis of similarity comparison for classification. Such vectorization is a highly computationally intensive operation, as is the comparison of the distance between two vectors (using techniques like cosine distances [16]). Typically, in most implementations of CSP solutions, vectorization is done a priori, and at runtime, only vector distances are computed. The computational intensity of this distance computation itself renders such solutions infeasible to reside behind online applications.

In the RD case, since our concepts are not known beforehand, particularly in the case of user inputs, *we need to compute the user input concept vector at runtime* and then subsequently compute the distance between this and the dictionary concept vectors (where these vectors can be computed a priori). Vector computation is known to be quite compute intensive [36], even though much effort has been expended in building efficient schemes. The results of this effort have been several published variants of LSI, such as pLSI [12] and *Latent Dirichlet Allocation* (LDA) [2]. Researchers have attempted to enhance the scale of the efficient variants by creating parallelized and distributed avatars [24]. Yet, even in the highest performing vectorization schemes, even for modest document sizes ($\approx 50{,}000$), response times are still on the order of multiple tens of seconds. In our case, we are looking at approximately 200,000 words and for corpora of that size, it appears infeasible that online speed or scale might be achieved with even state-of-the-art existing methods.

In this paper, we report the creation of the Wordster Reverse Dictionary (WRD), a database-driven RD system that attempts to address the core issues identified above. The WRD not only fulfils new functional objectives outlined above, *it does so at an order of magnitude performance and scale improvement* over the best concept similarity measurement schemes available *without impacting solution quality*. We also demonstrate that the WRD is far better in solution quality than the two commercial RDs [26], [6] available.

The remainder of this paper is structured as follows: In Section 2, we describe our solution approach and in Section 3, we describe the architecture of the WRD system. In Section 4, we analytically demonstrate the scalability of our approach. In Section 5, we present the results of our experiments, including both quality and performance results. Finally, we conclude in Section 6.

2 Proposed Solution Approach

Intuitively, our reverse dictionary system is based on the notion that a phrase that conceptually describes a word should resemble the word's actual definition, if not matching the exact words, then at least conceptually similar. Consider, for example, the following concept phrase: "talks a lot, but without much substance." Based on such a phrase, a reverse dictionary should return words such as "gabby," "chatty," and "garrulous." However, a definition of "garrulous" in a dictionary might actually be "full of trivial conversation," which is obviously close in concept, but contains no exact matching words.

In our RD, a user might input a phrase describing an unknown term of interest. Since an input phrase might potentially satisfy the definition of multiple words, a RD should return a set of possible matches from which a user may select his/her choice of terms. This is complex, however, because the user is unlikely to enter a definition that exactly matches one found in a dictionary. However,

the meaning of the phrase the user entered should be conceptually similar enough to an actual dictionary definition to generate a set of possible matches, e.g., returning to the "talks a lot, but without much substance" example, our reverse dictionary should return words like "garrulous." How can we generate such potential matches? How can we do this such that the system can provide online responsiveness?

One approach might be to compare the user input phrase to every definition in a dictionary, looking for definitions containing the same words as the user input phrase. Such an approach has two major problems: 1) it requires the user's input phrase to contain words that exactly match a dictionary definition; and 2) it does not scale well—for a dictionary containing more than 100,000 defined words, where each word may have multiple definitions, it would require potentially hundreds of thousands of queries to return a result.

As a first step, we can improve the efficiency of the simple method proposed above by reducing the set of definitions we compare to the user input phrase by *knowing which definitions a given word appears in*. For example, suppose that a user input phrase contains the word "carnival." If we know which dictionary definitions the word "carnival" appears in, i.e., if we have an index that maps from a word to all the dictionary words in whose definitions it appears, we can use such an index to limit the set of definitions we compare to the input phrase to only those definitions containing the word "carnival." We call such an index a *reverse mapping*, since it is designed to support a reverse dictionary. This is a form of inverted index [1], [39], where we incorporate stemming to ensure that we use the most generic form of a word possible. For example, if the word "spirited" occurred in a definition, then in the reverse mapping we use the word "spirit." This removes the possibility of concepts that are actually similar, but are not found to be similar based on differing grammatical usage.

This optimization, though useful, only takes us so far. The user's input phrase must still exactly match words in a dictionary definition, which is very unlikely. We can extend the input phrase to consider conceptually similar words using a form of query expansion [20] in order to improve the potential that relevant definitions will be identified. For example, if the user's input phrase contains the word "carnival," relevant definitions containing the conceptually similar word "festival" would not be returned. If instead we considered the set of words where "carnival" or "festival" appear in definitions, we can increase the probability of identifying relevant definitions.

These two approaches form the basic framework of our method, which we discuss in detail in the remainder of this section. We first define several concepts that will be useful in describing our methods.

We define the notion of a *term* t as any legitimate word in the English language, e.g., "cow," "boy," "reading," "jargon," and "merrily" are examples of terms. We further define a phrase P as a sequence of one or more terms, i.e., $P = <t_1, t_2, ..., t_i, ...t_n>$. For convenience, we designate the fact that term t is part of the phrase P by the notation $t \in P$ (even though P is not technically a set).

A dictionary D is a set of mappings $P \rightarrow P$. For clarity, we will distinguish between the two classes of phrases, word phrases (W) and sense phrases (S), where W refers to a word or sequence of words indexed for lookup in a dictionary, and S refers to a definition of a W in a dictionary. Under this classification, a dictionary D can be redefined as a set of mappings $W \rightarrow S$. In particular, the mapping $W_i \rightarrow S_j$ expresses the following relationship: *S_j denotes a sense (or meaning) of the (word) phrase W_i*. For example, consider the following dictionary mapping: "to retreat" $\rightarrow$ "to turn back in defeat." This denotes the fact that the meaning of the (word) phrase "to retreat" is expressed the (sense) phrase "to turn back in defeat."

Forward mapping (standard dictionary): Intuitively, a forward mapping designates all the senses for a particular word phrase. This is expressed in terms of a forward map set (FMS). The FMS of a (word) phrase W, designated by $\mathcal{F}(W)$ is the set of (sense) phrases $\{S_1, S_2, \ldots S_x\}$ such that for each $S_j \in \mathcal{F}(W_i)$, $(W_i \rightarrow S_j) \in D$. For example, suppose that the term "jovial" is associated with various meanings, including "showing high-spirited merriment" and "pertaining to the god Jove, or Jupiter." Here, $\mathcal{F}(jovial)$ would contain both of these phrases.

Reverse mapping (reverse dictionary): Reverse mapping applies to terms and is expressed as a reverse map set (RMS). The RMS of t, denoted $\mathcal{R}(t)$, is a set of phrases $\{P_1, P_2, \ldots, P_i, \ldots, P_m\}$, such that $\forall P_i \in \mathcal{R}(t), t \in \mathcal{F}(P_i)$. Intuitively, the reverse map set of a term t consists of all the (word) phrases in whose definition t appears. For example, suppose forms of the word "spirit" appear the definitions of both "languid," which might be defined as "lacking spirit or liveliness" and "jovial," which might be defined as "showing high-spirited merriment." Here, $\mathcal{R}(spirit)$ would include both "languid" and "jovial."

We note that we consider the words "spirit" and "spirited" as conceptually equivalent for the purposes of concept matching. We do this by running each word through a standard stemming algorithm, e.g., the Porter stemmer [29], which reduces each word to its base form by removing common modifications for subject-verb agreement, or variation in parts of speech (e.g., "standard," "standards," "standardizing," "standardization"). We denote the output of stemming a term as $\hat{t}$. Further, we maintain a count of definitions in which a stemmed term $\hat{t}$ appears, denoted $N(\hat{t})$.

As we noted above, a user is unlikely to enter an exactly matching definition, but should be able to enter something conceptually similar to the dictionary meaning. Thus, we need ways to describe how words can be conceptually related to one another. We defines several types of relatedness below.

Synonym set: A set of conceptually related terms for t. $\mathcal{W}_{syn}(t) = \{t_1, t_2, \ldots t_j, \ldots, t_n\}$, where t_j is a synonym of t, as defined in the dictionary. For example, $\mathcal{W}_{syn}(talk)$ might consist of the set of words { speak, utter, mouth, verbalize}.

Antonym set: A set of conceptually opposite or negated terms for t. $\mathcal{W}_{ant}(t) = \{t_1, t_2, \ldots t_j, \ldots, t_n\}$, where each t_j is an antonym of t, as defined in the dictionary, e.g., $\mathcal{W}_{ant}(pleasant)$ might be consist of {"unpleasant," "unhappy"}.

Hypernym set: A set of conceptually more general terms describing t. $\mathcal{W}_{hyr}(t) = \{t_1, t_2, \ldots t_j, \ldots, t_n\}$, where t_j is a hypernym of t, as defined in the dictionary. For example, $\mathcal{W}_{hyr}(red)$ might consist of {"color"}.

Hyponym Set: A set of conceptually more specific terms describing t. $\mathcal{W}_{hyo}(t) = \{t_1, t_2, \ldots t_j, \ldots, t_n\}$, where t_j is a hyponym of t, as defined in the dictionary. For example, $\mathcal{W}_{hyo}(red)$ might consist of {"maroon," "crimson"}.

Forward mapping sets, as well as synonym, antonym, hypernym, and hyponym sets can be retrieved from an existing corpus. In this paper, we draw these sets from the WordNet [22] database.

User phrase: A user phrase U is a phrase defined by a sequence of terms $<t_1, \ldots, t_k, \ldots, t_x>$ input by a user. This string serves as input to our method.

Given an input U, we note that some common words are not useful for the purposes of indexing. Such words are called *stop words* [33], and are typically removed from indexes and queries over indexes.

Stop word sets: We define two sets of stop words, *Level 1 stop words* (L_1), which are always removed during index building and querying, and *Level 2 stop words* (L_2), which may or may not be useful in indexing. An example of an area where a word may or may not be useful is gender, because the gender is important in some cases but not others. For example, the input "man who is married" should logically return the word "husband," but not "wife;" however, the input "man who cares for patients" should return both "physician" and "nurse."

Negation word set: Negation is also a concern. For example, a user might enter an input phrase using the word "not," e.g., when an antonym of the negated term would be more precise. We wish to identify such cases, We define a set of negation words, denoted G.

Query: A query phrase Q is a Boolean expression based on an input U.

Output: The output O of our method consists of a set of Ws, such that a definition of each W in the output satisfies Q.

Table 3 provides examples of level 1 and level 2 stop words, as well as negation terms, and Table 4 summarizes the notation used throughout the remainder of the paper, respectively.

Having described our notational building blocks, we now move on to describe the details of our reverse dictionary creation techniques. The details of this conceptual similarity discovery follow.

2.1 Solution Overview

At a high level, our approach consists of two sequential steps. Upon receipt of a user input phrase, we first *find candidate words* from a forward dictionary data source, where the definitions of these candidate words have some similarity to the user input. We then *rank the candidate words* in order of quality of match. The *find candidate words* phase consists of two key substeps: 1) build the RMS; and 2) query the RMS. We now describe the details of our approach.

2.2 Building Reverse Mapping Sets

Intuitively, building the RMS of a term t, $\mathcal{R}(t)$, is a matter of finding all Ws in whose definition t appears. Given the large size of dictionaries, creating such mappings on the fly is infeasible. Thus, we precreate these $\mathcal{R}$s for every relevant term in the dictionary. This is a one time, offline event; once these mappings exist, we can use them for ongoing lookup. Thus, the cost of creating the corpus has no effect on runtime performance.

For an input dictionary D, we create $\mathcal{R}$ mappings for all terms appearing in the sense phrases (definitions) in D. Algorithm 1 describes this process. We build $\mathcal{R}$ indexes by iterating through all the terms (or Ws) in D. For each W defined in D, we consider each sense phrase S_j, where $(W_i \rightarrow S_j)$ (line 2). For each term t appearing in S_j (lines 3-4), we first determine the most generic form of t by applying stemming to t (line 5). This results in a stemmed form of t: $\hat{t}$. We then place W in $\mathcal{R}(\hat{t})$, and increment the count of appearances $N(\hat{t})$ (line 6).

Algorithm 1. `BuildRMS`

1: **Input:** a dictionary D.
2: **for all** mapping $W_i \rightarrow S_j \in D$ **do**
3: Extract the set of terms $\{t_1 \ldots t_x\}, t_k \in SP_j$
4: **for all** t_k **do**
5: Apply stemming to convert t_k to convert it to its general form $\hat{t_k}$
6: Add W_i to $\mathcal{R}(\hat{t_k})$ and increment $N(\hat{t_k})$ by 1.

2.3 Querying the Reverse Mapping Sets

In this section, we explain how we use the $\mathcal{R}$ indexes described in Section 2.2 to respond to user input phrases. Upon receiving such an input phrase, we query the $\mathcal{R}$ indexes already present in the database to find candidate words whose definitions have any similarity to the input phrase. We explain the intuition behind this step with an example. Given an input phrase "a tall building," we first extract the core terms present in this phrase: "tall" and "building" (the term "a" is ignored, since it is a stop word). We then consult the appropriate $\mathcal{R}$ indexes, $\mathcal{R}(tall)$ and $\mathcal{R}(building)$, to find those words in whose definitions the words "tall" and "building" occur simultaneously. Each such word becomes a candidate word.

Two additional issues are important for the reader to understand. First, the fact that two terms occur simultaneously in a word W does not imply that W should be part of the final output. At this step, we are only concerned with finding candidate words. In the ranking phase, we will test candidate words' definitions for quality of match with the input phrase. Second, different dictionaries might use different terms to define the same W. For instance, a dictionary might use the word "high" rather than "tall," and "edifice" instead of "building." So, in this candidate identification step, we consider not only the terms extracted from the user input phrase, but also terms similar to them in meaning.

Upon receiving an input phrase U, we process U using a stepwise refinement approach. We start off by extracting the core terms from U, and searching for the candidate words (Ws) whose definitions contain these core terms exactly. (Note that we tune these terms slightly to increase the probability of generating Ws) If this first step does not generate a sufficient number of output Ws, defined by a tuneable input parameter α, which represents the minimum number of word phrases needed to halt processing and

return output, we then broaden the scope of Q to include, in turn, the synonyms, hyponyms, and hypernyms of the terms in Q. If these steps still do not generate a sufficient α number of output Ws, we remove terms from Q to increase the scope of possible outputs until either a sufficient number of output Ws has been generated, or Q contains only two terms.

When the threshold number of Ws α has been reached, we sort the results based on similarity to U, and return the top β Ws, where β is a tuneable input parameter defining the maximum number of Ws to include in returned results. The details of sorting and ranking Ws are described in Section 2.4.

Algorithms 2, 3, 4, 5, and 6 implement the above procedure. We begin our discussion with the `GenerateQuery` algorithm (Algorithm 2), which controls the overall process. It takes as input U and minimum and maximum output thresholds α and β, respectively, and expands the query in a stepwise fashion until a sufficient number of W outputs have been generated. We next describe the steps in Algorithm 2 in detail.

Algorithm 2. `GenerateQuery`(U, α, β)
1: $U = U$ s.t. $\forall t_i \in U, t_i \notin L_1$
2: Form a boolean expression Q by adding all $t_i \in U$ to Q, separated by `AND`
3: $Q = Q$ `OR ExpandAntonyms`(Q)
4: **for all** $t_i \in U$ **do**
5: Apply stemming to t_i to obtain $\hat{t}_i$
6: Replace t_i in Q with (t_i `OR` $\hat{t}_i$)
7: Reorder terms in Q s.t. all nouns appear before verbs, and verbs before adjectives and adverbs
8: $O =$ `ExecuteQuery`(Q)
9: **if** $|O| > \alpha$ **then**
10: Return `SortResults`(O)
11: **for all** $t_i \in Q$ s.t. $t_i \in L_2$ **do**
12: Remove t_i from Q
13: $O = O \cup$ `ExecuteQuery`(Q)
14: **if** $|O| > \alpha$ **then**
15: Return `SortResults`(O)
16: $O = O \cup$ `ExpandQuery`(Q, "synonyms")
17: **if** $|O| > \alpha$ **then**
18: Return `SortResults`(O)
19: $O = O \cup$ `ExpandQuery`(Q, "hyponyms")
20: **if** $|O| > \alpha$ **then**
21: Return `SortResults`(O)
22: $O = O \cup$ `ExpandQuery`(Q, "hypernyms")
23: **if** $|O| > \alpha$ **then**
24: Return `SortResults`(O)
25: Create a list l of all terms $t_i \in Q$
26: Sort l in descending order based on $N(t_i)$
27: **while** $|Q > 2|$ **do**
28: Delete t_i from Q, where t_i has the highest value of $N(t_i)$ in l
29: $O = O \cup$ `ExecuteQuery`(Q)
30: **if** $|O| > \alpha$ **then**
31: Return `SortResults`(O)
32: **else**
33: Delete t_i from l

Algorithm 3. `ExecuteQuery`(Q)
1: Given: a query Q of the form $Q = t_1 \odot_1 t_2 \odot_2 t_3 \ldots t_{k-1} \odot_{k-1} t_k$, where $\odot_i \in \{\texttt{AND}, \texttt{OR}\}$
2: $O_e = \mathcal{R}(t_1) \otimes_1 \mathcal{R}(t_2) \otimes_2 \mathcal{R}(t_3) \ldots \mathcal{R}(t_{k-1}) \otimes_{k-1} \mathcal{R}(t_k)$, where if $\odot_i =$ `AND`, $\otimes_i = \cap$ and if $\odot_i =$ `OR`, $\otimes_i = \cup$
3: return O_e

Algorithm 4. `ExpandAntonyms`(Q)
1: Given: a query Q of the form $Q = t_1 \odot_1 t_2 \odot_2 t_3 \ldots t_{k-1} \odot_{k-1} t_k$, where $\odot_i \in \{\texttt{AND}, \texttt{OR}\}$
2: Create a new query Q', where Q' is a copy of Q
3: **for all** negated $t_i \in Q$ **do**
4: $F = \mathcal{W}_{ant}(t_i)$
5: **if** $F \neq \emptyset$ **then**
6: Create a new subquery $q = \emptyset$
7: **for all** $t_j \in F$ **do**
8: Add t_j to q, connected to existing terms with `OR`
9: Replace t_i and the word that negates it in Q' with q
10: **if** $Q' \neq Q$ **then**
11: Return Q'
12: **else**
13: Return $\emptyset$

Algorithm 5. `ExpandQuery`(Q, SetType)
1: Given: a query Q of the form $Q = t_1 \odot_1 t_2 \odot_2 t_3 \ldots t_{k-1} \odot_{k-1} t_k$, where $\odot_i \in \{\texttt{AND}, \texttt{OR}\}$
2: **for all** $t_i \in Q$ **do**
3: **if** SetType is "synonyms" **then**
4: $F = \mathcal{W}_{sym}(t_i)$
5: **if** SetType is "hyponyms" **then**
6: $F = \mathcal{W}_{hyo}(t_i)$
7: **if** SetType is "hypernyms" **then**
8: $F = \mathcal{W}_{hyr}(t_i)$
9: Create a new subquery $q = (t_i)$
10: **for all** $t_j \in F$ **do**
11: Add t_j to q, connected with `OR`
12: Replace t_i in Q with q
13: Return `ExecuteQuery`(Q)

Algorithm 6. `SortResults`(O, U)
1: Create an empty list K
2: **for all** $W_j \in O$ **do**
3: **for all** $S_k \in W$ **do**
4: **if** $\exists t_i \in W_j$ s.t. $t_i \in L_1$ **then**
5: Remove t_i from W_j
6: Compute $Z(S)$ and $Z(U)$
7: **for all** pairs of terms (a, b), where $a \in Z(S)$ and $b \in Z(U)$ **do**
8: **if** a and b are the same part of speech in $Z(S)$ and $Z(U)$, respectively **then**
9: Compute $\rho(a, b) = \frac{2 \times E(A(a,b))}{(E(a)+E(b))}$
10: Compute $\lambda(a, S) = \frac{(d(Z(S))-d_a)}{d(Z(S))}$
11: Compute $\lambda(b, U) = \frac{(d(Z(U))-d_b)}{d(Z(U))}$
12: Compute $\mu(a, S, b, U) = \lambda(a, S) \times \lambda(b, U) \times \rho(a, b)$
13: Use $\mu(a, S, b, U)$ values to measure the phrase

similarity $M(S,U)$ of S and U following the algorithm described in [5].
14: Insert the tuple (S,U,M) in to the list K
15: Sort the tuples in K in descending order based on the value of M
16: For the top β word senses in K, return the corresponding words

We first remove all L_1 stop words (line 1). Since these stop words do not add specificity to the input, removing them does no harm. We next generate a query Q by connecting all the terms $t_i \in U$ with AND (line 2).

We next consider instances where a negation word (as listed in the examples in Table 3) modifies a term $t_i \in Q$. We note that if there are any antonyms in $\mathcal{R}_{ant}(t_i)$, these terms might help to generalize Q and improve the result set. For example, "not pleasant" can be expanded to include conceptually similar terms "unpleasant" or "unhappy." In line 3, we call the ExpandAntonyms algorithm (described in Algorithm 4 below), which creates a copy of Q expanded to include antonyms of negated terms, and concatenate the result to Q with OR. For example, given $Q =$ ("wetness" AND "not" AND "pleasant"), ExpandAntonyms will return ("wetness" AND ("unpleasant" OR "unhappy")). We combine this result with Q to form (("wetness" AND "not" AND "pleasant") OR ("wetness" AND ("unpleasant" OR "unhappy"))).

Next, we apply stemming to all terms $t_i \in Q$ (lines 4-6), to include the most general form of each t_i in Q.

In line 7, we reorder the query such that all nouns appear first, then verbs, then adjectives and adverbs. Since nouns and verbs represent more concrete concepts than adjectives or adverbs, they have smaller average mapping sets than adjectives and adverbs. By starting set comparison processing with the items with smaller average set sizes, we reduce the number of set operations required, as compared to orderings that process adjectives and adverbs earlier.

In lines 8-10, we evaluate Q by calling ExecuteQuery. If this generates sufficient output, we call SortResults to sort the results by similarity to the U, and return the sorted list. We describe ExecuteQuery and SortResults next.

If sufficient results are not generated, we expand Q to consider synonyms (lines 16-18), hyponyms (lines 19-21), and hypernyms (lines 22-24) using the ExpandQuery algorithm (described in detail below). We evaluate Q at each stage to determine if processing needs to continue to gather further results (lines 17, 20, and 23).

If we still do not have sufficient results, we begin to generalize Q by successively removing a single term t_i from Q, in descending order of incidence in definitions of Ws, i.e., in decreasing order of $N(t_i)$. This removes the more common words first, based on the assumption that the least commonly occurring words will be the most important in finding suitable output Ws. We evaluate Q at each step, and continue until we have sufficient output, or until Q contains only two terms.

2.3.1 Executing a Query

The ExecuteQuery algorithm, shown in Algorithm 3, takes any generic Q as input, where Q is a Boolean expression consisting of terms $t_1 \ldots t_k$, and produces an output set O_e, which contains a set of Ws.

To evaluate Q, we take the output of $\mathcal{R}(t_i)$ for each term $t_i \in Q$, and merge the $\mathcal{R}$ sets for two terms based on how they are connected together in Q. If the two terms are connected with AND, we merge the corresponding $\mathcal{R}$ sets using set-intersection. If the two terms are connected with OR, we merge the corresponding $\mathcal{R}$ sets using set-union.

For example for the $Q =$ ("wetness" AND "not" AND "pleasant") OR ("wetness" AND ("unpleasant" OR "unhappy")), $O_e = (\mathcal{R}(\text{wetness}) \cap \mathcal{R}(\text{not}) \cap \mathcal{R}(\text{pleasant})) \cup (\mathcal{R}(\text{wetness}) \cap (\mathcal{R}(\text{unpleasant}) \cup \mathcal{R}(\text{unhappy}))$.

2.3.2 Replacing Negation with Antonyms

The ExpandAntonyms algorithm, described in Algorithm 4, takes a query Q, and returns a copy of Q, where each negated term t_i is expanded to include its antonyms. We first create a copy of Q, called Q' that will be modified to include antonyms of negated terms (line 2). Then, for each negated term t_i in Q, we find the set of antonyms of t_i, called F (line 4). If there exist any antonyms for t_i, we create a new Boolean expression q (line 6) that contains all the antonyms of t_i connected by OR (lines 7-8), and we replace t_i and the word that negates it in Q' (line 9). This generates an alternate version of Q containing antonyms rather than negated terms, which we return (lines 10-13).

For example, if query $Q =$ ("wetness" AND "not" AND "pleasant"). In this case the negation word "not" is before the adjective "pleasant." The antonym set of pleasant is $\mathcal{W}_{ant}(pleasant) = \{\text{"unpleasant," "unhappy"}\}$, and $q =$ ("unpleasant" OR "unhappy").

2.3.3 Expanding a Query

The ExpandQuery algorithm, described below in Algorithm 5, returns a set of output Ws, based on an expansion of Q to include conceptually similar terms of type SetType, which can be "synonyms," "hyponyms," or "hypernyms." We expand each term in Q in turn. For each $t_i \in Q$, we find the appropriate set of related terms, using $\mathcal{W}_{syn}(t_i)$ for synonyms (lines 3-4), $\mathcal{W}_{hyo}(t_i)$ for hyponyms (lines 5-6), and $\mathcal{W}_{hyr}(t_i)$ for hypernyms (lines 6-7). We then create a new subquery q, initially containing only t_i, and concatenate the related terms to q using OR (lines 9-11). We replace t_i in Q with q, evaluate Q by running ExecuteQuery, and return the results (line 13).

For example, consider the case of synonyms. If $Q =$ ("talk" AND "much"), $\mathcal{W}_{syn}(talk) = \{\text{speak, utter, mouth, verbalize}\}$ and $\mathcal{W}_{syn}(much) = \{\text{very, often, lot, excessively}\}$, we evaluate the following boolean expression containing expanded synonyms: (("talk" OR "speak" OR "utter" OR "mouth" OR "verbalize") AND ("much" OR "very" OR "often" OR "lot" OR "excessively")). Similar logic applies to hyponyms and hypernyms.

2.4 Ranking Candidate Words

The SortResults algorithm, shown in Algorithm 6, sorts a set of output Ws in order of decreasing similarity to U, based on the semantic similarity of the definitions $S_1 \ldots S_x$ of W (i.e., $\mathcal{F}(W)$) as compared to U.

To build such a ranking, we need to be able to assign a similarity measure for each (S,U) pair, where U is the user

TABLE 1
Example Parser Output for P_1

```
(ROOT
  (S
    (NP
      (NP (DT the) (NN dog))
      (SBAR
        (WHNP (WP who))
        (S
          (VP (VBD bit)
            (NP (DT the) (NN man)))))))))
```

TABLE 2
Example Parser Output for P_2

```
(ROOT
  (S
    (NP
      (NP (DT the) (NN man))
      (SBAR
        (WHNP (WP who))
        (S
          (VP (VBD bit)
            (NP (DT the) (NN dog)))))))))
```

input phrase and S is a definition for some W in the candidate word set O. If simply considering the pairwise similarity of words in two phrases could produce a meaningful measure of similarity, we could directly apply the work in [5]. We note, however, that the sequence of words in a phrase is important, and that some words contribute more to the meaning of a phrase than others. Consider two phrases: "the dog who bit the man" and "the man who bit the dog." They contain the same words, so the method in [5] would consider them virtually equivalent, but they convey very different meanings.

Thus, we need measures of both *term similarity* across two terms generally, and *term importance*, i.e., how critical the term is in the context of the phrase of which it is a part. If we can build a measure of similarity that is weighted by the importance of each term in the context of the phrase where it occurs, then we can leverage the similarity framework suggested in [5] with these importance measures. Specifically, we can generate a weighted similarity measure for each pair of terms (a, b), where $a \in S$ and $b \in U$, such that if a and b are conceptually similar, and they are both important in the phrases in which they occur, then the phrase pair (S, U) would be measured to be more similar than another pair of phrases (S, U'), where U and U' contain the same set of terms, but $a \in S$ is more important to the meaning of S than $a \in S'$ is to the meaning of S'.

We approach the problem of building such a weighted similarity measure by first developing a measure of general conceptual similarity *term similarity*, $\rho(a, b)$, between two terms a and b. We then develop a measure of *term importance*, $\lambda(t, P)$, for a term t in a phrase P. For example, $\lambda(a, S)$ describes the importance of a in the phrase S and $\lambda(b, U)$ describes the importance of b in the phrase U. We combine these measures by defining $\mu(a, S, b, U)$, which is a single importance-weighted similarity measure.

We first consider *term similarity*. We compute the similarity $\rho(a, b)$ (based on [40]) for two terms a and b based on their locations in the WordNet hierarchy, where this hierarchy organizes words in the English language from the general at the root to the most specific at the leaf nodes. Intuitively, two terms have very little similarity if their least common ancestor (LCA) in this hierarchy is the root, and increasingly greater similarity the deeper their LCA is in this hierarchy.

We define several functions for use in computing the similarity of two terms $\rho(a, b)$. $A(a, b)$ returns the least common ancestor shared by both a and b in the WordNet hierarchy. $E(t)$ returns the depth of a term t in the WordNet hierarchy. Using these two functions, we define a similarity function that describes how similar two terms are in the English language:

$$\rho(a, b) = \frac{2 \times E(A(a, b))}{(E(a) + E(b))}. \quad (1)$$

That is, $\rho(a, b)$ will be larger the closer a and b are to their LCA in the hierarchy.

We next consider the notion of *term importance*, $\lambda(t, P)$, for a term t in a phrase P. We return to the example of the two phrases: $P_1 =$ "the dog who bit the man" and $P_2 =$ "the man who bit the dog." Here, the subject of the phrase is intuitively more important than the object, i.e., in P_1, "dog" is more important than "man," and in P_2, "man" is more important than "dog." If we were to add adjectives or adverbs, these would add detail, but be less critical to the meaning of the phrase than the subject or object. For example, the phrase $P_3 =$ "the agitated dog who bit the man" differs very little in basic meaning from P_1—we have just added a bit of information that might explain why the dog bit the man.

To generate the importance of each term, we can use a *parser*, e.g., OpenNLP [30], that can return the grammatical structure of a sentence. Given an input phrase, such a parser returns a *parse tree*, where the words in the input phrase that add the most to the meaning of the phrase appear higher in the parse tree than those words that add less to the meaning of the input phrase. The details of this how this parser works are beyond the scope of this paper, but we note that the output representation is similar to a functional programming language with part-of-speech notation as described in [25]. Table 1 shows the parser output $Z(P_1)$ for the phrase $P_1 =$ "the dog who bit the man," while Table 2 shows the parser output $Z(P_2)$ for the phrase $P_2 =$ "the man who bit the dog."

Here, we can clearly see that the word "dog" is more important than the word "man" in P_1, and less important

TABLE 3
Examples of Stop and Negation Words

Level 1 stop words	a, be, person, some, someone, too, very, who, the, in, of, and, to, that, for, with, this, from, which, when, what, than, into, these, where, those, how, during, without, upon, toward, among, although, whether, else, anyone, beside, whose, whom, onto, anybody, whenever, whereas
Level 2 stop words	woman, female, male
Negation words	lack, lacking, never, no, non, none, not, seldom, without

TABLE 4
Notation

Notation	Meaning
t	any legitimate word in the English language
$\hat{t}$	stemmed (generic) version of t
P	a sequence of one or more terms
D	a set of mappings $P \rightarrow P$
W	a word phrase
S	a sense phrase
$\mathcal{F}(W)$	a forward dictionary, i.e., a set of mappings $W \rightarrow S$
$\mathcal{R}(t)$	reverse mapping for t, i.e., all the Ws that include t in their definitions
$N(\hat{t})$	count of definitions in which a stemmed term $\hat{t}$ appears
$\mathcal{W}_{syn}(t)$	set of synonyms of t
$\mathcal{W}_{ant}(t)$	set of antonyms of t
$\mathcal{W}_{hyr}(t)$	set of hypernyms of t
$\mathcal{W}_{hyo}(t)$	set of hyponyms of t
U	a user input phrase
G	set of negation words
Q	boolean expression query, based on U
L_1	the set of level 1 stop words
L_2	the set of level 2 stop words
O	set of output WPs satisfying Q
α	minimum threshold number of $WP \in O$ required to stop processing
β	maximum number of WP to include in output
$\rho(t_1, t_2))$	term similarity of the terms t_1 and t_2
$\lambda(t, P)$	importance of t in P
$Z(P)$	parse tree of a phrase P
d_t	depth of t in a parse tree
$d(Z(P))$	overall depth of a parse tree for P

than the word "man" in P_2. Thus, for a parse tree $Z(P)$, we can use the depth d_t of a term $t \in P$ in $Z(P)$, where $Z(P)$ has a total depth $d(Z(P))$, to generate a measure of the importance of t to the meaning of P. We define a term importance function $\lambda(t, P)$, for a term t in a phrase P as follows:

$$\lambda(t, P) = \frac{(d(Z(P)) - d_t)}{d(Z(P))}, \tag{2}$$

where $\lambda(t, P)$ produces a larger value, the higher t appears in the parse tree $Z(P)$.

Finally, we define a weighted similarity factor $\mu(a, S, b, U)$ where $a \in S$ and $b \in U$ as follows:

$$\mu(a, S, b, U) = \lambda(a, S) \times \lambda(b, U) \times \rho(a, b). \tag{3}$$

We use this weighted similarity factor $\mu(a, S, b, U)$ to generate a weighted similarity measure for each term pair (a, b) where $a \in S$ and $b \in U$. We can then use this weighted similarity matrix as input to the generic string similarity algorithm described in [5] to obtain a phrase similarity measure $M(S, U)$ for a sense phrase S and a user input phrase U. We can then rank the elements of the candidate set O by similarity to U, and return the β best matches.

3 Solution Architecture

We now describe our implementation architecture, with particular attention to design for scalability. Fig. 1 presents the architecture of the system. The Reverse Dictionary Application (RDA) is a software module that takes a user phrase (U) as input, and returns a set of conceptually related words as output.

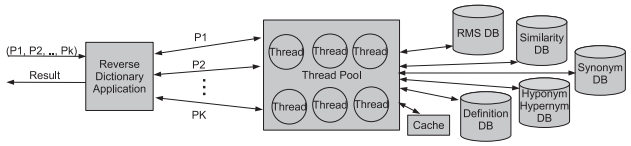

Fig. 1. Architecture for reverse dictionary.

To perform the processing described in Section 2, the RDA needs access to information stored in a set of databases:

1. the *RMS DB*, which contains a table of mappings $\hat{t} \rightarrow \mathcal{R}(\hat{t})$, as well as dictionary definitions and computed parse trees for definitions;
2. the *Synonym DB*, which contains the synonym set for each term $\hat{t}$;
3. the *Hyponym/Hypernym DB*, which contains the hyponym and hypernym relationship sets for each term $\hat{t}$;
4. the *Antonym DB*, which contains the antonym set for each term $\hat{t}$; and
5. the actual dictionary definitions for each word in the dictionary.

The mappings for the RMS, synonyms, hyponyms, hypernyms, and antonyms are stored as integer mappings, where each word in the Wordnet dictionary is represented by a unique integer. This both condenses the size of the mapping sets, and allows for very fast processing of similarity comparisons, as compared to string processing.

This architecture has three characteristics designed to ensure maximum scalability of the system. First, a cache stores frequently accessed data, which allows a thread to access needed data without contacting a database. It is well known that some terms occur more frequently than others. The synonym, hyponym, hypernym, and RMS sets of these popular terms will be stored in the cache and the query execution in the database will be avoided. Second, the implementation of a thread pool allows for parallel retrieval of synonym, hyponym, hypernym, and RMS sets for terms. Third, separate databases increase the opportunity for parallel processing, and increase system scalability. If a single machine is not capable of handling the necessary loads, the database can easily be further distributed across multiple servers using partitioning methods to improve overall system scalability.

When the RDA needs word-related data, it delegates the job of retrieving this data to a thread in a thread pool. The thread first checks the local cache to determine whether the appropriate data exist in the cache. If so, the cache returns the needed data. If not, the cache returns a null set. If the thread receives a null set from the cache, it contacts the appropriate database to obtain the needed data.

Having described the architecture, we estimate the cache size for the data for the most frequently used data types, i.e., RMS sets, synonyms, hyponyms, hypernyms, and antonyms. If we can store these data objects entirely in cache (using MD5 hashing) in a reasonable memory size, then the `GenerateQuery` algorithm can execute entirely on in-memory data objects until it calls `SortResults`,

which requires accessing full definitions (sense phrases) for each candidate $W \in O$.

We first describe the cache overhead for a single cache entry, and then consider the overall cache sizes for each of the mappings used in our method. Rather than representing a word by its string in the mappings for reverse mappings, synonym mappings, etc., we use integers (a unique 4-byte integer is assigned to each word). We implement the cache with a Java HashMap, where each cache entry contains one integer word-id, and is associated with a key and other HashMap-related data structures. For small stored values (and integers are certainly small values), a Java HashMap with an integer key is associated with overheads of approximately 2.5 times (including the space required for the key). For a small 4-byte stored integer value, this amounts to a total of 10 bytes for each cached entry (i.e., 6 bytes of overhead).

Cache size for synonyms: On average, each of WordNet's 155,287 words has 3.63 synonyms. To store the synonym words for the WordNet corpus, we need a hashtable of size $15{,}5287 \times 3.63 \times 10$, i.e., 7.2 MB.

Cache size for antonyms: On average, each WordNet word has 0.045 antonyms. This gives us an in-memory hashtable size of $15{,}5287 \times 0.045 \times 10$, i.e., 69 KB, to store all antonym sets.

Cache size for RMS: Based on the RMS sets we have built on the WordNet corpus, we have found that, on average, each term has an RMS set of 10 words. Thus, storing the RMS data objects in cache will require $15{,}5287 \times 10 \times 10$, i.e., 15.5 MB to store the RMS maps for the entire WordNet dictionary. Note that `ExecuteQuery` queries the RMS, so such caching will help to enhance the performance of the system, whereas the actual definitions, which are not accessed until the `SortResults` step, are kept in the database.

Cache size for Hyponym-Hypernym database: For each word in the WordNet dictionary, we need to store on average 2.74 hyponyms, 2.75 hypernyms, and the depth of the word in the WordNet hierarchy. To cache this information in a hashtable structure will require $15{,}5287 \times (2.75 + 2.74) \times 10 + 155{,}287 \times 4$, i.e., 9.2 MB.

Thus, with a cache size of $7.2 + 0.069 + 15.5 + 9.2 = 32$ MB, we can cache all the data required to run `GenerateQuery` (up until it calls `SortResults`) can execute entirely on in-memory data objects. For this portion of the data, the database system will be used as a backup only in case there is a memory dump or a new dictionary is added into the system.

`SortResults` will access a larger volume of data, making it impractical to cache all the data that it might require for processing (the dictionary data, which includes all definitions, requires 25 GB of storage). Here, we can apply more traditional caching such that we store the most frequently accessed dictionary definitions (and other data objects required) in cache to speed processing for the most frequently requested words, and retrieve less-frequently accessed data from the database.

We have implemented this architecture, and used it as the basis of our experimental platform. We describe our performance experiments, which look at the accuracy and runtime performance of our methods, next.

4 Scalability Analysis

In this section, we try to understand the scalability of our approach by considering the expected scaling behavior of each of the algorithms that support our approach.

We first consider the expected scaling behavior each of our algorithms. We begin our analysis with the `GenerateQuery` algorithm (Algorithm 2), since it calls all the other online algorithms. The key components of `GenerateQuery` that require attention in terms of our scalability analysis are `ExecuteQuery` and `SortResults`. Of these, `ExecuteQuery` is called multiple times, while `SortResults` is called only once (just before results are returned to the user).

`ExecuteQuery` involves two main operations: 1) querying the data source to retrieve the RMS of each element in Q; and 2) executing set logic on the returned RMSs. The second operation is not computationally intensive. Thus, the scalability of the `ExecuteQuery` will depend on the number of elements in Q.

Let m be the number of elements in Q. Let $\bar{s}$ be the mean number of synonyms per term, $\bar{h}$ be the mean number of hyponyms per term, $\overline{hr}$ be the mean number of hypernyms per term, and $\bar{a}$ be the mean number of antonyms per term. Given these, the mean number of elements in Q will be $\bar{q} = m \times (\bar{s} + \bar{h} + \overline{hr} + \bar{a})$.

To better estimate $\bar{q}$, we have computed the distributions of synonyms, hyponyms, hypernyms, and antonyms based on counts in the WordNet database. These distributions are very much skewed. So the mean does not represent the distribution appropriately. We therefore consider the medians of these distributions ($s_{md} = 2$, $h_{md} = 0$, $hr_{md} = 1$, $a_{md} = 0$), and we get $\bar{q} = 30$ for $m = 10$, i.e., on average 30 RMS database queries per Q. In essence, then, we can expect the scalability of `ExecuteQuery` to depend on the scalability of RMS database in executing on average 30 queries per user input phrase if the average $|Q| = 10$.

Finally, we consider `SortResults`(Algorithm 6), which depends on the scalability of the computation of $\rho(a, b)$ and λ. The λ computation parses dictionary definitions and user input phrases for grammatical structure, and generates parse tree. We precompute the parse trees for dictionary definitions a priori and offline, and store them in the RMS database. Then, we need to compute only the parse tree of each input user phrase, which is typically very short in length, at runtime. Parsing the user input phrase can be done quite efficiently without much overhead particularly because the context free grammar has the worst case time complexity of cubic time [7].

The value of $\rho(a, b)$ is based on the hypernym-hyponym database, which contains following table structures: `TABLE_HYP = (word, parent-word)`; and `TABLE_DEPTH = (word, depth)`.

Given two terms a and b, the depth of these terms in the WordNet hierarchy, $E(a)$ and $E(b)$, can be found from the `TABLE_DEPTH`. Given two words a and b, the common list of ancestors can be found from `TABLE_HYP` using a nested SQL query (some further optimization of this is possible

using different table structures—this is left to future work). With the list of common ancestors from TABLE_HYP and the depth of each of these ancestor words from TABLE_DEPTH, we can compute $A(a,b)$. Thus, computing $\rho(a,b)$ will require executing three select statements on TABLE_HYP and TABLE_DEPTH. With proper indexing, hashing and caching these executions can be made quite efficient.

In summary, the scalability of our approach depends on the RMS queries on the RMS databases and the queries related to WordNet hierarchy from hyponym-hypernym database. We demonstrate this scalability in practical terms in our experimental results, in Section 5.

5 Experimental Study

In this section, we first demonstrate the scalability and responsiveness of our approach with an existing approach, and then demonstrate the quality of our solution compared to an existing approach.

To achieve both of these goals, we needed a reverse dictionary solution. We were unable to find a direct reverse dictionary solution described in the academic literature; however, there are two industry solutions publicly accessible online: onelook.com and dictionary.com. While we can (and do) compare our reverse dictionary solution to the results from these real sites in terms of quality, performance testing is problematic. So, we have applied existing techniques from document similarity and classification problems to develop reverse dictionary solutions for performance comparison purposes.

The vector-based approach is a classic document indexing, searching, and classification technique. We chose the Support Vector Machine (SVM) technique, which is a document clustering implementation based on regression analysis, and statistical document clustering/classification techniques such as LSI [11] and LDA [2]. Our goal here is to test whether our approach provides meaningful performance benefits over existing approaches, and if so, whether these benefits are achieved at the expense of quality. Since SVM provides the best benchmark for quality of results, we use this as the baseline comparison for both performance and accuracy comparisons. We include LDA and LSI in our performance comparisons as well, since these methods are known to provide performance improvements over SVM.

Our evaluation consists of four parts. We first demonstrate the responsiveness of the WRD system in comparison to the SVM approach. We then demonstrate how the WRD system scales with the availability of more hardware resources in comparison to the SVM, LSI, and LDA. Once we have demonstrated that the WRD performance and scale is substantially better than any of the existing approaches, we demonstrate that WRD achieves this performance without sacrificing quality by comparing the accuracy of the WRD with the SVM approach. Finally, we compare the WRD with some of the existing RD systems available in the internet and demonstrate that the accuracy of the WRD is significantly better than that of existing online RD systems.

5.1 Experimental Environment

Our experimental environment consisted of two 2.2 GHz dual-core CPU, 8 GB RAM servers running Windows XP. On one server, we installed our implementation our algorithms (written in Java). The other server housed an Oracle 11i database, configured to store our dictionary data. This DBMS hosted all database instances, e.g., for RMS, synonym, antonym, hypernym, hyponym databases.

5.2 Workload Generation

We drew our experimental workload of user inputs based on a set of 20,000 user input traces generated by a set of approximately 1,000 Wordster beta users (college students in a North American city). We divided the user input phrases into three bins, based on the length of the phrase disregarding any stop words: $|U| \leq 2$; $2 < |U| \leq 4$; and $5 \leq |U|$. We selected 300 unique user input phrases following a two step process: 1) choose a bin based on a uniform random distribution; and then 2) choose a phrase from that bin based on a uniform random distribution. Our workload contains 87 user inputs from the $|U| \leq 2$ bin, 108 user inputs from the $2 < |U| \leq 4$ bin, and 108 user inputs from the $5 \leq |U|$ bin.

5.3 Experimental Overview

We make a strong argument in Section 1 regarding the infeasibility of directly applying vector-based approaches to the CSP problem, so we compare the performance of our approach to an implementation based on the SVM using SVMLite [13] and $\text{SVM}^{\text{multiclass}}$ [14] here. In the learning phase, SVM pre-creates a context vector for each word in WordNet by including words from definitions, synonyms, antonyms, hypernyms, and hyponyms. Each synset (synonym set) in Wordnet is identified by a class number, and this class number is assigned to any word within that synset. This class number, along with the context vector for each word, is the input in SVM's learning phase. The learning phase happens as a preprocessing step, which gives the SVM approach every possible advantage. At runtime, given a U, we create the context vector using words in U. We use SVM to classify U in one of the synset classes. The words belonging to that synset are returned as the output of SVM-based Reverse Dictionary implementation.

Since the SVM approach has access to all the word-relationship data available and compares the input phrase to the context vector for each word in the WordNet corpus, we can use the SVM quality as a rough benchmark for high-quality output. We show the results of comparing the quality of output from our approach to that of the SVM method in the quality experiments below. First, however, we consider the performance of the SVM approach as compared that of our approach.

5.4 Response Time Results

To test how our system performs, we performed a classic response-time measurement experiment. The results are reported in Fig. 2, where we plot the average response times (measured in milliseconds) against system load (measured in requests/second) for three different values of α for our approach (labeled Wordster), and for SVM. In this experiment, load is modeled by varying the rate of user inputs (drawn at random from the experimental set) incident on the system, in requests arriving per second.

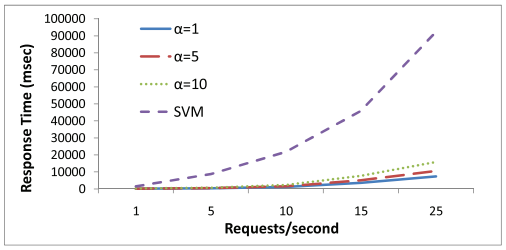

Fig. 2. Response time performance as request rate increases.

The curves all show classic exponential growth with increasing load. However, the response time of the Wordster system is about *an order of magnitude* better than the response time for the SVM approach, which requires vector similarity computation of the user defined phrase across the context vectors for all clusters (synsets). In contrast, the Wordster approach identifies a limited number of candidate words using the RMS and word-relationship data sets, and computes full similarity only against the definitions of these words in the final step. This requires significantly less processing than the SVM approach.

The differences between the different curves for Wordster occur because the different values for α require different amounts of processing in the `GenerateQuery` algorithm execution. Smaller values of α show faster response times, since the algorithm can short-circuit in earlier steps, after a small number of candidate words have been identified. Larger values show higher response times because `GenerateQuery` must execute further into the steps that consider synonyms, hypernyms, etc., to generate the minimum number of candidate words. Further, with smaller values of α, the size of the output O is smaller, which in turn results in faster execution of `SortResults`. A larger value of α will generate a larger number of candidate words in O, requiring more time and resources in `SortResults`.

5.5 Scalability Results

To demonstrate the scalability of our approach, we set up an environment where we can control the total CPU usage by the system. We demonstrated how the performance of the Wordster and other systems changes with the increased availability of CPU resources. In addition to SVM, in this experiment we also considered two advanced document clustering and classification techniques: LSI and LDA. For LSI, we used the Semantic Vector implementation available at [38]. For LDA, we used GibbsLDA++ [27]. In both the LSI and LDA cases, as we did for the SVM case, we precreated the model for each word in WordNet by including words from definitions, synonyms, antonyms, hypernyms, and hyponyms. In LSI [38], once the model has been created and given input to the system, it can be directly searched by a user input U. In LDA [27], we input all the dictionary words and their corresponding context vectors to GibbsLDA++, and receive a set of classes as output. We then provide the user input U to identify the class to which U belongs, based on GibbsLDA++ inference. The dictionary words corresponding to that class are identified as the Reverse Dictionary output.

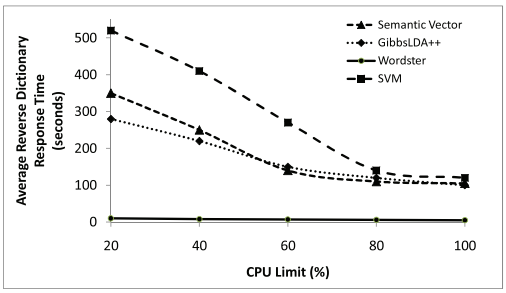

Fig. 3. Scalability as CPU limit increases.

At runtime, we compared the user input phrase with the model for each word to find the set of words that most closely represents the concepts in the user defined phrase.

We ran each of the Wordster, SVM, Semantic Vector, and GibbsLDA++ implementations in a Linux environment running on a VMWare virtual machine running on a host Windows XP Quad processor 2.4 GHz machine. Using the VMWare configuration settings, we limited the percentage CPU allocated to the Linux VM from 20 to 100 percent. This effectively enables us to control the available CPU for each approach considered in this experiment. We report the average response time for each approach for each CPU limit in Fig. 3. Here, the response time of SVM, Semantic Vector, and GibbsLDA++ show decreases in response time as more CPU resources are allocated. However, even when all CPU resources are available (the best-case performance of the SVM, Semantic Vector and GibbsLDA++ approaches), the Wordster performance still provides about an order of magnitude improvement. The key reason for such a big difference is that in all other cases, the user input phrase needs to be compared with the large number of clusters from the dictionary entries, which becomes highly CPU-intensive. In the Wordster approach, we identify a subset of candidate definitions for comparison early on in processing.

5.6 Quality Results

We now turn our attention to reporting the results of experiments to judge solution quality. To determine the optimal result sets for our 300 experimental query strings, we computed the optimal results manually (by expert professional lexicographers) as a yardstick for the accuracy of our algorithms. We measure solution quality with the standard Information Retrieval metrics *precision* and *recall*.

In our first quality experiment, we compare the output quality of the SVM, Cosine [20], and Okapi [37] approaches to our approach (labeled Wordster). For the Okapi and Cosine approaches, we developed library modules within the Wordster system for each of these approaches. In Fig. 4, we plot precision and recall against α. Here, we compare the first α results produced by each method in terms of both precision and recall. Precision and recall have opposing trends—as alpha increases, we progressively consider larger and larger numbers of possible output phrases, thereby increasing the probability of including false positives, which causes precision to decrease as α increases, while simultaneously increasing the probability finding expected results, which causes recall to have an increasing trend as α increases.

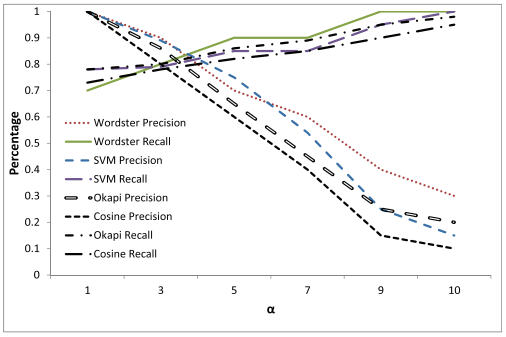

Fig. 4. Accuracy as the value of alpha is varied.

At low values of α the SVM, Cosine, and Okapi approaches give better results because they exhaustively compute similarity with all the clusters in the entire dictionary corpus. At higher values of α, the Wordster measure of word-similarity using the importance of the word in the user defined phrase starts to pay off with improved results in comparison to the other methods tested.

In our second quality experiment, we compare the quality of our approach with that of the two existing reverse dictionaries available: 1) onelook.com and 2) dictionary.com. Because these two reverse dictionary do not rank their output, we compare the output with a very high value of α, $\alpha = 50$, and categorize the result based on the length of user input phrase (ignoring stop words). We report the results in Figs. 5 and 6.

In all three cases, both the precision and recall increase with the length of the user input phrase because additional words add context—as the length of the user phrase increases, it is possible to more accurately identify the semantic meaning of the user phrase in all cases.

We note that in all length-cases Wordster outperformed both Onelook.com and Dictionary.com. We also note that Onelook.com performs better than the Dictionary.com. However, since we lack any meaningful information regarding the implementation of Onelook.com or Dictionary.com, we cannot offer any further explanation.

6 Conclusion

In this paper, we describe the significant challenges inherent in building a reverse dictionary, and map the problem to the well-known conceptual similarity problem. We propose a set of methods for building and querying a reverse dictionary, and describe a set of experiments that show the quality of our results, as well as the runtime performance under load. Our experimental results show that our approach can provide significant improvements in performance scale without sacrificing solution quality. Our experiments comparing the quality of our approach to that of Dictionary.com and OneLook.com reverse dictionaries show that the Wordster approach can provide significantly higher quality over either of the other currently available implementations.

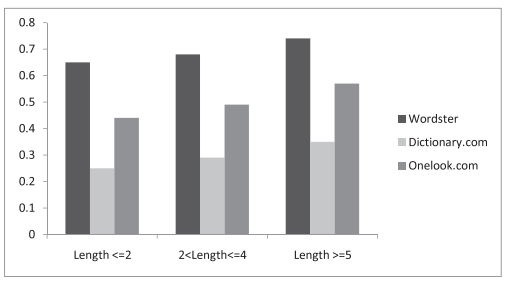

Fig. 5. Precision comparison.

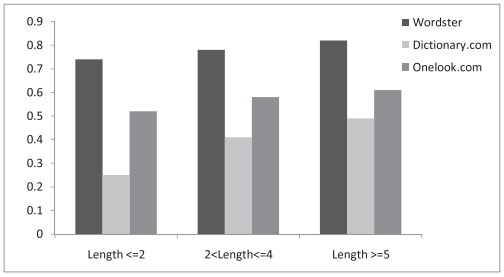

Fig. 6. Recall comparison.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* ACM Press, 2011.
[2] D.M. Blei, A.Y. Ng, and M.I. Jordan, "Latent Dirichlet Allocation," *J. Machine Learning Research,* vol. 3, pp. 993-1022, Mar. 2003.
[3] J. Carlberger, H. Dalianis, M. Hassel, and O. Knutsson, "Improving Precision in Information Retrieval for Swedish Using Stemming," Technical Report IPLab-194, TRITA-NA-P0116, Interaction and Presentation Laboratory, Royal Inst. of Technology and Stockholm Univ., Aug. 2001.
[4] H. Cui, R. Sun, K. Li, M.-Y. Kan, and T.-S. Chua, "Question Answering Passage Retrieval Using Dependency Relations," *Proc. 28th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 400-407, 2005.
[5] T. Dao and T. Simpson, "Measuring Similarity between Sentences," http://opensvn.csie.org/WordNetDotNet/trunk/ Projects/Thanh/Paper/WordNetDotNet_Semantic_Similarity.pdf (last accessed 16 Oct. 2009), 2009.
[6] Dictionary.com, LLC, "Reverse Dictionary,"http://dictionary.reference.com/reverse, 2009.
[7] J. Earley, "An Efficient Context-Free Parsing Algorithm," *Comm. ACM,* vol. 13, no. 2, pp. 94-102, 1970.
[8] Forrester Consulting, "Ecommerce Web Site Performance Today,"http://www.akamai.com/2seconds, Aug. 2009.
[9] E. Gabrilovich and S. Markovitch, "Wikipedia-Based Semantic Interpretation for Natural Language Processing," *J. Artificial Intelligence Research,* vol. 34, no. 1, pp. 443-498, 2009.
[10] V. Hatzivassiloglou, J. Klavans, and E. Eskin, "Detecting Text Similarity over Short Passages: Exploring Linguistic Feature Combinations Via Machine Learning," *Proc. Joint SIGDAT Conf. Empirical Methods in Natural Language Processing and Very Large Corpora,* pp. 203-212, June 1999.
[11] T. Hofmann, "Probabilistic Latent Semantic Indexing," *Proc. Int'l Conf. Research and Development in Information Retrieval (SIGIR),* pp. 50-57, 1999.
[12] T. Hofmann, "Probabilistic Latent Semantic Indexing," *SIGIR '99: Proc. 22nd Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 50-57, 1999.
[13] T. Joachims, "Svmlight," http://svmlight.joachims.org/, 2008.
[14] T. Joachims, "Svm^{multiclass}," http://svmlight.joachims.org/svm_multiclass.html, 2008.
[15] J. Kim and K. Candan, "Cp/cv: Concept Similarity Mining without Frequency Information from Domain Describing Taxonomies," *Proc. ACM Conf. Information and Knowledge Management,* 2006.
[16] T. Korneius, J. Laurikkala, and M. Juhola, "On Principal Component Analysis, Cosine and Euclidean Measures in Information Retrieval," *Information Sciences,* vol. 177, pp. 4893-4905, 2007.

[17] J. Lafferty and C. Zhai, "Document Language Models, Query Models, and Risk Minimization for Information Retrieval," *Proc. 24th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 111-119, 2001.
[18] D. Lin, "An Information-Theoretic Definition of Similarity," *Proc. Int'l Conf. Machine Learning,* 1998.
[19] X. Liu and W. Croft, "Passage Retrieval Based on Language Models," *Proc. 11th Int'l Conf. Information and Knowledge Management,* pp. 375-382, 2002.
[20] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval.* Cambridge Univ. Press, 2008.
[21] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-Based and Knowledge-Based Measures of Text Semantic Similarity," *Proc. Nat'l Conf. Artificial Intelligence,* 2006.
[22] G. Miller, C. Fellbaum, R. Tengi, P. Wakefield, and H. Langone, "Wordnet Lexical Database," http://wordnet.princeton.edu/wordnet/download/, 2009.
[23] D. Milne and I. Witten, "Learning to Link with Wikipedia," *Proc. 17th ACM Conf. Information and Knowledge Management,* pp. 509-518, 2008.
[24] R. Nallapati, W. Cohen, and J. Lafferty, "Parallelized Variational em for Latent Dirichlet Allocation: An Experimental Evaluation of Speed and Scalability," *Proc. IEEE Seventh Int'l Conf. Data Mining Workshops,* pp. 349-354, 2007.
[25] U. of Pennsylvania, "The Penn Treebank Project," http://www.cis.upenn.edu/ treebank/, 2009.
[26] OneLook.com, "Onelook.com Reverse Dictionary," http://www.onelook.com/, 2009.
[27] X. Phan and C. Nguyen, "A c/c++ Implementation of Latent Dirichlet Allocation (lda) Using Gibbs Sampling for Parameter Estimation and Inference," http://gibbslda.sourceforge.net/, 2010.
[28] J. Ponte and W. Croft, "A Language Modeling Approach to Information Retrieval," *Proc. 21st Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 275-281, 1998.
[29] M. Porter, "The Porter Stemming Algorithm," http://tartarus.org/martin/PorterStemmer/, 2009.
[30] O.S. Project "Opennlp," http://opennlp.sourceforge.net/, 2009.
[31] P. Resnik, "Semantic Similarity in a Taxonomy: An Information-Based Measure and Its Application to Problems of Ambiguity in Natural Language," *J. Artificial Intelligence Research,* vol. 11, pp. 95-130, 1999.
[32] S. Berchtold, D.A. Keim, and H.-P. Kriegel, "Using Extended Feature Objects for Partial Similarity Retrieval," *The VLDB J.,* vol. 6, no. 4, pp. 333-348, Nov. 1997.
[33] S. Lawrence and C.L. Giles, "Searching the World Wide Web," *Science,* vol. 280, no. 5360, pp. 98-100, 1998.
[34] G. Salton, J. Allan, and C. Buckley, "Approaches to Passage Retrieval in Full text Information Systems," *Proc. 16th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 49-58, 1993.
[35] F. Sebastiani, "Machine Learning in Automated Text Categorization," *ACM Computing Surveys,* vol. 34, no. 1, pp. 1-47, Mar. 2002.
[36] N. Segata and E. Blanzieri, "Fast Local Support Vector Machines for Large Datasets," *Proc. Int'l Conf. Machine Learning and Data Mining in Pattern Recognition,* July 2009.
[37] S.E. Robertson, S. Walker, S. Jones, M.M. Hancock-Beaulieu, and M. Gatford, "Okapi at Trec-3," *Proc. Third Text REtrieval Conf.,* Nov. 1994.
[38] D. Widdows and K. Ferraro, "Semantic Vectors," http://code.google.com/p/semanticvectors/, 2010.
[39] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.
[40] Z. Wu and M. Palmer, "Verbs Semantics and Lexical Selection," *Proc. 32nd Ann. Meeting Assoc. for Computational Linguistics,* pp. 133-138, 1994.
[41] R. Zwick, E. Carlstein, and D. Budescu, "Measures of Similarity Among Fuzzy Concepts: A Comparative Analysis," *Int'l J. Approximate Reasoning,* vol. 1, no. 2, pp. 221-242, 1987.

Ryan Shaw received the BComp degree from the School of Computing, National University of Singapore (NUS) in 2005 and the MComp degree from NUS. He is a software engineer at Google in Mountain View, California. His research area includes information retrieve, natural language processing, and data mining. Prior to receiving the MComp degree, he worked in Oracle in Singapore as a software engineer. He is a member of the IEEE.

Anindya Datta received the PhD degree from The University of Maryland. He is currently an associate professor in the Department of Information Systems at the National University of Singapore. He is a serial entrepreneur backed by Tier 1 venture capitalists. His research has formed the basis of state-of-the-art commercial solutions in database and internet systems. Previously, he was on the faculty of the University of Arizona and Georgia Institute of Technology. He is a member of the IEEE.

Debra VanderMeer received the PhD degree from the Georgia Institute of Technology. She is an assistant professor in the Department of Decision Sciences and Information Systems in the College of Business at Florida International University. Her research interests involve applying concepts from computer science and information systems to real-world problems; her work is published widely in these fields. She is a member of the IEEE.

Kaushik Dutta received the PhD degree from the Georgia Institute of Technology. He is an associate professor in the Department of Information Systems, National University of Singapore. He has extensive industry experience in leading the engineering and development of commercial solutions in the area of caching, business process monitoring, and text processing/mining. Previously, he was a faculty member at Florida International University. He is a member of the IEEE.