CrowdQ: A Search Engine with Crowdsourced Query Understanding

Daniel Haas  Daniel Bruckner  Jonathan Harper
{dhaas, bruckner, jharper}@cs.berkeley.edu

University of California Berkeley
CS262a: Advanced Topics in Computer Systems

ABSTRACT
Providing direct answers to web search queries rather than links to related documents has been shown to increase user satisfaction [3]. However, most search engines only provide answers to a few simple query types and keyword queries provide little semantic information. To support direct answers for more complex and less common queries, we have developed CrowdQ, a novel hybrid human-machine system that leverages the crowd to gain knowledge of query structure and entity relationships without the need for expert query annotation. Keyword query understanding techniques provide candidate semantic interpretations of queries which are generalized as structured query templates. Where existing query understanding techniques result in ambiguity we utilize web users for disambiguation via tasks on a crowdsourcing platform. We describe our prototype of CrowdQ, demonstrating that we can answer a wide range queries with high precision in interactive latencies, and evaluate the quality of the crowd interface, correctness of template answers and responsiveness of live template matching and query execution.

1. INTRODUCTION
Modern web search engines are beginning to provide structured results in response to users’ queries—rather than merely links to documents—in order to more directly meet their information needs. These direct answers, however, are only available for a small set of common query types [6], leaving less common, more complex queries (for example, “birthdays of bay area mayors”) unaddressed. Uncommon queries actually form the bulk of a search engine’s workload: White et. al.[19] found that over six months, 97% of unique queries occurred 10 or fewer times. A solution is needed to interpret and directly answer these long tail queries.

Recent crowd-based systems have demonstrated the effectiveness of crowds for solving common database tasks like gathering missing data and entity resolution [14, 11]. Moreover, crowdsourcing techniques have been used to link entities from keyword queries to semantic datastores [8]. While these techniques have all focused on fixing data, we believe they can be extended to understand queries as well. Existing approaches to keyword query understanding link queries with entities and relationships in structured Linked Data stores [18], though the best approaches require expert annotation [16].

Crowds provide an opportunity for deeper query understanding: every web user has intimate experience with crafting and executing keyword searches on the internet, making them experts at understanding queries. We propose injecting this expertise into each of the hard problems that compose query understanding: recognizing entities, identifying the relationships between entities, and linking these entities and relationships to structured semantic stores. Further, we observe that queries can be grouped by their semantics, and these groups can be utilized to extend the crowd’s understanding of a single query to enable automated direct answering of other queries in the same semantic grouping.

To this end, we introduce CrowdQ, a system for crowdsourced query understanding and answering. CrowdQ consists of an offline query analyzer and an online answer lookup engine. The offline analyzer uses the crowd to build structured query templates that are used for efficient answer retrieval from a structured datastore. Our initial prototype takes advantage of the crowd to verify the results of automated inter-entity relationship extraction. The result is a structured template for each semantic class of queries that is used by the online answer engine to efficiently find answers in real time. Users interact with the system via a standard web interface (see Figure 4) that provides query answers as well as contextual information and alternate interpretations of the query.

The main contributions of this work are as follows:

- An abstraction (“1-Hops”) that enables efficient representation and discovery of query interpretations
- An efficient crowd interface for verifying query semantics with high accuracy
- An online query answer engine that maps queries to learned templates with latencies acceptable for interactive web search
Crowdsourcing has been used previously for query processing in the context of traditional database systems in CrowdDB [11] and Qurk [14]. These systems, however, focus on using the crowd for gathering, cleaning, or ranking the data. CrowdQ attempts to address using the crowd for understanding query semantics, especially in the case of keyword queries which have less context than natural language queries.

3. CHALLENGES

Complexity of query semantics. Parsing natural language is a non-trivial problem, and keyword queries provide a more difficult challenge as they lack grammatical context to identify relationships between terms. This means that a keyword query of even relatively short length can represent considerably involved semantic structure. For example, the query “bay area mayors” has only two components (“bay area”, an entity, and “mayors”, a class), but implies a much more complicated interpretation (“list of people who are political leaders of type mayor in cities which are located in the San Francisco Bay Area”).

As such, our initial approach to understanding queries does not attempt to interpret arbitrarily complex keyword groupings. Rather, we make two key restrictions on the problem space. First, we only attempt to understand queries containing two terms in order to restrict the space of possible term-term interactions. Second, we only attempt to understand queries with relatively simple term-term interactions. Namely, we only search for query interpretations with \textit{1-Hop semantics} (which we define below). Our approach can be extended to more complicated queries, however, as we propose in section 7.

Data quality issues. In order to provide structured answers to a keyword search, we must transform the search into a structured query against a knowledge database. Our approach uses a database populated with publicly available semantic web (RDF) datasets. These datasets have two major drawbacks.

First, data is often missing, incomplete, or incorrect. We do not address the issues associated with constructing a robust knowledge database in this work, as they could easily fill up a paper of their own. Rather, we evaluate a query interpretation as successful if, were the datasets clean and available, the query could be answered correctly. We also note in section 7 that extra crowdsourcing steps could be added to the CrowdQ workflow in order to dynamically clean data as users encounter errors.

Second, data is often organized inconsistently. For example, in the DBPedia dataset leaders of cities are related to the cities via the “leader” relation. However, to find current mayors of these cities one must check the “leaderTitle” relation as well. To further confuse things, some cities have a “mayor” relationship which directly gives the mayor (though this is only true for a minority of cities). This issue is a direct consequence of the flexibility of the RDF model (and its lack of a formal schema), but it has unfortunate implications for this work. When we attempt to generalize our understanding of a single query to other similar queries, we assume that the data for all such queries is stored consis-
tently and can therefore be queried consistently. At worst, this could lead us to generate query templates that were correct for some initial query, but which incorrectly interpret subsequent queries (and return incorrect results). More commonly, inconsistent data representation merely leads to less general templates and fails to yield results for queries that do not match the template’s implicit schema.

**Unreliable crowds.** Any work that relies on crowd-based feedback must cope with the fact that the crowd often provides conflicting and incorrect responses. We seek to mitigate this introduced error in three ways. First, we generate candidate query interpretations ourselves and ask the crowd to disambiguate them, rather than giving the crowd the open-ended task of generating interpretations from the query text alone. Second, we provide clear and explicit English descriptions of each query interpretation, to prevent misunderstanding. Finally, we require multiple votes from the crowd before we accept an interpretation as correct. However, this methodology is by no means perfect, and as we suggest in section 7, additional quality assurance steps could be added to improve crowd accuracy.

**Latency requirements.** Our goal is to build a system that provides live, ad-hoc query answering, much like a modern search engine. This requires that users receive answers in interactive timescales, and has important implications for CrowdQ’s system design. Namely, we cannot embed crowd-sourcing steps that require feedback from other workers into the online workflow, and we must represent and store templates in a format that can be retrieved, matched against, and executed against the knowledge base while providing sub-second latencies. This acknowledgment led us to break CrowdQ into an offline query processor that uses the crowd to generate and store templates (see sections 5.1, 5.2.1) and an online query processor that simply matches queries against templates, executes them, and returns results (see section 5.2.2). Evaluation of the latency characteristics of CrowdQ can be found in section 6.3.

### 4. QUERY SEMANTICS

To make query understanding tractable, an answer system must make assumptions about queries it can handle. Queries that do not conform to the assumptions are ignored by the system. Traditionally, search engines have restricted direct answers to specific content domains, for example, flight times or the weather. Given a restricted domain, the system can leverage specific semantic structure to give high quality answers to covered queries. The drawback is that domain specific systems are, by definition, hard to extend to other domains.

Another approach to restricting the answerable query space is to make assumptions about the linguistic syntax of input queries. For example, the Tail Answers system mines query logs for queries beginning with question words like “how,” “who,” and “where” [3]. Hardcoded syntax rules can be leveraged by a parser to achieve high quality understanding, but they also make the system rigid and difficult to extend.

CrowdQ is designed to be more extensible than other answer systems by making weaker assumptions about input. CrowdQ is like domain specific systems in that it depends on an underlying data store to generate answers. CrowdQ’s maximum answerable query space is bounded by the content of its store. Unlike domain specific systems, however, CrowdQ makes no assumptions about the structure of its store, and instead uses search algorithms and the crowd to learn semantic structure.

The schema of a data store is limited in its ability to assist with query understanding because queries may contain an arbitrary number of semantic “hops.” To illustrate, consider that any noun phrase, say “CS262a,” can be expanded indefinitely (for example to “students in CS262a,” or “favorite football teams of cousins of professors of students in CS262a”). To make CrowdQ’s job tractable, we limit it to answering queries with at most one semantic hop, plus an additional constraint hop. We believe this is a good assumption because queries with less indirection occur more frequently. We intend to extend CrowdQ to handle more complex queries in the future.

To make the limit of “one semantic hop, plus a constraint” precise, we introduce the 1-Hop abstraction (see figure 1). A 1-Hop is a sub-graph consisting of four nodes and three edges. Consider the query “students in CS262a.” The source node corresponds to an explicit entity found in the query (in this case “CS262a”). The answer node is an entity in the database that has an attribute that answers the query (any student in CS262a). The 1-Hop semantics require that the answer node be one hop—the source predicate—from the source node, where a hop corresponds to one edge in a semantic graph data store or to one foreign-key relationship in a relational database. Answer nodes can optionally be required to meet some constraint in the form of being one hop (the filter predicate) from a filter node. Filters often enforce a type constraint, for example, “is a student” may be applied to weed out professors and TAs in case the source predicate is too general (e.g. “participants in CS262a”).

The 1-Hop abstraction is used by the components of CrowdQ to add new semantics to the system and extend its query coverage. Often, it is useful to group 1-Hops that are identical except for their answer nodes—these groups represent answer sets. The relationship extraction system generates candidate answer sets. The crowd verification system vets these candidates and chooses correct ones to be made into templates. The template system maps query patterns to
generalized versions of selected candidates—in these templates, the source node is abstracted as a wild card, for example, “students in <course number>" (see section 5.2.2 for more details). Adding new templates enables CrowdQ’s on-line component to quickly answer new classes of queries, thus expanding the space of answerable queries. The following section will describe this process and the CrowdQ system components in greater detail.

5. THE CROWDQ SYSTEM

The major components of CrowdQ are an off-line query analyzer, a crowd-based verification system, and an on-line query answer engine. To answer a live search query, the answer engine searches a template database for an answer template that matches the query. The template in turn maps the query to a parameterized structured query on an underlying answer store. This latter query is executed to return the result to the user.

CrowdQ can be extended to answer more queries by adding new templates to its template database. Given an exemplar query, the query analyzer generates candidate templates that match the 1-Hop semantic abstraction described in the previous section. Candidate templates are then passed to the crowd to be cleaned, filtered, and verified. Templates that pass the crowd are added to the template database to be used by the on-line engine. The following section explores these component systems in greater detail.

Figure 2 illustrates how these components fit together in CrowdQ’s architecture (as proposed originally in [9]). To this point, CrowdQ has been implemented against semantic web answer stores, namely DBPedia [4], YAGO [17], and MusicBrainz [1]. We put these data sets into the Apache Jena [15] triple store, and access the data with the SPARQL query interface.

5.1 Query Analyzer

Given a new exemplar query, the CrowdQ query analyzer attempts to interpret the query to generate a set of subgraph patterns in the answer database. In particular, the analyzer parses the query to identify relevant entities, then searches for relationships between those entities that conform to the 1-Hop abstraction. The 1-Hop objects that the analyzer identifies are used as input to the crowd interface for candidate templates that answer the exemplar query and others with equivalent semantics.

5.1.1 Entity Extraction

The first step when given an exemplar query is to parse it and map n-grams in it to objects in the answer database. For example, given the query “actors in Top Gun,” we would like to map “Top Gun” to its corresponding entity in DBPedia, and “actors” to an actor type or relation. In practice, an n-gram in the query may map to multiple objects—the YAGO data set, as an example, contains hundreds of categories about actors, including “ActorsFromNewMexico” and “ActorsFromCalifornia.”

To accommodate the 1-Hop abstraction, we expect the entity extractor to find exactly two objects, a source and a sink, both linked to n-grams in the query. If the extractor finds more objects, the query is discarded, since CrowdQ does not handle more complex queries at present. Likewise, if no objects are found, the query is dropped as trivial. In cases where only a single entity is found, select information is displayed on the search results page about that entity.

Named entity recognition (NER) is a challenging problem that has been studied deeply in the NLP field. Many tools exist to perform NER with acceptable accuracy, but they require careful training and set up to achieve accuracy on a new corpus. Working within the scope of a class project, we opted to avoid the potential rabbit hole of exploring sophisticated NLP extraction techniques. Therefore, the present implementation of CrowdQ expects that the source portion of exemplar queries be tagged and mapped by hand. The sink text must also be labeled, but not mapped to any database object, since it is used simply as input to a full-text search for matching objects.

5.1.2 Candidate Generation

The procedure to produce candidate templates is a search of the answer database for sub-graphs that connect a query’s source and sink and that match the 1-Hop graph pattern. The 1-Hop abstraction greatly simplifies the search, which, given a triple store answer database, reduces to a handful of SPARQL queries. There are four cases that need to be checked. In all cases, the query’s source is identified with the 1-Hop’s source node. Then each case identifies the query’s sink as one of the source predicate, answer node, filter predicate, or filter node. In fact, the sink text is identified with these objects indirectly, by means of a full-text search query. Apache JENA supports full-text search powered by Lucene. Non-string objects are returned by the search when their text labels match the search term.

These four semantic case are illustrated below, for mapping the query “CS262a room number” to different database schemas:

1. Sink matches source predicate. This case covers the simple situation where the answer to a query is an attribute of the source node, for example, if the CS262a entity has an attribute labeled “Room Number” that points to the string “Soda 310”

2. Sink matches answer node. Here, the sink matches an entity directly; for example, if the property location of CS262a points to an entity named “Soda Hall Room number 310.”

3. Sink matches filter node. This case is slightly more complex because the sink text points to a node two hops away from the source. An example is if CS262a’s location property points to multiple entities, named perhaps “Soda Hall” and “310.” The correct answer node here is distinguished because its type property points to “Room Number.”

4. Sink matches filter predicate. This is an odd variant of the previous case, and would apply to the case where the “310” node above has, say, a property called “Is Room Number” that points to true. In practice this case introduces many low quality candidates, and so it is not used in our current implementation.

1http://musicbrainz.org
Figure 2: CrowdQ Architecture, as proposed in our original vision paper [9]. Off-line processing starts at the top-right with a search log as input. On-line processing starts at top-left with a user keyword query.

Note that we do not find the answer if it is more than one hop from the source node, for example, if CS262a points to an abstract ClassLocation entity that itself points to location details like a Soda Hall node and a Room 310 node.

The candidate generator in CrowdQ is a Python module that generates and executes SPARQL queries corresponding to the cases above. The queries are executed against JENA. Duplicate results are removed and those with equivalent graph patterns but different answer nodes are grouped. These groups are joined with their associated tagged exemplar query and passed to the crowd system for verification.

5.2 Template Verification

5.2.1 Crowd Verification

Candidate generation produces a set of potential 1-Hop graphs without any indication of which are most likely to correspond to the intent of the query. In order to find the answer graph with the most appropriate semantics, natural language representations of the 1-Hop graphs are generated and presented to the crowd. The most common answer from this crowd verification step is used to determine the answer pattern for similar queries, while other common answers may be presented as alternatives on the search results page.

Figure 3 shows the interface presented to the crowd to verify the correct query meaning. Due to the tendency of crowd workers to pick one of the first few answer choices, the list of answer candidates is randomized for each crowd worker. Additionally, since multiple answer graphs are sometimes correct for a query, the interface has the option of multiple choice or single-select answers from the crowd. Since displaying 1-Hop graph patterns to the crowd might be confusing, we build explicit natural language sentences to describe each answer candidate. For example, in the query “actors in Top Gun”, the 1-Hop with a source node of “Top Gun”, a source predicate of “starring”, the filter node “ActorsFromNewYork” would generate the natural language text “objects related to Top Gun with the starring relationship, to ActorsFromNewYork with the type relationship”.

5.2.2 Templates

At this point, CrowdQ has matched the query to a 1-Hop subgraph representation, with an accompanying structured SPARQL query that would answer the query if run against our knowledge database. We observe, however, that by abstracting parts of the 1-Hop graph structure, we can answer a wide range of related queries. Particularly, by converting the source node of a 1-Hop to a wildcard, we can answer any query that would match the rest of the 1-Hop subgraph. For example, “actors in Top Gun” would generate a 1-Hop with the <Top_Gun> entity as its source node. By replacing <Top_Gun> with the wildcard <MOVIE>, we can trivially match “actors in Forrest Gump”, “actors in Cloud Atlas”, or “actors in Saving Private Ryan”. We call a 1-Hop graph with wildcard replacements a generalized 1-Hop, and the specific 1-Hop from which it was created the origin 1-Hop. Given a generalized 1-Hop, we can trivially produce an equally general structured SPARQL query:

1. For each entity $e_i$ that was generalized from the origin 1-Hop, generate a variable $v_i$.
2. Replace each occurrence of $e_i$ in the origin 1-Hop’s SPARQL query with $v_i$.

So we could generalize the SPARQL query for “actors in Top Gun”:
SELECT distinct ?actor  
WHERE {  
  ?actor a class:Actor .  
  <Top_Gun> rel:starring ?actor  
}  
to a query for “actors in <MOVIE>”:

SELECT distinct ?actor  
WHERE {  
  ?actor a class:Actor .  
  ?movie rel:starring ?actor  
}

We term these generalized SPARQL queries query templates. Thus, given an incoming keyword query, CrowdQ’s query analyzer builds and stores a corresponding query template.

Templates are only useful, however, if incoming queries can be matched against them. The challenge for this matching process is twofold—incoming queries must match against templates and look up results in web search timescales, and matching must be accurate to avoid bombarding users with irrelevant answers. As an initial approach, CrowdQ uses regular expressions to match only queries that bear strong textual similarity. Regular expressions are generated along with the query templates, and simply represent the exact text of the query with potential wildcards. For example, "actors in Top Gun" would generate the regular expression \("actors in (.+)\). This would match "actors in Top Gun", "actors in Forrest Gump", etc. The group of text which matches (.+) corresponds to the movie name, and can be linked using our Lucene index to an entity in the knowledge database.

This approach is intentionally limited: it cannot match "Top Gun actors", for example. However, regular expression matching still allows templates to answer a large number of related queries (see section 6). We have considered other, more sophisticated approaches as well—see section 7 for a discussion.

5.3 Query Answering

Once the system has stored a body of query templates, it can immediately begin processing incoming queries. In order to successfully answer a new query, CrowdQ must match the keyword query against the set of stored templates, evaluate the query against matching templates, and present results to the user.

In order to successfully match a query against a template, we simply evaluate the query against the template’s associated regular expression. We attempt to match the incoming query against each template in the store. If there are multiple matches, we have several possible interpretations of the query. In such a case, when results are displayed to the user, only the first interpretation is displayed, with a note that other interpretations are available and the option to switch between them.

To evaluate a template, we link wildcard entities in the incoming query to our knowledge database using a Lucene
index, substitute them into the template’s SPARQL query, and execute the query against our database. If the result set of such a query is non-empty, we parse the results and display them to the user via the browser interface (figure 4). If there were no returned results, there are several possible explanations: the query might be unanswerable, our data store might not cover the requested data, or we may have matched against an incorrect template. In such a case, we return an empty result to the user (falling back to a straight listing of Google results) and queue the query for inspection to determine the cause of the failure. This step is currently manual, though it could be implemented as a crowdsourced task.

6. EVALUATION
CrowdQ has been evaluated using a set of queries from the QALD-2 Challenge\(^2\) for question answering on linked data. The QALD-2 data set contains 200 natural language queries with associated important keywords and gold standard SPARQL queries for DBPedia [4] and Musicbrainz\(^3\) datastores. Since our system doesn’t attempt to answer queries involving aggregation, any queries requiring aggregation were removed from the dataset. Our implementation of CrowdQ uses the Jena [15] framework for storing and querying RDF data and Lucene\(^4\) for text indexing of the data. All evaluations are run using the DBPedia [4] infobox and category data.

Evaluation of CrowdQ requires defining quality metrics for the crowd interface, template generation and template matching. Answerable query classification and query tagging aren’t addressed in the CrowdQ evaluation, but are dealt with in detail in related work [3, 12].

6.1 Crowd Interface
To evaluate the crowd interface, the “gold standard” answer graph (or graphs if multiple choices are equally correct) are hand-labeled before being submitted to the crowd interface and then compared to the results from the crowd. The query corpus used was the QALD-2 DBPedia and Musicbrainz datasets. In the first experiment, answer candidates were generated from the DBPedia graph with category data and in the second on DBPedia without categories. We found that categories resulted in far more false answer candidates, though there was a correct answer more often when using categories. Without categories, the most common answer was the best answer 64.1% of the time, and the correct answer was one of the top two most common 87.1% of the time. With the category data from DBPedia, the most common answer was the “gold standard” 26.9% of the time and one of the top two were 30.7% of the time. This demonstrates the danger of presenting too many potentially confusing results. The crowd often chose results belonging to categories too specific for the query.

6.2 Query Coverage
An answer system can only be as useful as the number of queries it is able to respond to. We have argued that the

\(^2\)http://greententacle.techfak.uni-bielefeld.de/ cunger/qald/index.php?q=challenge&x=challenge\(^3\)http://musicbrainz.org
\(^4\)http://lucene.apache.org

1-Hop abstraction allows CrowdQ to extend the size of its answerable-query space rapidly, by enabling templates with simple yet generalizable semantics. To test this hypothesis, we need a measure of the generality of a template. We define a template’s size to be the number of distinct entities that are in the source node position of a sub-graph that matches the templates 1-Hop pattern. For example, suppose a template fitting queries of the form “students in <course>” uses a 1-Hop where the source node points to answer nodes via the source predicate attendee. Then the size of this template will be the number of distinct entities that have one or more attendee properties (i.e., the number of classes for which we can find students).

We believe this notion of size is a good measure of the generality of a template because it is proportional to the number of distinct keyword queries—given successful entity extraction—that the template can answer. We measured the sizes of templates vetted by the crowd and found that the majority of our templates are larger than one thousand distinct source entities (see figure 5). Though our sample is not large (N=39), this is evidence for the efficiency of our approach: using templates enables CrowdQ to expand its query coverage by several orders of magnitude faster than a query-at-a-time system. And while these results were measured for templates covering only one data set, DBPedia is a worst-case for templates because it is mass-authored and has a very heterogeneous schema. In a more uniformly structured data set, for example a properly modeled relational schema, templates should perform better.

Finally, outliers in template size tend to reveal incorrect templates. On the one hand, a template with very small size is often an overfit. For example, for the exemplar query “members of The Prodigy,” the crowd selected a template with a filter node that required that all band members belong to the category “Members of The Prodigy.” Such a template cannot generalize to bands other than The Prodigy; its template size was 3 distinct entities (The Prodigy, and two bands that Prodigy members also belonged to). On the other hand, very large templates tend to be overly general, and perhaps lack a necessary filter constraint. In the future, we plan to use template size to assess template candidates and optimize crowd review.

Figure 5: The 1-Hop abstraction encapsulates a limited but common class of query semantics.
Closing the loop. Our current system uses a hand-tagged crowd, but there are a number of areas in which we can continue to build upon our ideas. In the scope of the semester, we were able to prototype and evaluate our approach to understanding queries with the tools would significantly improve recall, but such work is out of scope for our first prototype.

6.3 Template Matching

For template matching to be useful, the process of matching the query text to the correct template, linking the entity in the template to the data store, and querying the data store for the result set must occur with latency appropriate for a search results page. Figure 6 shows the latency results for local CrowdQ requests for randomly chosen templates and entities which match them (to avoid the effect of caching similar queries). Latency rarely climbs above 50ms, though after network latency this figure would be higher. This shows the practicality of our approach with an unoptimized storage solution.

6.4 Template Accuracy

Users of search engines demand high-quality, accurate results, and an answer engine that returns incorrect results only trains users to ignore its output. We use the standard IR metric of precision—the ratio of correct answers returned to total answers returned—to measure the accuracy of our templates. The QALD-2 benchmark provides questions and answers for the DBPedia data set. We assessed the precision of 13 of our templates against QALD-2 (the rest of our templates were either from the MusicBrainz data set, which did not have useable answer queries, or their QALD-2 answers returned no results). For these templates, we measured a precision of 83%, a respectable number, but one which we believe can be further improved with further optimization of the crowd pipeline.

Another standard IR metric is recall, or the ratio of answers returned to answers requested. On our test set, recall was a low 46%, owing primarily to limitations of the Lucene full-text search we depend on for entity extraction at query time. We believe that using more sophisticated entity extraction tools would significantly improve recall, but such work is out of scope for our first prototype.

7. FUTURE WORK

In the scope of the semester, we were able to prototype and evaluate our approach to understanding queries with the crowd, but there are a number of areas in which we can continue to build upon our ideas.

Closing the loop. Our current system uses a hand-tagged query corpus to generate templates. By implementing state-of-the-art NLP techniques for entity extraction and recognition, augmented with crowdsourced verification steps, we can automate this step. This will enable us to use our live query interface as a feedback loop for template generation: incoming queries that do not match any templates can be immediately queued for processing by the query analyzer, leading to the desirable behavior that users who find no results to a query may return minutes later to find that their queries can now be answered.

Improved use of the crowd. Our experiments showed that in the majority of cases, the crowd can be used successfully to interpret queries. However, we can continue to develop our interface to improve the crowd’s accuracy. For example, we detected a bias in the crowd’s choices towards earlier items in the list of candidates, possibly indicating that crowd workers don’t bother to read all candidates before answering. In order to ameliorate this effect, we could experiment with breaking candidates into subsets to minimize the number of candidates each worker must consider. Additionally, crowd accuracy can be improved by adding in extra verification steps, both to confirm initial interpretations of queries and to verify that generalized templates preserve their desired semantics.

Beyond simply using the crowd to generate accurate query templates, we can begin to leverage an engaged user base in order to fundamentally improve the system. By making the results page more interactive, we could allow users to indicate issues with the results returned by CrowdQ. Users responses would constitute input from a ‘secondary crowd’, which could correct errors in templates or knowledge base data, or even define the types and format of returned data.

Support for more complex query semantics. One of the major constraints of the current system is its inability to handle queries which have more than two terms, and which do not correspond to a 1-Hop semantic subgraph in the knowledge database. Future work could easily expand on this. Allowing more terms in search queries would complicate the graph search problem, as discovered subgraphs would need to match against more than just a single source node and sink text. Matching subgraphs with more complex structures than 1-Hop graphs would allow us to interpret keyword queries with more involved semantics (“bay area mayors”, for example), but would also result in more complexity in the graph exploration. A combination of heuristics for graph search and increased crowd involvement in the earlier phases of candidate generation could help limit this complexity.

Generation of more general templates without loss of accuracy. Templates currently can only match keyword queries with identical semantic structure, with text that matches a simple regular expression. While this limited model is remarkably effective at expanding knowledge of a single query to a large number of similar queries, the model could be improved. First, we could increase the number of queries that match a stored template by developing a more sophisticated matching algorithm. For example, following the work of Pound et. al. [16], we can model queries as ordered sets of semantic types (e.g. “capital of Canada” be-
comes {property} <relationship> <entity>}, then train a model to estimate whether an incoming query matches each template. This would allow us to match incoming queries against templates without requiring identical syntax. For example, our current system would build a template for “capital of Canada” that could match “capital of France”, but not “France’s capital”. A machine learning-based model could successfully determine that the two forms are identical.

Additionally, we could expand on the semantic structures recognized by a template by integrating NLP tools such as stemmers and WordNet to capture queries with synonymous (but not identical) semantics. For example, if our knowledge database represented the “acted in” relationship with both the rel:actedIn and the rel:starredIn relations, our current system might not find rel:starredIn for a query like “actors in Top Gun”, as there is little textual similarity between “actors” and “starredIn”. Using word stems and synsets, however, we could discover that “act in” is related to “star in”, and match the query against our template.

Evaluation of system scalability. While we showed that our current template system is sufficient to respond in interactive timescales to user queries, we currently store a limited number of templates. We could evaluate how the system performs at scale, and develop more sophisticated storage and lookup mechanisms to improve performance when there are many templates in the database and users query templates with high concurrency. We expect that templates would work well in a distributed, highly-available store, as user queries are read-only and templates are immutable.

8. CONCLUSION
We have introduced CrowdQ, a search engine that leverages the crowd to understand keyword queries and build them into general templates that can answer large classes of similar queries. By limiting the keyword queries we interpret to those that correspond to 1-Hop semantic subgraphs over our knowledge database, we ensure that our templates return precise and accurate answers to the queries they address. We have determined that with reasonable success, the crowd is able to correctly verify query semantics. Our prototype provides a low-latency interface for the delivery of structured answers to unstructured queries, especially those in the long tail of complex queries with which traditional keyword-based information retrieval systems struggle. Thus, CrowdQ represents a promising step towards building search engines that truly understand user queries, and can respond immediately with structured answers to complex questions.

9. REFERENCES