IQ: The Case for Iterative Querying for Knowledge

Yosi Mass[1,3] Maya Ramanath[2] Yehoshua Sagiv[3] Gerhard Weikum[2]

[1]IBM Haifa Research Lab
Haifa, Israel
yosimass@il.ibm.com

[2]Max-Planck Institute for Informatics
Saarbrücken, Germany
{ramanath,weikum}@mpi-inf.mpg.de

[3]The Hebrew University
Jerusalem, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Large knowledge bases, the Linked Data cloud, and Web 2.0 communities open up new opportunities for deep question answering to support the advanced information needs of knowledge workers like students, journalists, or business analysts. This calls for going beyond keyword search, towards more expressive ways of entity-relationship-oriented querying with graph constraints or even full-fledged languages like SPARQL (over graph-structured, schema-less data). However, a neglected aspect of this active research direction is the need to support also query refinements, relaxations, and interactive exploration, as single-shot queries are often insufficient for the users' tasks. This paper addresses this issue by discussing the paradigm of Iterative Querying, IQ for short. We present two instantiations for IQ, one based on keyword search over labeled graphs combined with structural constraints, and another one based on extensions of the SPARQL language. We discuss the suitability of these approaches for knowledge-centric search tasks, and we identify open research problems that deserve greater attention.

1. INTRODUCTION

Advanced users such as journalists or analysts have information needs which are often expressed (even in natural language) as a mix of vague, precise, and implicit requirements. For example, consider the following queries: i) "I want to know something about classical music composers who have composed music for western movies." ii) "Could the H1N1 vaccine interfere with blood-pressure medications such as Metolazone?" iii) "How are Israel and Italy related to each other, for example, by some international organizations?"

Although Web search engines now have limited and specialized support for natural-language questions and are moving towards more expressive entity-oriented search (e.g., by understanding product names or locations), the above questions cannot be answered easily. Recently emerging knowledge engines [5, 4] or knowledge-base search services such as WolframAlpha[1], Google Squared[2], or *sig.ma* cannot cope with such complex queries either. Some of them seem to perform entity-oriented information extraction on-the-fly, while others harness large knowledge bases such as DBPedia[3], Freebase[4], True Knowledge[5], or the CIA WorldFactbook[6]. These contain billions of RDF triples about entities and relationships, but cannot retrieve the necessary facts for answering the above queries or lack inferencing capabilities for composing proper answers. For example, trueknowledge provides a browser plug-in that supplements keyword-search results from Google or Bing with fact-retrieval answers from their knowledge base. This is good enough for returning the correct birth place of Barack Obama, but still far from handling our examples.

So neither Web search engines nor knowledge-base engines can directly answer such advanced questions. However, there is often a solution if the user is willing to engage in an entire workflow of query refinement, query relaxation, exploration of intermediate results, combining different results, etc. This is tedious but often works. The point is that, instead of running a single-shot query, we need a process of *iterative querying*, *IQ* for short. In fact, this is what search-engine users often end up doing, but there is not much support by the engine. Moreover, while IR researchers are advocating interactive retrieval for many years [13] (without compelling impact), in the structured-data world of knowledge bases and inference engines, the expectation by DB folks is that everything can be expressed in a single query of some super-powerful language (be it SQL, XQuery, SPARQL, or whatever).

An IQ task with a Web search engine would involve the following steps:
i) *exploration*: retrieving initial results by keyword search,
ii) *filtering*: refining the results with additional constraints,
iii) *aggregation*: combining results into a concise answer.

These steps may themselves have to be iterated. For example, for the composer question, one could proceed as follows: i) find a list of classical music composers; again i) find a list of composers for western movies; ii) pick out composers of western movies who also compose classical music; once more i) find biographies for each of the interesting composers; ii) ensure that the biographies match the list of com-

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

[1] *wolframalpha.com*
[2] *google.com/squared/*
[3] *dbpedia.org*
[4] *freebase.com*
[5] *trueknowledge.com*
[6] *www.cia.gov/library/publications/the-world-factbook/*

posers (as keyword search can easily return wrong results); iii) aggregate multiple pages about the same composer into a compact summary.

For the world of structured knowledge bases, this kind of IQ process seems totally neglected so far and widely open for research. Knowledge portals such as dbpedia.org or freebase.com offer APIs that support only single-shot querying, by means of SPARQL calls; their UIs, on the other hand, are merely Web-pages with fancy rendering but tedious navigation for the user. The irony is that despite semantically structured data, there is poor support for advanced questions about factual knowledge.

In this paper, we advocate the IQ paradigm for search against *semantic knowledge bases* or the linked-data cloud (*linkeddata.org*) [3] of structured data on the Web. We discuss the requirements for the exploration, filtering and aggregation steps, describe the development of effective tools in two case studies and identify challenges for future research.

2. DATA MODEL

The diversity of data available in different kinds of knowledge bases, linked data on the Web, as well as text in the form of contextual information, requires a very general and flexible data model. The natural choice is to use a directed graph where each *node* is allowed to have: a *name* (identifier or short string describing, for example, an entity name or a paper title), a *type* (the class to which the node belongs), and a *context* (additional text that is associated with the node). Similarly, each *edge* is allowed to have: a *name* (identifier), a *type* (a label denoting, for example, a relationship type), and a *context* (textual information, for example, denoting the context in which the fact denoted by the edge and its two end points was extracted from).

This unified model captures all the different approaches to graph search. It can represent, for example, XML (nodes labeled, edges unlabeled), RDF triples (nodes and edges labeled), relational databases (records as nodes, foreign-key relationships as edges), etc. Note that in general, there is no schema for these knowledge bases, apart from a generic triple representing an edge in the data graph (referred to as subject, predicate and object in RDF).

3. THE IQ PARADIGM

As in the case of current day Web search, we envision iterative querying as being composed of a combination of three steps: exploration, filtering and aggregation. However, unlike Web search where there is no structure at all, and unlike relational databases where the schema is fixed, in our data model there are entities (nodes) and relationships (edges), but there is no fixed schema. Therefore, the exploration, filtering and aggregation steps of the IQ paradigm should allow the user to work in two dimensions. First, the user should be able to select desired structural patterns (i.e., schemas) of answers and secondly, she should be able to select the relevant instances of those patterns.

Figure 1 illustrates our framework. Users query data in the underlying knowledge-bases through a UI. A system built on the IQ framework and supporting the exploration, filtering and aggregation steps interacts with the API and returns results back to the user. The goal of an IQ system should be to automate as many tasks as possible, and to involve the user only at certain critical steps when her

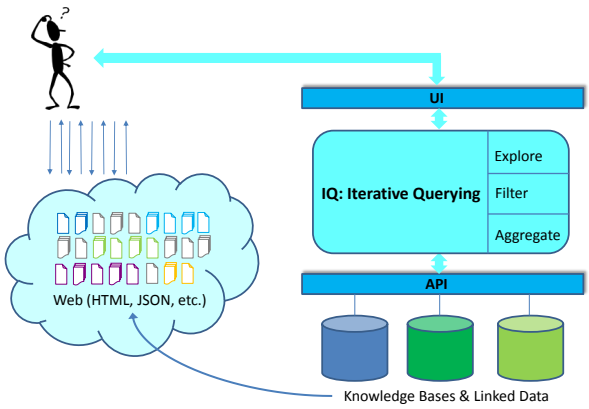


Figure 1: The IQ Paradigm

feedback is essential.

Clearly, while the UI has to be simple and intuitive, the API should ideally be expressive enough to handle complex tasks, including aggregation. The IQ system provides a bridge between the UI and API, and in certain cases may have to extend the API to support the needed functionality.

In the rest of this section, we describe the key ideas of each step, and defer technical details of how they could be achieved to the case studies in Sections 4 and 5.

Exploration. The first step is to help the user express her information need in a precise manner. A natural way to start the exploratory step is for the user to enter some relevant keywords, or phrases and for the system to return possible connections between them. For example, for the query on classical composers, a starting point is for the user to enter "classical music" "composer" and "westerns" or for the query on the relationship between Israel and Italy, to enter "Israel" and "Italy". It is now up to the system to search for possible connections between these keyphrases.

Given a set of diverse connections from the underlying data graph, the user would prioritize certain structural constraints while disallowing certain other kinds of connections. The system repeats the query with these additional (positive and negative) constraints. At the end of this iterative process, a precise query of the form "classical composer X composed for western Y" is formulated, where X and Y have to be substituted with a person and a movie, respectively. This query could then be mapped onto the API for a structured query language.

However, this may not be the end of the exploration step. Perhaps there are other closely related queries such as "classical *musician* X composed for western Y" or "classical *music conductor* X *directed music* for western Y", which can additionally be suggested to the user. Once the user converges on a subset of precise queries as representative of her information need, the system can now move on to the filtering step.

Filter and Refine. While the exploratory step helps users narrow in on possible interesting queries, the filtering step allows users to specify exactly which items in a result set are interesting. The system runs the queries formulated in the

previous step directly on the knowledge-base and returns a set of qualifying results. The user now refines her query in order to filter out the results of interest to her. The filtering could be as straightforward as selecting certain interesting results, or there may now be additional refinements (in the form of constraints) to restrict the size of the result set. For example, movies with a certain plot-line (say, involving "soldiers"), or musicians born in a particular continent, etc. Additionally, after seeing this set of results, the user may also disallow certain queries formulated in the previous step or give additional weight to certain other queries. Iteratively processing these refined queries may finally lead to a satisfactory result list. Alternatively, the user could repeat the exploration step if the results are still unsatisfactory to her.

Aggregation. Our data model (Section 2) is general enough to accommodate a wide variety of data including a combination of structured triples and unstructured text. Aggregating results can thus take on different meanings, ranging from a simple ranking of results (famous composers first), grouping of results (composers grouped by their nationalities), diversifying the top results, etc. to type-specific aggregations such as summarization of text (for example, biography summarization for composers).

Aggregating results from a single knowledge-base is complex enough by itself, but the situation becomes more complicated if there are multiple knowledge-bases which need to be queried. A likely scenario is that the local knowledge-base has insufficient information to answer the query, but has multiple pointers to other knowledge-bases (in the linked-data spirit) which are likely to contain the missing information. The system now has to decide whether and how the query should be split (perhaps there is overlapping information in the knowledge-bases which can be leveraged) and subsequently, how to merge the (partial) results. Both tasks are non-trivial because on-the-fly entity disambiguation [9] may be required due to different vocabularies.

With this overview of the IQ paradigm, we now illustrate two case studies of how a system built on these principles would work. In the first case study, the user starts the exploration step with SPARQL queries, and refines queries by adding keywords to the structured queries. In contrast, in the second case study, the user initially does not have any knowledge about the structure of the data graph. Hence, she starts the exploration of the knowledge-base with keywords, while the refinement step involves selecting and unselecting suitable structures showing the interconnections between the keywords. The selected interconnections can then be converted to SPARQL to further the search. Finally, the aggregation step for each case study highlights a different kind of aggregation—ranking and grouping of results in the first study, and merging results from multiple knowledge-bases in the second. In principle, however, both types of aggregation are applicable to either one of the two case studies.

4. CASE STUDY 1: IQ WITH EXTENDED SPARQL

The W3C-endorsed query language SPARQL is a natural starting point for searching knowledge bases or the world of linked-data Web sources. SPARQL is designed for structured RDF data, but does not need a prescriptive schema for its data, and can cope with high heterogeneity. The basic building block in a SPARQL query is a triple pattern: essentially an SPO triple with one or several of the S, P, and O components replaced by variables. Multiple triple patterns are combined in a conjunctive manner, thus supporting select-project-join queries. The key point, compared to traditional database querying, is that even properties—the counterparts of relation or attribute names—can be variables. For example, when searching for composers of film music, we may not know how exactly the relationship between composer and movie is named, e.g., composed, isComposerOf, wroteMusicFor, contributedToSoundtrack, etc. Or the data may be so heterogeneous that no single property name is suitable for high recall. With SPARQL, we could express a sub-query for the composers question as follows:

```
SELECT ?c, ?m
WHERE {
  ?c hasType composer . ?m hasType movie .
  ?m hasGenre Western . ?c ?prop ?m . }
```

where variables start with a question mark and the appearance of the same variable in different triple patterns denotes a join condition.

Exploration. Despite the schema-agnostic option for query formulation, users (or programmers on behalf of users) still need some awareness of the underlying RDF structures. If composers are first related to their compositions, say by a property composedPieceOfMusic, and the compositions are in turn related to movies by a property appearedInSoundtrackOf, even the wildcard pattern ?c ?prop ?m would not return any matches as all variable bindings need to come from a single RDF triple. Fortunately, it is not too difficult to extend SPARQL, to support these cases while preserving the general flavor of SPARQL. Following the proposal by [2], we could introduce variables that can be bound to entire paths in the RDF triples graph. Moreover, as we do not simply want connectivity but have semantic requirements, we would combine this with filter conditions on the property names in the qualifying paths. For our example, this could be phrased as follows:

```
SELECT ?c, ?m
WHERE {
  ?c hasType composer . ?m hasType movie .
  ?m hasGenre Western . ?c ??prop ?m .
  Filter regex(??prop, {"compose"}) .
  Filter pathlength(??prop, 3) . }
```

where ??prop is a path variable (note the two question marks), regex is a regular expression (simple substring matching in our case) on the path of property names bound to ??prop, and the last condition sets an upper bound on the path length. The Filter construct is standard SPARQL, to include conditions beyond exact-match on URIs or literals. Here we deliberately use it for extensibility.

Further, the system could support *query relaxation*, and suggest close-and-related queries. For example, for the swine-flu vaccine question, a user formulation like

```
SELECT ?c, ?d
WHERE {
  ?c hasType H1N1vaccine .
  ?c interferesWith ?d.
  ?d hasType BPMedication . }
```

could be automatically relaxed into:

Figure 2: Western movies: i) filtered by "soldier", ii) unfiltered

```
SELECT ?c, ?d
WHERE {
  ?c hasType H1N1vaccine .
  ?c (interferesWith|notRecommendedWith|createsRiskWith) ?d .
  ?d hasType BPMedication . }
```

Filter and Refine. The exploratory step works reasonably well when everything the user wants can be expressed in triple patterns, but is somewhat inflexible. It may not be possible to express certain kinds of constraints using just triple patterns, or the user may not know how to (for example, Westerns with a particular plot line involving soldiers). On the other hand, the knowledge base itself does not have every conceivable property. For example, there may be a lot of facts about compositions, but no predicates with classical music.

The key to overcoming this obstacle is to extend the knowledge base with textual contexts from the Web. For every RDF triple in the database, we can reach out to the Web and gather text snippets where the triple occurs.

Now we associate words and phrases from these textual "witnesses" with each triple, and make them queryable as if they were a fourth dimension added to the three SPO dimensions. Inspired by XQuery Full-Text, we have developed such a SPARQL extension with the following specific syntax [6, 7]:

```
SELECT ?c, ?m
WHERE {
  ... ?c composed ?m{"classical music"} . }
```

where "classical music" is a phrase to be matched in one or more of the witnesses for the triples that qualify for the triple pattern. Similarly, filtering Westerns with a particular plot line involves adding the appropriate keywords. An example from our system [7] is shown in Figure 2—the order in which results are shown differs on whether a filter condition ("soldier") has been added to the triple pattern or not.

However, it now seems that the user must be familiar with the specific terminology "classical music" in the witnesses. But, the query relaxation for SPO patterns introduced previously, can naturally be extended to keyphrases as well if the user wishes to explore this. First, we could offer a dialog to the user with suggestions on related words and phrases such as "operas", "symphonies", "cantatas", "string quartets", etc. In IR this is known as query expansion; it can be done with explicit user involvement, or transparently to the user. The semantics for a qualifying triple is that its witnesses should contain at least one of the text terms, not necessarily all. We can plug in a suitable ranking model based on IR principles.

Aggregate. The results of the previous steps may overwhelm the user with too many answers (especially if query relaxation is used). Therefore, it is crucial to aggregate the results into an easily digestible form. Grouping and ranking are obvious ideas, and can even be utilized together.

Different groups (clusters) have different statistical evidence for their validity and informativeness. In the example question about swine-flu vaccines, there are definitely many conflicting sources, and a good answer needs to be backed by statistical mass and/or authoritative sources. This issue is important for aggregation into groups, but also shows up for ranking individual answers. A good answer needs to reflect salient entities and properties rather than exotic facts about long-tail entities. Estimating salience is non-trivial, especially if the corpus does not have redundancy—that is, confidence in a fact can no longer be estimated using frequency measures on the corpus.

Moreover, if the system has to handle "close-but-related" queries as well as textual conditions (both of which users can specify in the filter-and-refine step), developing an efficient ranking mechanism which integrates all these features becomes challenging. Recent efforts in this direction include [6, 7].

5. CASE STUDY 2: IQ WITH GRAPH-BASED KEYWORD SEARCH

In the previous case study, the user needs to have some knowledge about the structure of the data graph. For instance, in the composer example above, the user needs to know that there are triples matching the pattern ?c ?prop ?m. Even with the extension ?c ??prop ?m that supports path variables, it is still assumed that the path is from a composer (which is bound to ?c) to a movie (which is bound to ?m). It is possible, however, that the data graph is heterogeneous and in some cases, the edges are reversed, namely, there are triples matching ?m ?prop ?c. In such cases, the path variable will not assist in finding existing answers (unless more patterns are added). In summary, it is rather hard to start a search with the approach of the previous case study, when the user lacks any knowledge about the structure of the data graph.

In this case study, we assume that the user has either no knowledge or a very limited one about the underlying data graph. Thus, the *query* is just a set of keywords. The API may include, in addition to those keywords, some control characters for advanced search. From the user point of view, the search is carried out in two dimensions, namely, the types (i.e., schemas) of answers and their particular instantiations. For example, for the keywords "Italy" and "Israel," there are several types of answers. One is that the two countries are members in some organization. Another type is that the two countries are located on the shores of the same sea.

The data model is a directed graph, as described in Section 2. An *answer* is a *non-redundant* subtree t of the graph, such that t *contains* all the keywords of the given query. Containment means that each keyword may appear anywhere in the tree, that is, in the name, type or context of a node (or an edge). Non-redundancy means that an answer does not have a proper subtree that also contains all the

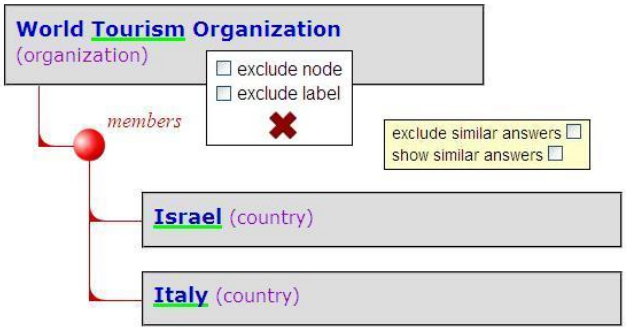
keywords of the query. Note that non-redundancy does *not* imply minimality, and a query could have a large number of answers. The *schema* of an answer is the tree obtained by ignoring the names and contexts, that is, each node (and edge) has only a type.

In this section, we consider the system demonstrated in [1], which is based on [8]. There are other systems for keywords search over data graph [11], but only [1] facilitates search in both dimensions, namely, answers and their schemas.

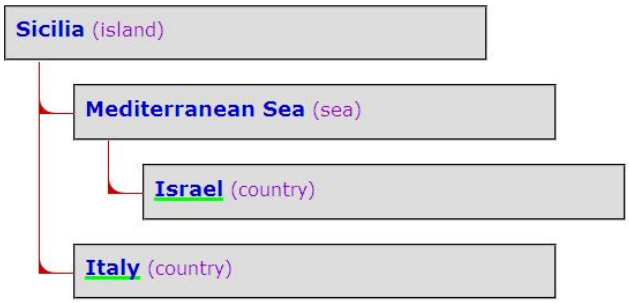
Exploration. A major problem with typical keyword search over data graphs is that the user may be inundated with too many answers that have the same "flavor" (i.e., schema). For example, both Italy and Israel are members in many international organizations. Hence, for the query "Italy Israel," there are going to be many answers—one for each organization. The user might have to browse through many pages until she gets something different. In [1], there are search options that enable the user to zoom in on the "flavors" of her choice. In particular, the user can choose an answer and specify that either she does not want to see more similar answers or all she needs are additional similar answers ("similar" means "with the same schema"). She can also specify that subsequent answers should not include particular names or types (i.e., labels). Figure 3(a) shows an answer for the query "Italy Israel." Note that names are capitalized, whereas labels (identifying types) are not; in addition, a label is shown inside parentheses when there is also a name. The schema of the answer, in Figure 3(a), connects two countries through membership in some organization. (In the data model of [1] only nodes have names, types and contexts.) By checking the option "show similar answers," the user can browse through all organizations in which both Italy and Israel are members. Alternatively, the user can exclude the labels "organization" and "members" so that subsequent answers will not include them. Consequently, the user will immediately get an answer that does not show any information about members of organizations. One such answer is shown in Figure 3(b), where the two countries are linked due to the fact that Israel and an island of Itlay are located on the shores of the Mediterrannean Sea. There is a subtle difference between choosing the option "exclude similar answers" versus excluding the labels "organization" and "members." The former, but not the latter, allows subsequent answers to be about organizations and members as long as their schemas are not identical to the one already seen.

In [1], there is also a mechanism for diversifying answers. The main idea is to apply adaptive ranking that takes into account similarity to answers that have already been shown to the user. Two answers are similar if they have the same schema, or if they have two subtrees that are identical or have the same schema. The ranking function is augmented with a parameter that takes the degree of similarity into account, and as a result, the next page of answers is likely to show results that are substantially different from previous ones. This mechanism does not require user intervention (other than turning it on). The user, however, has the option of tweaking the parameters that measure similarity.

A query may contain keywords that do not appear in the data graph. For example, the user is interested in "Italy" and "Israel" in the context of "music," but the data graph only contains the first two keywords. Similarly to Section 4,



(a) Members in the same organization



(b) Located on the shores of the same sea

Figure 3: Two answers with diverse schemas for the query "Italy Israel"

we can extend the data graph with textual context from the Web. But now there is a need for a suitable ranking function that also takes the context into account. One approach is to first explore the data graph without the context, while ignoring the keywords of the query that appear only in the context. Once some specific schemas are selected, the ranking takes all the keywords of the query into account and uses textual "witnesses" (as described in Section 4). However, this approach may not be as effective and quick as an exploration that uses all the keywords from the outset. An alternative is to extend the ranking technique of [8] by applying IR methods to the textual context.

Filter and Refine. The exploratory options described above (e.g., "show similar answers") are effective in practice, but are quite rudimentary. A more expressive way is to use trees (of answers produced thus far) in order to create SPARQL queries, similar to the approach of Section 4. There is a natural corresponds between a tree and a conjunction of triple patterns. There is, however, some latitude when creating triple patterns from a tree. We can use the names of the nodes and edges of the given tree, thereby creating a conjunction that matches just that particular tree. Alternatively, for some nodes and edges, we can take just the type or use a variable, and consequently, the generated SPARQL query would match many answers. We can also use the contexts of the nodes and edges in order to create additional selection criteria, as described in Section 4. The user need not be aware of this translation into a SPARQL API, because the system would do it automatically based on some interaction with the user through a high-level UI. Note that the SPARQL queries of Section 4 correspond, in

general, to graphs and not just trees. Thus, the user can actually construct a SPARQL query corresponding to a graph, based on the insight she has obtained from several answers with diverse schemas.

Aggregate. We focus here on how to combine results that come from different data sources. As an example, consider the query "find movies that were shot in countries that border Italy." We can decompose it into two sub-queries. (1) "?X = find countries that border Italy," and (2) "find movies that were shot in ?X." The first sub-query could be executed on the CIA World Factbook, and the second—on the IMDB (i.e., Internet Movie Database).

In general, the challenge is how to decompose the original query, and how to join the results of the sub-queries. One approach is to add support in the UI for decomposing the query into several subsets of keywords, according to the available data graphs, and join the results of the sub-queries as follows. Initially, the user executes the first sub-query, which is "countries that border Italy" in our example. Once she zooms in on the relevant answers, she selects specific nodes that are used for creating instantiations of the next query. In the above example, each country returned by the first sub-query is used to instantiate ?X in the second sub-query "movies shot in ?X."

6. CHALLENGES AND OUTLOOK

The IQ paradigm for search is a natural consequence of the complexity of the information needs of users as well as the underlying data. There has been a considerable amount of research on the many different aspects that a system built on this paradigm should support. Indeed, examples of these were illustrated in our case studies. However, there are still several hard problems which need to be solved. We list a few of these below.

Exploration. In order to make user interactions as easy as possible, the system has to understand natural-language questions—a hard problem on which some progress has been made in the last few years (see, for example, [10, 12]). The questions need to be mapped to a query language in order to make the processing more precise and efficient, but there will be parts of the question which may not map to any structure. We believe that data models such as the text-augmented RDF where both the structured, as well as unstructured text are queriable in a unified manner can help in more robust translations. One way to leverage this is, for example, to map the question into structured triple patterns when possible, and leave the rest as keywords attached to certain triple patterns. The triple patterns themselves may in turn be automatically derived by doing a keyword search on graphs and extracting the most interesting patterns from it.

Even in the case of expert users who may enter formal queries directly, there is still the issue of query correctness and whether the query will return a non-empty result set. Interactions in the form of close-but-correct query suggestions may be needed to guide the user.

Deriving these queries over a single knowledge-base is hard enough, and even more challenging is the case when there are several (possibly overlapping) sources. Choosing the appropriate set of sources itself becomes difficult, unless the user is in the loop.

Filter and Refine. Learning the history of the user alleviates user frustration, by providing, for example, a personalized ranking of results, from which the user can quickly select interesting ones. Moreover, answers can be tuned based on whether the user asking a query is an expert or a child.

On-the-fly personalization, where the system learns as the user goes through the search process is also a challenge. For example, the query on composers of western music could return a long list. But as the user chooses one or two composers who also compose classical music, the system could immediately change the ranking and return classical composers higher up.

Aggregate. The notion of whether two results are identical is important in some of the aggregations that we mentioned. For example, we may need to perform joins on results returned by two different knowledge sources, or group near-duplicate results into clusters.

Even in cases of joins involving just entities, there has to be an implicit entity disambiguation step. Are the two attributes the same entity despite having different names, or are they different entities despite having the same name?

In the universe of knowledge rather than Web pages, the notion of duplicates is more involved. For example, we may find the same composer related to operas in one answer and to symphonies in another answer. Since both operas and symphonies are considered classical music, are these two results duplicates? Apart these semantic issues, there is also no support for grouping in SPARQL, and the notion of grouping in text-augmented RDF data is totally unexplored. We doubt that generic clustering methods are suitable here.

User Interface. UI issues play a big role in identifying the basic functionality of the system. For example, consider the connection between doing aggregation of results and the display of results. Suppose a user asks for countries bordering Italy. Rather than seeing one answer for each bordering country, she would like to see all the bordering countries in one answer. However, if the user also wants a typical food recipe for each country, then aggregating all the countries and their recipes in one answer might create a display which is too cluttered. In general, there is a tradeoff between many similar answers and one aggregated answer. The former has a simple and clear display of each answer, but the list of those answers could be long. An aggregated answer might be too cumbersome to display neatly. So, it is not clear how aggregation can be carried out automatically at the level that is most suitable to the user.

Not all browsing and searching happens while a user sits at her machine, nor is every user patient enough to follow through on each step. How does the system need to adapt, if, for example, the only interface between the user and the system is a cell-phone? The obvious consequence of this is that no user would start with an exploration step, but rather convey her entire information need in one shot, and in speech. Apart from having to translate from speech to text, the system has to automatically determine when user feedback is needed and to minimize the interactions as much as possible.

7. CONCLUSIONS

Iterative querying is a natural paradigm for users with advanced information needs and users already do this with

search engines. Our goal in this paper has been to explore some of the issues involved in building a system supporting this paradigm in the context of *structured knowledge-bases*. We showed how this could be done with a couple of case studies and highlighted some of the tools which can already be used to implement this kind of system.

In conclusion, there is already considerable excitement over the availability of large and structured knowledge-bases. The main bottleneck is to figure out how to make use of them effectively. We believe that studying the problems of iterative querying, both at the conceptual as well as the user level will substantially advance the process of finding answers to questions (of any complexity!).

Acknowledgements. This work was partially supported by The German-Israeli Foundation for Scientific Research & Development (Grant 973–150.6/2007).

8. REFERENCES

[1] H. Achiezra, K. Golenberg, B. Kimelfeld, and Y. Sagiv. Exploratory keyword search on data graphs. In *SIGMOD*, 2010.

[2] K. Anyanwu, A. Maduko, and A. P. Sheth. Sparq2l: towards support for subgraph extraction queries in rdf databases. In *WWW*, 2007.

[3] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3), 2009.

[4] M. Cafarella. Extracting and querying a comprehensive web database. In *CIDR*, 2009.

[5] A. Doan et. al. Information extraction challenges in managing unstructured data. *SIGMOD Record*, 37(4), 2008.

[6] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, and G. Weikum. Language-model-based ranking for queries on RDF-graphs. In *CIKM*, 2009.

[7] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum. Searching rdf graphs with SPARQL and keywords. *IEEE Data Engineering Bulletin*, 33(1), 2010.

[8] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, 2008.

[9] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti. Collective annotation of wikipedia entities in web text. In *KDD*, 2009.

[10] Y. Li, H. Yang, and H. V. Jagadish. Nalix: A generic natural language search environment for xml data. *ACM Trans. Database Syst.*, 32(4), 2007.

[11] B.-C. Ooi, editor. *Special Issue on Keyword Search.* IEEE Data Eng. Bull., March 2010.

[12] J. Pound, I. Ilyas, and G. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, 2010.

[13] A. Schaefer, M. Jordan, C.-P. Klas, and N. Fuhr. Active support for query formulation in virtual digital libraries: A case study with daffodil. In *ECDL*, 2005.