Entity Ranking and Relationship Queries Using an Extended Graph Model

Ankur Agrawal
IIT Bombay
ankuragrawal.iitb@gmail.com

S. Sudarshan
IIT Bombay
sudarsha@cse.iitb.ac.in

Ajitav Sahoo
IIT Bombay
ajitavsahoo@gmail.com

Adil Anis Sandalwala
IIT Bombay
sandalwalaadil@gmail.com

Prashant Jaiswal
IIT Bombay
prash.jai@gmail.com

ABSTRACT

There is a large amount of textual data on the Web and in Wikipedia, where mentions of entities (such as Gandhi) are annotated with a link to the disambiguated entity (such as M. K. Gandhi). Such annotation may have been done manually (as in Wikipedia) or can be done using named entity recognition/disambiguation techniques. Such an annotated corpus allows queries to return entities, instead of documents. Entity ranking queries retrieve entities that are related to keywords in the query and belong to a given type/category specified in the query; entity ranking has been an active area of research in the past few years. More recently, there have been extensions to allow entity-relationship queries, which allow specification of multiple sets of entities as well as relationships between them.

In this paper we address the problem of entity ranking ("near") queries and entity-relationship queries on the Wikipedia corpus. We first present an extended graph model which combines the power of graph models used earlier for structured/semi-structured data, with information from textual data. Based on this model, we show how to specify entity and entity-relationship queries, and defined scoring methods for ranking answers. Finally, we provide efficient algorithms for answering such queries, exploiting a space efficient in-memory graph structure. A performance comparison with the ERQ system proposed earlier shows significant improvement in answer quality for most queries, while also handling a much larger set of entity types.

1. INTRODUCTION

Over the last decade, there has been a lot of work on keyword search over structured and semi-structured data. Some of this body of work focuses on finding a closely connected set of data items containing specified keywords, for example [4, 10, 1, 9]. In contrast ObjectRank [2] extended the idea of PageRank to compute keyword specific ranks for objects in a connected graph. A similar idea of near queries was also mentioned briefly in [11]. All the above work focused primarily on structured data.

In recent years, search over annotated text data has received increasing attention. This work is motivated in part by the availability of annotated text in Wikipedia, and by the availability of text annotators for named entity recognition/disambiguation, such as [13, 19], which can work on web scale data. Such annotations add semantic links to text, identifying mentions of entities in text, and organizing the entities into a type or category hierarchy. For example, the occurrence of the words "Kleinberg" in text may be identified as a mention of the person entity "Jon Kleinberg".

Suppose the annotation on a text corpus have identified occurrences of person entities (amongst other types of entities). We can then run queries such as "find persons near Web search"; the basic idea is to find mentions of entities of type person close to the words Web and search, and aggregate over multiple such occurrences to rank persons in terms of their proximity to the words web and search. Work in this area includes [5, 6, 8] and [7]; see Section 6 for more details.

In general, entity ranking involves finding specific entities as answers to queries. The user submits the search keywords and also the target type of the desired answers. (We use the words type and category interchangeably, since both terms have been widely used in prior work.) Some examples of such queries as obtained from the INEX 2008 track are: "Find a list of musicians who appeared in at least one of the Blues Brothers movies", and "Find a list of the state capitals of the United States of America".

Wikipedia is often used as the source of entities, and the YAGO category hierarchy [18] (which provides a cleaned up version of Wikipedia categories combined with the WordNet ontology) is used to associate entities with a hierarchy of categories. Several systems, such as Yago, also extract relationships from unstructured information and represent them, for example, using RDF or even relational schemas. Structured queries are then run on the structured data, by systems such as Naga [12], [3] and [17]. However, the number of extracted relationships are limited, and the integration of unstructured and structured information is limited. See Section 6 for more details.

The ERQ system [14, 15] has worked on more complex queries called entity-relationship queries, that can look for relationships between entities. Queries can specify entities

The 18th International Conference on Management of Data (COMAD), 14th-16th Dec, 2012 at Pune, India.

in a manner similar to entity ranking queries, but additionally specify desired relationships through keywords. As an example from [15], a query can ask for "persons related to Stanford who have founded companies in silicon valley"; more formally the query asks for "person entities near Stanford that are related to company entities near silicon valley by the term founded".

The systems mentioned above exploit entity annotations, but do not exploit the graph structure of the underlying data. For example, they cannot answer a query of the form "find universities near Nobel prize" unless there are mentions of the term Nobel prize near the university name. If person entities related to Nobel prize, are also related to a university entity, we would consider the university to be related to Nobel prize. Such transfer of prestige does occur in graph based systems such as Object Rank [2] and BANKS [11], but those systems do not support nodes containing annotated text. Our goal is to have a unified model that handles both graph information and annotated textual data.

As a first attempt to address the issue, we treated the Wikipedia corpus as a graph, with documents as nodes and inter-page links as edges, and ran the near query implementation of [11] on the graph. However, the results were very disappointing; the main reasons were (a) the graph is very densely connected and (b) it makes no sense to consider a link at the end of a long Wikipedia page to be related to a word that occurs early in the page.

To address the above problem, we introduce the notion of a graph where nodes contain words and edges, occurring at specified offsets. When we traverse the graph to answer a query, we take the offsets into account, in a way that we describe later in the paper. This extended graph model is well suited to Wikipedia data, to annotated Web pages, as well as to traditional structured data, and can be used in systems that integrate different types of data.

We then show how to use the extended graph model to define scoring models for entity and entity-relationship queries, and to derive efficient algorithms for answering such queries.

The contributions of this paper are as follows:

1. We present (in Section 2) a new graph model that allows nodes to contain terms as well as links at specified offsets. This model combines the best features of the graph model, and the document models, both of which have been widely used in the past.

2. We present (in Section 3) new methods for scoring answers to near queries taking the new graph model into account.

   Unlike earlier work on entity ranking and entity-relationship queries, our model does not require the user to provide a precise specification of the desired type of the results; instead, we allow type-keywords to describe the desired type. Answers are scored based on how well the type matches the given type-keywords, and the entity matches the remaining keywords.

3. We then present (in Section 4) a scoring model for entity-relationship queries, again based on the extended graph model.

   We also present efficient algorithms for answering entity-relationship queries in the above graph model.

4. We present (in Section 5) several optimizations improve result quality.

5. We present a performance study (in Section 7) which shows that our techniques give good result quality, outperforming [15] on most queries.

2. DATA MODEL

We now describe our extended graph model, and then outline how semi-structured datasets from Wikipedia and YAGO [18] can be represented in the extended graph model.

2.1 Extended Graph Model

The basic data model we use is a labelled directed multigraph $G = (V, E)$, where V is a set of vertices and E a multiset of edges. Our multigraph model has two further extensions to better handle documents.

1. Vertices can have an associated text description, modeled as a document; such vertices have an associated set of *(term, offset)* pairs. The offset denotes the relative position of the term from the start of the document. Vertices that do not represent documents can still have text descriptions, with all terms assumed to be at offset 0.

   Vertices can have associated labels; for example, when modeling Wikipedia, these labels can be used to distinguish regular entity nodes from category nodes. Vertices can also store other information, for example a node prestige may be associated with each node.

2. Edges are directed. Edges can represent a hyperlink from one document to another; each such edge $e = v1 \rightarrow v2$ has an associated offset $e.offset$ which is an offset within the document represented by $v1$ where the hyperlink occurs. There can be multiple edges from one vertex to another, at different offsets, which is why we use a multigraph model. Edges that do not represent hyperlinks are assumed to have an offset of 0.

   Edges can also have associated labels; for example, when modeling Wikipedia data, edge labels can be used to distinguish edges linking a node to its category, from edges linking a node to a non-category node. Edges can also have an associated edge weight.

We call the above graph model as the *extended graph model*.

The extended graph model only stores nodes and edges with offset information. The mapping from terms to nodes (including offset information for term occurrences) is stored separately, in a full text Lucene index. The term frequency (TF) of each term in a document can also be stored in Lucene; for example, terms in a document title can be given a higher TF. The node prestige of a node can also be used to boost the score of the corresponding document in Lucene.

2.2 Representing Wikipedia Data

In Wikipedia, every entity is stored as a separate document (a Wikipedia page) also called articles. Wikipedia articles are all linked or cross-referenced. These articles are categorized according to the type of entity it represents. Wikipedia provides us with category types, into which an author could categorize the pages.

In our model, each Wikipedia page/document represents an entity, which is the basic unit of our search and thus, it is represented by a node in the graph. There are two types of nodes in our model:

- *category nodes* (representing Wikipedia categories)
- *entity nodes* (representing all other Wikipedia pages)

Each vertex has a label identifying whether it is a category vertex or a entity vertex. In addition, each vertex has a separate label denoting its page-rank, pre-computed as described later. If we integrate other Web pages into our graph, we could use a new node type, *web-page* node, to represent such Web pages,

Labels are also associated with the edges to identify the edge type; the different types of edges in the graph are as follows.

1. Document to entity edges, which link from a document to entities referenced in the document. Each entity has an associated document in Wikipedia. The offset associated with such an edge is the token offset of the start of the link in the document.
2. Edges denoting the 'belongs to' relation from an entity to a category The offset of such edges is 0.
3. Edges denoting category to category hierarchy; the offset of such edges in 0.

Since, a single data graph is built for both entities and categories different parts of the graph can be traversed based on the edge type.

Edges linking entities to categories that denote the "belongs to" relationship are treated specially for the purpose of ranking.

As in [4, 11], the node prestige of a node is a measure of its importance disregarding query keywords, and is computed using a biased PageRank computation with edge weights, as described in [11], with teleport probability of 0.3. Offset information is ignored, and all edges in the original graph are treated as being of unit weight.

As in [11], for each directed edge $u \rightarrow v$ in the original graph, we introduce a reverse edge $v \rightarrow u$, if such an edge is not already present. Each reverse edge is assumed to be at offset 0, and its weight is defined as the indegree of v.

The Wikipedia category hierarchy has a number of problems, such as cycles, and improper nesting of categories. For example, Jerry Yang, the founder of Yahoo! is in the category Yahoo!, and thus indirectly (after a few more levels in the hierarchy) under the category Companies. Based on the hierarchy, we expect each entity to belong to higher level categories also. However, we would certainly not expect Jerry Yang to be categorized as a company.

To avoid these problems we used the category hierarchy of the YAGO ontology [18]. YAGO includes all Wikipedia entities, as well as conceptual categories from Wikipedia, but replaces the Wikipedia category hierarchy by the WordNet hierarchy, suitably integrated with the Wikipedia categories (which now form the leaf level of the category hierarchy). This not only improved the quality of results, linking entities only to relevant categories in most cases, but also reduced the execution time significantly.

3. NEAR QUERIES

In this section, we first describe our model for *near queries*, and then describe how answers are scored using our extended graph model.

3.1 Near Query Model

A *near query* q can be specified as

find C **near** (K)

Where C is one or more keywords specifying the target entity type for the answer, and K is a set of keywords; phrases enclosed in double quotes can also be used in place of keywords to ensure that the keywords appear together in the order specified.

Example. Consider a user searching for the list of movies in which actor Robert De Niro has played a part and is directed by famous Hollywood director Martin Scorsese. The **near query** formulation of this query will be:

find films **near** (directed "martin scorsese" "robert de niro")

Here the keyword *films* gives the type information C, and the set K is equal to {directed, martin scorsese, robert de niro}.

Near queries using the above syntax were supported in the BANKS system [11]. However, as mentioned earlier, when we attempted to use the BANKS near query model on the Wikipedia corpus, with Wikipedia pages modeled as nodes, the performance was very poor; the reason is that nodes have many keywords and many links, and a keyword occurring early in a page often has little connection to a link occurring late in the page. In Section 3.2 we describe how to score answers based on proximity of keywords to links or entity mentions.

We use the following terminology in the rest of the paper:

- *categoryKeywordList*: Keyword (or set of keywords) C before the meta-word *near* which specifies the target categories (entity types).
- *nearKeywordList*: The set of keywords following the meta-word *near*. Each keyword is separated by space within the parenthesis. Keywords within quotes are considered as phrases and as a result, single keywords.
- *nearKeywordOriginSet*: The document pages that contain the keywords in the *nearKeywordList*.
- *relevantCategorySet*: The set of categories relevant to *categoryKeywordList*.

We could use either of the following alternatives to decide which documents form the nearKeywordOriginSet:

- AND semantics: Every document in the *nearKeywordOriginSet* must contain all the keywords in *nearKeywordList*.
- OR semantics: Every document in the *nearKeywordOriginSet* must contain at least one keyword from the *nearKeywordList*.

In our implementation we use the default scoring mechanism of Lucene, which corresponds to the OR semantics.

3.2 Scoring model

We now see how to score answers to near queries, using the idea of activation spreading, as well as the relevance of a category to the category keywords in the query. Our technique extends the spreading activation technique used for near queries in [11], by taking the proximity between keywords and links (calculated using offset values) into account. Our scoring models have a number of parameters; default values are specified for some of the parameters when they are introduced, but the values used in our experiments are given later, in Section 7.3.

3.2.1 Activation Spreading

As described in [11], activation spreading is initiated from the nodes containing keywords, and spreads activation to neighboring nodes. The following are the key features: (a) The initial activation from a given keyword is spread to nodes containing that keyword, in proportion to the node prestige (PageRank) of each such node. Nodes that receive the maximum activation form the results of the near query. (b) Each node retains part of its incoming activation, and spreads the remaining to its neighbors; the fraction spread to each neighbor is inversely proportional the weight of the directed edge from the node to its neighbor. (c) Activation received from multiple neighbors is combined using a combining function. Activation spreading continues until the amount spread falls below a specified threshold.

We now describe how the above scheme is modified in our context.

3.2.1.1 Initial Activation.

In our context, activation spreading starts from nodes representing the documents which contain the keywords. The initial hit set for query keywords is obtained using the searcher available in Lucene. Lucene also returns the score of the documents that are obtained as hits during the search.

The initial activation of a node is a combination of of the relevance of the node to the keywords, given by the Lucene score for the node, and the node prestige of the node. The *initialActivation* value for each node is calculated from these two scores by combining them either additively:

$$NodePrestige * \alpha + LuceneScore * (1-\alpha) \qquad (1)$$

or multiplicatively:

$$[LuceneScore^{\alpha}] * [NodePrestige^{(1-\alpha)}] \qquad (2)$$

Here α is a distribution factor that can be tuned to give more weight to the desired score. By default we use multiplicative combination with $\alpha = 0.5$.

Note that the above model is a little different from the near query model of [11]; that model allowed each near keyword to appear in a different tuple, and spread activation separately for each near keyword. The activation scores were combined across multiple keywords either multiplicatively (for the AND semantics) or additively (for the OR semantics). In the context of search on Wikipedia and other document collections, it makes more sense to compute the initial activation across all keywords, and then spread activation only once.

3.2.1.2 Proximity and Spreading of Activation.

When spreading activation from a node, an attenuation factor μ is used. Every node spreads a fraction $1-\mu$ of its activation to its neighbors and retains the remaining μ fraction for itself. By default, we set $\mu = 0.75$. As in BANKS, the fraction of activation spread to each neighbor depends on the edge weights. However, the spreading of initial activation is special cased. The fraction of the initial activation spread to each outlink depends on the proximity of the outlinks to the near keywords. Intuitively, if a keyword and a link to an entity occur in proximity in a document, we believe that the entity is related to the keyword; the closer the occurrences, the higher is the estimate of relevance of the entity to the keyword. We use this idea to define the amount of activation transferred to each of the entities linked with the document.

The position offset of each term of a document is stored with the index. And the offset information for every link in a document is stored in the graph during pre-processing phase. This offset is calculated with respect to the start of the document. The amount of activation spread to the entity pointed to by the link is proportional to the distance between the link and the query keyword in the document.

The function to calculate the proximity of a link with respect to a keyword must be such that its value degrades as the distance between the link and the keyword increases.

Formally, if a word w occurs at position i, and a link to an entity at position j, then if the position j is closer to i, the propagated activation for word w at that position would be larger than the propagated activation at a position farther away. The issue of how the activation should decay with distance is studied in [16]. We use the Gaussian kernel function to calculate the proximity score.

$$k(i,j) = exp[\frac{-(i-j)^2}{2\sigma^2}]$$

The initial activation associated with a node in nearKeywordOriginSet is spread to the outlinks of the node in proportion to proximity (using the formula defined above) based on the distance between the outlink and the nearest occurrence of the near keyword; with multiple keywords, we take the distance as the minimum, across all keywords, of the distance as above.

3.2.2 Category Relevance

The answers to a keyword query must satisfy the target type information specified in the query. In the near query model, a user specifies the target type for the answers by providing relevant keywords. In the context of near queries, this target type specifies one or more categories, and the result entity must belong to one of these categories. Each category has a category relevance score, which is used in entity ranking.

The categories are indexed separately, as documents, and the categoryKeywordList specified in the query is used to retrieve relevant categories; we call the set of categories returned as the *relevantCategorySet*. We use the relevance score that Lucene returns for each category as the relevance of that category.

To calculate relevance score of an entity, the set of categories to which this entity belongs is retrieved. It is then checked if any of these categories belongs to *relevantCategorySet* and the maximum of the Lucene scores of such categories is taken as the category relevance of that entity.

3.2.3 Combining Activation and Category-Relevance Scores

After spreading of activation, the result of activation spreading is stored in a priority heap ResultHeap. To get the final score *score* of each entity, the activation score *actScore* and the the category relevance score *relScore* of each node in ResultHeap are combined additively as follows:

$$score(e) = actScore(e) * \eta + relScore(e) * (1 - \eta) \qquad (3)$$

The parameter η denotes the weight given to the score. Entities in the result are sorted by their scores $score(e)$, and output in descending order.

3.3 Discussion

Our scoring model for near queries spreads activation from entities to other entities that are referenced in the Wikipedia page of the entity (only links in or before the infobox are considered, since Wikipedia pages often have less relevant links later in the document).

For example, if we search for *Universities near "web search"*, we may find many references to a person working on web search techniques near keywords "web search". Spreading activation from such person entities can then give us a university as an answer.

Earlier systems such as [5, 8, 7] and [15] (described in more detail in Section 6) cannot do this, since they only look for co-occurrences of entities and keywords to determine their association.

4. PROCESSING ENTITY-RELATIONSHIP QUERIES

In this section, we focus on issues involved in answering entity-relationship queries. The query model we use is basically the same as that described in [14, 15], but we use a different scoring system, as well as a different system design and implementation to solve such queries.

In our formulation of the entity-relationship queries, as in [15], we have a list of *entity variables*. Unlike in [15], each entity variable is associated with a list of keywords specifying the category of the desired entities called *categoryKeywordList*, and these category keywords are used to identify one or more categories to be considered for the entity variable.

Each entity variable can be associated with zero or more predicates. There are two kinds of predicates in an entity-relationship query :

- **Selection Predicate :** A selection predicate consists of an entity variable and a list of keywords specifying the criterion on the selection of entities. We call the list of keywords as the *NearKeywordList*.
- **Relation Predicate :** A relation predicate consists of two or more entity variables and a list of keywords specifying the relationship between the entities described by these variables.

As an example consider the following query from [15]: *"Find companies and their founders, where the companies are in Silicon Valley and founders are Stanford graduates"*. Simple entity ranking systems are not adequate for such complex information needs. Li et al. [15] provide a solution to this problem, by designing an entity-centric structured query mechanism called *entity-relationship queries*. The above query expressed in the language of [15] is as follows:

> **select** X, Y
> **from** person X, companies Y
> **where** X:[Stanford graduate]
>   **and** Y:["Silicon Valley"]
>   **and** X,Y: [founder]

In the above query, X and Y are entity variables, bound to specific entity types, while the keywords act as predicates.

The above query can be expressed in our syntax as follows:

> **find** person(x) **near** (Stanford graduate) **and**
>   company(y) **near** ("Silicon Valley")
> **such that** x,y **near** (founder)

In this query, there are two entity *variables* named x and y. The *categoryKeywordList* for variable x contains the word *"person"* and for variable y, it contains the word *"company"*. Variable x has a *selection predicate* consisting of keywords *"Stanford"* and *"graduate"* while variable y has a *selection predicate* consisting of keyword *"Silicon Valley"*. The query also has a *relation predicate* on variables x and y consisting of keyword *"founder"*.

As in ERQ [15], an entity variable can have more than one selection predicates. For example

> **find** person (x) **near** ("Turing Award")
>   **and near** (IBM)

If we had instead used **near** ("Turing award", IBM), we would only get entities mentioned near co-occurrences of Turing Award and IBM. In contrast, by using separate selection predicates, the set of documents that establish that a person is associated with "Turing Award" can be different from the set of documents that establish that the person is associated with IBM.

4.1 Scoring ERQ Answers

Scoring and ranking of the results is an important task. The important concepts involved in ranking the entity search results are:

- **Proximity**: Entities and keywords should be placed close to each other in the text. Intuitively, the closer they are to each other, the more likely is their association with each other.
- **Relevance to category**: As the category itself is specified in the form of keywords, there is uncertainty involved regarding the relevance of an entity to the specified category keywords.
- **Number of Evidences**: The more number of times a set of entities appears with the keywords in the text, the more likely is their association.

First, we score each answer entity tuple for each predicate separately. Finally while merging the single predicate results, we calculate the aggregate score for each answer tuple by taking the product of the single predicate scores for the entities involved.

4.1.1 Selection Predicate Scoring

A selection predicate in an entity-relationship query is basically a near query, which we saw in Section 3. To compute score of an answer entity e on a selection predicate p, we use the scoring model for near queries described in Section 3. We combine the activation score *actScore* and the category relevance score *relScore* using the additive combination:

$$score_p(e) = actScore(e) * \eta + relScore(e) * (1 - \eta)$$

The combined score is a normalized score and the value is always between 0 and 1.

If there is more than one selection predicates over the same variable, we use the following formula, where $p_1, p_2, \ldots p_n$ denote the selection predicates on a single entity variable.

$$Score_{p_1,p_2,\ldots,p_n}(e) = (\Pi_{i \in 1 \ldots n} actScore_{p_i}(e)) * \eta \\ + relScore(e) * (1 - \eta)$$

4.1.2 Relation Predicate Scoring

Consider a relation predicate answer tuple $< e_1, e_2, ..., e_n >$, and the set of occurrences O of the entities in the answer tuple and the keywords corresponding to the predicate appearing together in the text. We calculate the score for the relation predicate p as:

$$score_p(< e_1, e_2, ..., e_n >) = \sum_{o \in O} exp[\frac{-(TokenSpan(o))^2}{2\sigma^2}]$$

where $TokenSpan(o)$ is the number of tokens present in the minimal scope in o covering all the entities and keywords. λ is an input parameter specifying the threshold for the maximum allowed value of $TokenSpan$ and all occurrences beyond this threshold are ignored.

4.1.3 Aggregating Single Predicate Scores

After computing single predicate scores for each predicate result, we finally merge the results and calculate the aggregate score for the final answer tuples. The aggregate score *aggScore* is calculated as :

$$aggScore = \prod_{p \in selPreds} score_p * \prod_{p \in relPreds} score_p^{\gamma}$$

where *selPreds* and *relPreds* denote the selection and relation predicates, and γ is an input parameter controlling the weightage given to the relation predicate scores.

4.2 Query Evaluation Algorithm

Given an entity-relationship query, our approach is to first evaluate all the selection predicates individually to find the list of entities for each entity variable involved in the query. We then use these entity lists to evaluate the relation predicates to find tuples of related entities. Finally we take a join of the individual predicate result list on entities for same entity variable. In the process, we also collect offset information to finally score the answer tuples and rank them accordingly. We look at the steps involved in evaluating an entity-relationship query in the following sections.

4.3 Evaluating Selection Predicates

A selection predicate in Entity-Relationship Query is exactly a near query. So we directly use the near query evaluation algorithm described in Section 3 to get the list of answer entities for each entity variable, along with their scores.

After this step, we will have a list of <*entity, score*> pairs for each variable. For our example query, the lists would be:
variable x : <Scott_McNealy, 1.0>, <Ken_Kesey, 0.9973>, <John_Steinbeck, 0.9946>, ...
variable y : <Microsoft, 1.0>, <Hewlett-Packard, 0.9944>, <Metro_Newspapers, 0.9942>, ...

4.4 Evaluating Relation Predicates

Relation predicates specify a relationship between two or more entities in terms of keywords. There are two alternative approaches to solve a relation predicate.

Approach 1

- Use the Lucene index to find documents containing the relation keywords, along with their offsets in the documents.
- For each Lucene hit page :
  - Find entity references near those keyword occurrences, using the outlinks from the entity pages (outlink information along with offsets is available in the extended graph representation, stored in-memory).
  - Check whether these entities belong to the selection predicate answer entity list for any of the variables involved in this relation predicate and put them in a list for the corresponding entity variable.
  - Perform a cross product of the lists for the entity variables, to get the answer tuples.
  - Note the offsets of the keywords and the entity links for score calculation.

The problem in this approach is that in most cases, the keywords specifying the relationship are very general (e.g. "join", "found" etc.) and generate a very large number of hits. However only a small fraction of these pages contain links to at least one entity from the selection predicate answer list for each entity variable involved in this relation predicate. Thus processing each document as above causes a lot of useless processing.

We solve this problem using Approach 2 described below.

Approach 2. The result of a single relation predicate, taking into account selection predicates on all the associated entity variables, can be computed as follows.

- Find lists of pages containing reference to at least one of the entities in the selection predicate answer list for each entity variable; this can be done using the inlinks of the corresponding entity nodes, fetched from the in-memory graph representation.
- Intersect these lists to find list of pages containing links to at least one entity from the selection predicate answer list for each entity variable.
- Intersect this list with the hit list for the relation keywords to find all such pages also containing the relation keywords.
- For each page in this list:
  - Perform a cross product of the entity lists for each entity variable present in this page to get the answer tuples.
  - Note the offsets of the keywords and the entity links for score calculation.

```
1: Inputs: List of entity variables: $eVars$,
   List of Keywords: $nKeywords$,
   Mapping of variable to Entity list: $varToEntityMap$
2: Define: $VarToPageMap$: a mapping from entity
   variables to list of pages
3: Define: $VarPageToEntityMap$: a mapping from
   <entity-variable, page> pairs to a list of entities
4: for all $v \in eVars$ do
5:   for all $entity \in varToEntityMap[v]$ do
6:     $pageSet \Leftarrow$ Find all pages pointing to $entity$
7:     for all $page \in pageSet$ do
8:       $VarToPageMap[v]$.Add($page$)
9:       $VarPageToEntityMap[v, page]$.Add($entity$)
10:    end for
11:  end for
12: end for
13: $allLinkPageList \Leftarrow \cap_{v \in eVars} VarToPageMap[v]$
    /* Computes intersection of lists*/
14: $LuceneHitArray \Leftarrow$ Find all pages which contain the
    the keywords $nKeywords$ using the Lucene Index.
15: for all $luceneHitPage \in LuceneHitArray$ do
16:   if $luceneHitPage \in allLinkPageList$ then
17:     Define $varToEntitiesMap$: a map from entity
        variables to a list of entities
18:     for all $v \in$ eVars do
19:       $entityList \Leftarrow VarPageToEntityMap[v, NodeId]$
20:       $varToEntitiesMap[v]$.Add($entityList$)
21:     end for
22:     $answerTuples \Leftarrow \times_{v \in eVars} varToEntitiesMap[v]$
        /* Compute cross product ($\times$) of entity lists;
        optimization using band join described in text*/
23:     $ResultHeap$.addAll($answerTuples$)
24:   end if
25: end for
```

Algorithm 1: Evaluating a relation predicate

An optimization of the this step is to perform a band merge of lists sorted on their offsets, to only match entity and relation keyword occurrences that are present close to each other (in terms of their offsets), instead of performing a cross product of the entity lists. This can reduce costs greatly for pages with many entity references and relation keyword occurrences.

The above intuition is formalized in Algorithm 1.

After this step, we have the list of entity-tuples with their relation predicate scores. For our example query, we will have a list like:

x,y : <(Bill_Gates, Microsoft), 0.9896>,
<(David_Filo, Yahoo!), 0.9745>,
<(Vinod_Khosla, Sun_Microsystems), 0.9257>, ...

4.5 Handling Complete Queries

If a query does not involve any relation predicate, processing is straightforward. If the entity variable in such a query has more than one selection predicate, we need to combine the results of each selection predicate; we use a simple merge join of the results.

If the query involves only one relation predicate, Algorithm 1 gives the desired final answers. In case the query has more than one relation predicate, we process each relation predicate as above, and then do an equijoin on the results of each selection predicate. Currently we do not optimize the join order, since none of our benchmark queries has more than 2 relation predicates, but this could be a topic of future work.

As an optimization, if a query has an entity variable with more than one selection predicate, as well as a relation predicate involving the same entity variable, we can avoid the join of the selection predicate results; instead, when we process the relation keyword we get a list of neighboring entities for each keyword occurrence, and look up such entities in the result entity lists for each of the selection predicates on that entity variable.

5. HEURISTIC OPTIMIZATIONS

We now describe a few heuristics aimed at improving the scoring of results. The effect of these optimizations is studied empirically in Section 7.

Using Wikipedia Infoboxes. In our initial implementation, every term in a Wikipedia article was assumed to be relevant to the entity. However, our initial experiments showed that most Wikipedia articles have a lot of terms that are not very relevant. However, the terms early in the article, in particular those that occur in the Wikipedia infoboxes, are highly relevant. We could have chosen to tailor the ranking scheme of Lucene, but instead chose to use our extended graph model to exploit this information, as follows.

When we build the graph, we assume that a self-link to the same Wikipedia entity is present near each term in the infobox, at a small offset (with default value as 5). Thus, if we find some keyword in the infobox, we add some initial activation to the entity itself. Similarly, we create self links to the Wikipedia page from terms in the first few sentences of each article; for concreteness, we use all sentences that appear before the infobox in the article, since these generally constitute a highly relevant summary of the entity.

Exploiting Wikipedia category specificity by matching near keywords. Another area of performance improvement is the specificity of Wikipedia categories. Wikipedia provides a large collection of categories, many of which are associated with very specific entities. For example, *Novels_by_Jane_Austen*, *Films_directed_by_Steven_Speilberg*, *Universities_in_Catalunya* are all Wikipedia categories.

Users are generally not aware of the presence of such categories, and would query on a higher level category, for example novels, even if they are specifically looking for novels by Jane Austen.

Thus, we look for the *near keywords* in the category titles also. If we find any category whose title contains all the *near keywords*, we judge entities belonging (directly) to the category as being more relevant to the near keywords. If such a category is a subcategory of the original query categories, the resulting entities are directly answers to the original query. But even otherwise, we wish to give extra weight to such entities for the purpose of spreading activation through entities that occur close to occurrences of the near keywords.

To handle both the above goals, we add a constant value (0.2) to the initial activation to entities directly belonging to the above category; if an entity belongs to more than one such category, its initial activation gets increased only once.

We demonstrate the effect of this feature in Section 7.

Spreading activation from articles with title containing the near keywords. Intuitively, if the title of an article contains all the near keywords, all the content in the article can be assumed to be related to the keywords with high probability. We exploit this intuition by spreading activation from such articles to its out-neighbors.

In our spreading activation mechanism, the activation decays for links farther away from the keyword occurrence. In the special case of keywords in the article title, we treat all outlinks early in the article (up to and including the infobox for the article) as closely related to the keyword, even if they are somewhat further off in terms of token offset.

We demonstrate the effect of this feature in Section 7.

6. RELATED WORK

Several systems such as ObjectRank [2], the system of [5] and Entity Search [7] have been developed which return a ranked list of entities as answers to keyword queries.

ObjectRank works on a graph model of data, with entities as nodes, and is based on a biased random walk model which is an extension of the random walk model of PageRank. Nodes also contain descriptive text, with the starting nodes of the walk being determined by which nodes contain the given keywords. The biased random walk determines the score of each entity. The near query model of BANKS, briefly mentioned in [11] uses a similar model, and has similar goals, although details vary.

Chakrabarti et al. [5] describe an entity querying system based on a model where documents have terms as well as entity mentions. Queries can specify the type of the desired entities, and keywords that they should be associated with. For example, a query may ask for "cities near Eiffel tower". The occurrence of an entity mention near the given keywords provides support for the relevance of that entity. The support for an entity is aggregated across multiple occurrences of mentions of that entity near the given keywords. Chakrabarti et al. [5] also describe a query language which allows more complex queries to be created, allowing for example entities that occur near entities retrieved by a subquery. The implementation in [5] worked on a Web scale corpus, but was limited to a small number of entity types; that limitation was subsequently removed [6].

EntityRank [8] has a similar goal, and also works on Web scale data, but allows recognition of multiple entities cooccurring with given keywords. Specifically, it allows the query to specify multiple target entity types, such as #professor, #university, along with keywords such as "database". All entities and keywords should co-occur near each other in the same document. Entity Search [7] has goals similar to that of [5], but focuses on efficient evaluation of queries by creating appropriate indices.

Users are however often interested in relationships between entities, where the keywords that select entities may occur separately from keywords that specify the desired relationships; we give an example shortly. If the relationships have been extracted already, it is possible to represent the information using a graph model such as RDF, and then a query language such as NAGA [12] can be used to execute such queries on the graph. The NAGA query language allows complex connections to be specified, and allows aggregation of evidence from multiple parts of the graph. However, a problem with this approach is that relationships have to be extracted ahead of time, and at Web scale the number of potential relationships is enormous. The YAGO dataset [18] used in the NAGA system only extracted a few tens of relationships. Other related work which considers integration of structured information and textual data includes the ESTER system [3] and Pound et al. [17]. Both these systems focus on relationships that have already been extracted, using the YAGO dataset, and thus support only a limited number of relationships. However, both these system allow queries to combine some form of textual search with the queries on structured data.

The ERQ system [14, 15] presents an alternative approach where the corpus is stored uninterpreted except for identification of entities. Relationships are specified by keywords, and found by keyword search on the corpus, in effect performing a simplified on-the-fly extraction of relationships.

ERQ uses three position-based features for ranking answers tuples. The first is *proximity* which emphasizes the fact that if the entities and keywords are close to each other in an evidence, then it is more likely to form a valid evidence. The second feature is the *ordering pattern* of entities and phrases in an evidence. The ordering patterns which appear more often are better indicators of valid evidences. The third feature is the *mutual exclusion* rule which dictates that when evidences of different entities cooccur in the same sentence, at most one colliding pattern is effective. Our scoring model takes proximity into account, but does not currently implement ordering patterns and mutual exclusion.

Although the ERQ system does not limit the number of relationships, the evaluation algorithm used in ERQ requires separate indices per entity type, and the implementation of [15] indexed only 10 selected entity types. Thus the number of queries that can be expressed is limited. In contrast, in our system, we handle all possible categories specified in Wikipedia/YAGO. To our knowledge, all the earlier systems require the answer types to be precisely specified in the query. In the real world, such specification is not easy, since users are not aware of what types are available. Our system allows type specification to be done based on keywords that match types, and all matching types are answer candidates; a match score for each answer type is taken into account along with entity scores, to get the overall answer ranking.

We use Lucene as a document-centric indexing system, and exploit our extended graph model to efficiently find entity mentions in proximity to keyword occurrences. The in-memory graph also provides a mapping between entities and their categories.

7. EXPERIMENTAL EVALUATION

In this section, we present a detailed analysis of the effectiveness our approach for solving near queries and entity-relationship queries. We look at the contribution of different factors involved in the approach. We also show a comparison of the quality of our results with those generated by the ERQ system of Li et al. [15].

7.1 Experimental Setup

We have implemented our algorithms in Java using servlets; we call our system WikiBANKS. Our system is available for access over the Web at the URL www.cse.iitb.ac.in/banks. The machine we used has 12 GB of RAM and an Intel Xeon

E5504, 2 GHz processor, with 1 TB Hard disk with SATA interface, running Ubuntu 10.04 LTS with a Linux 2.6.32-34 kernel. The database system used is PostgreSQL 8.3.7.

The Wikipedia graph was created out of a Wikipedia dump as of January 2009, and has following characteristics: number of nodes: 16.28 million, number of edges: 179.5 million, average indegree: 5.4303, maximum indegree: 374882. The graph takes about 4 GB space and takes about 85 seconds to load.

We index Wikipedia data using Lucene. For each document in Wikipedia, a virtual document is created, containing three fields: (a) Nodeid: the Wikipedia article ID, (b) Title: the title of the article, and (c) Content: the textual content of the article. The title and content field are indexed using Lucene, while Nodeid is stored but not indexed. The index stores term offsets for each occurrence of a term in each document that it occurs in. The index building is done after assigning prestige scores to the nodes of the graph. These node prestige scores are also included in indexing to boost the hits of the relevant nodes. We use Lucene *Collectors* to collect the Lucene scores for documents and *SpanQuery* to get the offsets of the search terms.

7.2 Query Set

Unlike the ERQ system of Li et al. [15], which supports only a limited number of categories, our system supports all the Yago categories, numbering nearly 150,000. Thus our system can answer a vastly larger number of queries than ERQ. However, to compare the two systems, we chose a set of 27 queries from the "Own28" set of Li et al. [15], available online at http://idir.uta.edu/erq/, as a performance benchmark. The query set includes:

- Q1 - Q16 : *Single selection predicate queries*, i.e. Near queries with only one selection predicate.
- Q17 - Q21 : *Multiple selection predicate queries*, i.e. Near queries with multiple selection predicates on the same entity variable.
- Q22 - Q27 : *Entity-relationship queries*, also known as multi-predicate queries with join.

For each query, we have a manually collected a set of correct answers, which we believe is fairly complete. We consider these sets as the ground truth when evaluating the performance. While [15] has only a limited number of entity categories available for use in queries, when we expressed the queries in our system for a few of the queries we made use of the richer set of categories available to us; the specific set of changes were: actor instead of person, football player instead of player, and football club instead of club, in queries where such substitutions are appropriate.

7.3 Parameter Settings

There are a number of input parameters involved in our query processing and scoring model. We have executed a large number of queries with different parameter settings, and manually chose the optimum values for these parameters, i.e. the values that gave the best precision. For selection predicates, we set the token span $\lambda = 12$, and the value $\sigma = 6$ for proximity scoring using the Gaussian kernel. For near queries with a single selection predicate we use the weightage for activation $\eta = 0.1$, while for near queries with multiple selection predicates we use $\eta = 0.6$. For entity-relationship queries, we set $\eta = 0.8$ for selection predicates; for relation predicates we set $\lambda = 16$, $\sigma = 8$, and the multiplicative weighting factor for relations predicates $\gamma = 0.6$.

7.4 Measures of Performance

We have used the following precision measures to compare the performance:

- **Precision at k :** Also referred to as $P@k$, it is the precision at a given cut-off rank. It is calculated as:

$$P@k = \frac{|relDocs \bigcap topKDocs|}{k}$$

where *relDocs* is the set of all relevant documents (here, entities) and topKDocs is the set containing the top-K documents (here, entities) that are retrieved. Our precision at K graphs stop at $K = 10$.

- **Recall :** Recall is the fraction of the documents that are relevant to the query that are successfully retrieved:

$$recall = \frac{|relDocs \bigcap retrievedDocs|}{|relDocs|}$$

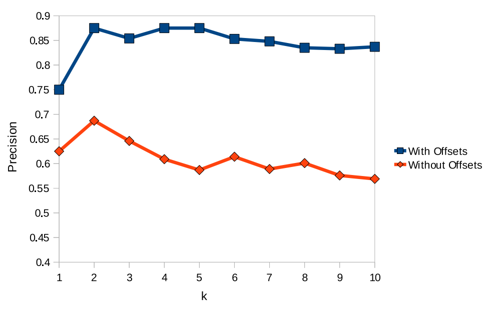
where *retrievedDocs* is the set of all documents (here, entities) that are retrieved. To compare precision and recall, we have plotted precision at specific values of recall. To calculate this, we find the precision at the point when we have retrieved just enough answers to achieve a particular recall value (i.e. particular fraction of the set of all correct answers). When the system is unable to retrieve enough answers to achieve a particular recall value, we define the precision at that recall value as zero.

7.5 Experimental Results

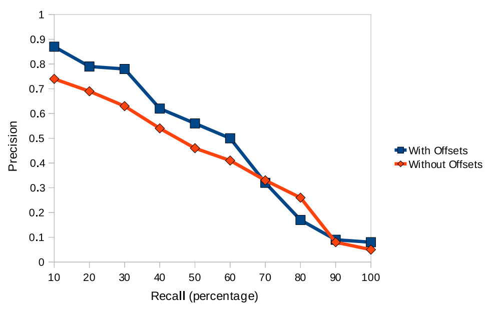
Figure 1 compares the performance of the basic system (without the optimizations described in Section 5) with and without using offsets information. The comparison is for near queries Q1 through Q16. For the case where offsets are not used, we explore all nodes linked from the nodes containing near keywords, without regards to the token distance between the near keyword and the links. This causes a large number of irrelevant nodes to be explored, and increases query execution time as well as memory utilization. Figure 1 (a) shows that the average precision at k is much lower without using offset information, for k up to 10.

However, Figure 1 (b) indicates that the average precision is lower for "with offsets" than "without offsets" at 80% recall. This is because, in case of "without offsets", a large number of nodes are explored and hence it generates higher fraction of correct answers. The "with offsets" version spreads activation only to nodes that are within a limited offset (span), which is set to 12 by default. As a result this version explores fewer nodes, and fails to generate several answers which the "without offsets" technique is able to generate; as per our definitions, the precision is 0 at this point for these queries, reducing the average precision significantly. Since users are likely to only view the top-k results for some small value of k, the version with offsets is definitely preferable.

Next, we compare the performance of our system with ERQ [15]. We have experimented with 5 different versions of our system to isolate the effect of various optimizations described in Section 5. The different versions are as follows:



(a) Precision at k



(b) Precision vs. recall

Figure 1: Effect of offsets on near queries with single selection predicate (no optimization)

k	Basic	Near Titles	Infobox	Near Categories	All 3	ERQ
1	0.704	0.666	0.814	0.851	0.851	0.741
2	0.741	0.777	0.759	0.833	0.814	0.833
3	0.703	0.728	0.753	0.79	0.814	0.796
4	0.731	0.75	0.741	0.796	0.833	0.75
5	0.733	0.748	0.733	0.807	0.822	0.76
6	0.703	0.715	0.703	0.802	0.814	0.716
7	0.693	0.714	0.692	0.793	0.804	0.72
8	0.675	0.694	0.689	0.777	0.81	0.734
9	0.678	0.691	0.695	0.765	0.802	0.71
10	0.681	0.685	0.696	0.751	0.785	0.698

Table 1: Precision at k for All Queries

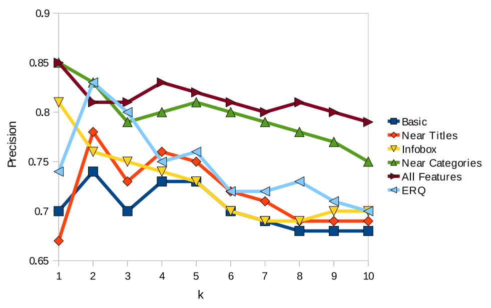


Figure 2: Precision at k for all queries

- **Basic:** In this version, we use the basic model without any of the optimizations.

- **Near Titles:** In this version, along with the basic features, we also spread activation from articles whose titles contain the near keywords to all its out-neighbors.

- **Infobox:** In this version, we use the infobox information and add some initial activation to the node whose infobox contains the near keywords.

- **Near Categories:** In this version, we exploit the Wikipedia category specificity as explained earlier.

- **All Features:** This version uses all the above optimizations along with the basic version.

The precision and precision versus recall numbers for ERQ are obtained by running the queries on their system, available online at http://idir.uta.edu/erq/.

Table 1 gives the precision at k values for our complete set of test queries. The table data clearly shows that each of the additional features improves the precision. Specially, the NearCategories feature improves the performance by a large margin. Using all the features together gives us the best performance.

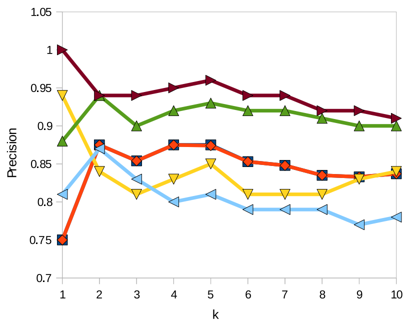
Figure 2 shows the plot for the precision at k values across all queries, with different system features turned on. Figure 3 shows the same information, but separately for different types of queries. The graphs indicates that our system clearly outperforms ERQ for near queries, with single selection predicate as well as with multiple selection predicates.

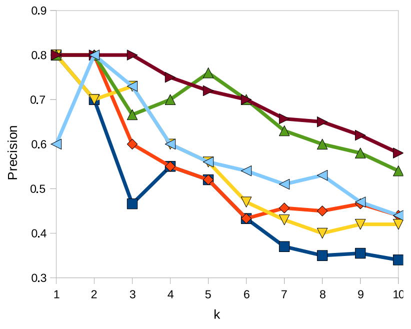
For entity-relationship queries, the ERQ system provided better precision. One reason for the lower precision is that our system allows flexible specification of categories. On Q28 from the OWN28 set (not included in the performance results) most of the films returned were in fact academy award winning movies adapted from novels, but in place of novels, the query returned other movies. This is because these movies are in categories such as “movies adapted from novels”, and since such a category is treated as a valid category for the category keyword “novel”, the query treats the movie itself as a novel. Requiring exact match for categories improved the result quality drastically, with 9 out the top 10 answers being correct. Our scoring system needs to be improved to avoid problems due to non-exact category matches.

However, the ERQ system requires categories to be precisely specified, which is not an easy task for a casual user, whereas we can handle queries where the categories are not precisely specified. (In addition our implementation can handle a very large number of categories in contrast to the limited number of categories handled by the current ERQ implementation.)

We also found anecdotally that the mutual exclusion and ordering pattern heuristics used in [15] would have been useful in improving precision, had we implemented them. Implementing these heuristics is an area of future work.

Table 2 shows average query execution time for various types of queries. Execution time is the response time measured when the query is input to the servlet. Table 3 shows average memory utilization for the queries in terms of num-


(a) near queries with single selection


(b) near queries with multiple selections

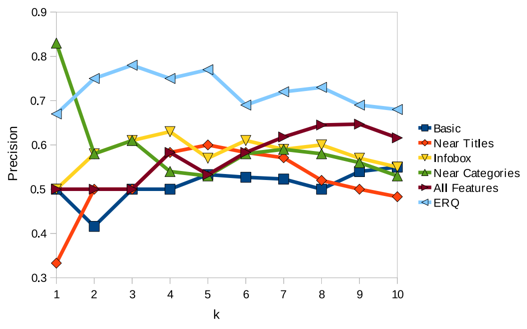

(c) entity-relationship queries

Figure 3: Precision at k by query type

Query Set	COLD CACHE	WARM CACHE
Single Selection	4.546	1.694
Multiple Selection	12.112	5.837
Entity-Relationship	14.44	9.317
All	8.233	4.284

Table 2: Average Query Execution Time (in seconds)

Query Set	Nodes Explored	Size of Target Queue
Single selection	7474	210
Multiple selection	48132	4074
Entity relationship	84003	9635
All	32010	3020

Table 3: Average Memory Utilization

ber of nodes explored during activation spreading and size of the target queue. Target queue size determines the collection of nodes which are processed during ranking to produce final set of answers i.e. it is set of probable answers before ranking. Table 4 gives the average recall i.e. average of fraction of correct answers reported by the system for various types of queries.

Figure 4 shows the plots of precision versus recall across all queries, while Figure 5 shows the same information for each query type. Since some queries do not have 100% recall up to the number of answers retrieved, we show the precision as 0 at recall percentages that are higher. Figure 4 show that WikiBANKS outperforms ERQ overall, while Figure 5 shows that WikiBANKS outperforms ERQ in case of near queries with single and multiple selection predicates, but ERQ achieves better performance for relationship queries.

We also tested our system on a number of other queries, many of which we could not run on the ERQ system since they used types such as medicines, airports, languages, animals, currencies, and so on which are among the many types not currently supported in the ERQ implementation. The precision at k and execution time for these queries were similar to the results we saw earlier for queries from the ERQ system. We omit details for lack of space.

Query Set	Basic	Near Titles	Info-box	Near Cate-gories	All 3	ERQ
Single selection	0.639	0.662	0.645	0.77	0.788	0.635
Multiple selection	0.448	0.488	0.448	0.565	0.598	0.414
Entity-relation-ship	0.511	0.537	0.500	0.511	0.533	0.672
All	0.575	0.602	0.577	0.674	0.696	0.602

Table 4: Average Recall

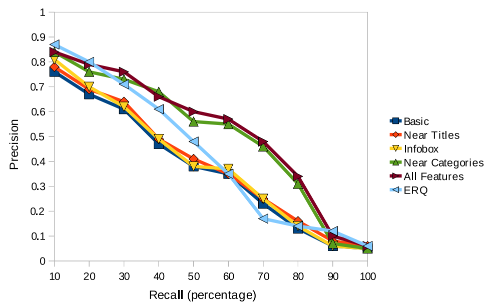
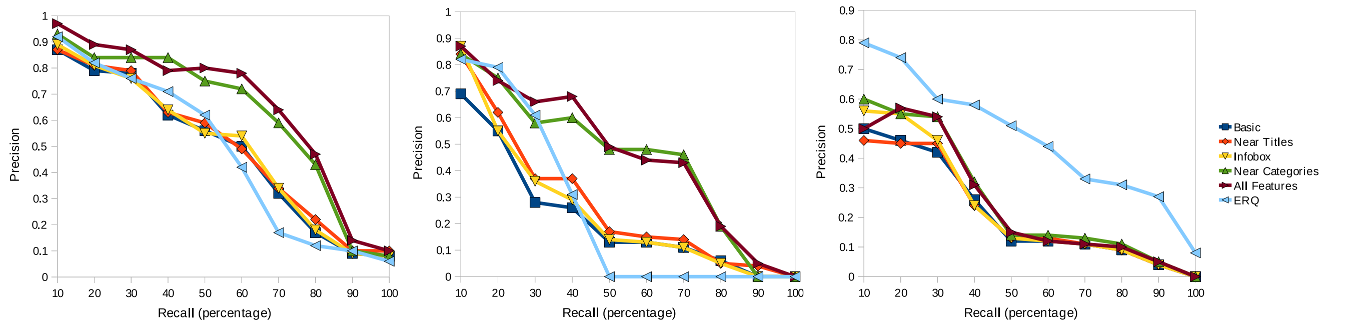


Figure 4: Precision vs. Recall for all queries

We also ran some queries to test the impact of spreading activation through the graph, a feature we support, but other entity ranking techniques do not support. However, we did not find much difference since in most cases the Wikipedia corpus has direct links to the desired entities from pages containing the near keywords, and adding indirect activation did not help. For example, for the query "University near Nobel prize", the infoboxes of Nobel prize winners pages mentioned the Nobel prize and had a link to their institutions. We expect these results will be different if we include Web pages instead of just Wikipedia pages, an area of ongoing work.

8. CONCLUSIONS AND FUTURE WORK

We have proposed a novel extended graph representation,



(a) near queries with single selection (b) near queries with multiple selections (c) entity-relationship queries

Figure 5: Precision vs. recall by query type

and showed how to exploit it to answer entity ranking (near) queries and entity-relationship queries on the Wikipedia corpus. Unlike earlier systems we allow type specification through keywords, and develop novel scoring mechanisms based on spreading activation. Our performance study shows good result quality, beating earlier work on entity queries. Improving performance on entity-relationship queries is an area of current work.

The Wikipedia corpus has a limited number of explicit entity mentions, limiting the effect of spreading activation from keywords to nearby entities. We are currently extending our system to work on the annotated Web corpus of [6], which would provide a much richer set of keyword-entity associations. With such a system, we cannot keep the entire Web graph in memory; however, the number of entities is still relatively small (since we use Wikipedia as the source for entities), and we can keep these entities, along with their category hierarchy, in memory. We have developed versions of our algorithms tailored for such an environment, with data partitioned across multiple machines. Initial results demonstrate the feasibility of such a system, both in terms of answer quality, and interactive response times. We are currently working on improving the answer quality of the system. Extending our implementation to add the mutual exclusion and ordering pattern heuristics of [15] is another direction for future work.

Acknowledgment: This work was partially supported by IMPECS (Indo-German Max Planck Center for Computer Science).

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: authority-based keyword search in databases. In *VLDB*, 2004.

[3] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[5] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, pages 717–726, 2006.

[6] S. Chakrabarti, D. Sane, and G. Ramakrishnan. Web-scale entity-relation search architecture. In *WWW (Companion Volume)*, pages 21–22, 2011.

[7] T. Cheng and K. C.-C. Chang. Beyond pages: Supporting efficient, scalable entity search with dual-inversion index. In *SIGMOD*, 2010.

[8] T. Cheng, X. Yan, and K. C.-C. Chang. EntityRank: Searching entities directly and holistically. In *VLDB*, 2007.

[9] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[10] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, 2002.

[11] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[12] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[13] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti. Collective annotation of wikipedia entities in web text. In *KDD*, pages 457–466, 2009.

[14] X. Li, C. Li, and C. Yu. Entityengine: answering Entity-Relationship queries using shallow semantics. In *CIKM*, pages 1925–1926, 2010.

[15] X. Li, C. Li, and C. Yu. Entity-relationship queries over wikipedia. *ACM Trans. on Intelligent Systems and Technology*, 3(4), Sept. 2012.

[16] Y. Lv and C. Zhai. Positional language models for information retrieval. In *SIGIR*, pages 299–306, 2009.

[17] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD Conf.*, pages 423–434, 2010.

[18] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago - a core of semantic knowledge. In *WWW*, 2007.

[19] M. A. Yosef, J. Hoffart, I. Bordino, M. Spaniol, and G. Weikum. AIDA: An online tool for accurate disambiguation of named entities in text and tables. *PVLDB*, 4(12):1450–1453, 2011.