Efficiently Linking Text Documents With Relevant Structured Information

Venkatesan T. Chakaravarthy Himanshu Gupta Prasan Roy Mukesh Mohania
IBM India Research Lab
New Delhi, India
{vechakra, higupta3, prasanr, mkmukesh}@in.ibm.com

ABSTRACT

Faced with growing knowledge management needs, enterprises are increasingly realizing the importance of interlinking critical business information distributed across structured and unstructured data sources. We present a novel system, called EROCS, for linking a given text document with relevant structured data. EROCS views the structured data as a predefined set of "entities" and identifies the entities that best match the given document. EROCS also embeds the identified entities in the document, effectively creating links between the structured data and segments within the document. Unlike prior approaches, EROCS identifies such links even when the relevant entity is not explicitly mentioned in the document. EROCS uses an efficient algorithm that performs this task keeping the amount of information retrieved from the database at a minimum. Our evaluation shows that EROCS achieves high accuracy with reasonable overheads.

1. INTRODUCTION

Faced with growing knowledge management needs, enterprises are increasingly realizing the importance of interlinking critical business information distributed across structured and unstructured data sources. In this paper, we address the problem of linking a document with related structured data in an external relational database. We introduce a novel system, called EROCS (Entity RecOgnition in Context of Structured data), that views the structured data in the relational database as a set of predefined "entities" and identifies the entities from this set that best match the given document. EROCS further finds embeddings of the identified entities in the document; these embeddings are essentially linkages that interrelate relevant structured data with segments within the given document. Unlike prior entity recognition approaches [20, 7, 12], EROCS identifies such linkages even when the relevant entity is not explicitly mentioned in the document.

As an example, consider a retail organization where the structured data consists of all information about sales transactions, customers and products. The organization, with a network of multiple stores, has a steady inflow of complaints into a centralized complaint repository; these complaints are accepted using alternative means, such as a web-form, email, fax and voice-mail (which is then transcripted). Each such complaint is typically a free-flow narrative text about one or more sales transactions, and is not guaranteed to contain the respective transaction identifiers; instead, it might divulge, by way of context, limited information such as the the store name, a partial list of items bought, the purchase dates, etc. Using this limited information, EROCS discovers the potential matches with the transactions present in the sales transactions database and links the given complaint with the matching transactions.

Such linkage provides *actionable* context to a typically fuzzy, free flow narrative which can be profitably exploited in a variety of ways.

- In the above example, we can build an automated complaint routing system. Given the transaction automatically linked with the complaint, this system retrieves from the relational database additional information about the transaction (such as type and value of the items purchased, specific promotions availed and the customer's loyalty level), and routes the complaint to an appropriate department or customer service representative based on the same.
- Consider a collection of complaints that have been linked to the respective transactions in the relational database. This association can be exploited in OLAP analytics to derive useful information such as regions or product categories that have shown a recent upsurge in complaints.

In addition to the database-centric uses mentioned above, the additional information provided by these linkages can be effectively exploited in entity-based search [8], question answering, document understanding and a host of other related problems in information retrieval [3] and extraction [20].

Overview. EROCS takes as input (a) the given document, suitably filtered to retain only the relevant terms, and (b) the given database, viewed as a set of predefined entities and associated context information. These entities are defined in terms of a collection of *entity templates* that specify the location of each entity and its context information in the relational database. In the retail organization example, the each sales transaction is an entity and the customer, store

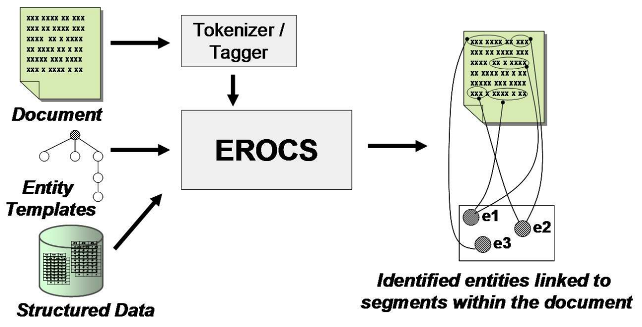

Figure 1: EROCS Overview

and product information associated with a given transactions forms its context. Given this input, as illustrated in Figure 1, EROCS matches the context information of the candidate entities with the document and finds the best matching entities and their embeddings.

This task performed by EROCS is similar in spirit to dictionary-based named-entity recognition [7, 1, 12, 17], but differs in the following crucial and challenging aspects.

- EROCS identifies an entity even if it is not explicitly mentioned in the document; it exploits the available context information to match and identify the entities. As a consequence, while entity matching in named-entity extraction essentially involves a dictionary lookup, entity matching in EROCS involves a significantly more complex search over a multi-table database. Notice that given the large number of candidate entities (in the retail example, the number of sales transactions could easily be in hundreds of thousands) and the size of context information associated with each entity, it is impractical to apriori materialize and index the entire context of each candidate entity; this materialization would involve expensive joins across multiple tables, and would have high maintenance and storage overheads.

- In named-entity recognition, the terms matching an entity are assumed to appear as contiguous phrases in the document and follow regular patterns. In EROCS, this assumption does not hold and the terms matching an entity could be arbitrarily spread out in the document. The additional challenge, in course of identifying the best matching entities, is thus to interrelate multiple terms across the document as belonging to the same entity. Note that the number of entities present in the document, or their relative order is not known apriori.

Contributions. Our main contributions in this paper are as follows.

- We propose an efficient algorithm that, while keeping the amount of information retrieved from the database at a minimum, finds the best matching entities as well as their optimal embedding in the document. The proposed algorithm successively queries the database to incrementally build the contexts of only the potentially matching entities as it converges to the final solution.

- We present an experimental study that clearly illustrates the practicability of the proposed approach and its superiority over less sophisticated alternatives, in terms of both the accuracy of the result as well as the execution and space overheads.

Organization. The remainder of the paper is organized as follows. We begin with a discussion of related work in Section 2. A description of framework appears in Section 3. This is followed by the details of the proposed algorithm in Section 4. In Section 5, the context cache is introduced. In Section 6, algorithms that effectively exploit the context cache to resolve the database performance bottleneck are presented. A preliminary experimental study is presented in Section 7. Finally, Section 8 presents the conclusions.

2. RELATED WORK

Matching entities across structured data and unstructured text documents falls under the general area of semantic integration, which deals with the problem of identifying common concepts across heterogeneous information sources. The mainstay of this area has traditionally been integration of data across heterogeneous structured databases [13]. Recent work has also addressed the semantic integration within and across text documents [16]. However, we are not aware of any previous work that directly addresses semantic integration across structured data and unstructured documents, the focus of this paper.

Record matching is the task of matching records and identifying whether they correspond to the same real-world entity [23]. Record matching techniques are primarily relevant to structured records [10], but can be extended to semi-structured records (such as paper citations and street addresses) by including text-segmentation as a pre-processing step [1, 17]. EROCS, in contrast, matches structured records with informal, unstructured text (such as an email) which is not amenable to segmentation using these methods because it could contain arbitrarily intermixed entity references.

Semantic integration in text [16] is the task of identifying whether different entities identified using NER within or across documents correspond to the the same real-world entity; this identification is done by matching multiple features extracted using natural language rules. Note that the real-world entity is not available for reference, which is an inherent limitation. In EROCS, the focus is to match mentions of an entity in the document with a well-defined structured representation of the entity, which acts as a reference.

Recently, there has been significant work on keyword search in relational databases [2, 6, 14]. Given a set of terms as input, the task is to retrieve the candidate join-trees (group of rows across multiple tables in the database interlinked through foreign-key relationships) that collectively contain these terms; these candidate join-trees are also ranked against each other based on relevance. The entity identification task of EROCS is quite different; it interprets the input document as a sequence of terms, and not only identifies the best matching entities, but also their embeddings in the document. Note that entities embedded in disjoint portions of the document do not compete against each other. Moreover, in keyword search, the number of input terms are few (typically, less than 5) and hence, for each given term, an algorithm can afford to fetch all rows containing the term. Whereas, in EROCS, the input document may contain hun-

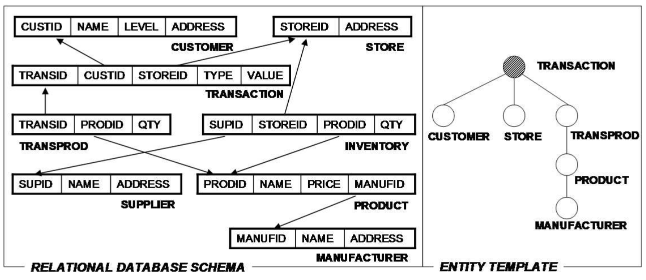

Figure 2: Example of a Relational Database Schema and an associated Entity Template

dreds of terms and so, we need to restrict the above database lookup to a minimum.

The entity matching task in EROCS is similar to dictionary-based named-entity recognition (NER) [7, 1, 12, 17, 9], where the focus is to identify sequences of terms within the documents as named-entities such as person name, company name, location, etc. by exploiting explicit lists (dictionaries) of single and multi-word terms and patterns. Unlike NER, EROCS identifies entities even when they are not explicitly mentioned in the document. This makes entity matching in EROCS significantly more complex than the dictionary lookup in NER, as discussed earlier in Section 1. However, EROCS can exploit NER to preprocess the input document to increase the efficiency of the entity matching task; the details appear in Section 3.2.

3. FRAMEWORK

In this section, we give details of the models that build up the underlying framework of EROCS.

3.1 Entity Model

An *entity* is a “thing” of significance, either real or conceptual, about which the relational database holds information [11]. An *entity template* specifies (a) the entities to be matched in the document and (b) for each entity, the context information that can be exploited to perform the match.

Formally, an entity template is a rooted tree with a designated root node. Each node in this tree is labeled with a table in the given relational database schema, and there exists an edge in the tree only if the tables labeling the nodes at the two ends of the edge have a foreign-key relationship in the database schema. The table that labels the root node is called the *pivot table* of the entity template, and the tables that label the other nodes are called the *context tables.* Each row e in the pivot table is identified as an entity belonging to the template, with the associated context information consisting of the rows in the context tables that have a path to row e in the pivot table through one or more foreign-keys covered by the edges in the entity template.

For instance, the sales transactions entity template, shown in Figure 2, has the root node labeled by the `TRANSACTION` table (the pivot table), and the non-root nodes labeled by the `CUSTOMER`, `STORE`, `TRANSPROD`, `PRODUCT` and `MANUFACTURER` tables (the context tables) that provide the context for each transaction in the `TRANSACTION` table. Note that the template definition also provides the information that the `SUPPLIER` table, though reachable from the `TRANSACTION` table via both the `PRODUCT` and `STORE` tables, carries no contextual information about a given transaction; this is valuable domain knowledge that is hard to figure automatically.

Multiple nodes in the template can be labeled with the same table. This is needed to differentiate the different roles a table might play in the context of the entity. Suppose the document mentions product names not only to identify a transaction, but also to identify the store in which the transaction occurred; further suppose it mentions the manufacturer in the former case, but not in the latter. Then, the template in Figure 2 would extend the `TRANSACTION`→`STORE` path to `TRANSACTION`→`STORE`→`INVENTORY`→`PRODUCT`. Now there exist two nodes in the template labeled with the same table `PRODUCT` representing the two roles the table plays; also, one includes a child labeled with the table `MANUFACTURER`, the other does not.

Currently, we assume that the entity templates are specified by a domain expert; this is a one-time low-overhead activity that can be a part of the initial customization and configuration. For instance, reverse engineering of existing relational databases to derive the entity-relationship model is a much studied problem [18], and several commercial data modeling tools support this feature (e.g. Microsoft Office Visio [22]); specification of entity templates can be readily integrated with this reverse engineering effort.

The entity templates are closely related to *view objects* defined earlier by Barsalou et al. [5], who also propose information theoretic techniques to automatically identify what tables to include in the view object for a given pivot table [4]. We plan to explore similar techniques to automate the specification of entity-templates as a future work.

Further discussion in this paper assumes only a single entity template is defined. This is only for ease of exposition; the techniques can be readily generalized for a collection of entity templates.

3.2 Document Model

EROCS views a document as a sequence of sentences, where a sentence is a bag of terms. Some terms in a sentence are potentially useful since they occur in the database as well, and thus may occur in the context of a candidate entity; other terms are not useful and are filtered out. EROCS uses a part-of-speech parser to identify *noun-phrases* in a sentence, and filters out the rest; the assumption, which usually holds, is that only nouns appear as values in the database. Further, each noun thus identified is looked up in the database and annotated with the database columns it occurs in.

This filtering and annotation pre-processing reduces the amount of work to be performed in the matching step. This could be further enhanced by (a) incorporating NER techniques [1] that can identify potential matches with the database terms without the database lookups needed currently, and (b) incorporating semantic integration in text [16] to matches the terms in the document to identify whether they belong to the same entity; this dependency information can potentially reduce the number of database queries needed in the current implementation.

3.3 Entity-Document Matching Model

EROCS defines the weight of a term t as:

$$w(t) = \begin{cases} \log(N/n(t)) & \text{if } n(t) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where N is the total number of distinct entities in the relational database, and $n(t)$ is the number of distinct entities that contain t in their context.

A *segment* is a sequence of one or more consecutive sentences in the document. We now discuss how a segment d is scored with respect to an entity e.

Let $T(d)$ denote the set of terms that appear in the segment d, and let $T(e,d) \subseteq T(d)$ denote the set of such terms that appear in the context of e as well. Then, the score of the entity e with respect to the segment d is defined as:

$$score(e,d) = \sum_{t \in T(e,d)} tf(t,d).w(t)$$

where $tf(t,d)$ is the number of times the term t appears in the segment d, and $w(t)$ is the weight of the term t as defined above. This definition of $score(e,d)$ is in the spirit of the "tf-idf" scores commonly used in the Information Retrieval literature [3].

4. IDENTIFYING BEST MATCHING ENTITIES AND THEIR EMBEDDINGS

In this section, we first formulate the entity identification and embedding problem and then provide an efficient algorithm for solving the same.

4.1 Problem Formulation

We are given as input (a) a document D, (b) a relational database, and (c) an entity template that interprets the database as a set of entities E.

In this paper, we make the assumption that each sentence in the document D relates to at most one entity; this is a stylistic assumption about the document, and seems to be reasonable in practice. Given this assumption, we can formally model an annotation for D as follows: An *annotation* for the document D is a pair (S,F) where S is a set of non-overlapping segments of D and F is a mapping that maps each segment in $d \in S$ to an entity $F(d) \in E$.

EROCS defines the score of an annotation (S,F) as:

$$score(S,F) = \sum_{d \in S}(score(F(d),d) - \lambda)$$

where the entity-document matching score is as defined in Section 3.3 and $\lambda \geq 0$ is a tunable parameter; a justification for this parameter will appear in a moment. The problem being addressed can now be stated as follows.

Problem Statement. *Find an annotation with the maximum score among all annotations of the document D.*

Before moving on to discuss the solution of this problem in the next section, let us reflect upon the utility of the parameter λ in the above formulation. For a given annotation (S,F), we can think of $score(F(d),d)$ as the support for a given $d \in S$. Introducing λ in the annotation scoring function guarantees that in the annotation with the maximum score, no segment will have a support less than λ. In Section 7, we show that setting $\lambda > 0$ eliminates a significant amount of irrelevant annotations, leading to improved accuracy.

The naive algorithm to solve the proposed problem is to enumerate all annotations, and pick the annotation that has the maximum score. This is clearly impractical since the number of possible annotations is $|E|^{|D|}$, where $|E|$ is the number of entities and $|D|$ is the number of sentences in the document. EROCS solves this problem by first effectively pruning the search space, and then searching on the reduced space using an efficient algorithm; the next two sections give the details of the solution.

4.2 Pruning the Search Space

In this section, we present some properties of the best annotation of a given document D. These properties will help in effectively pruning the size of the search space, which will be useful in developing an efficient algorithm for finding the best annotation in the next section.

We call an annotation (S,F) *canonical* iff (a) S is a partition[1] of D, and (b) F maps each $d \in S$ to its best matching entity. Now, consider an annotation (S,F) that is not canonical. If S does not cover some sentences of D, then we can transform the annotation by adding these non-covered sentences to adjacent segments in S. Further, if S contains a segment d such that the associated entity $F(d)$ is not its best matching entity, then we can transform the annotation by taking $F(d)$ as d's best matching entity instead. Notice that neither of these transformations can possibly decrease the annotation's score. Thus, we see that any annotation can be converted into a canonical annotation without decreasing its score. It follows that there exists a canonical annotation that achieves the maximum score. We formalize the discussion in the following claim.

CLAIM 1. *For any document D, there exists a canonical annotation (S,F) such that (S,F) is an optimal annotation for D.*

We can thus restrict the search space to only canonical annotations without any loss in generality. The problem being addressed can now be revised as follows.

Revised Problem Statement. *Find a canonical annotation with the maximum score among all canonical annotations of the document D.*

In the next section, we present an efficient algorithm to solve this problem.

4.3 Best Annotation Computation

In this section, we present an efficient dynamic programming algorithm to find the best canonical annotation for a given document. As argued in the previous section, the restriction to canonical annotations does not result in any loss of generality, and the best canonical annotation computed is guaranteed to be the best annotation overall. Accordingly, we consider only canonical annotations in the rest of this paper. Moreover, since a canonical annotation (S,F) is completely specified by its segment set S, we shall occasionally identify a canonical annotation (S,F) by S alone, and refer to F as the *canonical mapping* for S.

For $1 \leq i \leq j \leq |D|$, let $D_{i,j}$ denote the segment in D that starts at the i^{th} sentence and ends at the j^{th} sentence (both inclusive). The segments $D_{1,1}, D_{1,2}, \ldots, D_{1,|D|}$, where $D_{1,|D|}$ is the document D itself, are termed the *prefixes* of the document D.

Now, let S_k be the best annotation for the prefix $D_{1,k}$ and let r_k be its score. Further, let $e_{i,j}$ be the best matching entity for the segment $D_{i,j}$ and let $s_{i,j}$ be its score. The following claim gives a recurrence relation for r_k in terms

[1]A set of non-overlapping segments that cover each sentence in the document

```
Procedure BestAnnot($D$)
Input: Document $D$
Output: (best annotation, score)
Begin
A01 For $i$=1 to $|D|$
A02     For $j$=$i$ to $|D|$
A03         Let $e_{i,j} = \arg\max_{e \in E} score(e, D_{i,j})$
A04         Let $s_{i,j} = score(e_{i,j}, D_{i,j})$
A05 Let $S_0 = \phi$
A06 Let $r_0 = 0$
A07 For $k = 1$ to $|D|$
A08     Let $j = \arg\max_{0 \leq j \leq k-1}(r_j + s_{j+1,k} - \lambda)$
A09     Let $S_k = S_j \cup \{D_{j+1,k}\}$
A10     Let $r_k = r_j + s_{j+1,k} - \lambda$
A11 For each $d \in S_{|D|}$
A12     Let $F_{|D|}(d) = \arg\max_{e \in E} score(e, d)$
A13 Return $((S_{|D|}, F_{|D|}), r_{|D|})$
End
```

Figure 3: Best Annotation Computation Algorithm

of r_j, for $j \leq k-1$; as the base case for the recurrence, we define r_0 to be zero.

Claim 2. *For each $1 \leq k \leq |D|$, the score r_k can be recursively expressed as $r_k = \max_{0 \leq j \leq k-1}(r_j + s_{j+1,k} - \lambda)$.*

Proof. The claim is easily proved via induction on k. We make use of the fact that (a) r_k is the maximum annotation score possible for $D_{1,k}$, and (b) for any $1 \leq k \leq |D|$, there must exist a $1 \leq j \leq k$ such that $D_{j,k}$ appears in the best annotation of the prefix $D_{1,k}$. $\square$

This recurrence relation forms the basis of the dynamic programming algorithm presented in Figure 3.

The algorithm first computes the best matching entities for all the segments in the given document (Lines A01-A04). Then, in accordance with the recurrence relation of Claim 2, it iteratively computes the best annotation for increasingly larger prefixes of the document, making use of these best matching entities and the best annotations of strictly smaller prefixes computed in previous iterations (Lines A05-A10). The algorithm then constructs the canonical mapping $F_{|D|}$ for the computed best annotation $S_{|D|}$ (Lines A11-A12) and returns the pair along with its score $r_{|D|}$ (Line A13).

Discussion. The time complexity of the proposed algorithm is quadratic in the number of sentences in the document; this can be reduced to linear if we limit the size of the segments considered to be at most L sentences. However, this efficient algorithm is not enough to make the solution scalable. Finding the entity in E that best matches a given segment (Line A03) involves a search (rather than a simple lookup) on the database; this is an expensive operation for nontrivial database sizes, and performing it for every segment in the document is clearly a performance bottleneck. In the next section, we describe how EROCS resolves this critical scalability issue.

5. THE CONTEXT CACHE

As discussed in the previous section, the operation of finding the entity with the best score for a given document segment is an expensive operation. This operation needs to be performed for every segment in the document; if naively done, this is likely to be a severe performance bottleneck. Since a document is not likely to have repeated segments, caching the result of the operation is not effective. Moreover, the large number of candidate entities, and the size of context information associated with each entity, makes it impractical to apriori materialize and index the entire context of each candidate entity; this materialization would involve expensive joins across multiple tables, and thus would have high computation, maintenance and storage overheads. In this section and the next, we show how EROCS resolves this critical issue.

EROCS uses an entity-term association cache, formally referred to as the *context cache*, to reduce the database access overhead. This cache can be visualized as a collection of relationships of the form (e, t) meaning that the term t is contained in the context of the entity e. This cache is indexed both on entities as well as terms. In Section 5.1, we describe how this cache is populated.[2]

A naive, conventional use of the cache would be in eliminating the overheads of repeated database accesses. The crux of EROCS's algorithms lies in more sophisticated use of the contents of this cache to reduce the number of database accesses. At the core of these optimizations, which will be discussed in the next section, lie techniques that exploit the contents of the cache to bound the entity-segment matching scores as well as the annotation scores. These techniques are formally described in Section 5.2.

5.1 Context Cache Population

The algorithms in this paper access the relational database via the following two operations.

- *GetEntities(t)*: Given a term t appearing in D, this operation queries the database and returns the set of all entities that contain the term t in their context.
- *GetTerms(e)*: Given an entity e,[3] this operation queries the database and returns the set of all terms from D that are contained in the context of e.

GetEntities involves (a) identifying the rows containing the term t across all tables labeling the nodes in the entity, and (b) identifying the rows in the pivot table that have a join path (along the edges in the entity template) to any of the identified rows. Step (a) is performed using a text index over the tables in the database, while step (b) involves a union of multiple join queries, one for each node whose labeling table contains a row that contains the term t. Our current implementation exploits DB2 Net Search Extender [15] for combined execution of both steps in a single query. Computing the context of an entity in GetTerms, on the other hand, involves a join query based on the entity template. However, in presence of nested substructure, it is sometimes more efficient to retrieve the context using an outer-union query; such issues are well-known in the XML literature [21].

Clearly, both these operations are expensive, and caching their results makes sense. The result of GetEntities(t) for a term t is cached by inserting the pair (e, t) for each each entity e returned in the result. Similarly, the result of GetTerms(e) for an entity e is cached by inserting the pair (e, t) for each each term t returned in the result.

[2]In this paper, we focus on populating the cache with relevant data; techniques for shrinking the cache by removing garbage entities and terms exist, but are not discussed here.
[3]Henceforth, entity stands for the *identifier* of the entity.

5.2 Context Cache-based Score Bounds

In this section, we show how the contents of the cache at any given point can be used to obtain tight upper and lower bounds on the entity-segment matching scores as well as the annotation scores.

Consider a document segment d and let $T(d)$ be the set of terms in d. Further, let $T_C(d) \subseteq T(d)$ denote the set of terms in d for which GetEntities has been invoked so far. Now, consider an entity $e \in E$ and, as in Section 3.3, let $T(e, d) \subseteq T(d)$ denote the set of terms in the document that appear in the context of e.

To compute $score(e, d)$ exactly, we need to know $T(e, d)$. There are two cases, based on whether or not GetTerms has been invoked on e earlier. If GetTerms has not been invoked on e earlier, then our knowledge of the context of e is limited to the set of terms which were found to be in the context of the entity e by virtue of having invoked GetEntities on them in the past; this set, $T(e, d) \cap T_C(d)$, can be obtained using an index lookup on the cache using e. We use this information to get a lower bound on $score(e, d)$ as:

$$score^-_C(e, d) = \sum_{t \in T(e,d) \cap T_C(d)} tf(t, d).w(t)$$

Further, defining $W_C(d) = \sum_{t \in T(d) - T_C(d)} tf(t, d).w(t)$ and using the fact that $T(e, d) - T_C(d) \subseteq T(d) - T_C(d)$, we also have the following upper bound on $score(e, d)$:

$$score^+_C(e, d) = score^-_C(e, d) + W_C(d)$$

On the other hand, if GetTerms has been invoked on e earlier, then $T(e, d)$ (which is the set obtained as a result of that invocation) is available in the cache; in this case, we can compute $score(e, d)$ exactly and thus have:

$$score^-_C(e, d) = score^+_C(e, d) = score(e, d)$$

The bounds on $score(e, d)$ for entity $e \in E$ and segment d in document D derived above can be used to derive a lower bound $score^-_C(S, F)$ and an upper bound $score^+_C(S, F)$ for a given annotation (S, F) of D. These bounds follow trivially from the definition of $score(F, B)$ in Section 4.1 and are as follows.

$$score^-_C(S, F) = \sum_{d \in S} (score^-_C(F(d), d) - \lambda)$$

$$score^+_C(S, F) = \sum_{d \in S} (score^+_C(F(d), d) - \lambda)$$

In the next section, we show how these bounds can be gainfully used to effectively reduce the number of database access operations that need to be invoked in course of best annotation computation.

6. CONTEXT CACHE-BASED BEST ANNOTATION COMPUTATION

In this section, we present algorithms to compute the best annotation for a given document D that make effective use of the context cache contents to reduce database access operations. The algorithm AllTerms, presented in Section 6.1 uses the cache in a conventional manner to eliminate the overheads of repeated database accesses; this algorithm forms our baseline. Next, in Section 6.2, we present the algorithm AllSegments that reduces the number of GetEntities invocations while computing the best entity for a segment. Finally, in Section 6.3, we present the algorithm that additionally uses a greedy cache refinement strategy to rapidly converge to the best annotation; this algorithm is actually used in EROCS and is therefore called the EROCS algorithm.

6.1 Eliminating Repeated Database Operations

The most straightforward use of this cache is to eliminate repeated invocations of GetEntities(t) or GetTerms(e) for the same term t or the same entity e respectively. This suggests an algorithm that first invokes GetEntities(t) for each term t in the document, and populates the cache with the pairs (e, t) for each e in the result. After this cache population step, the index on entities is used to determine, for each entity in the cache, the terms in the document that belong to that entity. Using this information, the algorithm readily computes the best matching entity for each document segment; once this is done, BestAnnot is invoked to compute the best annotation for the document. We call this algorithm AllTerms.

Despite eliminating repeated invocations, AllTerms does not scale well (cf. Section 7). The reason is that there exist several terms in the document that appear in the context of a large number of entities. Invoking GetEntities on such terms is excessively expensive. Moreover, being low weight, these terms do not contribute much to the score of any entity in case they appear only a few times in the document. In the next section, we develop an optimization that can potentially avoid invocations of GetEntities on such terms.

6.2 Reducing GetEntities Invocations

Consider any algorithm for finding the best matching entity for the given segment d. Starting with no prior information about the best entity, the search space for this algorithm is the entire entity set E. The purpose of invoking GetEntities for terms in d is to build up a set of entities, hopefully much smaller in size than E, that would form a reduced search space in the algorithm's quest for the best matching entity; for the algorithm to be correct, however, it needs to ensure that this reduced search space contains the entity being sought.

The algorithm AllTerms follows a naive, conservative approach and invokes GetEntities for all terms in d — it thus builds up the set of *all* entities that are potentially relevant to d; clearly, the best matching entity is guaranteed to be in this set. Next, we present an optimization that can be used to build the search space in a manner that does not require GetEntities to be invoked on all terms in d, but at the same time ensures that the best matching entity is included in this search space.

6.2.1 *Term Pruning Strategy*

Let $E_C(d)$ be the set of entities such that each $e \in E_C(d)$ has appeared in the result of at least one GetEntities invoked so far for terms in the segment d. We call the cache *complete* with respect to the segment d iff the best matching entity for d is guaranteed to be present in $E_C(d)$.

The idea is to populate the cache by invoking GetEntities on the terms $t \in T(d)$ in decreasing order of $tf(t, d).w(t)$, stopping as soon as the cache becomes complete with respect to d. The challenge in implementing this optimization lies in efficiently checking for completeness of the cache at any point. Next, we present an efficient criteria for the same.

Consider any $e \in E - E_C(d)$. By definition, we have

$T(e,d) \cap T_C(d) = \phi$, which implies $score^-_C(e,d) = 0$ and, thus, $score^+_C(e,d) = W_C(d)$ based on the bounding analysis of Section 5.2. Now, if there exists an entity $e' \in E_C(d)$ such that $score^-_C(e',d) > W_C(d)$ then, clearly, $score(e,d) < score(e',d)$ and therefore e cannot be the best matching entity for d. We have thus proved the following:

Claim 3. *The context cache is complete with respect to the segment d if there exists an entity $e' \in E_C(d)$ such that $score^-_C(e',d) > W_C(d)$.*

Since both $E_C(d)$ and $W_C(d)$ can be progressively maintained as the cache is being populated, this criterion can be checked efficiently.

6.2.2 The AllSegments Algorithm

We now present an algorithm, called AllSegments, that applies the optimization discussed above while computing the best matching entities for each segment in the document.

For each segment d in the document, the algorithm successively invokes GetEntities on the terms $t \in T(d)$ in decreasing order of $tf(t,d).w(t)$. The algorithm progressively maintains the entities in $E_C(d)$ in a heap, max-ordered on $score^-_C(e,d)$, and also maintains $W_C(d)$ during the course of these invocations. As soon as the top entity in the $E_C(d)$ heap has a value greater than $W_C(d)$, the cache is flagged as complete with respect to d. When this happens, the algorithm stops invoking GetEntities on any more terms, and moves to the task of finding the best matching entity from the set $E_C(d)$.

At this point, we already have the entities in $e \in E_C(d)$ in a heap, max-ordered on $score^-_C(e,d)$. The algorithm proceeds by removing the topmost entity e from this heap, invoking GetTerms(e), and using the result find its exact score $score(e,d)$. The algorithm repeats the above process and maintains the entity $\overline{e}$ that has maximum score so far, stopping when the topmost entity e' is such that $score^+_C(e',d) < score(\overline{e},d)$. It is easy to see that, at this point, $\overline{e}$ is the best matching entity for d.

The algorithm starts with an empty cache, but carries the cache over while computing the best matching entity across different segments. When the best matching entities for every segment in the document have been computed, the algorithm invokes BestAnnot to compute the best annotation for the document.

6.3 Greedy Iterative Cache Refinement

In this section, we present the EROCS algorithm that uses a greedy iterative cache refinement strategy to converge to the best annotation for the given document without computing the best entities for all segments. The greedy iterative strategy for computing the best annotations is presented in Section 6.3.1. A cache refinement heuristic that attempts to refine the cache in a way that this greedy iterative strategy converges to the best annotation rapidly is presented in Section 6.3.2.

6.3.1 Greedy Iterative Strategy

Let us revisit the procedure BestAnnot outlined in Figure 3 that computes the best annotation (S^*, F^*) of a given document D. We modify the procedure BestAnnot so that Lines A03, A04 and A12 invoke the score upper-bound function $score^+(e,d)$ instead of the exact $score(e,d)$. Let us call this modified procedure BestAnnot$_C$.

```
Procedure BestAnnotErocs($D$)
Input: Document $D$
Output: (best annotation, score)
Begin
B01 Initialize the context cache as empty
B02 Let $((\bar{S},\bar{F}),s) = $ BestAnnot$_C(D)$
B03 While $score^-_C(\bar{S},\bar{F}) < score^+_C(\bar{S},\bar{F})$
B04         Call UpdateCache$(\bar{S},\bar{F})$
B05         Let $((\bar{S},\bar{F}),s) = $ BestAnnot$_C(D)$
B06 Return $((\bar{S},\bar{F}),s)$
End
```

Figure 4: The EROCS Algorithm

Let $(\bar{S},\bar{F})$ be the annotation returned by BestAnnot$_C$. We make the following claim.

Claim 4. $score^-_C(\bar{S},\bar{F}) \leq score(S^*,F^*) \leq score^+_C(\bar{S},\bar{F})$

Proof. The first inequality follows from $score^-_C(\bar{S},\bar{F}) \leq score(\bar{S},\bar{F})$ and $score(\bar{S},\bar{F}) \leq score(S^*,F^*)$. Also, since BestAnnot$_C$ overestimates the score of every segment in the document, the best score it computes must be an overestimate of the actual best score; this is the second inequality. □

This result suggests a greedy strategy that iteratively improves the cache contents so that the slack between the upper and lower bound scores of the successive best annotations $(\bar{S},\bar{F})$ computed by BestAnnot$_C$ decreases with every iteration.

The resulting procedure, called BestAnnotErocs, appears in Figure 4. Starting with an empty cache, it repeatedly calls BestAnnot$_C$, which computes a best matching annotation $(\bar{S},\bar{F})$ based on the current cache, and then calls the subroutine UpdateCache (discussed later in Section 6.3.2), that updates the cache using $(\bar{S},\bar{F})$. The procedure terminates whenever we find that the best annotation returned by BestAnnot$_C$ has $score^+_C(\bar{S},\bar{F}) = score^-_C(\bar{S},\bar{F})$; at this point, by Claim 4, we know that $(\bar{S},\bar{F})$ is the best annotation.

Since $score^-_C(e,d)$ and $score^+_C(e,d)$ are computed based on the contents in the context-cache, each invocation to BestAnnot$_C$ can be executed efficiently. In fact, since $score^+_C(d,e)$ for most segments d and entities e remains the same across successive invocations of BestAnnot$_C$, EROCS actually uses lazy, incremental techniques in BestAnnot$_C$ to compute the successive best annotations efficiently.

6.3.2 Cache Refinement

For a given annotation (S,F), let us define $slack_C(S,F) = score^+_C(S,F) - score^-_C(S,F)$. Let $(\bar{S}_1,\bar{F}_1)$ and $(\bar{S}_2,\bar{F}_2)$ be the best annotations returned by BestAnnot$_C$ on two successive invocations in the course of the algorithm. The goal of the cache refinement strategy, to be developed in this section, is to choose a cache update at the intervening cache refinement step such that the decrease in slack, $slack_C(\bar{S}_1,\bar{F}_1) - slack_C(\bar{S}_2,\bar{F}_2)$, is maximized.

To start with, let us assume that the only cache update operation allowed is GetEntities. Then, the task reduces to finding the term $t^* \in T(D) - T_C(D)$ for which GetEntities should be invoked next; here $T(D)$ is the set of terms in the document D, and $T_C(D)$ is the set of terms in D for which GetEntities has already been invoked.

Now, given our constraint, the state of the cache at each iteration would be such that it has been populated using only

```
Procedure UpdateCache($\bar{S}, \bar{F}$)
Input: current best annotation
Begin
C01 If the current cache is complete with respect to each $d \in \bar{S}$
C02     Let $d^* = \arg\max_{d \in \bar{S}} slack_C(\bar{F}(d), d)$
C03     Call GetTerms($\bar{F}(d^*)$)
C04 Else
C05     Let $t^* = \arg\max_{t \in T(D)-T_C(D)} tf(t, D).w(t)$
C06     Call GetEntities($t^*$)
End
```

Figure 5: Cache Refinement Algorithm

invocations of GetEntities. It is easy to show that in such a state, for *all* annotations (S, F) of the document D, we have $slack_C(S, F) = W_C(D) = \sum_{t \in T(D)-T_C(D)} tf(t, D).w(t)$ (cf. Section 5.2). Clearly, the difference in slack between $(\bar{S}_1, \bar{F}_1)$ and $(\bar{S}_2, \bar{F}_2)$ is then always equal to $tf(t^*, D).w(t^*)$, where t^* is the term picked by our cache refinement strategy. This gives us an *optimal* cache refinement strategy: at each cache refinement step, invoke GetEntities on the term $t^* \in T(D) - T_C(D)$ with the maximum $tf(t^*, D).w(t^*)$.

Alternatively, let us assume that the cache is complete with respect to each segment in the document D, and the only cache update operation allowed is GetTerms. Then, the task reduces to finding the entity $e^* \in E - E(D)$ for which GetTerms should be invoked next; here, $E(D)$ is the set of entities for which GetTerms has already been invoked. However, now no obvious global property of the kind seen above holds, and we need to devise heuristics, taking cue from the current best annotation $(\bar{S}_1, \bar{F}_1)$. Instead of trying to maximize the slack decrease across two successive optimal annotations, we make the assumption that the current annotation $(\bar{S}_1, \bar{F}_1)$ would remain the optimal even after the update. Under this assumption, the goal of the cache refinement strategy is to find the entity e^* such that invoking GetTerms on this entity maximizes the slack decrease for the current best annotation $(\bar{S}_1, \bar{F}_1)$.

For a given segment d and entity e, define $slack_C(e, d) = score_C^+(e, d) - score_C^-(e, d)$. Then, we have $slack_C(\bar{S}_1, \bar{F}_1) = \sum_{d \in \bar{S}_1} slack_C(\bar{F}_1(d), d)$. Clearly, a principled way to maximize the reduction of $slack_C(\bar{S}_1, \bar{F}_1)$ is to maximize the reduction of the largest $slack_C(\bar{F}_1(d), d)$ in the summation. This leads to the following cache refinement strategy: at each cache refinement step, invoke GetTerms on the best matching entity for the segment in $\bar{S}_1$ with the maximum slack.

In EROCS, we follow a cache refinement strategy that is a hybrid of the two. Specifically, at each cache refinement step, it checks whether the cache is complete with respect to all the segments in the current best annotation $(\bar{S}, \bar{F})$. If this is false, it invokes GetEntities on the term $t^* \in T(D) - T_C(D)$ with the maximum $tf(t^*, D).w(t^*)$; if this is true, it invokes GetTerms on the best matching entity for the segment in $\bar{S}$ with the maximum slack. This strategy appears formally as the procedure UpdateCache in Figure 5.

Until the point the cache becomes complete with respect to at least one segment in the document, this strategy enforces the GetEntities-only constraint and makes decisions that are the best with respect to that constraint. After the point when the cache is complete with respect to all the segments in the document, it enforces the GetTerms-only constraint and makes decisions that are (heuristically) the best with respect to that constraint. In general, the strategy favours GetEntities initially and GetTerms later, which is justified since the initial terms contribute more towards decreasing the slack than the latter terms.

In Section 7, we show that this hybrid strategy works well in practice. As a part of our future work, we plan to address the problem of finding a provably good strategy for cache refinement.

7. EXPERIMENTAL STUDY

The techniques proposed in this paper are based on two main contributions. The first contribution is the proposition that entity identification should be done at a fine-grained level, i.e. by matching entities with segments within the document, rather than simply matching with the entire document. The second contribution is the efficient implementation of this proposition as a greedy iterative cache refinement strategy. In this section, after discussing the experimental setup, we evaluate the efficacy of fine-grained matching (Section 7.1) and of greedy iterative cache refinement (Section 7.2) against less sophisticated alternatives. Next, we study the effect of varying the value of the parameter λ on the accuracy of the techniques (Section 7.3). Finally, we study the improvement in the accuracy of the current best annotation as the greedy iterative cache refinement progresses (Section 7.4).

Platform. The implementation was done in Java with J2SE v1.4.2 (approx. 2000 lines of code) and executed on a 2.4 GHz Machine with 4GB RAM running Windows XP SP1. The relational DBMS used was IBM DB2 UDB v8.1.5 co-located on the same machine. Communication between EROCS and DB2 was through JDBC. We used the IBM DB2 UDB Net Search Extender v8.1 as our text index over the database; this was because of convenience and easy availability. A more specialized text index on the database, we believe, would achieve even better performance than that reported in this section.

Structured Dataset. This study used a subset of the *Internet Movie Database*[4], with movies as the entities of interest. The database contains roughly 4 million records and has size 2GB across eight tables. The `Movies` table (401660 rows) contains names of movies and the `Persons` table (287398 rows) contains a list of persons along with their names. The remaining six tables relate rows in these two tables. The `Actors` (1619647 rows) and `Actresses` (776396 rows) tables relate the movies with their cast along with the corresponding character names. The `Directors` (244394 rows), `Producers` (150838 rows), `Writers` (232420 rows) and `Editors` (46795 rows) tables relate the movies with their designated crew. The entity template is defined so that the `Movies` table is the pivot table (the total number of entities are therefore 401660) and other tables are the context tables.

Document Dataset. The documents used in this study were assorted movie reviews downloaded from the *Greatest Films*[5] website. These reviews were processed using a part-of-speech tagger, and the noun-phrases identified as the relevant terms. We removed the names of the movies

[4]http://www.imdb.com
[5]http://www.filmsite.org

in the review text, but retained this information separately for evaluation purposes. To control the quality and have multiple entities in the documents, we decomposed these base documents into segments of approximately 8 sentences each, and classified each segment as *good* or *bad* based on the average weight of terms contained in the segment. This gave us a set of good and a set of bad documents for each movie. Now, given the number of entities per document (K) and the fraction of good segments (α) as parameters, we generated a random document by picking a random sequence of K distinct movies, and for each movie in the sequence, including a good segment with probability α and a bad segment with probability $(1 - \alpha)$. Note that the length of the document is a multiple of K, and is not considered as a separate parameter. Our final repository of documents included 50 documents for each combination of $K = 1, 2, \ldots, 10$ and $\alpha = 0.0, 0.1, \ldots, 1.0$.

Parameter Settings. EROCS has only one parameter, λ; unless otherwise stated, its value is fixed at 4. In Section 7.3, we justify this choice, and also study the effect of varying the value of λ.

Accuracy Metric. We use an annotation accuracy measure that not only considers the accuracy of the set of entities identified, but also the accuracy of the embeddings of these entities in the document as specified by the annotation. Let e be the entity a given sentence actually belongs to, and let E' be the set of entities that best match the segment containing this sentence according to the given annotation.[6] Then, precision and recall for this sentence are computed as $|E' \cap \{e\}|/|E'|$ and $|E' \cap \{e\}|$ respectively. The accuracy of the annotation for the document is then computed as the harmonic mean of the average precision and recall over the sentences in the document.[7]

7.1 Efficacy of Fine-Grained Entity Matching

In this experiment, we show that EROCS's strategy for identifying entities that best-match the document at a fine-grained, individual segment level leads to greater accuracy than a technique that simply identifies the entities that best-match the entire document.

For the sake of this experiment, assume that the number of entities (K) in the given document is known apriori. Given this information, our alternative algorithm (called TopK) picks all the entities with top K matching scores with respect to the entire document considered as a single segment, and then uses $\textsc{BestAnnot}_C$ to compute the best annotation considering only these entities.

We compared the accuracy of EROCS and TopK for documents with varying quality (α), as well as varying entities contained within (K). We first fixed $\alpha = 0.8$ and varied $K = 1, 2, \ldots, 10$, and then fixed $K = 10$ and varied $\alpha = 0.0, 0.1, \ldots, 1.0$. For each combination, we computed the average accuracy of each algorithm for the 50 documents in the repository for that combination. The results are plotted in Figure 6 and Figure 7 respectively.

Discussion. We discuss the results in Figure 6 first. When

[6]We assume that an annotation may associate multiple entities with a segment in case all these entities have the maximum score.

[7]The harmonic mean of precision and recall is a popular measure of accuracy in the Information Retrieval literature, and is known as the *F-measure* [3].

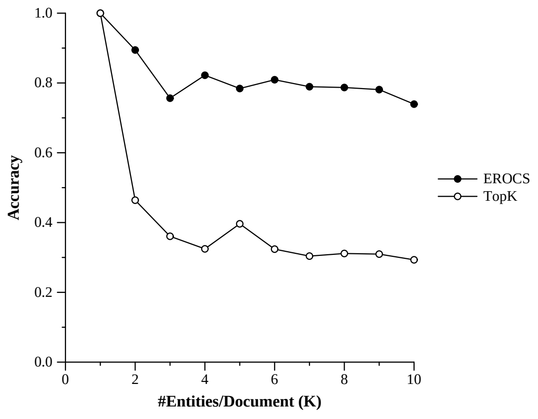

Figure 6: Accuracy of EROCS, TopK for varying K

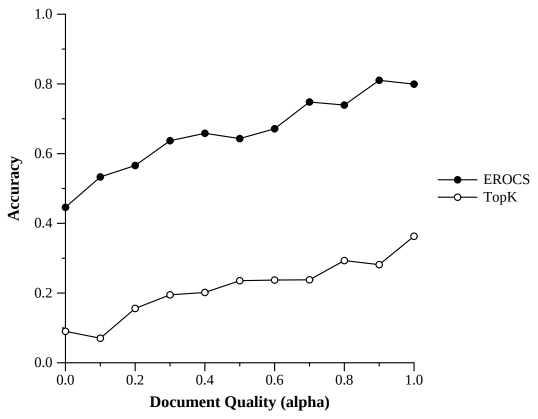

Figure 7: Accuracy of EROCS,TopK for varying α

the document has exactly one entity, the two algorithms are identical, and both are able to identify the best entity perfectly. However, for document with multiple entities, TopK's accuracy falls drastically. This is because in TopK considers the entire document as a single segment. As a result, unrelated terms in different, well separated parts of the document interfere, leading to irrelevant entities being scored high. EROCS, in contrast, compares the best score for the entire document as a single segment with the best score for each possible partitioning of the document; this allows it to exploit the substructure within the document to filter out such irrelevant entities. From the figure, we see that EROCS is able to maintain an accuracy close to 0.8 with increasing K, whereas the accuracy of TopK deteriorates to about 0.3 for the same documents.

Next, consider the results in Figure 7. The increase in accuracy for both the algorithms with increasing α is because documents with higher α have better clues in terms of higher weight terms. However, the significant gap between the accuracy of EROCS and TopK persists irrespective of the quality of the documents. Even for the lowest-quality documents considered, i.e. $\alpha = 0$, EROCS was able to achieve an accuracy of 0.45; in contrast, for the highest-quality documents considered, i.e. $\alpha = 1$, TopK could merely achieve an accuracy of 0.36.

Overall, these experimental results clearly show that the fine-grained, segment-level entity matching strategy proposed

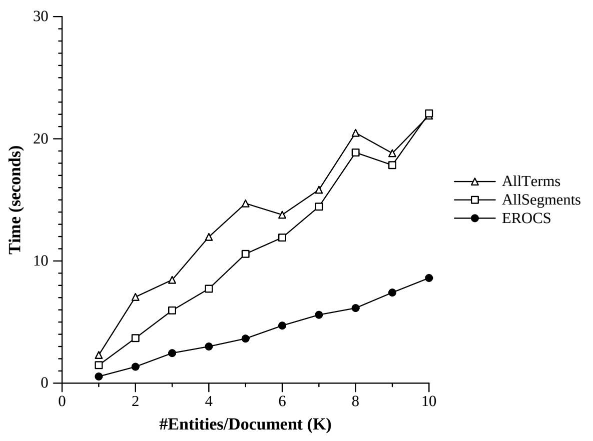

Figure 8: Execution Efficiency of AllTerms, AllSegments, EROCS for varying K

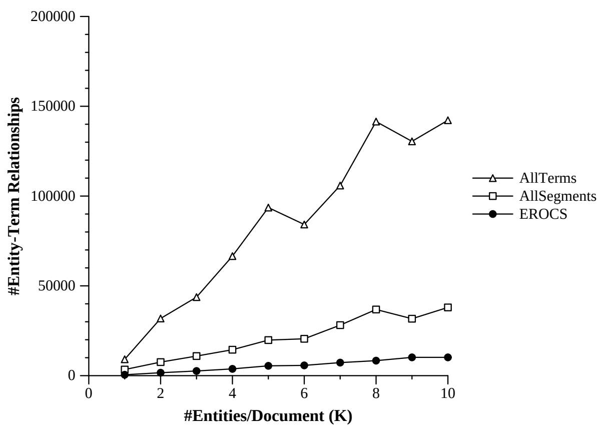

Figure 9: Space Overhead of AllTerms, AllSegments, EROCS for varying K

in this paper has significant advantage in terms of accuracy over the simpler-minded alternative.

7.2 Efficacy of Greedy Iterative Cache Refinement

In the previous section, we established the need for segment-level entity matching. The dynamic-programming algorithm proposed in Section 4.3 is clearly an efficient algorithm in terms of complexity for performing the same. In this section, we validate the efficacy of greedy iterative cache refinement (Section 6.3) as an implementation strategy for this dynamic programming algorithm. The validation is against the following alternatives that were used to motivate the strategy in Section 6.

- AllTerms, that does not exploit term-pruning (Section 6.1).
- AllSegments, that exploits term-pruning, but does not exploit greedy cache-refinement (Section 6.2).

We compare EROCS, which uses greedy iterative cache refinement, with these two alternatives with respect to execution efficiency and space overheads. The execution efficiency was measured in terms of clock time while the space overheads were measured in terms of the number of (entity, term) pairs computed and maintained in the cache.

As in the previous experiment, we first fixed $\alpha = 0.8$ and varied $K = 1, 2, \ldots, 10$, and then fixed $K = 10$ and varied $\alpha = 0.0, 0.1, \ldots, 1.0$. For each combination, we computed the average time and space overheads of each algorithm over the 50 documents in the repository for that combination.

The execution efficiency and space overhead results are plotted for varying K in Figure 8 and Figure 9, and for varying α in Figure 10 and Figure 11 respectively.

Discussion. Figure 8 shows that for all values of K considered, EROCS achieves at least 60% reduction in execution time over its nearest competitor, AllSegments; this large gap between EROCS and AllSegments shows that the greedy cache-refinement strategy is indeed effective in reducing the number of segments for which the scores were computed exactly. Notice that EROCS takes 0.55s for $K = 1$ and 8.6s for $K = 10$, scaling linearly with increasing K; this implies a sub-second execution time for each additional entity, which is encouraging. The gap between AllSegments and AllTerms is relatively small, and decreases as K increases. This is because an increase in K implies increase in document size, which leads to an increase in the number of segments for which AllSegments needs to compute the exact score; as a result, any gains AllSegments has over AllTerms due to the term-pruning optimization become negligible as K increases.

Figure 9 compares the space overheads of the three algorithms for varying K. It is interesting to see the rapid rate at which the space overhead for AllTerms grows with increasing K – from 9111 at $K = 1$ to 142211 at $K = 10$. This is because AllTerms gets the entity set for each distinct term in the document, and maintains this information in the cache. This includes even low weight terms that occur in a large fraction of the entities in the database; moreover, many of these entities have only this one word in the document. AllTerms thus results in a large number of (entity, term) relationships in the cache, a majority of them unnecessary. AllSegments and EROCS, in contrast, exploit the term-pruning optimization for each segment, avoiding terms with lower weight. For AllSegments, this optimization results in a highly reduced space overhead of 3398 at $K = 1$ and 37992 at $K = 10$. EROCS further exploits the greedy cache-refinement strategy to avoid computing exact scores for several segments, resulting in space overhead of a mere 505 at $K = 1$ (a reduction of almost 95% over AllTerms and 85% over AllSegments), and 10181 at $K = 10$ (a reduction of almost 93% over AllTerms and 73% over AllSegments). Moreover, for EROCS, these overheads scale up linearly, with a slope of about 1000 per additional entity, which is reasonable.

In Figure 10, we notice that with increasing document quality (α), the execution time for EROCS and AllSegments decreases, while the execution time for AllTerms increases. Figure 11 shows a similar trend for space overheads as well. This can be explained as follows. As the number of higher weight terms in the document increase, the distribution of weights in most segments becomes more skewed; this can be effectively exploited by the term-pruning optimization. On the other hand, increase in higher weight terms in the document also implies increase in the number of distinct terms in the document (recall that a higher weight term is present in lesser number of entities); since AllTerms necessarily queries

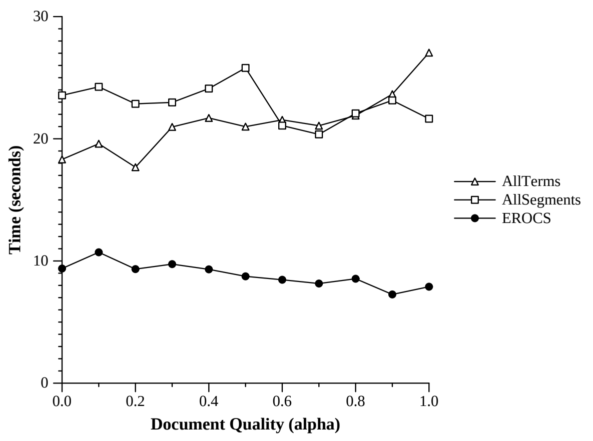

Figure 10: Execution Efficiency of AllTerms, AllSegments, EROCS for varying α

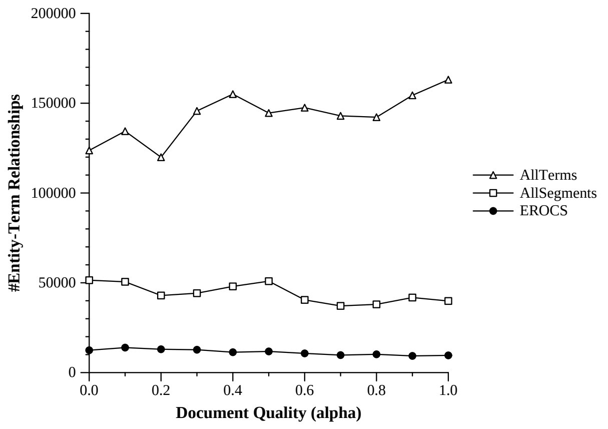

Figure 11: Space Overhead of AllTerms, AllSegments, EROCS for varying α

all distinct terms, this leads to an increase in its execution time and space overhead.

The results in Figure 10 show that AllSegments performs worse than AllTerms at smaller values of α. This happens because our implementation of AllTerms batches multiple invocations of GetEntities together. At smaller values of α, the weights of the terms in the document are low, and thus the term-pruning strategy is not effective. As a result, both AllTerms and AllSegments invoke GetEntities for almost the same number of terms. However, because of the batched implementation, AllTerms is able to perform these invocations more efficiently as compared to AllSegments. For larger α, as discussed above, the term-pruning strategy becomes effective, resulting in improved performance of AllSegments as compared to AllTerms.

Overall, the experimental results in this section clearly establish the efficacy of the greedy cache-refinement strategy used in EROCS.

7.3 Effect of Varying λ

The annotation scoring function for EROCS has a parameter, λ; this parameter specifies the penalty to be paid by an annotation for each segment it contains. In this experiment, we explore the effect of varying the value of λ on EROCS's accuracy.

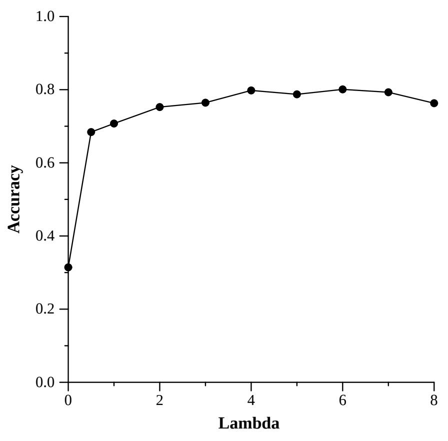

Figure 12: Accuracy of EROCS for varying λ

We generated set of 50 documents with $K = 10$ and $\alpha = 0.8$, and studied the average accuracy of EROCS on documents in this set as λ was varied from 0 to 8. Figure 12 reports the accuracy of the algorithms as a function of λ.

Discussion. For $\lambda = 0$, the precision and recall values are rather poor, suggesting it is a good idea to penalize each segment with $\lambda > 0$; this guarantees that annotations containing segments with score $< \lambda$ are not output as the optimal. We further note that, interestingly, as long as $\lambda > 0$, the accuracy remains stable with respect to variations in λ. This robust behavior is desirable, since it obviates the need of tuning the parameter. The plot shows that for our document set, $\lambda = 4$ gives the best accuracy; this is the reason for our fixing this value as the default in this experimental study.

7.4 Progressive Accuracy Improvement with Greedy Iterative Cache Refinement

The greedy iterative cache refinement strategy in EROCS has the advantage that at any point in time, the algorithm has a "current best" annotation. With successive iterations, this current best annotation keeps improving in accuracy, converging to the final best annotation when the algorithm terminates. In this experiment, we explore how the accuracy of this current best annotation progresses during the course of the algorithm.

We generated set of 50 documents with $K = 10$ and $\alpha = 0.8$, and studied the average accuracy of EROCS on this set as a function of the fraction of completion time. The results are plotted in Figure 13.

Discussion. The results show that the accuracy increases roughly linearly with time; that is, the improvement in accuracy is uniformly distributed across the iterations, except at the very beginning and at the very end. We also experimented with other values of K and α, and found that this general trend remains the same.

This behavior lets us reason about a variant of the given algorithm that, given a time bound, returns the best annotation it is able compute in this time. The results of this experiment suggest that the accuracy of this variant would gracefully (i.e. linearly) degrade with the time quota allotted. This is a useful property that can be exploited while deploying EROCS in real-time scenarios.

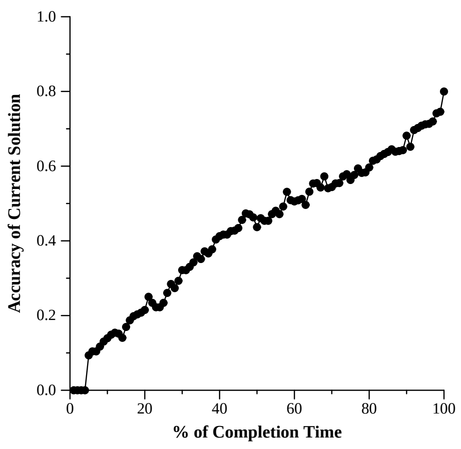

Figure 13: Accuracy of current best annotation as EROCS progresses

8. CONCLUSION

In this paper, we presented a system for inter-linking information across structured databases and documents. We presented efficient techniques for identification and embedding of entities relevant to a given document. This involved principled development of an effective successive approximation algorithm that tries to keep the amount of information retrieved from the database in course of the computation as small as possible. The linkages discovered as a result of the ideas in the paper can be profitably exploited in several applications in the database-centric as well as IR-centric domains, and may serve a purpose of bridging the two.

Acknowledgments. We thank Sreeram Balakrishnan for interesting discussions and help, and Laura Haas and Nelson Mattos for their continued support. Meghana Kshirsagar contributed to the initial codebase of EROCS. We also thank the anonymous reviewers of this paper for their valuable feedback.

9. REFERENCES

[1] Agichtein, E., and Ganti, V. Mining reference tables for automatic text segmentation. In *SIGKDD* (2004).

[2] Agrawal, S., Chaudhuri, S., and Das, G. DBXplorer: A System for Keyword-Based Search over Relational databases. In *ICDE* (2002).

[3] Baeza-Yates, R., and Ribeiro-Neto, B. *Modern Information Retrieval.* Addison Wesley/ACM, 1999.

[4] Barsalou, T. View objects for relational databases. Tech. Rep. STAN-CS-90-1310, CS Dept, Stanford University, 1990. Ph.D. thesis.

[5] Barsalou, T., Keller, A. M., Siambela, N., and Wiederhold, G. Updating relational databases through object-based views. In *SIGMOD* (1991).

[6] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., and Sudarshan, S. Keyword Searching and Browsing in Databases using BANKS. In *ICDE* (2002).

[7] Borthwick, A., Sterling, J., Agichtein, E., and Grishman, R. Exploiting diverse sources via maximum entropy in named entity recognition. In *Sixth Workshop on Very Large Corpora* (1998).

[8] Chakrabarti, S. Breaking through the syntax barrier: Searching with entities and relations. In *PKDD* (2004).

[9] Chandel, A., Nagesh, P., and Sarawagi, S. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE* (2006).

[10] Chaudhuri, S., Ganti, V., and Motwani, R. Robust identification of fuzzy duplicates. In *ICDE* (2005).

[11] Chen, P. P.-S. The Entity-Relationship Model–Toward a Unified View of Data. *ACM TODS 1*, 1 (1976).

[12] Cohen, W., and Sarawagi, S. Exploiting dictionaries in named entity extraction: Combining semi-markov extraction processes and data integration methods. In *SIGKDD* (2004).

[13] Doan, A., and Halevy, A. Semantic Integration Research in the Database Community: A Brief Survey. *AI Magazine: Special Issue on Semantic Integration* (2005).

[14] Hristidis, V., Gravano, L., and Papakonstantinou, Y. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB* (2003).

[15] IBM. *IBM DB2 UDB Net Search Extender : Administration and User Guide (version 8.1)*, 2003.

[16] Li, X., Morie, P., and Roth, D. Semantic Integration in Text: From Ambiguous Names to Identifiable Entities. *AI Magazine: Special Issue on Semantic Integration* (2005).

[17] Mansuri, I., and Sarawagi, S. Integrating unstructured data into relational databases. In *ICDE* (2006).

[18] Premerlani, W. J., and Blaha, M. R. An Approach for Reverse Engineering of Relational Databases. *CACM 37*, 5 (1994).

[19] Roy, P., Mohania, M., Bamba, B., and Raman, S. Towards automatic association of relevant unstructured content with structured query results. In *CIKM* (2005).

[20] Sarawagi, S. Automation in information extraction and integration (tutorial). In *VLDB* (2002).

[21] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., and Naughton, J. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB* (1999).

[22] Walker, M. H., and Eaton, N. J. *Microsoft Office Visio 2003 Inside Out.* Microsoft Press, 2003.

[23] Winkler, W. E. The state of record linkage and current research problems. Tech. rep., U.S. Census Bureau, 1999.