Exploring a Few Good Tuples From Text Databases

Alpa Jain[1], Divesh Srivastava[2]
[1]Columbia University, [2]AT&T Labs-Research

Abstract— **Information extraction from text databases is a useful paradigm to populate relational tables and unlock the considerable value hidden in plain-text documents. However, information extraction can be expensive, due to various complex text processing steps necessary in uncovering the hidden data. There are a large number of text databases available, and not every text database is necessarily relevant to every relation. Hence, it is important to be able to quickly explore the utility of running an extractor for a specific relation over a given text database before carrying out the expensive extraction task. In this paper, we present a novel exploration methodology of *finding a few good tuples* for a relation that can be extracted from a database which allows for judging the relevance of the database for the relation. Specifically, we propose the notion of a *good(k, ℓ)* query as one that can return any k tuples for a relation among the top-ℓ fraction of tuples ranked by their aggregated confidence scores, provided by the extractor; if these tuples have high scores, the database can be determined as relevant to the relation. We formalize the access model for information extraction, and investigate efficient query processing algorithms for *good(k, ℓ)* queries, which do not rely on any prior knowledge about the extraction task or the database. We demonstrate the viability of our algorithms using a detailed experimental study with real text databases.**

I. Introduction

Oftentimes, collections of text documents such as newspaper articles, emails, web documents, etc. contain large amounts of structured information. For instance, news articles may contain information regarding disease outbreaks which may be put together using the relation *DiseaseOutbreak*$\langle$*Disease, Location*$\rangle$. To access these relations, we must first process documents using an appropriate *information extraction system*. Examples of real-life extraction systems include Avatar[1], DBLife[2], DIPRE [3], KnowItAll [10], Rapier [5], Snowball [2]. These relations that are extracted from text documents differ from traditional relations in one important manner: not all tuples in the extracted relations may be valid instances of the target relation [14], [16]. To reflect the confidence in an extracted tuple, extraction systems typically assign a score along with an extracted tuple (e.g., [2], [9], [10]).

Example 1.1: Consider the task of extracting information about recent disease outbreaks to generate the DiseaseOutbreak *relation. We are given an extraction system, that applies the pattern "*$\langle$*Disease*$\rangle$ *outbreak is sweeping throughout* $\langle$*Location*$\rangle$*" to identify the target tuples. As a simple example of confidence score assignment, consider the case where the extraction system uses edit-distance-based similarity matching between the context of a candidate tuple and the pattern. Specifically, if* max *is the maximum number of terms in an extraction pattern and x is the number of word transformations for a tuple context to match the pattern, then the confidence score is computed as* $1 - \frac{x}{max}$ *(*$x \leq \text{max}$*). Given a text snippet,* "A Cholera outbreak is sweeping throughout Sudan as of ...," *and* max $= 4$*, the extraction system generates the tuple* $\langle$*Cholera, Sudan*$\rangle$ *with a score of 1.0 (*$x = 0$*). On the other hand, for a text snippet,* "... checking for Measles is America's way of avoiding ...," *the extraction system generates the tuple* $\langle$*Measles, America*$\rangle$ *with a lower confidence score of 0.25 (*$x = 3$*).* □

A tuple may also be associated with multiple scores: facts are often repeated across text documents and this redundancy can be used to further boost the confidence score of a tuple [9].

Example 1.2: **(continued).** *Using the same extraction system in Example 1.1 to process another document in the collection that contains the text snippet,* "A cholera outbreak sweeping throughout Sudan resulted in ..." *will result in extracting the same tuple in Example 1.1 but now with a score of 0.75 (*$x = 1$*).* □

The total confidence score of a tuple is an *aggregation* of individual scores derived from processing different documents. We can amass available scores for tuples and rank them using an aggregate scoring function, while viewing extraction systems as "black-boxes" that take as input a document and produce tuples along with some confidence scores as illustrated in the examples above. Typically, good tuples such as $\langle$Cholera, Sudan$\rangle$ will occur multiple times [9] and thus, have their overall confidence scores boosted; on the other hand, low quality tuples such as $\langle$Measles, America$\rangle$ will be sparse and not have their scores boosted.

This notion of a score per tuple provides an opportunity for ranking the extracted tuples and, in turn, allowing for a new query paradigm of exploring a *few high-ranking good tuples*. As a concrete example scenario for this query paradigm, in this paper, we consider the problem of exploring the potential of a text corpus for extracting tuples of a specific relation. Information extraction on a large corpus is a time-consuming process because it often requires complex text processing (e.g., part-of-speech or named-entity tagging). Before embarking on such a time-consuming process on a corpus that may or may not yield high confidence tuples, a user may want to extract a few good tuples from the corpus, allowing the user to make an informed decision about deploying an information extraction system. For instance, given the task of extracting the *DiseaseOutbreak* relation, a test collection that contains Sports-related documents from a newspaper archive is unlikely to generate tuples with high confidence scores. Knowing the

[1]`http://www.almaden.ibm.com/cs/projects/avatar`
[2]`www.dblife.cs.wisc.edu`

nature of tuples that are among the high ranking tuples can help decide whether processing the entire corpus is desirable.

How can we identify a few good tuples for a relation buried in a text database? The first solution that suggests itself is to draw a random sample of tuples in the database, by processing a random set of database documents using an appropriate extraction system. Unfortunately, this approach leads to a negatively biased view of the database: typically text databases (even those that are relevant to a relation) contain a relatively large number of low scoring, incorrect tuples, which, in turn, leads to large number of low scoring tuples (false positives) in a random sample of tuples. Another solution is to derive the top-k ranking tuples of the relation based on the aggregated scores of each tuple. However, using top-k query models is undesirable since, as we will see, deriving the top-k answers in our extraction-based scenario requires near complete processing of a database, for each relation.

In this paper, we propose a novel approach to identify a few good tuples for a relation based on returning any k tuples among the top-ℓ fraction of tuples ranked by their aggregate confidence scores. We refer to this as the *good(k, ℓ)* model, and investigate the problem of *efficiently identifying good(k, ℓ) answers from a text database*. An intuitively appealing solution to process a *good(k, ℓ)* query is to draw a set of $\frac{k}{\ell}$ tuples from the database and return the top-k tuples among the candidate set. Such a two-phase approach allows to focus the expensive extraction process on only a small subset of the database. However, following this approach poses several important challenges. First, we need to build efficient algorithms to identify the final k answers among a candidate set. Second, we need to identify an appropriate size for the candidate set, so that it contains enough answer tuples and at the same time avoids wasteful processing of documents.

To identify the set of k answers from a candidate set, we propose two algorithms. Our first algorithm, $E\text{-}Upper$, is a deterministic algorithm, that adapts an existing top-k algorithm, namely Upper [4], to our setting. The second algorithm, Slice, is a probabilistic algorithm which recursively processes documents identified through query-based access, returning promising subsets of the candidate tuple set that are likely to be in the $good(k, \ell)$ answer set, while performing early pruning on subsets that are unlikely to be in the answer set. This repeated triage is achieved by modeling the evolution of ranks of tuples as a sequence of *rank inversions*, wherein pairs of adjacent tuples in the current rank order independently switch rank positions with some probability. Within this framework, query processing time can be reduced by trading off time with answer quality, in a user-specific fashion.

Identifying a "right-sized" candidate set is non-obvious because of the skew in the distribution of tuple occurrences. Depending on how we choose to aggregate individual scores of a tuple we may be more (or less) likely to observe a frequently occurring tuple at a given rank. To account for the effect of the aggregate scoring function and the skew in the number of tuple occurrences, we present a iterative method that first learns the relevant parameters of the data using a small sample, and then uses the learned parameters to adaptively choose a

$$\left(\begin{array}{cccccccccccc}
 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & d_{10} & d_{11} \\
t_1 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.75 & 0 & 0.6 \\
t_2 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.8 & 0 & 0 & 0 & 0 \\
t_3 & 0.2 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
t_4 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0.1 & 0 & 0 & 0 \\
t_5 & 0.1 & 0.1 & 0 & 0 & 0.2 & 0 & 0.6 & 0 & 0 & 0 & 0 \\
t_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0 \\
t_7 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0.7 & 0 \\
t_8 & 0 & 0 & 0 & 0 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 \\
t_9 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0.2 & 0.2 & 0 & 0 \\
t_{10} & 0.05 & 0 & 0.15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
t_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.05 \\
t_{12} & 0 & 0.9 & 0 & 0 & 0.9 & 0 & 0 & 0.9 & 0 & 0.9 & 0 \\
t_{13} & 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 \\
t_{14} & 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
t_{15} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 \\
t_{16} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.7 \\
t_{17} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0 \\
t_{18} & 0.3 & 0 & 0 & 0.2 & 0 & 0.3 & 0 & 0 & 0.4 & 0 & 0 \\
t_{19} & 0 & 0.5 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\
t_{20} & 0 & 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0
\end{array}\right)$$

Fig. 1. Sample matrix of confidence scores.

"right-sized" candidate set. In summary, the contributions of this paper are:

- We formalize a new query model, i.e., *good(k, ℓ)* for the task of exploring databases for an extraction system.
- We present two query processing algorithms for the $good(k, \ell)$ queries, one deterministic and the other probabilistic, which do not rely on any prior knowledge of the extraction task or the database, making them suitable for our data exploration task.
- We evaluate the effectiveness of our algorithms using a detailed experimental study over a variety of real text data sets and relations.

The rest of this paper is organized as follows: Section II introduces and formally defines our data, access, and query models. In Section III, we present an overview of our query processing approach. In Section IV we present *good(k, ℓ)* processing algorithms, which consist of a deterministic and a probabilistic algorithm. In Section V we discuss how we pick an initial candidate set of tuples to be processed by the proposed algorithms. We then present our experimental results in Sections VI. Section VII discusses related work. Finally, we conclude the paper in Section VIII.

II. Data, Access, and Query Models

We first define the data and access models which form the basis of our query processing scenario. We then formalize the *good(k, ℓ)* query model and the problem we address.

A. Data model

The primary type of objects in our query processing framework are tuples extracted from a text database D using an extraction system E. Given a tuple t, we describe t as a vector $\mathbf{sv(t)} = [s_1, s_2, \cdots, s_{|D|}]$, where s_j $(0 \leq s_j \leq 1)$ is the score assigned to the tuple after processing document d_j $(j = 1 \ldots |D|)$. For documents that do not generate a tuple, the associated score element is 0.

Example 2.1: Consider a sample matrix of scores for tuples across documents with documents as columns and tuples as rows (see Figure 1). Each element (i, j) in the matrix represents the score for tuple t_i after processing document d_j. For instance, tuple t_1 represents the tuple $\langle$Cholera, Sudan$\rangle$ from Example 1.1 which was extracted from two documents, denoted as d_1 and d_9 from our previous examples. Additionally, t_1 also occurs *in a third document d_{11} with a score of 0.6. Thus,* $\mathbf{sv}(\mathbf{t_1})$ = *[1.0, 0, 0, 0, 0, 0, 0, 0, 0.75, 0, 0.6].* □

Given a tuple t, we derive its final score $s(t)$ using an aggregate function. In our discussion, we consider two aggregate functions. Our first function is summation, a common choice, which computes the final score of a tuple t as the sum of the elements in the associated score vector. Specifically,

$$s(t) = \sum_{i=1}^{|D|} s_i \tag{1}$$

where s_i are the elements in $\mathbf{sv}(\mathbf{t})$. We refer to this function as *sum*. For instance, the final score for the tuple t_1 in Figure 1 using *sum* is 2.35. For our second aggregate function, we view the observed occurrences of a tuple along with their confidence scores as independent events. To compute the aggregated confidence, we derive the probability of the union of these events occurring based on the Inclusion-exclusion principle. Specifically,

$$s(t) = \sum_{i=1}^{n} s_i - \sum_{\substack{i,j: \\ 1 \leq i < j \leq n}} s_i \cdot s_j - \sum_{\substack{i,j,k: \\ 1 \leq i < j < k \leq n}} s_i \cdot s_j \cdot s_k + \ldots \tag{2}$$

where $n = |D|$. We can derive this expression as $s(t) = 1 - \prod_{i=1}^{|D|}(1 - s_i)$. We refer to this function as *incl-excl*. Using *incl-excl*, the final score of the t_1 in Figure 1 is 1.0. This function allows for a probabilistic interpretation of the confidence scores observed for a tuple: if s_i is the probability that a tuple t is a good tuple, then $s(t)$ is the probability of the tuple being correct as gathered from multiple evidence.

B. Access model

We define two complementary access methods for exploring tuples and their scores based on two general ways of retrieving database documents for extraction, namely, a scan- or query-based retrieval [15], [16]. Our first access method, *S-access*, uses a scan-based retrieval and sequentially retrieves documents from the database. Upon discovering a tuple t, we can query for a specific set of documents that contain the terms in the tuple, by constructing an appropriate conjunctive text search query for t and issuing it to the search interface of the database. We refer to this access method as *Q-access*. In principle, *Q-access* can retrieve all documents that contain a given tuple, provided there is no limit on the maximum number of results returned by the search interface.

Example 2.2: Following Example 1.1, we could have retrieved document d_1 using S-access and derived the tuple ⟨Cholera, Sudan⟩. Then, using Q-access we can issue the query "[Cholera and Sudan]" to the search interface of the database to derive other documents, namely, d_9 and d_{11} that can contain this tuple. Note that we could possibly retrieve some other documents that also satisfy this query but do not generate the tuple. □

The conjunctive search queries generated using the extracted tuples rarely match all database documents, allowing *Q-access*, as compared to *S-access*, to reduce the number of documents to process in order to derive the complete score vector of a tuple. *Q-access* is an appealing access method for another important reason: using *Q-access* we can quickly determine an upperbound, $|H(t)|$, on the number of documents in which a tuple can occur. As we will see, the number of matching documents for a tuple serves as an important guidance for our query processing algorithms.

C. Query model

We now introduce a novel query model for the goal of finding a few good tuples to aid in data exploration tasks. We also argue why existing top-k query processing algorithms are infeasible for deriving *good(k, ℓ)* answers.

Definition 2.1: **[*good(k, ℓ)* query]** *A good(k, ℓ) query returns any k tuples that belong to the top-ℓ ($0 < \ell \leq 1$) fraction of the tuples in the database, where the ranking score of each tuple is the aggregation of the extraction scores of the extracted instances of the tuple, according to some given aggregation function.* □

An approach to process a *good(k, ℓ)* query is to use existing top-k query processing algorithms: given a *good(k, ℓ)* query, we can construct a top-k query and derive answers that are among valid *good(k, ℓ)* answers. Unfortunately, the access methods outlined above retrieve tuple scores in an unsorted fashion, making top-k algorithms an undesirable option, as discussed next.

Proposition 2.1: Using a sequential, unsorted access to the tuples in a database, in the worst case, a top-k query execution would need to observe all tuples in the database. If the score distribution is known, in the average case a top-1 query would need to observe at least half the tuples in the database. □

Consider the processing of a top-1 query (i.e., we need to retrieve the best tuple in the database for the query). Consider also that there is no absolute upper bound on the extraction scores[3]. In the absence of any score distribution information, we need to observe all possible tuples and scores in order to pick *the best* tuple. Because of the lack of the sorted access, even if the score distribution is known, the expected number of tuples to observe is $\frac{|T|}{2}$, where $|T|$ is the total number of distinct tuples that can be extracted from the database. Generalizing to top-k tuples, assuming the score distribution is known, the expected number of tuples to observe is $\frac{k}{k+1} \cdot |T|$. If tuples are uniformly distributed across the database documents, fetching the desired top-k tuples would imply near-complete processing of a database.

As top-k answers are a restrictive class of the *good(k, ℓ)* answers, using top-k algorithms to process *good(k, ℓ)* queries can result in a significant increase in the amount of work necessary in order to return the answers. With this in mind, we introduce a two-phase query processing approach that first identifies a candidate set of tuples and focuses the expensive extraction process to derive the scores of only this subset of the tuples. This, in turn, reduces the amount of work necessary in order to return the answers.

[3]If an absolute upper bound on the extraction scores exists, we could observe the highest scoring tuple after processing only a single document in the best case; but on an average, we would need to observe at least $\frac{|T|}{2}$ tuples.

Example 2.3: Consider again the score matrix from Figure 1 and the sum *function. Given a good(1, 0.1) query that requests for one tuple from the top 10 percent of the ranked tuples, a valid answer includes either tuple in $\{t_1, t_{12}\}$. When using top-k processing algorithms, we would focus on deriving answers for a top-1 query, which is t_{12}. For this, we ought to observe all scores for the 20 tuples. In contrast, when using a two-phase approach, we can pick (in expectation) a random sample of 10 tuples and focus only on exploring the complete scores for these 10 tuples, thus reducing the necessary processing.* □

The above example also underscores two important observations. First, the total number of tuples to extract in a *good(k, ℓ)* query execution can be bounded. For most databases, this translates to a bound on the number of documents to process as well. Second, the query model does not require any prior knowledge of the total number of tuples in the database, which may not be readily available. In Example 2.3, to process a *good(1, 0.1)* query, we need to focus on just 10 tuples irrespective of the total number of tuples in the database.

We now formally define the problem on which we focus:

Problem 2.1: Consider a text database D with a Boolean search interface and an extraction system E. Given a document d_i ($d_i \in D$) E extracts tuples from d_i and returns confidence scores for each tuple extracted from d_i. Given a good(k, ℓ) query over D, our goal is to efficiently process the query and derive answers that satisfy the user query with high probability.

Given a *good(k, ℓ)* query, finding exactly which tuples are in the top-ℓ fraction of all tuples would require processing every document in D. Faster solutions for working with quantiles [12] do approximations. We follow a similar approach and not insist on deriving k answers from the top-ℓ fraction in a deterministic manner. In the remainder of the paper, we will present algorithms for processing *good(k, ℓ)* queries based on the access models described in this section.

III. Processing good(k,l) Queries: An Overview

To improve upon an exhaustive approach that processes each database document, we note two important properties of a desirable query execution approach. First, we need to identify a *candidate set of tuples* such that it contains at least k answers. Second, we need to identify the total number of non-zero elements in a tuple's score vector as this will allow query processing algorithms to recognize tuples that do not need further processing as all possible scores for them have been explored.

For the first task of identifying a set C of candidate tuples, we introduce a *getCandidate(k, ℓ, δ)* procedure that extracts a set of tuples C from database D such that C contains at least k answers from the top-ℓ fraction with probability at least $(1 - \delta)$. Later, in Section V, we discuss the *getCandidate(k, ℓ, δ)* procedure in detail.

Proposition 3.1: Consider a database D and a good(k, ℓ) query q. Then the top-k tuples among those that are generated by procedure getCandidate(k, ℓ, δ) are valid good(k, ℓ) answers from q over D with probability at least $1 - \delta$. □

For the second task of determining the number of non-zero elements in the score vector for a candidate tuple, we resort to *Q-access*. Specifically, for each tuple t we construct an appropriate query q and fetch the number $|H(t)|$ of documents that match q using *Q-access*. We use $|H(t)|$ to derive an upper bound on the number of non-zero elements, as database documents that did not match q will not generate t. We summarize our two-phase approach for processing a *good(k, ℓ)* query over D below:

(1) Retrieve a candidate set C of tuples from D using *getCandidate(k, ℓ, δ)* such that C contains *good(k, ℓ)* answers with probability $(1 - \delta)$.
(2) Initialize a map $M = \emptyset$.
(3) For each tuple $t \in C$:
    a) Generate a search query q from t and using *Q-access* retrieve matching documents $H(t)$ for q.
    b) Use $|H(t)|$ as the upper bound on the number of non-zero elements in the score vector for t and update M.
(4) Identify the top-k tuples in C according to M, using some processing algorithm (see below).

To this end, yet another simple solution is possible: for each tuple t in C, we retrieve the matching documents $H(t)$ using *Q-access* and process them using E. Because $H(t)$ is a superset of all the documents that can generate t, we can explore all possible scores for t and hence compute its aggregate score $s(t)$. After deriving the aggregate scores for each candidate tuple, we can sort them and return the k highest ranking tuples as the answer. However, we may not necessarily need to process every document that matches every tuple in the candidate set. With this in mind, we discuss two possible algorithms for the task of identifying k answers from a candidate set.

To compare the query processing algorithms, we define the *execution cost* of an algorithm as:

Definition 3.1: **[Execution Cost]** *Consider a database D. The* execution cost *of an algorithm for a good(k, ℓ) query over D is the fraction of database documents that are processed by the algorithm, $\frac{|documents\ processed|}{|D|}$.* □

The execution cost of a query processing algorithm relies on several factors such as, the time to process a document, the time to retrieve a document, or the length of a document. To keep the necessary statistics manageable, we make a simplifying assumption that these factors are constant across different documents. As our query processing framework involves a single extraction system, the number of documents processed serves as a good enough surrogate for the execution cost of the query processing algorithms.

IV. Identifying Answers from Candidate Tuples

In this section, we discuss two algorithms, namely, *E-Upper* (Section IV-A) and *Slice* (Section IV-B) which focus on identifying k answers for a *good(k, ℓ)* query from a candidate set of tuples for the query.

A. E-Upper Algorithm

E-Upper is a deterministic algorithm in that it returns the top-k tuples in a candidate set of tuples. Thus, given a *good(k,*

Algorithm 1: **E-Upper**(C, k, M)

```
Output: top-$k$ tuples in $C$
$A = \emptyset$
while $|A| < k$ do
    Retrieve $t_u$ from $C$ with $U(t_u) = \max_{t \in C} U(t)$
    if all the matching documents for $t_u$ in $M$ have been processed then
        /* $t_u$ belongs to top-$k$ answers */
        $A = A \cup \{t_u\}$
    end
    else
        /* $t_u$ needs to be processed to decrease $U(t_u)$ */
        Retrieve and process a matching document $d$ for $t$ from $M$
    end
end
return $A$
```

Fig. 2. A deterministic algorithm for identifying the top-k answers from a candidate set of tuples.

ℓ) query, if a candidate set contains k answers from the top-ℓ fraction, *E-Upper* returns perfect answers.

E-Upper is based on the principles of Upper [4], a top-k query processing algorithm for queries over Web-accessible sources. Upper assumes that there exists at least one source that produces objects *sorted* according to one of the attributes, and that each source can be "probed" to identify the object score of the corresponding attribute. ("Sources" are associated with attributes in the Upper model.) Upper is based on the observation that tuples that are certainly not in the top-k answer need not be evaluated any further. To determine when it is safe to discard a tuple, Upper maintains two possible scores for each tuple: a score upper bound and a score lower bound. At any given point in time, Upper can discard a tuple if the score upper bound of the tuple is strictly below the score lower bound of k other tuples. Based on this property, Upper focuses on efficiently identifying and discarding such unnecessary tuples. Interestingly, once a tuple has been discovered, Upper is able to process a top-k query via "probing" to complete the score of the tuple. This observation is particularly important for our query processing setting. Specifically, once a candidate set of tuples has been identified, we can adapt Upper to identify the *good(k, ℓ)* answers from the set based on the information available via *Q-access*.

Consider a tuple t in a candidate set C with a set $H(t)$ of matching documents retrieved using *Q-access*. Assume that we have processed i documents in $H(t)$. Thus, the score vector $\mathbf{sv(t)}$ contains actual scores for i elements, a value of 0 for $|D| - |H(t)|$ elements, and unknown values for $|H(t)| - i$ elements. At any given time during query processing, we can define the two scores associated with t as follows: (a) *U(t)*, highest possible score for t, computed by assuming that all the $|H(t)| - i$ unseen scores are assigned the maximum value of 1, and (b) *L(t)*, lowest possible score for t, computed by assuming that all the $|H(t)| - i$ unseen scores are assigned the minimum value of 0.

Figure 2 shows the *E-Upper* algorithm. At any given iteration, *E-Upper* picks for exploration a tuple t_u with the highest score upper bound: if all the documents processed with t_u has been processed, t_u is an answer tuple; otherwise, we must lower its score before we can return the top-k results, to make sure that t is not in the answer. *E-Upper* progresses until it reaches a stage where there is no non-answer tuple t for

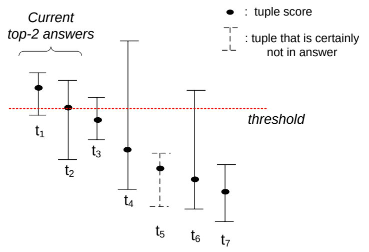

Fig. 3. Snapshot of an *E-Upper* execution, identifying the top-2 tuples from a candidate set of 7 tuples.

which its score upper bound, $U(t)$, is higher than the actual score of the k^{th} answer tuple. Figure 3 shows a snapshot of the *E-Upper* algorithm in progress. (For illustration purposes, we used a figure similar to that in [4].) The figure shows the scores for 7 tuples ordered by their actual scores for the task of identifying top-2 answers. A tuple whose upper bound is lower than the lower bound of at least 2 tuples can certainly not belong to the final answer. We discard such tuples at each iteration. In order to safely determine the top-2 answers, *E-Upper* will pick t_4 as the tuple to explore next.

By discarding tuples that do not belong to the final answer, *E-Upper* improves upon the *probe-all* strategy which processes all documents associated with every candidate tuple. However, *E-Upper* can be time-consuming under some scenarios where we need to process a relatively large number of documents to lower the upperbound of a tuple that does not belong to the answer. This observation has been made in the top-k processing case as well, leading to developing probabilistic algorithms [20]. Unfortunately, existing probabilistic algorithms typically require some apriori knowledge about the score distribution of the tuples. In our case, where we are exploring the potential of a new database, such information is not available. What we need is a method to make decisions as we observe scores at query processing time. With this in mind, we design a probabilistic algorithm, that determines early on two types of tuples, i.e., tuples that cannot belong to the final answer as well as tuples that very likely belongs to the final answer, based on the relative standing of the candidate tuples.

B. Slice Algorithm

In this section, we introduce a probabilistic algorithm, *Slice*(C, a, ϵ), that returns a answers from the set of candidate tuples C such that (in expectation) at least $a - \epsilon$ are contained in the top-a tuples in C. Using *Slice* we can provide a set R of approximate answers for a *good(k, ℓ)* query such that R contains at least k answers and no more than E extraneous tuples, where E is a user-specified tolerance threshold. To ensure that we obtain k answers with high probability (see Section II), we call *Slice* over a candidate set C of tuples that contains at least $k + E$ tuples from the top-ℓ fraction, and use $a = k + E$ and $\epsilon = E$. Among the answers returned by *Slice*, we expect to observe at least k answers picked from the top-$(k + E)$ tuples in C, which, in turn, belong to the top-ℓ fraction of the database.

Slice is a recursive algorithm and the main idea behind it is as follows: at each level of recursion the algorithm *slices*

```
Algorithm 2: Slice($C, a, \epsilon$)
Input: Candidate set of tuples $C$ that contains at least $a$ tuples, desired number
       of answers $a$, acceptable error $\epsilon$
Output: Tuple set $R$, such that (i) $|R| \leq a$ and (ii) $R$ has at least $a - \epsilon$ tuples
        that are among the correct top-$a$ tuples
/* base cases */
if $a \leq \epsilon$ then
    return $\emptyset$
if $|C| = a$ then
    return $C$
/* end base cases */
if $\epsilon = 0$ then
    Process all remaining documents for each tuple in $C$
    return top-$a$ tuples in $C$
foreach tuple $t \in C$ do
    Process $\alpha$ fraction of matching documents for $t$ in $M$
Rank tuples in $C$ based on observed scores so far
$\tilde{\epsilon}$ = $getNextError(\epsilon)$
Pick $\tau_g$ and $\tau_b$ using $\tilde{\epsilon}$
Get set $X$ of tuples in $C$ with rank at or above $\tau_g$
Get set $Y$ of tuples in $C$ with rank at or above $\tau_b$
return $X \cup$ Slice($C - (X \cup Y), a - |X|, \epsilon - \tilde{\epsilon}$)
```

Fig. 4. A recursive algorithm for deriving top-k approximate answers from a candidate set of tuples.

the candidate set of tuples by putting aside a set X of tuples expected to belong to the top-a tuples in C, and a set Y of tuples expected to *not* belong to the top-a tuples in C. *Slice* generates these sets after processing only *a fraction of the documents* associated with each candidate tuple, and the tuples in X and Y need not be further processed by the subsequent recursion levels, thus reducing the overall execution cost. The final answer is a union of the tuples in X set aside by each recursion step.

Figure 4 shows the *Slice* algorithm. To understand the algorithm, consider a tuple t. At any recursion level r, we process a new fraction α of documents from $H(t)$. Using the (collective) scores observed at all levels up to r, we compute the aggregate score for t. Based on these observed aggregate scores for the candidate tuples, we generate a ranked list. To derive the set X we decide a rank boundary, τ_g, such that all the candidate tuples with rank at or less than τ_g belong to X (rank position 1 represents the highest-scoring tuple). Similarly, we derive the set Y by deciding a rank position τ_b such that all the candidate tuples with rank positions at or greater than τ_b belong to Y.

Example 4.1: To illustrate how Slice works, we discuss a hypothetical scenario involving the sample score matrix in Figure 1. Assume that we want to retrieve 2 tuples from the top-4 tuples in a candidate set (i.e, k = 2 and $E = 2$, so $a = 4$). If sum *is the aggregate function, the actual answer for this query consists of any 2 tuples from the set $\{t_1, t_2, t_7, t_{12}\}$. Consider a candidate set that contains 8 tuples, namely, t_1, t_2, t_7, t_{10}, t_{12}, t_{14}, t_{16}, and t_{20} from Figure 1. Slice begins by processing a fraction of the documents associated with each candidate tuple. Say Slice processes documents $d_{11}, d_6, d_6, d_3, d_2, d_3, d_{11}$, and d_6 for the above tuples, which results in the observed scores shown in Figure 5. For instance, for tuple t_1, we observe a score of 0.6 from document d_{11}, whereas for tuple t_{10} we observe a score of 0.15 from document d_3. Using these observed scores, we construct a ranked list (see Figure 5): tuple t_{12} is at rank 1 and tuple t_{16} is at rank 2; similarly, t_{20} is assigned a rank of 8. These observed ranks are not identical to the actual ranks*

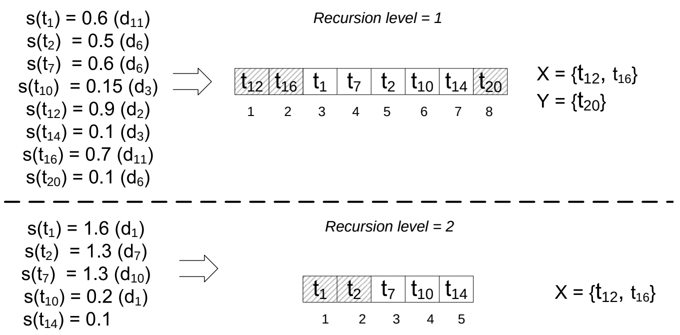

Fig. 5. Sample recursion snapshots for *Slice*.

of these candidate tuples (e.g., the actual aggregate score for t_1 is higher than that for t_{16}), but some of the tuples, such as t_{12} and t_{20}, are already in the right position.

At the first recursion level, using an appropriate "oracle," Slice picks $\tau_g = 2$ and $\tau_b = 8$. As a result, it generates the sets $X = \{t_{12}, t_{16}\}$ and $Y = \{t_{20}\}$. In the next recursion, the candidate set is reduced by eliminating these three tuples, and the goal now is to derive the top-2 tuples from the new candidate set consisting of 5 tuples. Slice processes a new fraction of documents for these candidate tuples; say Slice processes document documents d_1, d_7, d_{10}, d_1 for the tuples t_1, t_2, t_7, t_{10}, respectively, and updates the scores for the candidate tuples as shown in the figure. Tuple t_1 is the top-ranking tuple now and t_2 is ranked at 2. At the second level of recursion, Slice picks $\tau_g = 2$. This choice of τ_g generates a new X set with tuples t_1 and t_2. At this point, we have no more answer tuples to fetch and the final answer contains the set, $\{t_1, t_2, t_{12}, t_{16}\}$, of which at least 2 tuples are correct. $\square$

At each iteration, *Slice* uses partial score information and thus, X may contain tuples that are not among the top-a tuples in C and, similarly, Y may contain tuples that are among the top-a tuples in C. We want to pick τ_g and τ_b such that these errors are bounded by some $\tilde{\epsilon} \leq \epsilon$, where ϵ is the user-specified error-tolerance. The remainder $\epsilon - \tilde{\epsilon}$ is passed on as the error "budget" for the subsequent recursion levels. To derive an appropriate value for $\tilde{\epsilon}$ given ϵ, we rely on a procedure $getNextError(\epsilon)$; we will discuss this procedure later in this section.

Given $\tilde{\epsilon}$, we pick τ_g such that the expected number of *false positives* (i.e., tuples with actual rank greater than a but observed rank less than τ_g) is expected to be no larger than $\tilde{\epsilon}$. Similarly, we pick τ_b such that the expected number of *false negatives* (i.e., tuples with actual rank less than a and observed rank greater than τ_b) is expected to be no larger than $\tilde{\epsilon}$. For simplicity, we use the same value for $\tilde{\epsilon}$ for both false positives and false negatives. However, the algorithm can be easily extended to handle different values for these error bounds. Later in this section, we discuss how we pick τ_g and τ_b for a given a and $\tilde{\epsilon}$.

To examine the correctness of our algorithm, we define the following theorem.

Theorem 4.1: Given a candidate set of tuples C that contains at least a tuples, desired number of tuples a, and acceptable error ϵ, $Slice(C, a, \epsilon)$ returns a set of tuples $R \subseteq C$

such that $|R| \leq a$ and $|R|$ contains at least $a - \epsilon$ tuples that are among the top-a tuples in C.

To prove this theorem, we define the following lemmas.

Lemma 4.1: Let $C' \subseteq C$ be a set of tuples such that C' contains β tuples from the top-α tuples in C ($\beta \leq \alpha$). Then, the top-β tuples in C' are contained in the top-α tuples in C.

Proof: We will prove this by contradiction. Assume that there exists a tuple t_1 in the top-β of C' such that t_1 is not in the top-α of C. Since C' contains β tuples from the top-α of C and we assume not all of them are in the top-β of C', then there exists a tuple t_2 such that t_2 is in the top-α of C but not in the top-β of C'. Now $t_1 \in$ top-β tuples in C' and $t_2 \notin$ top-β tuples in $C' \implies s(t_1) > s(t_2)$. But $t_2 \in$ top-α tuples in C and $t_1 \notin$ top-α tuples in $C \implies s(t_2) > s(t_1)$. Since $s(t_1)$ cannot be both greater and smaller than $s(t_2)$, we arrived at a contradiction. ■

Lemma 4.2: Let $C' \subseteq C$ be a set of tuples such that C' contains β tuples from the top-α tuples in C ($\beta \leq \alpha$) and let $R' \subseteq C'$ be a set of tuples such that R' contains at least δ tuples from the top-γ tuples in C' ($\delta \leq \gamma \leq \alpha$). Then, R' contains at least $\delta - max\,[(\gamma - \beta), 0]$ tuples from the top-α tuples in C.

Proof: By Lemma 4.1, {top-β of C'} $\subseteq$ {top-α of C}. We distinguish two cases:

Case 1: $\gamma > \beta$**:** Then, {top-β of C' } $\subset$ {top-γ of C' }. This means that at least β tuples from the top-γ of C' are contained in the top-α of C. So, at most $\gamma - \beta$ tuples from the top-γ of C' are not contained in the top-α of C. Therefore, out of the δ tuples from R' that are contained in the top-γ of C' at most $\gamma - \beta$ can not be contained in the top-α of C.

Case 2: $\gamma \leq \beta$**:** Then, {top-γ of C'} $\subseteq$ {top-β of C'} $\subseteq$ {top-α of C}. Hence, all δ tuples in R' are contained in the top-α of C. ■

We are now ready to prove Theorem 4.1.

Proof: By induction on the recursion height.

Base cases: (a) If $a \leq \epsilon$, *Slice* returns $R = \emptyset$, which trivially satisfies the condition that $|R| \leq a$ and R contains at least $a - \epsilon$ tuples from the top-a of C. (b) If $|C| = a$, *Slice* returns $R = C$, which trivially are the top-a tuples in C.

Induction step: *Slice* picks $\tilde{\epsilon}$, X, and Y (see Figure 4). Let X_b and Y_g denote the false positives in X and the false negatives in Y, respectively. Let R' be the set of tuples returned by the recursion call *Slice*$(C - X - Y, a - |X|, \epsilon - \tilde{\epsilon})$. From the induction hypothesis we know that $|R'| \leq a - |X|$, and, since $R = R' \cup X$, we can conclude that $|R| \leq a$. To calculate the number of tuples in R that belong to the top-a of C, we identify the following properties used in our proof:

(1) $|X|$ contains $|X| - |X_b|$ tuples from the top-a tuples in C.

(2) C' contains $(a - |X - X_b| - |Y_g|)$ tuples from the top-a tuples in C.

(3) By the induction hypothesis, R' contains at least $a - |X| - (\epsilon - \tilde{\epsilon})$ from the top-$(a - |X|)$ tuples in $C - X - Y$.

Applying Lemma 4.2 to C', where $C' = C - X - Y$, with $\alpha = a$, $\beta = a - (|X| - |X_b|) - |Y_g|$ (by property **(2)**), $\gamma = a - |X|$, and $\delta = a - |X| - (\epsilon - \tilde{\epsilon})$ (by property **(3)**), we get that the set R' contains at least $a - |X| - (\epsilon - \tilde{\epsilon}) - max[(a - |X|) - (a - (|X| - |X_b|) - |Y_g|), 0]$ tuples that belong to the top-a of C. We further solve this to derive the total number of tuples in R' that are in the top-a of C to be:

$$\begin{aligned}
&a - |X| - (\epsilon - \tilde{\epsilon}) - max[|Y_g| - |X_b|, 0] \\
&= a - (|X| - |X_b|) - (\epsilon - \tilde{\epsilon}) - max[|Y_g|, |X_b|] \\
&\geq a - (|X| - |X_b|) - (\epsilon - \tilde{\epsilon}) - \tilde{\epsilon} \qquad \text{(by } |Y_g|, |X_b| \leq \tilde{\epsilon}\text{)} \\
&= a - (|X| - |X_b|) - \epsilon
\end{aligned}$$

Furthermore, we know that $R = R' \cup X$. Using property **(1)**, we can conclude that R must contain at least $a - (|X| - |X_b|) - \epsilon + |X| - |X_b| = a - \epsilon$ tuples from top-a of C. ■

1) Deriving Rank-based Boundaries: We now discuss how we pick the rank-based boundaries τ_g and τ_b in order to generate the sets X and Y at each recursion.

Given a candidate set C and the goal of fetching top-a tuples in C, we observe that using a rank boundary τ_g generates a false positive when a tuple at a rank greater than than a is observed at a rank less than τ_g. To compute the expected number of false positives given τ_g and a, we study the probability of such *rank inversions* taking place. Formally, we are interested in deriving the probability $Pr_{inv}\{j, i\}$ that a tuple at rank j is observed at rank i where $j > i$ after processing $\alpha \cdot r$ fraction of documents, where r is the number of recursion levels so far.

A rank inversion between positions j and i would, in turn, require a series of consecutive *rank swaps* to take place, i.e., the tuple must "swap" ranks with all the tuples with ranks q, where $j > q > i$. Figure 6 illustrates the generation of a false positive for a set C of ranked tuples, with 1 being the top rank and $|C|$ the largest possible rank. For a given a, the figure illustrates one possible choice of τ_g which results in a single false positive at position 3. This false positive was generated when a tuple with actual rank = a+1 swapped ranks with *all* other tuples with smaller ranks until it arrived at the observed rank of 3. In practice, the relation between rank swaps and a rank inversion of a specific length can be arbitrarily complex depending on the nature of the text database, the tuple score distribution, etc. To avoid relying on any apriori knowledge or sophisticated statistics, we make a simplified assumption of independence across different rank swaps. Specifically, we can derive $Pr_{inv}\{j, i\}$ as:

$$Pr_{inv}\{j, i\} = \prod_{q=j}^{i} Pr_{inv}\{q + 1, q\} \qquad (3)$$

The probability of a single rank swap i.e., a tuple with actual rank j swaps ranks with a tuple with actual rank $(j+1)$, depends on the fraction $(\alpha \cdot r)$ documents processed for the candidate tuples. Specifically, we denote the probability of a single rank swap as a function $f_s(\beta)$ of the fraction of documents $\beta = (\alpha \cdot r)$ processed for the candidate tuples. Following this, the expected number of false positives when using τ_g as the boundary is:

$$E[\textit{false positives}|\beta, \tau_g] = \sum_{j=1}^{\tau_g} Pr_{inv}\{a{+}1, j\} = \sum_{j=1}^{\tau_g} f_s(\beta)^{(a+1-j)}$$

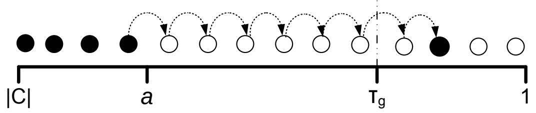

Fig. 6. A false positive generated by a series of consecutive rank swaps.

This is a conservative upperbound on the expected number of false positives as we compute the rank inversions between the ranks τ_g and $a+1$; tuples with ranks greater than $a+1$ are less likely to switch over as false positives than the tuple at rank $a+1$.

To compute $f_s(\beta)$ given the fraction β of documents processed, we estimate the probability of a rank swap for varying values of β at runtime. Specifically, we begin with a small sample S of tuples (e.g., 5 or 10 tuples) and examine the probability of rank swap by varying the values for β. Given an aggregate function, and a β value we process β fraction of the documents associated with the tuples in S. Using the aggregate function along with the scores observed so far, we derive a ranked list of the tuples in S. To derive $f_s(\beta)$, we now compute the total number of rank inversions observed in this ranked list and normalize it by the maximum number of inversions possible in a sequence of size $|S|$. As we will see later, this step of estimating $f_s(\beta)$ can be "piggy-backed" with the process of deriving other parameters necessary for query processing (Section V).

So far, we discussed how we derive the rank boundary τ_g for generating the set X. To derive the rank boundary τ_b, we proceed in a similar fashion after computing the expected number of false negatives for a given τ_b and a.

2) Choosing an Error Budget: We now discuss the issue of picking $\tilde{\epsilon}$, i.e., the maximum allowed false positives (or false negatives) at each recursion level. For this, we first discuss the general effect of the choice of $\tilde{\epsilon}$.

Given two different recursion levels r_1 and r_2 where $r_2 > r_1$ and a $\tilde{\epsilon}$ value, the set X_2 picked by *Slice* at recursion level r_2 is larger than the set X_1 picked at level r_1. This is mainly because at r_2, *Slice* would have observed and processed more documents than at r_1, thus moving closer to the actual ranking of the candidate tuples and reducing the chance of picking a false positive. This, in turn, allows *Slice* to pick a larger value for τ_g at r_2 than at r_1 for the same $\tilde{\epsilon}$ value, resulting in X_2 such that $|X_2| > |X_1|$. This observation gives us an incentive to increase the number of recursion levels by picking a small value for $\tilde{\epsilon}$ during the initial levels of recursion. However, deeper recursion levels come at the cost of processing more documents as *Slice* processes a new fraction of documents at each level.

On the other hand, given a recursion level r and two different values for the error budget $\tilde{\epsilon}_2$ and $\tilde{\epsilon}_1$, where $\tilde{\epsilon}_2 > \tilde{\epsilon}_1$, the set X_2 picked by *Slice* using $\tilde{\epsilon}_2$ is larger than the set X_1 picked by *Slice* using $\tilde{\epsilon}_1$. This is because using $\tilde{\epsilon}_2$ allows for more *slack* and enables a more aggressive approach at building X by picking a larger value for τ_g compared to that picked using $\tilde{\epsilon}_1$. While using larger values for epsilon earlier on can reduce the number of recursion levels (and thus the cost of the algorithm), we run into the risk of exhausting the error budget too soon and having to process all the remaining documents: when $\tilde{\epsilon} = 0$, *Slice* is not permitted any false positives or false negatives, and *Slice* must process *all* documents for each tuple in C.

We studied alternative definitions for the $getNextError$ and found the function $\tilde{\epsilon} = \frac{\epsilon}{2}$ to work well; halving the available error budget at each level ensures we use error budgets in proportion to the expected error.

V. Generating Candidate Sets

Our algorithms of Section IV rely on a *getCandidate(k, ℓ, δ)* procedure to generate a candidate set of tuples C that, with probability at least $1-\delta$, contains at least k tuples from the top-ℓ fraction of the database tuples. We now present two methods to derive such a candidate set, a Naive method (Section V-A) and an Iterative method (Section V-B).

A. The Naive Approach

Given the goal of constructing a candidate set that contains k tuples from the top-ℓ fraction of the tuples in the database, we begin with drawing a random sample of the tuples via *S-access*. Specifically, our goal is to process documents retrieved by *S-access* until we have extracted k tuples from the top-ℓ fraction. For this, we observe that the number of tuples in C that belong to the top-ℓ fraction of the tuples in the database is a random variable V_H that follows a hypergeometric distribution. Thus,

$$Pr\{V_H < k\} = \sum_{j=0}^{k-1} Hyper(T, T \cdot \ell, |C|, j) \qquad (4)$$

where $Hyper(D, S, g, i) = \frac{\binom{g}{i}\cdot\binom{D-g}{S-i}}{\binom{D}{S}}$. For a desired confidence $(1-\delta)$ in the candidate set, we can draw samples C such that the probability that C contains at least k answers exceeds $1-\delta$. Hence, we are looking for:

$$min\{|C| : (1 - Pr\{V_H < k\}) \geq (1-\delta)\} \qquad (5)$$

In practice, though, deriving the cumulative distribution function of a hypergeometric distribution does not yield a closed-form solution. To optimize the process of selecting a candidate set we can model the sampling process using a binomial distribution. Besides simplifying the candidate set size derivation, using a binomial model provides an added benefit of not requiring the knowledge of the exact number of tuples in the database. This is particularly appealing in a data exploration setting where the total number of tuples is not known a priori. Under the binomial model, the number of tuples in C that belong to the top-ℓ fraction of the tuples that can be extracted from the database is a random variable V_B with probability of success $p = \ell$ such that:

$$Pr\{V_B < k\} = \sum_{j=0}^{k-1} \binom{|C|}{j} \cdot p^j (1-p)^{|C|-j} \qquad (6)$$

Equation 5 gives us the size of candidate set to draw. Finally, for $p \geq 0.5$ we can further use Chernoff bounds [18] to derive an upper bound on $|C|$.

The approach outlined above assumes no skew in the data and that a set of tuples derived using *S-access* is an unbiased random sample of the tuples in the database. In particular, it assumes that (a) each tuple occurs only once or the frequency of the tuples is uniform, and (b) the choice of the aggregate function does not affect the likelihood of observing a tuple that belongs to the top-ℓ fraction of the tuples that can be extracted from the database. Next, we present another approach for constructing the candidate set that relaxes these assumptions.

B. The Iterative Approach

In some cases, we may have a skewed database such that some tuples occur more frequently than the others. In fact, [15] showed that the extracted tuples in a database follow a long tail distribution, i.e., a power-law distribution. In this setting, it becomes important to examine the effect of the choice of the aggregate function on the rank of a frequently occurring tuple or that of a rarely occurring tuple. Specifically, we want to examine for a given function the relation between the frequency of a tuple and the fraction of the ranked tuples it belongs to.

Consider the case of summation. Informally, a tuple that occurs very frequently is more likely to have a high score and thus occur towards the top of the ranked list of tuples (i.e., small values of ℓ) than a tuple that occurs only once. Furthermore, a frequently occurring tuple is more likely to be observed in a random sample than a rarely occurring tuple. As a consequence, for small values of ℓ, the samples drawn using *S-access* may contain more tuples than necessary to derive a *good(k, ℓ)* answer. This, in turn, implies that we can *down sample* when selecting the candidate set for smaller values of ℓ when using summation as the aggregation function. As a contrasting example, consider the case where the scoring function is *min*, i.e., the final score of a tuple is the minimum score assigned to it across all documents. In this case, a rarely occurring tuple is more likely to belong towards the top of the ranked list of tuples, and this would require us to *over sample* when constructing the candidate set for small values of ℓ.

To account for the scoring function effect, we developed a two-step candidate generation approach. Given the goal of constructing a candidate set containing k tuples from the top-ℓ fraction of the tuples in the database, we pick a small value s ($s \ll k$) and construct an initial candidate set that contains s tuples that belong to the top-ℓ fraction of the tuples in the database. We generate this initial candidate set using the Naive method outlined in Section V-A. Using *Q-access*, we derive the matching documents for each tuple in this set, and process them to derive for each candidate tuple the actual aggregate score. To this end, we compute an *adjust factor* $a(\ell)$. Specifically, we calculate the actual number of tuples in the initial candidate set that belong to the top-ℓ fraction of all the tuples and divide this number by s. When using a function like *sum* that calls for down sampling, $a(\ell) \geq 1$. We apply this adjust factor when constructing a candidate set for the remainder $k - s$ tuples to be fetched. Specifically, we now target for $\frac{k-s}{a(\ell)}$ tuples instead of $(k - s)$ tuples to generate the remainder of the candidate set.

The two-step approach discussed above can naturally be extended to a fully iterative approach where we refine our

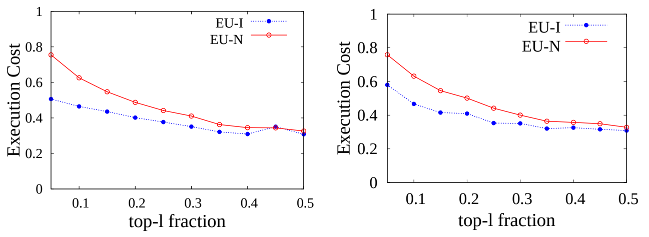

Fig. 7. Average execution cost using *E-Upper* for varying ℓ using (a) *sum* and (b) *incl-excl*.

estimate for $a(\ell)$ iteratively. Using the adjust factor obtained at iteration i from s_i, we can decide an appropriate scaling factor when fetching the s_{i+1} tuples in the $i + 1^{st}$ iteration. Interestingly, our experiments reveal that fixing the number of iterations to two (i.e., $n = 2$) results in candidate sets with sizes close to those we can obtain if we had perfect knowledge of the scoring function effect.

VI. Experimental Evaluation

We now describe the settings for our experiments and report the experimental results.

Information Extraction Systems: We trained Snowball [2] for two relations: *Headquarters*$\langle$*Company, Location*$\rangle$, and *Executives*$\langle$*Company, CEO* $\rangle$

Data Set: We used a collection of newspaper articles from The New York Times from 1995 (NYT95) and 1996 (NYT96), and a collection of newspaper articles from The Wall Street Journal (WSJ). The NYT96 database contains 135,438 documents and we used it to train the extraction systems. To evaluate our experiments, we used a subset of 49,527 documents from NYT96, 50,269 documents from the NYT95, and 98,732 documents from the WSJ.

Queries: To generate *good(k, ℓ)* queries, we varied ℓ from 0.05 to 0.5, in steps of 0.05, and used k ranging from 20 to 200, in steps of 20. For each *good(k, ℓ)* query, we report values averaged across 5 runs.

Metrics: To evaluate a processing algorithm, we measure the *precision* and the *recall* of the answers generated by the algorithm for a given *good(k, ℓ)* query. Given a *good(k, ℓ)* query, if G is the actual set of tuples in the actual top-ℓ fraction, and R is a set of answers, we define:

$$Precision = \frac{|G \cap R|}{|R|}; \quad Recall = min(\frac{|G \cap R|}{k}, 1.0) \qquad (7)$$

In addition to the precision and recall, we also derive the execution cost of deriving an answer using Definition 3.1.

Combining query processing algorithms and candidate set generation: To evaluate our query processing algorithms, we considered two possible settings for each algorithm, depending on the choice of candidate set generation method, namely, Naive or Iterative (Section V). We denote the combination of *E-Upper* with Naive as *EU-N*, and that with Iterative as *EU-I*. Similarly, we denote the combination of *Slice* with Naive as *SL-N*, and that with Iterative as *SL-I*.

E-Upper: Our first experiment evaluates the *E-Upper* algorithm from Section IV-A. For both settings of the algorithm,

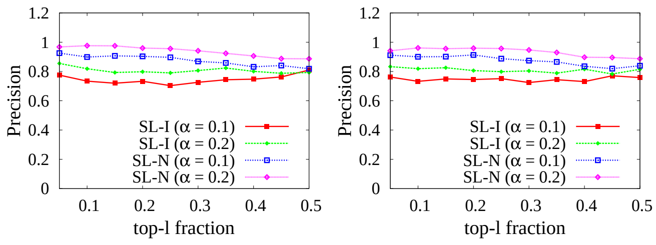

Fig. 8. Average precision using *Slice* for varying ℓ using (a) *sum* and (b) *incl-excl*.

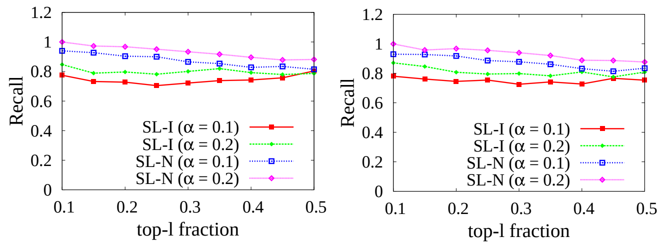

Fig. 9. Average recall using *Slice* for varying ℓ using (a) *sum* and (b) *incl-excl*.

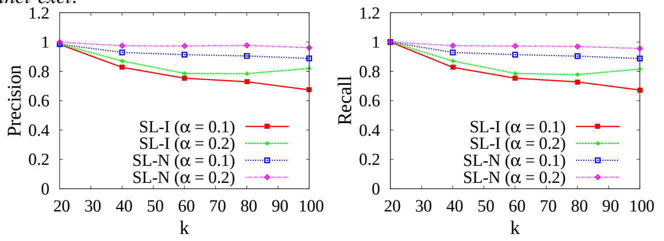

Fig. 10. Precision (a) and Recall (b) using *Slice* when $\ell = 0.25$ for varying k using (a) *sum* and (b) *incl-excl*.

namely, *EU-N* and *EU-I*, *E-Upper* correctly returns the top-k tuples in the candidate set. (We will examine the number of *good(k, ℓ)* answers that a candidate set of tuples contains later in this section.) We examined the execution cost for these settings. Figure 7(a) shows the execution cost for *EU-N* and *EU-I*, for different values of ℓ, when using *sum*; the execution cost is an average across a set of k values, ranging from 20 to 200. Figure 7(b) shows the average execution cost derived for *EU-N* and *EU-I* when using *incl-excl* (see Section II), for different ℓ values. The execution cost for both *EU-N* and *EU-I* tends to be high for lower values of ℓ, when the candidate set of tuples is large. Larger candidate sets, involve more database documents to be processed before we can generate the final answer, hence the higher execution costs. Figures 7(a) and 7(b) also show a promising direction towards reducing the execution cost by using the Iterative method for generating the candidate set; as seen in the figure, the execution cost for *EU-N* is higher than that for *EU-I* for all values for ℓ.

Slice: Our second experiment evaluates the *Slice* algorithm from Section IV-B. For both settings of the algorithm, *SL-N* and *SL-I*, we considered two different values for α (recall, that this is the fraction of matching documents processed by *Slice* at each recursion level). Finally, we set E (i.e., which is the user-provided acceptable error budget) to be 5% of the value of k. To ensure that the results contain k answers, we called *Slice* with candidate sets that contain $k + E$ tuples from the top-ℓ fraction of all the tuples (Section IV-B).

Figure 8(a) shows the average precision for *SL-N* and *SL-I*, for $\alpha = 0.1$ and $\alpha = 0.2$ and different values of ℓ, when using the *sum* aggregate function. The precision value is an average across a set of k values, ranging from 20 to 200. In general, the precision of the answers generated by any setting for *Slice* is at or above 0.75, with lower precision values for *SL-I* and close to perfect precision for *SL-N*. Using the Iterative method of candidate generation reduces the overall precision, as the candidate set is not as "rich" in answer tuples as for the Naive method. As discussed in Section IV-B.2, the value of α directly affects precision: the higher the value for α, the closer the observed ranking of candidate tuples to the actual ranking, and thus the higher the precision values. Figure 8(b) shows the average precision value for different values of ℓ when using *incl-excl* as the aggregate function. Precision follows a trend similar to that for *sum*.

Figure 9(a) shows the average recall for *SL-N* and *SL-I*, for $\alpha = 0.1$ and $\alpha = 0.2$, for different values of ℓ, when using *sum*. The recall value is an average across a set of k values, ranging from 20 to 200. In general, the observed recall ranges from 0.7 to 1.0, and just as for precision, we observe higher recall values for *SL-N* than that for *SL-I*. Furthermore, α affects recall in the same manner as for precision, with higher values of α improving recall. Figure 9(b) shows the average recall for different values of ℓ when using *incl-excl*, and the observations are similar to those for *sum*.

In our experiments, we observed that, in general, the precision and recall values decrease when increasing k and ℓ. Interestingly, as we increase k precision and recall drop faster than when we increase ℓ. To illustrate this observation, we examined the precision and recall of the *Slice* answers when ℓ is fixed at 0.25 and k is varied from 20 to 100. Figures 10(a) and 10(b) show the resulting precision and recall, respectively. As shown in the figures, the performance of *Slice* for all settings is ideal for $k = 20$ and deteriorates as k is increased. We also examined the precision and recall when k is fixed at 25 and ℓ is varied from 0.1 to 0.5. Figures 11(a) and 11(b) show the resulting precision and recall, respectively. As shown in the figures, the performance of *Slice* is relatively constant across different values of ℓ, with only a small degradation for higher ℓ values. This means that *Slice* presents a competitive choice, in terms of performance, for low values of k. We observed similar trends for *incl-excl*, which we do not discuss further for brevity.

We also studied the execution cost of *Slice*. Figure 12(a) shows the average execution cost for *SL-N* and *SL-I*, using two values for α, namely, $\alpha = 0.1$ and $\alpha = 0.2$, and for different values of ℓ; the execution cost is an average across a set of k values, ranging from 20 to 200. The average execution cost for the worst case (i.e., *SL-N* and using $\alpha = 0.2$) ranges between 0.45 to 0.12; in contrast, the execution cost for *EU-N* ranged from 0.75 to 0.32, and that for *EU-I* ranged from 0.5 to 0.3. This shows that *Slice* can result in at least a two-times reduction in execution cost compared to *EU-N*. The execution cost for *Slice* is strictly lower than that for *EU-I*, with a two-

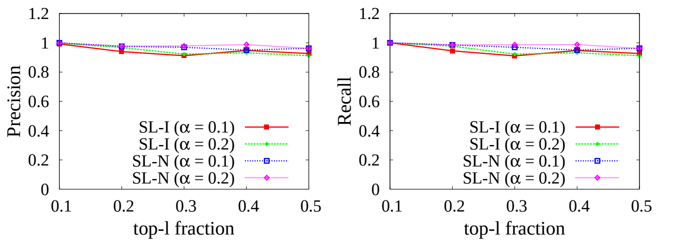

Fig. 11. Precision (a) and Recall (b) using *Slice* when $k = 25$ for varying ℓ using (a) *sum* and (b) *incl-excl*.

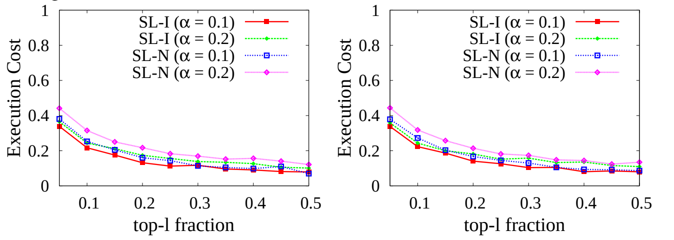

Fig. 12. Average execution cost using *Slice* for varying ℓ using (a) *sum* and (b) *incl-excl*.

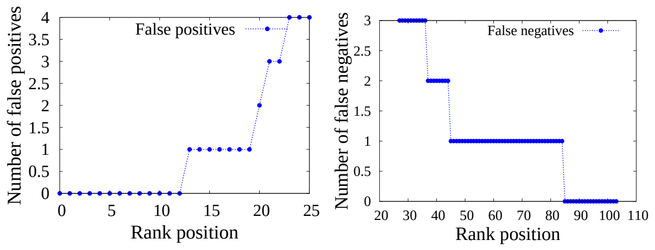

Fig. 13. The number of false positives (a) and the number of false negatives (b) at varying rank positions for $k = 25$, $E = 1$, and $\ell = 0.25$.

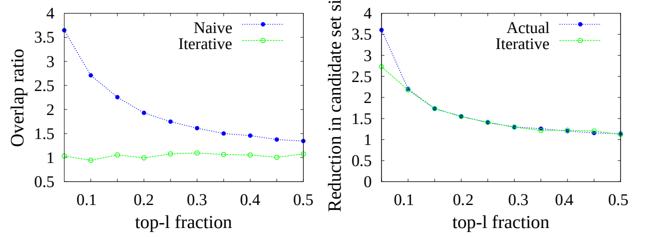

Fig. 14. Average (a) overlap ratio and (b) reduction in the candidate set sizes for varying ℓ.

times reduction in execution cost for higher ℓ values. These observations, when combined with our observations on the precision and recall of *Slice*, underscore an important point: using the most expensive setting of *Slice* results in a precision and recall value close to 0.9, but results in a significant speed up over any variation of the *E-Upper* algorithm. In fact, for $k < 100$, the precision and recall of *Slice* are similar to that for *E-Upper*.

An important factor that influences the accuracy of the *Slice* algorithm is the number of false positives and the number of false negatives that we observe at different rank positions. We examined the trend that the number of false positives and false negatives follow at different rank positions starting from the target rank position of k. In Section IV-B.1, we made a simplified assumption that this trend follows (an exponential trend) as defined by Equation 3. For a first level of recursion, Figure 13 shows these trends at varying rank positions for $k = 25$, $E = 1$ and $\ell = 0.25$ (with a candidate set contains 104, 26/0.25, tuples). Figure 13(a) shows the number of false positives, as we travel away from the target position of 25, and Figure 13(b) shows the number of false negatives at different rank positions. The figures reveal an important observation: they confirm our intuition underlying the *Slice* algorithm that the number of false positives and false negatives diminish as we travel farther from k. This observation encourages the idea of *slicing off* the tuples that lie at the extremities when the candidate tuples are ranked by their scores. For our data sets and relations, the number of false positives and false negatives follow a linear trend. Accounting for the exact trend that the number of false positives and false negatives follow would obviously require more sophisticated statistics, which may not be readily available. The above observations also provides insight into why the precision and recall for *Slice* is high for low k values: for low values of k, fewer tuples in the candidate set can swap around and thus reduce the number of false positives and false negatives.

Candidate Set Generation: Finally, we examine our candidate set generation methods, namely, Naive (Section V-A) and Iterative (Section V-B). First, we study the number of actual answers in the candidate sets generated using these methods. Specifically, for a given *good(k, ℓ)* query, we generate the candidate set using both methods and compute the overlap between these candidate sets and the tuples that actually belong to the top-ℓ fraction of the tuples in the database. We calculate the *overlap ratio* by normalizing this overlap by k, i.e., a value of 1 indicates that the candidate set contains k answer tuples and a value lower than 1 indicates fewer than k tuples in the candidate set. Figure 14(a) shows the average overlap ratio for both methods for varying ℓ values for the *sum* function. The overlap ratio for the Naive method is equal to or higher than 1 for all ℓ values, thus indicating that this method fetches more than required candidate tuples. In contrast, when using the Iterative method the overlap ratio falls slightly below 1 ($\approx$ 0.98) for some values of ℓ.

To examine the benefit of subsampling using the Iterative method, we compute the average cardinality of the set of candidate tuples, for different ℓ values. Figure 14(b) shows the relative reduction in the candidate set cardinalities, computed as the cardinality of the candidate set derived using the Naive method divided by the cardinality of the candidate set derived using the Iterative method. For $\ell = 0.05$, the Iterative method on average, reduces the candidate set size by more than half. This, in turn, reduces the overall execution cost of the algorithms, as we have discussed above. For reference, Figure 14(b) also shows the "actual" reduction in the candidate set cardinality computed using the actual knowledge the effect of the scoring function and the database skew (see Section V-B). Specifically, we assume that we know the exact value for the adjust factor $a(\ell)$ for an aggregate function and use that to identify that number of tuples to fetch for the candidate set. As shown in the figure, our Iterative method with the number of iterations fixed to 2 is close to the reduction using actual value for $a(\ell)$, except for when $\ell = 0.05$, and shows that using two iterations, which is more efficient than multiple iterations, tends to work well in practice.

Evaluation summary: Overall, we demonstrated the effectiveness of our query processing approach at deriving answers that meet specified *good(k, ℓ)* query constraints. We evaluated the performance of our candidate set generation methods, of which the Iterative method effectively allows us to identify the right-sized candidate set and save execution costs.

VII. Related Work

Information extraction from text has received significant attention in the recent years (see [7], [8] for tutorials). A majority of the efforts have considered the problem of improving the accuracy or the efficiency of the extraction process [7]. Some research efforts have also focused on building optimizers that allow users to provide requirements in terms of the desired recall [15], or some balance between the output quality and the execution time [16]. In general, these methods use a "0/1" approach where a tuple is either correct or not and ignore any important indicators from the underlying extraction system regarding the quality of the extracted tuple. Furthermore, these methods rely on some prior knowledge of the database, either in terms of the tuple frequency distribution or some database-specific statistics. In contrast, our work exploits the confidence information imparted by an extraction system allowing for a novel data exploration scenario not studied before. For this database exploration problem, naturally we cannot assume any prior information about the database.

There is a lot of work on deriving the confidence score of tuples extracted from the database [2], [9], [10], [19]. We believe that these methods are complementary to our general task of data exploration: just as in the case of top-k processing where a user may specify the aggregate functions, these scoring methods can also be incorporated as aggregate functions in our query processing framework.

Related effort to this paper is [1], which presents an approach to examine the quality of a relation that could be generated using an extraction system over a text database. Specifically, [1] builds language models for a text database and compares them against those for an extraction system to examine the relation quality. Our proposed algorithms are comparatively lightweight in that we eliminate the need for any such (potentially expensive) text analysis or the need for any apriori database- or extraction-related knowledge.

Our work is also related to the existing top-k processing methods [4], [6], [11]. In general, existing top-k processing algorithms following TA [11] (see also [13], [17]) assume a sorted access for at least one of the attributes: under the sorted access, the tuples in the database can be sequentially retrieved in nonincreasing order of their attribute values until we can safely establish the k-best ranking answers. As discussed in Section II-C, the data access model available in our setting does not allow for a sorted access. Some top-k processing algorithms such as Upper [4] support a combination of access methods, sorted as well as probed access. In this paper, we adapted the generic Upper algorithm to our setting as discussed in Section IV-A, and we established the feasibility of our adaptation at processing *good(k, ℓ)* queries, as discussed in Section VI. For processing top-k algorithms, a variety of probabilistic algorithms have also been explored [20], which exploit some a priori knowledge on the score distribution.

VIII. Conclusion and Discussion

In this paper, we introduced *good(k, ℓ)* query model, a novel query paradigm to address an important problem of exploring a text database for the task of extracting relations. Our query model works hand-in-hand with an extraction system and allows users to identify a few good tuples as determined by the collective confidence of an extraction system in a tuple. The key challenge in processing *good(k, ℓ)* queries, is that no apriori knowledge about the database characteristics or the score distribution is available in a data exploration setting. To process a *good(k, ℓ)* query, we adapted an existing algorithm for processing top-k queries, and introduced a new probabilistic algorithm. We proved the correctness of our probabilistic algorithm and empirically established the effectiveness of our algorithms. Our novel *good(k, ℓ)* query model is a potentially cheaper alternative to the more conventional top-k model in other application scenarios where top-k is currently used. We have established the foundations of this area, and exploring this line of research for other access models and cost models is an interesting direction of future work.

References

[1] E. Agichtein and S. Cucerzan. Predicting accuracy of extracting information from unstructured text collections. In *CIKM*, 2005.

[2] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *DL*, 2000.

[3] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB*, pages 172–183, 1998.

[4] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[5] M. E. Califf and R. J. Mooney. Relational learning of pattern-match rules for information extraction. In *IAAI*, 1999.

[6] S. Chaudhuri and L. Gravano. Evaluating top-*k* selection queries. In *VLDB*, 1999.

[7] W. Cohen and A. McCallum. Information extraction from the World Wide Web (tutorial). In *KDD*, 2003.

[8] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction (tutorial). In *SIGMOD*, 2003.

[9] D. Downey, O. Etzioni, and S. Soderland. A probabilistic model of redundancy in information extraction. In *IJCAI*, 2005.

[10] O. Etzioni, M. J. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll (preliminary results). In *WWW*, 2004.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[12] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, 2001.

[13] U. Güntzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.

[14] R. Gupta and S. Sarawagi. Curating probabilistic databases from information extraction models. In *VLDB*, 2006.

[15] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. Towards a query optimizer for text-centric tasks. *ACM Transactions on Database Systems*, 32(4), Dec. 2007.

[16] A. Jain, A. Doan, and L. Gravano. Optimizing SQL queries over text databases. In *ICDE*, 2008.

[17] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.

[18] S. M. Ross. *Introduction to Probability Models*. Academic Press, 8th edition, Dec. 2002.

[19] S. Sarawagi and W. Cohen. Semimarkov conditional random fields for information extraction. In *ICML*, 2004.

[20] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, 2004.