Exploring a Few Good Tuples From a Text Database

Alpa Jain
Columbia University
alpa@cs.columbia.edu

Divesh Srivastava
AT&T Labs-Research
divesh@research.att.com

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.

1. 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].

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

VLDB '08, August 24-30, 2008, Auckland, New Zealand

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 [13, 16, 17]. 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* $1-\frac{x}{max}$ *($x \leq$* 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 ($x = 3$).* □

This notion of a score per tuple provides an opportunity to explore 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 nature of tuples that are among the high ranking tuples can help decide whether processing the entire corpus is a desirable option.

An ideal solution to the problem of identifying a few good tuples is to support top-k queries over the relations buried in the text database. Top-k processing focuses on identifying the k best ranked objects (e.g., tuples, images, documents, etc.) based on the values of (one or more) attributes of an object. By representing the confidence score of each extracted tuple as an additional tuple attribute, we can pick from a broad family of top-k algorithms to explore a few good tuples in the

database. However, an important restriction in the case of extraction systems is that to access the tuples, we need to sequentially retrieve documents from the database, processing one document at a time, in no specific order. Using such an "unsorted *scan* access," identifying any top-k answer would require near-complete processing of the corpus, even if a tuple can be extracted only from a single document.

This observation lead us to investigate a query model that relaxes the "best" k constraint, yet maintains the original goal of exploring interesting tuples while avoiding uninteresting tuples. We introduce the notion of *good(k, ℓ)* queries which focus on deriving any k tuples from the top ℓ ($0 \leq \ell \leq 1$) fraction of the tuples. For a database where each tuple occurs only in one document, we can build a simple solution to process a *good(k, ℓ)* query: we draw a random sample of k/ℓ tuples, and return the top-k tuples among this set.

In practice though, a tuple may 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. **(contd.)** *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. Our goal is to 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 by the above examples. Typically, good tuples such as ⟨Cholera, Sudan⟩ will occur multiple times [9] and thus have their overall confidence scores boosted; on the other hand, low quality tuples such as ⟨Measles, America⟩ will be sparse and not have their scores boosted.

Given that there are multiple occurrences of a tuple, yet another method for accessing the tuples (in addition to the sequential scan) is possible: we can generate a search query using the tuple values to fetch and process (a superset of) documents that can generate this tuple. For instance, consider a tuple that occurs more than once in a database. We can scan for the first occurrence of this tuple and then query for the remaining occurrences using a search interface to the database, thus focusing only on the documents that are likely to contain the tuple. Now, for a database where a tuple may occur multiple times, using the query-based access we can build a simple solution to process a *good(k, ℓ)* query: we draw a random sample of k/ℓ tuples using scan, and then using appropriate queries for each tuple, we obtain all the confidence scores for all k/ℓ tuples. However, this approach may be expensive as a large number of documents may be retrieved using the query-based access.

In this paper, we identify and investigate the novel problem of *efficiently identifying* good(k, ℓ) *answers where tuples have multiple occurrences in a database.* We present a two-phase processing approach where we first pick an appropriate set of candidate tuples for a given *good(k, ℓ)* query, and then identify the final answers among these candidate tuples. To identify the set of k answers from a candidate set, we propose two algorithms. Our first algorithm, E-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, ℓ) 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.

As a particularly challenging aspect of processing *good(k, ℓ)* queries is identifying a "right-sized" candidate set. This 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 two-step method that first learns the relevant parameters of the data using a small sample, and then uses the learned parameters to adaptively choose a "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, ℓ) 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 2 introduces and formally defines our data, access, and query models. In Section 3 we present *good(k, ℓ)* processing algorithms under these models, which consist of a deterministic and a probabilistic algorithm. In Section 4 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 5 and 6. Section 7 discusses related work. Finally, we conclude the paper in Section 8.

2. DATA, ACCESS, AND QUERY MODELS

In this section, we first define the data and access models which form the basis of our query processing approach. Then we formally define the *good(k, ℓ)* query model and the problem of interest.

2.1 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

	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

Figure 1: Sample matrix of confidence scores.

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 ⟨Cholera, Sudan⟩ 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, $\mathbf{sv(t_1)} = [1.0, 0, 0, 0, 0, 0, 0, 0, 0.75, 0, 0.6]$. □

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 $\mathbf{sv(t)}$. 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,

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

where $n = |D|$. We can derive this using the form $s(t) = 1 - \prod_{i=1}^{|D|}(1 - s_i)$. 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 $s(t)$ derived using *incl-excl* is the probability of the tuple being correct as gathered from the multiple evidences.

2.2 Access model

To extract tuples from a text database, there are two main access methods available [14, 15, 16]: *scan* and *query*. Under the scan-based access, we sequentially retrieve documents from the database and thus draw random samples over the database. We refer to this access method as **S-access**.

Definition 2.1. [**S-access**] *Given a document d_i,* **S-access***(d_i) returns all the tuples t_j (with their scores in d_i) that can be extracted from d_i. Given a set of documents D,* **S-access***(D) returns $\cup_{d_i \in D}$***S-access***(d_i).* □

Upon discovering a tuple t using **S-access**, we can "probe" for a specific set of documents that contain the terms in the tuple. Specifically, we construct an appropriate conjunctive text search query for t and issue it to the search interface of the database. We refer to this access method as **Q-access**.

Definition 2.2. [**Q-access**] *Given a set of documents D, and a tuple t_j,* **Q-access***(t_j, D) retrieves the set $H(t_j) \subseteq D$ of documents that match the keyword query obtained from t_j.* □

Q-access, in principle, can retrieve all documents that contain the given tuple, provided there are no restrictions on the maximum number of results returned by the search interface.

Example 2.2. *Following our 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.* □

Naturally, the documents retrieved by **Q-access** may be a superset of the documents that generate the tuple, as an extraction system may fail to recognize any occurrences of the tuple in some of these documents. However, in general, the conjunctive search queries generated using the extracted tuples rarely match *all* database documents, thus using **Q-access**, as compared to **S-access**, can 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.

2.3 Query model

We now discuss a novel query model for the goal of finding a few good tuples to aid in data exploration tasks. But first we argue why existing top-k query model is not appropriate.

Proposition 2.1. *Using a sequential, unsorted access to the tuples in the database, in the worst-case, a top-*k *query would need to process all documents in the database. In the presence of the score distribution, in the average-case, a top-k query would need to process at least half the documents in the database.* □

Consider the simplest case of processing top-l (k = 1) query, i.e., retrieving the best tuple in the database. In the absence of any score distribution information, we need to observe all possible tuples and scores in order to pick *the best* tuple. 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 tuples present in the database. Generalizing to top-k tuples, when the score distribution is known, the expected number of tuples to observe is $k \cdot \frac{|T|}{k+1}$: informally, under random retrieval

the k answer tuples partition $|T|$ into $k+1$ equal partitions, each containing $\frac{|T|}{k+1}$ tuples. Assuming that the tuples are uniformly distributed across the database documents, to fetch the desired k top-ranked tuples, we must observe at least k partitions. This would imply near-complete processing of a database. Furthermore, another important limitation is that processing top-k would require the knowledge of $|T|$, number of all tuples in the database. For our goal of exploring a database, such information may not be readily available.

For the task of exploring the potential of a database for an extraction system, users may be willing to relax the criteria of "best" k and instead explore "good" k tuples, in return of some reduction in the query processing work. With this in mind, we introduce a *good(k, ℓ)* query defined as:

DEFINITION 2.3. *A* good(k, ℓ) *query returns any k tuples that belong to the top-ℓ ($0 \leq \ell \leq 1$) fraction of the tuples in the database, when ranked by their aggregate scores.* □

The *good(k, ℓ)* query model maintains the principle underlying the top-k queries: users can focus on the interesting tuples while avoiding the uninteresting tuples. However, an advantage of the *good(k, ℓ)* query model is the possible significant reduction in the amount of work necessary in order to return the answers.

EXAMPLE 2.3. *Consider again the scores matrix from Figure 1. Under the top-k query model an answer to the top-1 query using* sum *is t_{12}. For this, we ought to observe all scores for the 20 tuples. On the other hand, under the* good(k, ℓ) *query model a good(1, 0.1), i.e., one tuple from the top 10% percent of the ranked tuples, a valid answer includes any one of the tuples in $\{t_1, t_{12}\}$. To process the good(1, 0.1) query, 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 underscores two important observations. First, the amount of work, i.e., total number of tuples to extract can be bounded. For most databases, this translates to a bound on the number of documents to process as well. (Recall that processing top-k queries required near-complete processing of the database.) Second, the answer model does not require any a priori knowledge about the total number of tuples in the database, which is often unknown for most extraction tasks. For instance, in Example 2.3 to process the *good(1, 0.1)* query, we need to focus on just 10 tuples irrespective of the total number of tuples that can be extracted from the database.

To define query processing algorithms for *good(k, ℓ)* queries, we need to study the cost incurred in deriving the answer:

DEFINITION 2.4. **[Execution Cost]** *Consider a database D. The* execution cost *of an algorithm for a* good(k, ℓ) *query is defined as the fraction of database documents that are processed by the algorithm, i.e., $\frac{|documents\ processed|}{|D|}$. The execution cost ranges from a minimum of 0 to maximum of 1 when all the database documents have been processed.* □

We now formally define the problem on which we focus:

PROBLEM 2.1. *Given a* good(k, ℓ) *query, over a database D, our goal is to efficiently process the query and derive an answer set that meets the user query.*

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.

3. PROCESSING GOOD(K,L) OVER TEXT

The most straightforward solution to deriving *good(k, ℓ)* answers is to process all available database documents using **S-access**. For each retrieved tuple t, we derive its aggregate score $s(t)$ and sort the tuples by their aggregate scores. We then identify the set of tuples in the top ℓ fraction of this sorted list and return as answers any sample of k tuples from this set. This method is equivalent to solving the top-k problem (see Section 2.3). This method returns the correct answer for each *good(k, ℓ)* query but this approach can be unnecessarily expensive, especially since many tuples could have been discarded as unnecessary to derive an answer. As this approach explores each database document, the execution cost is the maximum cost of 1. In what follows, we present improvements over this exhaustive naive approach.

3.1 Overview of our approach

To improve upon the exhaustive approach, we need to be able to (a) identify and explore only a relatively smaller set of tuples and (b) identify the number of non-zero elements in a tuple's score vector. Identifying a smaller *candidate set of tuples* would reduce the execution cost by avoiding processing documents for tuples that do not belong in the final answer. On the other hand, identifying the number of non-zero elements in the score vectors will allow query processing algorithms to terminate once every possible score for a tuple has been explored.

For the first task, we introduce a *getCandidate(k, ℓ, δ)* procedure that retrieves a set C of candidate tuples such that C contains at least k answers from the top-ℓ fraction with probability $(1-\delta)$. Later in Section 4 we discuss *getCandidate(k, ℓ, δ)*. Given a candidate set of tuples, the original goal of processing *good(k, ℓ)* query is transformed into the problem of finding top-k answers among the candidate tuples.

PROPOSITION 3.1. *The top-k of* getCandidate*(k, ℓ, δ) is in the* good(k, ℓ) *set with probability at least $1-\delta$.* □

For the second task of determining the number of non-zero elements in the score vectors for the tuples in C, we employ **Q-access**. Specifically, for each tuple t we only retrieve the set $H(t)$ of documents that match the query constructed using t. Using $|H(t)|$ we can derive an upperbound on the number of non-zero elements. As discussed previously, this value may be an overestimate of the actual number of non-zero elements, which can be corrected using additional statistics on the fraction of the matching documents that retrieve a tuple. Our algorithms focus on the generalized setting where such statistics are not available a priori.

We summarize our generic algorithm for processing a *good(k, ℓ)* query below:

1. Retrieve a candidate set C of tuples using *getCandidate(k, ℓ, δ)* such that C contains at least k answers with probability $(1-\delta)$.
2. Initialize a map $M = \emptyset$
3. For each tuple $t \in C$
   (a) Using **Q-access** retrieve matching documents $H(t)$
   (b) $M[t] \rightarrow H(t)$
4. Identify the top-k tuples in C, given M, using some processing algorithm.

To this end, yet another straightforward solution is possible: for each tuple t in C, using **Q-access** to retrieve the matching documents $H(t)$, we process all of them to explore all scores for t and compute its aggregate score. We can then sort the tuples in C by their aggregate score and return the k highest ranking tuples as the answer. The cost of this *probe-all* strategy is $\frac{\sum_{t=1}^{|C|}|H(t)|+S}{|D|}$, where S is the number of documents processed in order to generate the candidate set. However, we may not necessarily need to process every document that matches every tuple in the candidate set. In the rest of this section, we discuss two algorithms, namely, E-Upper (Section 3.2) and Slice (Section 3.3) that focus on reducing the execution cost of this naive strategy.

3.2 E-Upper Algorithm

In this section, we present E-Upper, a deterministic algorithm that guarantees to return the top-k tuples in a candidate set. Thus, given a *good(k, 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] introduced in the context of web-accessible sources where there exists at least one source that retrieves the target tuples in a sorted fashion, and additionally sources may allow to "probe" for the scores of a tuple. The main idea is: 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 three possible scores for each tuple: (a) an upperbound which is the best-case score, a lowerbound which is the worst-case score, and an expected score. At any given point in time, we can discard a tuple if the upperbound of the tuple is strictly below the lowerbound 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 can process a top-k query only via "probing" for the rest of the scores for a tuple. This observation is particularly important for our query processing setting. Specifically, once a candidate set has been constructed, we can adapt Upper to generate the *good(k, ℓ)* answers based on the information available via **Q-access**.

Consider a tuple t in the candidate set C with $H(t)$ matching documents retrieved using **Q-access** of which we have processed i documents. Thus, the score vector $\mathbf{sv(t)}$ contains scores for i elements, a value of 0 for $(|D| - |H(t)|)$ elements, and the values for $(|D| - |H(t)| - i)$ elements are unknown. At any given time during the query processing, we can define the three scores associated with t as follows:

- U(t), highest possible score for t if all the unseen scores were assigned the maximum value of 1
- L(t), smallest possible score for t if all the unseen scores were assigned the minimum value of 0
- E(t), expected score of t if all the unseen scores were assigned some expected score based on the i items that we have observed

Given a partially filled score vector $\mathbf{sv(t)}$ deriving U(t) and L(t) is straightforward. For E(t), we use a "best" guess approach and pick the mean of the observed scores as the expected score. It is noteworthy that since the tuples are not retrieved in a specific order, we cannot use the last observed score in a sorted list to guide us in deriving the expected scores as in [4].

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

Output: k highest ranking tuples
Initialize answers = 0
while *answers* $< k$ **do**
  Retrieve t_H from C with $U(t_H) = \max_{t \in C} U(t)$
  $H(t_H) = M \rightarrow \{t_H\}$
  if *all documents in* $M(t_H)$ *have been processed* **then**
    // t_H belongs to top-k answers
    return = return +1;
  else
    // t_H needs to be processed to either decrease its U(t)
    $E[t_k]$ = get expected score of the current $(k - \text{answers})^{th}$ ranked tuple in C
    Retrieve and process a document d from $H(t_H)$

Figure 2: A deterministic algorithm.

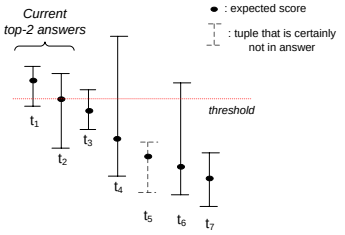

Figure 3: Working of the E-Upper algorithm by reducing the upper bound of a candidate tuple.

Figure 2 shows the E-Upper algorithm. E-Upper processes documents for candidate tuples until it reaches a stage where there are no non-answer tuples for which the upperbound, i.e., $U(t)$, is higher than the score of the k_{th} answer tuple. At any given iteration, E-Upper picks a tuple t_H with the maximum value for the upperbound, to explore next: if t_H belongs to the current set of answers, we must determine its actual score; alternatively, if t_H does not belong to the top-k answer, we must lower its score to return the current answer. Figure 3 shows a snapshot of the E-Upper algorithm in progress. (For illustration purpose, we used a figure similar to that in [4].) The figure shows the scores for 7 tuples for the task of identifying top-2 answers. A tuple whose upperbound is lower than the lowerbound of at least 2 tuples can certainly not belong to the final answer. We discard such tuples at each iteration. As shown in the figure, we have two tuples, t_4 and t_6 such that their maximum score could be above the expected score for t_2, the k_{th} answer tuple. So, these tuples need to be further explored and we pick the tuple with highest upper bound, i.e., t_5, and process an associated document.

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, under some scenarios, E-Upper can be time-consuming: con-

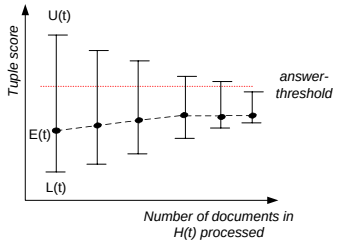

Figure 4: Working of the E-Upper algorithm by reducing the upper bound of a candidate tuple.

sider the case in Figure 4 which shows how the upperbound of a tuple is reduced with respect to the minimum threshold set by the expected score of the k_{th} answer. Figure 4 illustrates the change in a tuple's upperbound as the documents associated with this tuple are processed. The figure shows how even though the expected score of this tuple is below the required threshold and is reasonably stable, we still continue to process documents until its upper bound is reduced below the threshold. This scenario is more common in our settings as a tuple can match relatively large number of documents, which, in turn, results in large upperbounds and slower convergence. This observation has been made in the top-k processing case as well, leading to developing probabilistic algorithms [21]. However, typically, existing probabilistic algorithms would 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 build 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 can belong to the final answer, based on the relative standing of the candidate tuples.

3.3 Slice Algorithm

In this section, we introduce a probabilistic algorithm, Slice(C, a, ϵ), that returns a answers 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. Specifically, 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* the candidate set 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 documents* associated with each candidate tuple and the tuples in both 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.

Figure 5 depicts the pseudocode for Slice. 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 position boundary, τ_g such that we take all the candidate tuples with rank positions equal or above τ_g to 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 we take all the candidate tuples with rank positions equal or below τ_g to belong to Y. Since we use partial score information, 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 $\tilde{\epsilon} \leq \epsilon$ (ϵ is the user-specified error-tolerance). The remainder $\epsilon - \tilde{\epsilon}$ is

Algorithm 2: **Slice**(C, a, ϵ)

```
// base cases
if a ≤ ε then
    return ∅
if |C| ≤ a then
    return C_i
// end base cases
if ε = 0 then
    Process all remaining documents for each tuple in C
    return top-a in C
for {tuple t ∈ C} do
    H(t) = M → {t}
    Process α fraction of documents in H(t)
Rank C based on observed scores so far
ε̃ = getNextError(ε)
Pick τ_g and τ_b using ε̃
Get set X of tuples in C with rank higher than τ_g
Get set Y of tuples in C with rank lower than τ_b
return X ∪ Slice(C \ X \ Y, a − |X|, ε − ε̃)
```

Figure 5: A recursive algorithm for deriving approximate answers.

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(ϵ); we will discuss this procedure later in Section 3.3.1.

Given $\tilde{\epsilon}$, we pick τ_g such that the expected number of *false positives*, i.e., tuples with actual rank lower than a and observed rank higher 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 lower than a and observed rank lower than τ_b is expected to be no larger than $\tilde{\epsilon}$. For simplicity of our discussion, 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 Section 3.3.2 we discuss how we pick τ_g and τ_b for a given a and $\tilde{\epsilon}$.

EXAMPLE 3.1. *To illustrate the working of* Slice *we discuss a hypothetical scenario involving the sample score matrix in Figure 1. Assume the goal of deriving 2 tuples from the top-4 of a candidate set, i.e, $k = 2$ and $E = 2$ so $a = 4$. We will use* sum *as the aggregate function, and based on this 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 for this goal that contains a total of 8 tuples consisting of the tuples t_1, t_2, t_7, t_{10}, t_{12}, t_{14}, t_{16}, and t_{20} from Figure 1.* Slice *begins with processing a fraction of the documents associated with each candidate tuple. Say* Slice *processes documents $d_{11}, d_6, d_6, d_3, d_2, d_3, d_{11}, d_6$ for the above tuples, respectively, which results in the observed scores shown in Figure 6. For instance, for tuple t_1, we observe a score of 0.6 from document d_{11}, whereas for the 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 6): tuple t_{12} is at rank position 1 and tuple t_{16} is at rank position 2; similarly, t_{20} is assigned a rank position of 8. These observed ranks are not identical to the actual ranks 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 in the right position.*

At the first recursion level, the assigned error budget is $\tilde{\epsilon} = 1$. Using an appropriate "oracle," Slice *determines that the expected number of false positives for $\tau_g = 2$ is no more than 1; similarly it picks $\tau_b = 8$. As a result, it generates the sets $X = \{t_{12}, t_{16}\}$ and $Y = \{t_{20}\}$. In the next recursion, the*

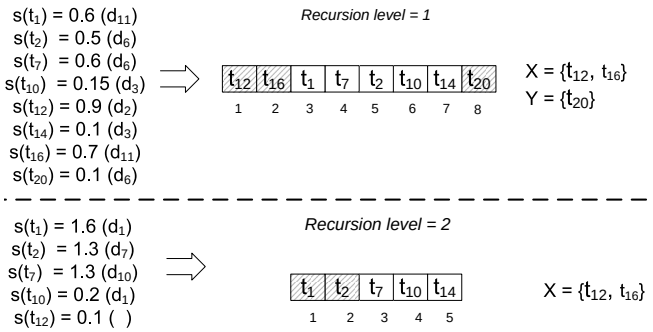

Figure 6: Sample recursion snapshots for Slice**.**

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. The tuple t_1 is the top ranking tuple now and t_7 is ranked at position 2. At the second level of recursion the assigned error budget $\tilde{\epsilon} = 1$, and* 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 atleast 2 tuples are correct.* □

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

THEOREM 3.1. Slice(C, a, ϵ) *returns a set R such that $|R| = a$ and atleast $a - \epsilon$ tuples in R are in the top-a of C.*

To prove this theorem, we define the following lemmas.

LEMMA 3.1. *Let $C' \subseteq C$ such that C' contains β elements from the top-α of C (where $\beta \leq \alpha$), then the top-β of C' are contained in the top-α of C.*

PROOF. *We will prove this by contradiction. Assume $\exists$ tuple $t_1 \in$ top-β of C' such that $t_1 \notin$ top-α of C. Since C' contains β tuples from top-α of C and we assume not all of them are in top-β of C', then $\exists$ tuple $t_2 \in$ top-α of C but $t_2 \notin$ top-β of C'.*

Now $t_1 \in$ top-β of C' and $t_2 \notin$ top-β of $C' \implies s(t_1) > s(t_2)$.

But $t_2 \in$ top-α of C and $t_1 \notin$ top-α of $C \implies s(t_2) > s(t_1)$.

Since $s(t_1)$ cannot be both greater and smaller than $s(t_2)$, we arrived at a contradiction. ∎

LEMMA 3.2. *Let $C' \subseteq C$ such that C' contains β elements from top-α of C ($\beta \leq \alpha$) and let $R' \subseteq C'$ such that R' contains at least δ elements from top-γ of C' ($\delta \leq \gamma \leq \alpha$). Then R' contains at least $\delta - max\,[(\gamma - \beta), 0]$ elements from the top-α of C.*

PROOF. *By Lemma 3.1, $\{$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 in top-γ of C' are contained in top-α of C, so at most $\gamma - \beta$ tuples in top-γ of C' are not contained in top-α of C. Therefore, out of the δ tuples from R' contained in top-γ of C' at most $\gamma - \beta$ can not be contained in 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 top-α of C. ∎

We are now ready to prove Theorem 3.1

PROOF. *By induction on the recursion height.*

Base cases: (a) If $|C| \leq a$, Slice *returns C which contains all tuples in top-a of C. (b) If $a = 0$, then $R = \emptyset$ is returned by* Slice*, which satisfies the condition trivially.*

Induction step: Slice *picks $\tilde{\epsilon}$, X, and Y (see Figure 5). Let X_b and Y_g denote the false positives in X and the false negatives in Y, respectively. Let R' be the set returned by the recursion call* Slice$(C \setminus X \setminus Y, a - |X|, \epsilon - \tilde{\epsilon})$*. We identify three properties used in our proof.*

By the induction hypothesis, $|R'| = a - |X|$ and R' contains at least $a - |X| - (\epsilon - \tilde{\epsilon})$ from top-$(a - |X|)$ of $C \setminus X \setminus Y$ **(1)***. To calculate the number of tuples in R that belong to the top-a of C, we observe that $|X|$ contains $|X| - |X_b|$ tuples from top-a of C* **(2)***, and C' contains $(a - |X \setminus X_b| - |Y_g|)$ tuples from top-a of C* **(3)***.*

Applying Lemma 3.2 to C', where $C' = C \setminus X \setminus Y$, with $\alpha = a$, $\beta = a - (|X| - |X_b|) - |Y_g|$ (by **(3)***), $\gamma = a - |X|$, and $\delta = a - |X| - (\epsilon - \tilde{\epsilon})$ (by* **(1)***), we get that R' contains at least $a - |X| - (\epsilon - \tilde{\epsilon}) - max[(a - |X|) - (a - (|X| - |X_b|) - |Y_g|), 0]$ elements from the top-a of C. Rewriting, we have the total number of tuples in R' from top-a is:*

$= a - |X| - (\epsilon - \tilde{\epsilon}) - max[|Y_g| - |X_b|, 0]$

$= a - (|X| - |X_b|) - (\epsilon - \tilde{\epsilon}) - max[|Y_g|, |X_b|]$

$\geq a - (|X| - |X_b|) - (\epsilon - \tilde{\epsilon}) - \tilde{\epsilon}$ *(by $|Y_g|, |X_B| \leq \tilde{\epsilon}$)*

$= a - (|X| - |X_b|) - \epsilon$

Using **(2)***, this implies that $R = R' \cup X$ contains at least $a - (|X| - |X_b|) - \epsilon + |X| - |X_b| = a - \epsilon$ elements from top-a of C.* ∎

3.3.1 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 the same $\tilde{\epsilon}$ value, we observe that 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 level r_2, Slice would have already observed and processed more documents than at level r_1, thus moving closer to the actual ranking of the candidate tuples and reducing the chance of picking a false positive. Reducing the chance of picking a false positive, in turn, allows Slice to pick a larger value for τ_g at level r_2 than at level 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 which can be done by picking a very small value for $\tilde{\epsilon}$ during the initial levels of recursion. However, deeper recursion levels come at the cost of processing additional documents since 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 assigned error budget $\tilde{\epsilon}_2$ and $\tilde{\epsilon}_1$, where $\tilde{\epsilon}_2 > \tilde{\epsilon}_1$, we observe that 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 this would require that Slice process *all* documents for each tuple

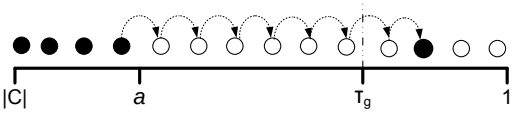

Figure 7: A false positive generated by a series of consecutive rank swaps.

in C (see Figure 5).

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.

3.3.2 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 higher than than a is observed at a rank lower 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 7 illustrates the generation of a false positive for a set C of ranked tuples, with 1 being the highest rank position and $|C|$ the least possible rank position. 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 higher ranking tuples 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:

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

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:

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

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 lower 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 "piggie-backed" with the process of deriving other parameters necessary for query processing (Section 4).

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.

4. CANDIDATE SET GENERATION

Our algorithms discussed in Section 3 rely on a *getCandidate(k, ℓ, δ)* procedure to generate a candidate set C that contains at least k tuples from the top-ℓ fraction of the database tuples, with probability $1-\delta$. We now present two methods to derive a candidate set, a Naive method (Section 4.1) and an Iterative method (Section 4.2).

4.1 Naive Approach

Given the goal of constructing a candidate set that contains k tuples from the top-ℓ fraction 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 observed 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 is a random variable V_H that follows a hypergeometric distribution. Thus,

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

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

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

In practice though, deriving the cumulative distribution function(cdf) 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. Along with simplifying the candidate set size derivation, using a binomial model provides an added benefit of not requiring the knowledge of exact number of tuples in the database. This is particularly appealing in our data exploration setting where the total number of tuples are not known apriori. Under the binomial model, the number of tuples in C that belong to the top-ℓ fraction is a random variable V_B with probability of success $p = l$ such that:

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

and use Equation 5 to pick the size of candidate set to draw.

Finally, for $p \geq 0.5$ we can further use Chernoff's bounds [19] to derive an upper bound on $|C|$.

The approach outlined above assumes no skew in the data, i.e., 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. Next, we present another approach for constructing the candidate set that relaxes these assumptions.

4.2 Iterative Approach

In some cases, we may have a skewed database such that some tuples occur more frequently than the others. In fact, [14, 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 on the rank 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, where an aggregate score is derived by summing up the scores assigned to a tuple by different documents. 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 be 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 such 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 approach. Given the goal of constructing a candidate set containing k tuples from the top-ℓ fraction, we pick a small value s for tuples ($s << k$) and construct an initial candidate set that contains s tuples from the top-ℓ fraction. To derive this initial candidate set, we use the approach outlined in Section 4.1. We then derive documents associated with the tuples in this initial candidate set using **Q-access**, and process them to derive the actual aggregate score for each candidate tuple using the aggregate function. Our goal now is to derive the *adjust factor* $a(\ell)$ for the tuples in this initial candidate set. For this, we calculate the actual number of tuples in the initial candidate set that belong to the top-ℓ fraction divided by the our initial goal of s. When using a function like *sum* that calls for down sampling, $a(\ell) \geq 1$, and when using a function like *min* that calls for over sampling, $a(\ell) \leq 1$. Using this adjust factor, we can now construct a candidate set for the remainder $k - s$ tuples from the top-ℓ fraction. 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 estimate for $a(\ell)$ iteratively. Specifically, we can split the original goal of deriving k tuples from the top-ℓ fraction across n iterations such that $\sum_{i=1}^{n} s_i = a$ where s_i are the number of tuples fetched at iteration i. Using the adjust factor obtained at iteration i using 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 ($n = 2$) results in candidate sets with sizes close to those we can obtain if we had perfect knowledge of the scoring function effect.

5. EXPERIMENTAL SETTINGS

We now describe the settings for the experiments in Section 6, focusing on the text collections, extraction systems, and the evaluation metrics.

Information Extraction Systems: We trained Snowball [2] for two relations: *Headquarters⟨Company, Location⟩*, and *Executives⟨Company, CEO ⟩*

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 5 to 50, in steps of 5, 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:

$$Precision = \frac{|G \cap R|}{|R|}; \qquad Recall = \frac{|G \cap R|}{k} \tag{7}$$

In addition to the precision and recall, we also derive the execution cost of deriving an answer using Definition 2.4.

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 4). We denote the combination of E-Upper with Naive as *EU-N*, and that with Iterative as *EU-I*. Similarly, we constructed two variants of Slice, (a) *SL-N* Slice combined with the naive and (b) *SL-I* Slice combined with the iterative method.

6. EXPERIMENTAL RESULTS

We now present our experimental results on evaluating the proposed algorithms and candidate set generation methods.

6.1 E-Upper

Our first experiment evaluates E-Upper (Section 3.2) for the task of extracting the HQ relation over NYT95. For both settings, *EU-N* and *EU-I*, the resulting precision and recall is perfect, as E-Upper is a deterministic algorithm. We examined the execution cost for these settings. Figure 8(a) shows the execution cost of for *EU-N* and *EU-I*, for varying values of ℓ, when using *sum* (see Section 2.1); the execution

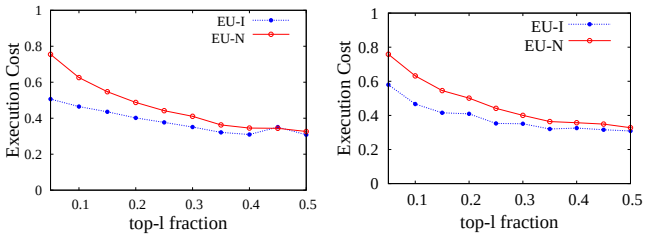

Figure 8: Average execution cost using E-Upper **for varying ℓ using (a) *sum* and (b) *incl-excl*.**

cost is an average across a set of k values, ranging from 20 to 200 (see Section 5). Figure 8(b) shows the average execution cost derived for *EU-N* and *EU-I* when using the *incl-excl*, for varying ℓ values. The execution cost for both *EU-N* and *EU-I* is high for low values of $\ell(\ell < 0.25)$ than that for $\ell > 0.25$. This is because, keeping k constant, the sizes of the candidate sets when $\ell < 0.25$ are larger than those for $\ell > 0.25$. Larger candidate sets, in turn, involves more documents to be processed before we can generate the final answer, and thus the higher execution costs. Figures 8(a) and 8(b) also show a promising direction towards reducing the execution cost by using the Iterative method for generating the candidate set; as seen the figure, the execution cost for the *EU-N* line is higher than *EU-I* for all values for ℓ. (Our observations for other datasets and relations were similar to those above, which we do not report due to space constraints.)

6.2 Slice

Our second experiment evaluates Slice (Section 3.3) for the task of extracting HQ from NYT95. For each of the two settings, *SL-N* and *SL-I*, we considered two different values for α, which is the fraction of documents processed by Slice at each recursion level. Finally, we set E, which is the acceptable error budget assigned by the user to be 5% of the k value.

Figure 9(a) shows the average precision for *SL-N* and *SL-I*, for $\alpha = 0.1$ and $\alpha = 0.2$, for varying values of ℓ, for *sum*. 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 equal or above 0.75, with lower precision values for *SL-I* and close to 1 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 in the case of a candidate set generated using the Naive method. As discussed in Section 3.3.1 the value of α directly affects the precision: higher the value for α, closer the observed ranking of candidate tuples to the actual ranking, and thus higher the precision values. Figure 9(b) shows the average precision value for varying ℓ, when using *incl-excl* as the aggregate function and the precision follows a trend similar to that in the case of *sum*.

Figure 10(a) shows the average recall for *SL-N* and *SL-I*, for α (0.1 and 0.2), for varying values of ℓ, when using the *sum* as the aggregate function; the recall value is an average across a set of k values, ranging from 20 to 200. In general, the observed recall ranges between 0.7 to 1.0, and just as in the case of precision, we observe higher recall values for *SL-N* than that for *SL-I*. Furthermore, α affects recall in the same manner as in the case precision, with higher values of alpha improving the recall. Figure 10(b) shows the average recall value for varying ℓ, when using *incl-excl*, and the observations

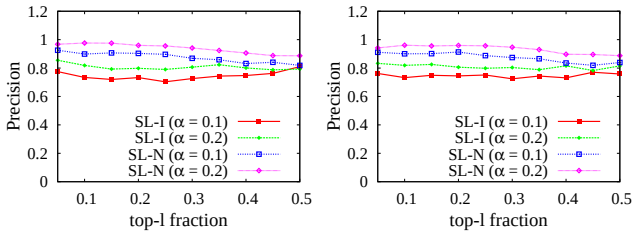

Figure 9: Average precision using Slice **for varying ℓ using (a) *sum* and (b) *incl-excl*.**

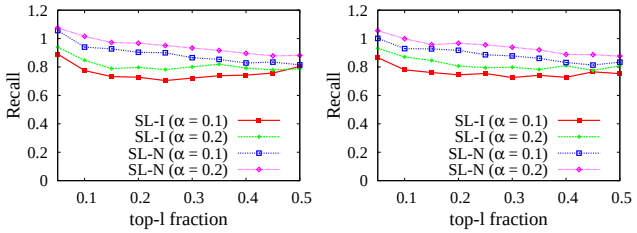

Figure 10: Average recall using Slice **for varying ℓ using (a) *sum* and (b) *incl-excl*.**

are similar to that in the case of *sum*.

In our experiments, we observed that in general the precision and recall values decrease for increasing k and ℓ values. Interestingly, as we increase k the precision and recall drops faster than in the case when we increase ℓ. To illustrate this observation, we examined the precision and recall of the Slice answers when ℓ is fixed at 25, and k is varied from 20 to 100. Figures 11(a) and 11(b) show the resulting precision and recall, respectively. As shown in the figure, the performance of Slice for all settings is ideal for k is 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 5 to 25. Figure 12(a) and 12(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 degeneration 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 when using *incl-excl*, which we do not discuss further.

We also studied the execution cost when using Slice. Figure 13(a) shows the average execution cost for *SL-N* and *SL-I*, using two values for α (0.1 and 0.2), for varying ℓ values; the

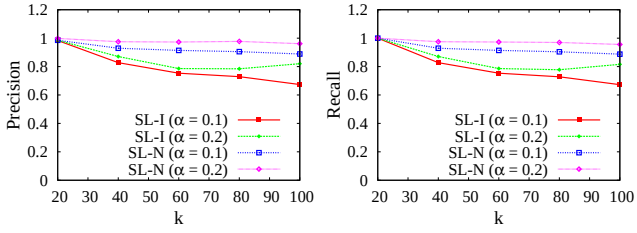

Figure 11: Precision (a) and Recall (b) using Slice **when $\ell = 0.25$ for varying k using (a) *sum* and (b) *incl-excl*.**

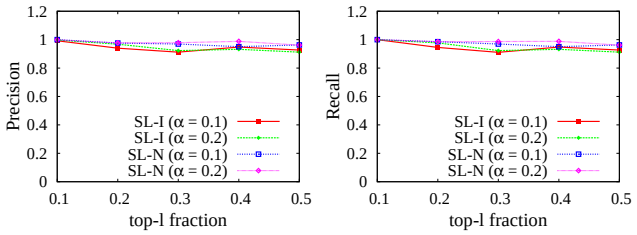

Figure 12: Precision (a) and Recall (b) using Slice **when $k = 25$ for varying ℓ using (a) *sum* and (b) *incl-excl*.**

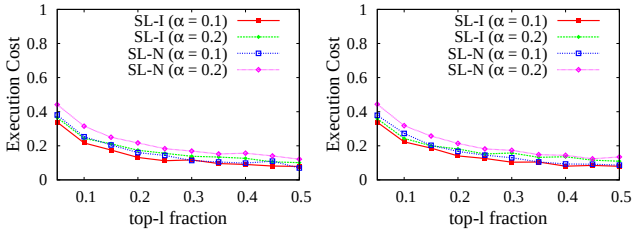

Figure 13: Average execution cost using Slice **for varying ℓ using (a) *sum* and (b) *incl-excl*.**

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 using Slice can result in at least two times reduction in the execution cost as compared to *EU-N*. As compared to *EU-I*, the execution cost for Slice is strictly lower than that for *EU-I*, with an improvement of two times for higher ℓ values. These observations when combined with our observations on the precision and recall of Slice underscore an important observation: 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 is similar to that for E-Upper.

The accuracy of Slice depends on the trend of the number of false positives and the number of false negatives at different rank positions, as we move away from the target, i.e., k. In Section 3.3.2 we made a simplified assumption that this trend follows (an exponential) trend as defined by Equation 3. For a first level of recursion, Figure 6.2 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 14(a) shows the number of false positives, as we travel away from the target position of 25, and Figure 14(b) shows the same for the number of false negatives at varying rank positions. The figures reveal an important observation: it confirms our intuition that the false positives and false negatives diminish as we travel farther from k, and thus encouraging the idea of *slicing off* the extremities of a candidate set. While neither false positives and false negatives follow a power trend, for this observations the trend is a linear one. Accounting for the exact nature of the false positives and false negatives distribution would obviously require more detailed information, e.g., slope of this trend. However, this allows us to gain insight into why the precision and recall for Slice is low for low k values:

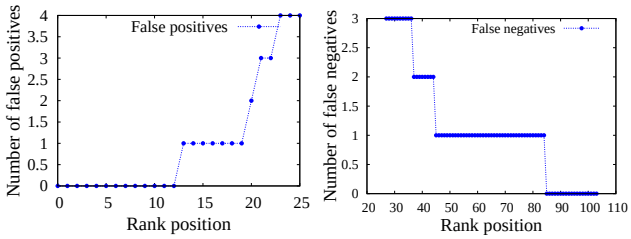

Figure 14: The number of false positives (a) and the number of false negatives (b) at varying rank positions for $k = 25$, $E = 1$, and $\ell = 0.25$.

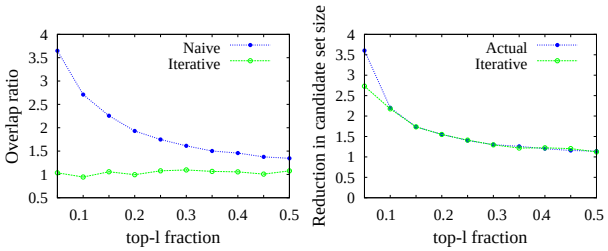

Figure 15: Average (a) overlap ratio and (b) reduction in the candidate set sizes for varying ℓ.

lower the k fewer tuples can swap around and thus fewer false positives and false negatives.

6.3 Candidate Set Generation

In our final set of experiments, we examine our methods, Naive (Section 4.1) and Iterative (Section 4.2) for generating the candidate sets. 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. 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 15(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 ℓ values, thus indicating that this method fetches more than required candidate tuples. On the other hand, using the iterative method the overlap ratio falls below 1 (≈ 0.98) at times, but by a very small margin.

To examine the benefit of subsampling using the iterative method, we compute average size of the candidate tuples, for varying ℓ values. Figure 15(b) shows the relative reduction in the candidate set sizes, computed as the size of the candidate set derived using the Naive method divided by the size of the candidate set derived using the Iterative method. As shown in the figure, for $\ell = 5$, using the Iterative method (on an average) reduces the candidate set size by more than half. This in turn, reduces the overall execution cost of the algorithms, as we have seen in our discussion above. For reference, Figure 15(b) also shows the "actual" reduction in the candidate set size computed using the actual knowledge. Specifically, we derive a candidate set using the exact value for

factor by which we must down sample. As shown in the figure, our iterative method with the number of iterations fixed to 2 is close to the true possible reduction is the candidate set size, except for when $\ell = 0.05$.

7. 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 for that allow users to provide requirements in terms of the desired recall [14], the output quality composition [17] 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, we can naturally not 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, 20]. 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 [12, 18]) 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 2.3, 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 3.2, and we established the feasibility of our adaptation at processing *good(k, ℓ)* queries, as discussed in Section 6. For processing top-k algorithms, a variety of probabilistic algorithms have also been explored [21], which exploit some a priori knowledge on the score distribution.

8. CONCLUSION

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.

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

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

[14] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl? Towards a query optimizer for text-centric tasks. In *SIGMOD*, 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. To appear.

[17] A. Jain and P. G. Ipeirotis. A quality-aware optimizer for information extraction. Technical Report CeDER-08-02, New York University, 2007.

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

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

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

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