Building Query Optimizers for Information Extraction: The SQoUT Project

Alpa Jain[1], Panagiotis Ipeirotis[2], Luis Gravano[1]
[1]Columbia University, [2]New York University

ABSTRACT

Text documents often embed data that is structured in nature. This structured data is increasingly exposed using *information extraction systems*, which generate structured relations from documents, introducing an opportunity to process expressive, structured queries over text databases. This paper discusses our SQoUT[1] project, which focuses on *processing structured queries over relations extracted from text databases.* We show how, in our extraction-based scenario, query processing can be decomposed into a sequence of basic steps: retrieving relevant text documents, extracting relations from the documents, and joining extracted relations for queries involving multiple relations. Each of these steps presents different alternatives and together they form a rich space of possible query execution strategies. We identify *execution efficiency* and *output quality* as the two critical properties of a query execution, and argue that an optimization approach needs to consider both properties. To this end, we take into account the user-specified requirements for execution efficiency and output quality, and choose an execution strategy for each query based on a principled, cost-based comparison of the alternative execution strategies.

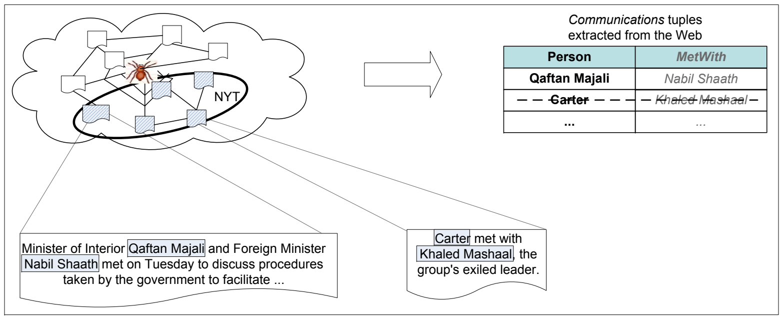
1. INTRODUCTION

Real-world applications frequently rely on the information in large collections of text documents. A financial analyst is interested in tracking business transactions regarding a specific sector from news articles; a company wants to trace the general sentiment towards a recently launched product from blog articles; a biomedical research group needs to identify disease outbreaks from recent health-related reports; an intelligence agency needs to study alliances between groups of people and their past professions by analyzing web pages or email messages. In general, users in the above scenarios are interested in accessing intrinsically structured information embedded in unstructured text databases. To uncover this structured data in text documents, we can use *information extraction systems.* Information extraction systems automatically extract structured relations from text documents, enabling the effective querying of the extracted data in more powerful and expressive ways than possible over the unstructured text. In this paper, we will discuss fundamental issues in defining and efficiently *processing structured queries over relations extracted from text databases*, in the context of our SQoUT project [10, 11, 12, 13, 14]. To understand the family of structured queries on which we focus in this paper, consider the following example.

[1]SQoUT stands for "*Structured Queries over Unstructured Text Databases.*"

EXAMPLE 1. *Consider an archive of newspaper articles (such as the New York Times (NYT), as shown in Figure 1) along with an appropriately trained extraction system to extract a* Communications(Person, MetWith) *relation, where a tuple* $\langle \alpha, \beta \rangle$ *indicates that the person* α *communicated (e.g., via a meeting or a phone call) with person* β. *Consider now an intelligence agent who is interested in recent communications reported in the NYT news articles involving a specific person, named Nabil Shaath. So an appropriate query could be expressed using SQL as* `SELECT C.Person FROM Communications C WHERE C.MetWith = 'Nabil Shaath'`. *In principle, such a query can be answered using the data embedded in the news articles. Specifically, to address this query we can extract information on communications using the extraction system over appropriate documents retrieved from the news archive, to generate tuples such as* $\langle$*Qaftan Majali, Nabil Shaath*$\rangle$ *and then project only the necessary columns, as shown in Figure 1.*

Query processing in an extraction-based scenario can be decomposed into a sequence of basic steps: retrieving relevant text documents, extracting relations from the documents, and joining extracted relations for queries involving multiple relations. During query processing, there are generally various alternatives for each of these steps, for multiple reasons. First, information extraction systems are often far from perfect, and might output erroneous information or miss information that they should capture. As extraction systems may vary in their output quality, we may consider more than one extraction system to extract a relation. Second, information extraction is a time-consuming task, so query processing strategies may focus on minimizing the number of documents that they process. For instance, to process the query in Example 1, we can follow an "exhaustive" execution that sequentially processes and extracts in-



Communications tuples extracted from the Web

Person	*MetWith*
Qaftan Majali	*Nabil Shaath*
~~Carter~~	~~*Khaled Mashaal*~~
...	...

Minister of Interior Qaftan Majali and Foreign Minister Nabil Shaath met on Tuesday to discuss procedures taken by the government to facilitate ...

Carter met with Khaled Mashaal, the group's exiled leader.

Figure 1: Processing a selection query on the relation *Communications(Person, MetWith)* from *The New York Times* news articles on the Web.

formation from every document in the New York Times archive. Alternatively, we can also follow a query-based execution that retrieves documents likely to contain the target information, or we can use a classifier-based approach that determines which New York Times articles are useful for the task at hand. Additionally, we may have multiple join algorithms to join relations extracted from text documents. By composing the various options for the query processing steps we can form a rich space of possible *query execution strategies*.

The candidate execution strategies for a query may differ substantially in their efficiency and output. The choice of extraction systems, as well as of the documents to be processed, affects not only the execution efficiency, but also the output *quality*. In the above example, the exhaustive execution can generate results that are more complete than those from a query- or classifier-based execution; however, the exhaustive execution is likely to be substantially slower than the (potentially less complete) alternatives. Thus, efficiency-related decisions meant to avoid processing useless documents may also compromise the output completeness.

As a natural consequence of this efficiency-quality trade-off, and depending on the nature of the information need, users may have varying preferences regarding the execution efficiency and output quality expected from the querying process: sometimes users may be after exhaustive, quality-oriented query answers, for which users may be willing to wait a relatively long time. Some other times, users may tolerate "quick and dirty" results, which should be returned fast. Therefore, a query execution strategy must be selected following a principled, cost-based comparison of the available candidate strategies, and taking user preferences into consideration.

We pose this problem of selecting an appropriate execution strategy for a query as a *query optimization* problem. To guide the query optimization task, we characterize query execution strategies by their execution efficiency and output quality. Selecting a desirable query execution strategy among all the candidates requires that we effectively estimate execution time and output quality for the candidate query execution strategies at query optimization time. Just as in the relational world, we need to decide which execution plan is best for a given query. Unlike in the relational world, though, we often lack direct access to detailed statistics (e.g., histograms) about the (not-yet-extracted) contents of the underlying database. So, the task of an optimizer is non-trivial, because a principled, cost-based plan selection requires addressing multiple challenges:

- *How can we account for the imprecise and incomplete nature of the information extraction output?*
- *How do we accurately predict the efficiency and output quality for an execution strategy? What database-specific information should we collect?*

The rest of the paper is structured as follows: Section 2 further motivates the need for a query optimizer by reviewing a broad family of information extraction systems, various document retrieval strategies, and join processing algorithms, which together serve as alternatives for critical operations in our query processing approach. To guide the task of query optimization, Section 3 shows how we can estimate important execution characteristics that allow us to compare query processing strategies and pick a strategy that closely meets user-specified preferences. Finally, Section 4 discusses some interesting future directions for research, and we conclude the discussion in Section 5.

2. THE NEED FOR A QUERY OPTIMIZER

For each of the important query processing steps in our extraction-based scenario, we generally have several options available. To understand this space of candidate execution strategies for a query, we discuss the choice of information extraction systems (Section 2.1), of methods to retrieve database documents to be processed during the extraction process (Section 2.2), and of algorithms to join the output from multiple extraction systems (Section 2.3). Given these alternatives, we show how we can pick a query execution strategy and discuss the underlying query optimization problem that needs to be addressed (Section 2.4).

2.1 Extracting Structured Data from Text

Information extraction automatically identifies structured data from inherently unstructured natural-language

The U.N. reported recently of an <DISEASE>**Ebola**</DISEASE> outbreak in <LOCATION>**Sudan**</LOCATION>.

Figure 2: Extracting *DiseaseOutbreaks* from text.

text documents. In general, extraction systems are trained for a specific task. An extraction system typically begins by preprocessing a given document using lexical analysis tools (e.g., to identify nouns, verbs, and adjectives), and named-entity taggers (e.g., to identify instances of organizations, locations, and dates). An extraction system then applies *extraction patterns*, which are extraction task-specific rules, to the tagged document to identify the structured information of interest.

EXAMPLE 2. *Consider the task of extracting a* DiseaseOutbreaks(Disease, Location) *relation from a text database, where a tuple* $\langle d, \ell \rangle$ *indicates that an outbreak of disease* d *occurred in location* ℓ. *Figure 2 illustrates how an instance of the* DiseaseOutbreaks *relation can be identified, after annotating the input, using the pattern "*$\langle$*DISEASE*$\rangle$ *outbreak in* $\langle$*LOCATION*$\rangle$*."*

The extraction patterns used by an extraction system often consist of "connector" phrases or words that capture the textual context generally associated with the target information in natural language, but other models have been proposed [5]. These extraction patterns may be constructed manually, as in KnowitAll [8], or automatically, notably by using bootstrapping as in DIPRE [3], Rapier [4], Snowball [1], or in the work of Pasca et al. [17].

Information extraction is generally a time-consuming process, as it can involve a series of expensive text processing operations (e.g., part-of-speech or named-entity tagging). As a result, several approaches have been proposed recently to improve the efficiency of an extraction task [6, 9, 19, 16, 20]. We revisit one important aspect of this issue in Section 2.2.

Information extraction is also a noisy process and the extracted relations are neither perfect nor complete [7, 12, 13, 14]. An extraction system may generate erroneous tuples due to various problems (e.g., erroneous named-entity recognition or imprecise extraction patterns). Additionally, the extraction system may not extract all the valid tuples from a document (e.g., because the language in the document does not match any of the extraction patterns). To examine the quality of the output generated by an extraction system, we can measure the number of *good* and *bad* tuples in the output and, in turn, the *precision* and *recall* of the extraction output. Intuitively, precision measures the fraction of tuples extracted by the system from a text database that are *good*, while recall measures the fraction of *good* tuples that the system manages to extract from the database. There is a natural trade-off between precision and recall, and extraction systems may be trained to favor one or the other. For instance, a *precision-oriented* extraction system might be preferable for critical data in the medical domain. In contrast, a *recall-oriented* extraction system might be appropriate for an analyst interested in tracking all company mergers as reported in newspaper articles. In some scenarios, we might have more than one extraction system for a relation and the choice of extraction system for the final execution depends on the characteristics (e.g., precision, recall, or efficiency) of these systems. Furthermore, in some cases, information extraction systems may export a tuning "knob" that affects the proportion of *good* and *bad* tuples observed in the extracted relation, and the choice of the knob setting used for the final execution depends on the characteristics associated with each of these settings [13].

2.2 Retrieving Documents for Extraction

We now discuss various document retrieval methods and how they impact the execution efficiency and output quality [10, 12].

Scan: *Scan* sequentially retrieves and processes each document in the database. While this strategy guarantees that we process all the database documents, it may be unnecessarily wasteful in that many useless documents will be processed by the information extraction systems. For instance, to extract the *DiseaseOutbreaks* relation (see Example 2) from a newspaper archive, *Scan* processes all articles in the archive, including those that are likely not to produce any tuples, such as the articles in the Sports section of the newspaper.

Filtered Scan: *Filtered Scan* uses a document classifier to decide whether a database document retrieved is relevant to an extraction task. Thus, *Filtered Scan* avoids processing useless documents not relevant to the task and tends to be more efficient than *Scan*. However, this gain in efficiency might result in a loss of recall, because classifiers are not perfect and might discard documents that are indeed useful for an extraction task.

PromD: *PromD* is a query-based document retrieval technique that also focuses on *promising* documents relevant to an extraction task, while avoiding *useless* documents not relevant to the task. Specifically, *PromD* exploits the search interface of the text database and, for a given extraction task, sends appropriate queries derived using machine learning techniques [2]. For example, we may derive the query [*outbreaks AND fatality*] to retrieve documents for the *DiseaseOutbreaks* relation. Just as *Filtered Scan*, *PromD* might be substantially more efficient than *Scan*, but at the expense of a potential loss of recall.

Const: *Const* is a query-based technique based on "pushing down selections" from the user query. *Const* finds constants (if any) in a user query to retrieve only documents that contain those constants. For the query `SELECT * FROM DiseaseOutbreaks WHERE Location = Sudan`, *Const* uses [*Sudan*] as a keyword query to focus only on documents that contain this word, because documents without it could not contribute useful tuples. *Const* thus avoids processing all documents, and

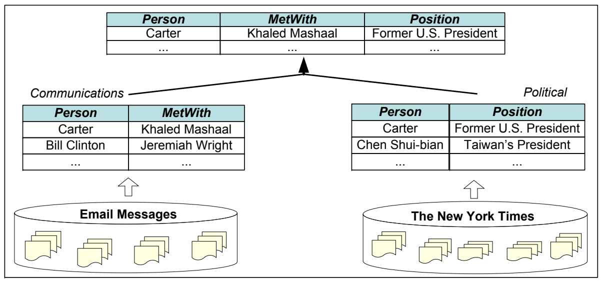

Person | MetWith | Position
Carter | Khaled Mashaal | Former U.S. President
... | ... | ...
Communications
Political
Person | MetWith
Carter | Khaled Mashaal
Bill Clinton | Jeremiah Wright
... | ...
Person | Position
Carter | Former U.S. President
Chen Shui-bian | Taiwan's President
... | ...
Email Messages
The New York Times

Figure 3: Joining information derived using two extraction systems.

its efficiency is determined by the selectivity of the constants in the query. Finally, we can naturally combine the *Const* queries with the *PromD* queries to generate keyword queries such as [*outbreaks AND fatality AND Sudan*] to generate results for the above query.

2.3 Joining the Extracted Data

Earlier, in Section 1, we discussed a single-relation query. By composing the output from multiple extraction systems, perhaps deployed over multiple text databases, we can also answer more complex, multiple-relation structured queries, as illustrated by the following example.

EXAMPLE 3. *Consider two text databases, a collection of email messages (EM), and the archive of The New York Times (NYT) newspaper (see Figure 3). These databases embed information that can be used to answer an intelligence analyst's query asking for all recent communications between two people, including information regarding their political role. To answer such a query, we can use information extraction systems to extract the* Communications *relation of Example 1 from EM and a* Political(Person, Position) *relation from NYT. So the analyst's query can be expressed in SQL as* `SELECT * FROM Communications C, Political P WHERE C.Person = P.Person`. *For Communications, we extract tuples such as ⟨Carter, Khaled Mashaal⟩, indicating that Carter met with Khaled Mashaal; for Political, we extract tuples such as ⟨Carter, Former U.S. President⟩, indicating that Carter has been a U.S. President. After joining all the extracted tuples, we can construct the information sought by the analyst.*

To process multiple-relation queries, we identify a variety of *join execution algorithms* that are adaptations of their relational world counterparts. Specifically, we explored three join algorithms that are naturally available in the context of text databases [14].

Independent Join: Our first algorithm joins two relations by first independently extracting their tuples and then joining them to produce the final join result. This algorithm does not exploit any knowledge from the extraction of one relation to influence the extraction of the other relation. For example, in Figure 3 we extract and join the *Communications* and *Political* relations following an independent join algorithm.

Outer/Inner Join: An alternate join execution algorithm resembles an *index nested-loops* join [18] and uses extracted tuples from the "outer" relation (e.g., tuple ⟨Carter, Khaled Mashaal⟩ for *Communications*) to build keyword queries to retrieve documents and guide the extraction of the "inner" relation (e.g., this algorithm may build query [*Carter*] to retrieve documents from which to extract *Political* tuples that will join with the *Communications* tuple).

Zig-Zag Join: Yet another alternate join execution algorithm that we consider follows a "zig-zag" execution. Specifically, this algorithm fully interleaves the extraction of the two relations in a binary join: starting with one tuple extracted for one relation (e.g., ⟨Carter, Khaled Mashaal⟩ for *Communications*), this algorithm retrieves documents—via keyword querying on the join attribute values—for extracting the second relation. In turn, the tuples from the second relation are used to build keyword queries to retrieve documents for the first relation, and the process iterates, effectively alternating the role of the outer relation of a nested loops execution over the two relations.

2.4 Selecting a Query Execution Strategy

Query processing, as discussed above, involves analyzing a rich space of candidate execution plans by seamlessly combining extraction systems along with appropriately chosen document retrieval strategies and join algorithms. A critical observation is that the choices of information extraction systems, document retrieval methods, and join algorithms influence the two main properties of a query execution discussed above, namely, execution efficiency and output quality. Thus, candidate execution strategies may differ in their efficiency and output quality, and no single query execution strategy may strictly dominate; in fact, which execution strategy among all candidates is the best option depends on user-specific requirements.

Following relational query optimization, we built a query optimizer that explores a space of candidate execution strategies for a query and selects a final execution strategy in a principled, cost-based manner. To compare candidate execution strategies, we characterize each execution strategy in terms of its execution efficiency and output quality. The execution efficiency of an execution strategy is naturally a function of the total time to run the strategy. The output quality of an execution strategy can be characterized using different metrics, such as precision and recall (see Section 2.1); alternatively, we can measure quality simply in terms of the output composition, namely, in terms of the number of *good* and *bad* tuples among the extracted tuples, or based on any other metric that uses these values. We explored such quality metrics in [12, 13, 14] and presented techniques to predict them, which we briefly review next in Section 3. Upon estimating the characteristics of the candidate execution strategies, which execution strategy is appropriate for a query depends on user-specific needs. As an important feature of our query processing approach, we explore a variety of query paradigms to capture user-specific requirements, which we discuss later in Section 4.

3. ESTIMATING QUERY EXECUTION CHARACTERISTICS: AN OVERVIEW

The candidate execution strategies for a query may differ substantially in their execution efficiency and output quality. The choice of execution strategy for a query heavily depends on the nature of the underlying text database. For instance, text databases that are *dense* in the information to be extracted may be good candidates for using *Scan* for document retrieval; on the other hand, text databases with sparse coverage of the information to be extracted may be better candidates for using *PromD* or *Filtered Scan*. Thus, to estimate the characteristics of an execution strategy, we need to identify key database-specific factors. An important challenge is that, unlike in the relational world, where we have histograms or statistics to estimate the necessary query execution properties, here we do not know a priori the database characteristics. Naturally, processing an entire text collection with all available information extraction systems in order to gather necessary database-specific statistics is not feasible or desirable for large text databases (e.g., for the Web at large), and so we need to build effective alternative methods to gather these statistics.

Estimating the execution time of a query execution strategy is relatively simple, using statistics such as the average time to retrieve, filter, and process a document, along with the total number of documents expected to be retrieved and processed [12, 13, 14]. These statistics vary depending on both the information extraction systems of choice (e.g., an extraction system that constructs a complete syntactic parse tree of the input documents is likely to be slower than an extraction system that returns tuples based on, say, the frequency of entity co-occurrences) and the document retrieval strategies (e.g., the time to retrieve and filter a document using *Filtered Scan* is likely to be higher than the time to retrieve a document using *Scan*). So, we must consider both these factors when deriving the database-specific statistics.

Estimating the output quality of a query execution strategy, on the other hand, is a more challenging task. We designed a sampling-based estimation method that considers an execution strategy "as a whole" and identifies various database-specific statistics for the entire strategy [12]. Specifically, for each information extraction system and each document retrieval strategy, we derive statistics on the average number of tuples and of *good* tuples that the extraction system generates after processing a document retrieved using the document retrieval strategy of choice. During estimation, these statistics can be extrapolated to the number of documents that the strategy will process and we can then estimate the expected output quality of the execution strategy. To derive these salient database-specific statistics, a critical observation is that document retrieval strategies conceptually divide a text database into disjoint sets of documents: the (tuple-rich) documents that match the *PromD* queries, and the rest of the documents, which are less likely to result in useful extracted tuples. Using stratified sampling, we can derive the necessary statistics from each set [12]. Given an appropriately constructed document sample, most of the database statistics can be automatically derived. However, a key new challenge arises when gathering statistics on the average number of *good* tuples. We will discuss this challenge later in Section 4.

The notion of output quality of an execution strategy is novel to this query optimization problem. Building an effective query optimization approach requires not only an understanding of how query execution strategies—as a single unit—differ, but also an in-depth understanding of the impact of each component of an execution strategy on the overall execution. With this in mind, we performed an in-depth study of query executions in an extraction-based scenario. As argued earlier, query optimization involves making choices for three important components, namely, information extraction systems, document retrieval strategies, and join algorithms. Therefore, knowing how these components work, both stand-alone and together, is crucial to query processing design and comprehension. In particular, when studying each component of the strategy our goal is to capture the impact of a component (and the parameters thereof) on the overall execution characteristics. Towards this goal, we built analytical models for each component that provide a concise view of its behavior. Specifically, we derived models for the output quality as a function of the information extraction systems and their parameters (e.g., knob settings), the document retrieval strategies, as well as the join algorithms. By understanding the impact on the execution time and output quality of a query execution strategy's components, we can also understand—and thus, predict—the characteristics of the overall query execution strategy [13, 14].

To summarize, given a structured query we explore the space of candidate execution strategies as discussed in Section 2. We estimate the key characteristics of each of the candidate strategies using the techniques reviewed above. Finally, to guide the choice of appropriate query execution strategies, we consider the user-specified preferences for execution efficiency and output quality.

4. DISCUSSION

Traditional relational optimizers focus on minimizing the time to execute a given query. In contrast, a query optimizer for processing structured queries over text databases must consider both execution efficiency *and* output quality. Thus, designing key components for a query optimizer in the context of text databases introduces interesting research directions, some of which we discuss next.

4.1 Automated Quality Evaluation

At the heart of the estimation processes discussed above lies the critical task of determining whether an observed tuple is correct or not. This task is important for two purposes: (a) to evaluate the output of the extraction systems and, in turn, the performance of the query optimizer during evaluation, and (b) to derive database-specific statistics used during query processing. To carry out this task, manually inspecting each tuple is, of course, tedious and prohibitively time-consuming.

As one possible automated verification approach, we can resort to using external *gold sets* to identify all the good tuples among a set of extracted tuples; tuples that are present in the gold set are good tuples, and the rest are bad tuples. However, in many real-life applications the extraction system needs to extract previously unseen tuples that simply do not appear in any gold set.

Earlier work has looked into the problem of verifying a tuple by gathering evidence from a given text database. In general, these approaches assign a confidence score to a tuple based on this evidence [1, 7, 17] and, using an appropriate threshold for the confidence score, we can determine whether a tuple is good or bad. In our earlier work [12, 13, 14], we also resorted to a partially automated verification approach. Specifically, we first manually define a small number of "high-precision" natural-language patterns for each relation. Then, to decide whether an extracted tuple is good or not, we instantiate the high-precision patterns for the relation with the attribute values for the tuple, and search for instances of the instantiated patterns using the database's search interface. We consider the presence of an instantiated pattern in a database as strong evidence of tuple correctness.

EXAMPLE 4. *Consider the task of extracting a* Headquarters(Company, Location) *relation from a text database, where a tuple $\langle c, \ell \rangle$ indicates that c is a company whose headquarters are located in ℓ. To verify tuples of the* Headquarters *relation, one possible template we can define is "$\langle$LOCATION$\rangle$-based $\langle$ORGANIZATION$\rangle$. Thus, the tuple $\langle$Microsoft Corp., Redmond$\rangle$, results in an instantiated pattern "Redmond-based Microsoft Corp.," which we issue as a query to the database's search interface.*

While this template-based verification task allows for some automation in the tuple-verification process, generating *enough* high-precision patterns for each relation is tedious and may be difficult to achieve. And using a restricted set of patterns is undesirable as it may result in few or no tuples being marked as correct and, in turn, may introduce some bias in the derived output quality.

Automated verification at a large scale is therefore a hard problem. Ideally, we would like to automatically verify a tuple based on the evidence gathered from the underlying database in a reliable manner. Approaches that leverage small-scale manual annotation (e.g., using on-demand services like Amazon's Mechanical Turk[2]) to improve scalable automatic verification techniques [21] are promising directions for improving the current state of the art.

[2]`http://www.mturk.com`

4.2 Query Formulation: Capturing User Needs

As argued earlier, candidate execution strategies for a given query may differ substantially in their execution efficiency and their output quality. Furthermore, different users may have different preferences regarding the desired execution efficiency and output quality of a query execution. Based on this observation, we can design query paradigms that reflect whether users are after output quality, efficiency, or an appropriate balance between these two query-execution characteristics. We briefly review some query processing paradigms that we have considered.

Threshold-based Model: Sometimes users may be after some minimum output quality, for which they desire a query execution that is as fast as possible. In [13, 14], we presented a query paradigm where users specify their preferences in terms of the minimum number of good tuples desired, as well as the maximum number of bad tuples that they are willing to tolerate. These thresholds then guide the query optimization process, which identifies query execution strategies that meet these user requirements and then picks the fastest among these candidate strategies. This relatively "low-level" query paradigm can be used as a building block for higher-level paradigms (e.g., users might request some minimum precision or recall, under some constraint on execution time, or some minimum value for a combination of precision and recall).

Efficiency-Quality "Mixture" Model: In [12], we presented a query paradigm where users can specify the desired balance between execution efficiency and output quality. Such "high-level" user preferences can allow users to explore a database without having to know the exact output quality thresholds as is required in the *Threshold-based Model* discussed above.

Score-based Model: Sometimes extraction system report each extracted tuple together with an extraction "score" that reflects the extraction system's confidence in the correctness of the extracted tuple. Given such score-based setting, users may be after a relatively small number of high-ranking tuples, where the tuple ranking is determined by the tuple confidence scores as exported by the extraction systems; furthermore, these tuples should be produced as fast as possible. In [15], we presented a query paradigm where users can specify the desired number of high-ranking tuples. This score-based model allows users to quickly receive a few desirable tuples, while avoiding uninteresting tuples that have low confidence scores.

The alternative query paradigms discussed above have relative strengths and weaknesses, particularly in terms of how natural and useful they are from a user's perspective. An important direction for future work is to conduct user studies to understand the relative merits of these paradigms, and also to help define additional ones that have not yet been explored.

5. CONCLUSION

In this paper, we presented an overview of our ongoing SQoUT project, with the general goal of processing structured queries over relations extracted from text databases. We identified and tackled a variety of problems that occur in our query optimization setting for information extraction. Specifically, we identified a rich space of query execution strategies and critical query execution characteristics, namely, efficiency and output quality. We also discussed methods to estimate these query execution characteristics, to build, in turn, a robust query optimizer for our text-based scenario. Our query optimizer takes into account user-specific requirements for execution efficiency and output quality, and chooses an execution strategy for each query in a principled, cost-based manner.

6. ACKNOWLEDGMENTS

This work has been performed in collaboration with AnHai Doan. This material is based upon work supported by a generous gift from Microsoft Research, as well as by the National Science Foundation under Grants No. IIS-0811038 and IIS-0643846. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Microsoft Research or the National Science Foundation.

7. REFERENCES

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

[2] E. Agichtein and L. Gravano. Querying text databases for efficient information extraction. In *ICDE*, 2003.

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

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

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

[6] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: An architecture for development of robust HLT applications. In *ACL*, 2002.

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

[8] 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.

[9] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. In *Natural Language Engineering*, 2004.

[10] 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.

[11] 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.

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

[13] A. Jain and P. G. Ipeirotis. A quality-aware optimizer for information extraction. *ACM Transactions on Database Systems*, 2009. To appear.

[14] A. Jain, P. G. Ipeirotis, A. Doan, and L. Gravano. Join optimization of information extraction output: Quality matters! In *ICDE*, 2009. To appear.

[15] A. Jain and D. Srivastava. Exploring a few good tuples from text databases. In *ICDE*, 2009. To appear.

[16] I. Mansuri and S. Sarawagi. A system for integrating unstructured data into relational databases. In *ICDE*, 2006.

[17] M. Paşca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and searching the world wide web of facts - step one: The one-million fact extraction challenge. In *WWW*, 2007.

[18] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[19] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, 2008.

[20] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *VLDB*, 2007.

[21] V. Sheng, F. Provost, and P. G. Ipeirotis. Get another label? Improving data quality and data mining using multiple, noisy labelers. In *KDD*, 2008.