Optimizing SQL Queries over Text Databases

Alpa Jain [#1], AnHai Doan [*2], Luis Gravano [#3]

[#] *Computer Science Department, Columbia University*
[1] alpa@cs.columbia.edu
[3] gravano@cs.columbia.edu

[*] *Department of Computer Sciences, University of Wisconsin-Madison*
[2] anhai@cs.wisc.edu

***Abstract*— Text documents often embed data that is structured in nature, and we can expose this structured data using information extraction technology. By processing a text database with information extraction systems, we can materialize a variety of structured "relations," over which we can then issue regular SQL queries. A key challenge to process SQL queries in this text-based scenario is efficiency: information extraction is time-consuming, so query processing strategies should minimize the number of documents that they process. Another key challenge is result quality: in the traditional relational world, all correct execution strategies for a SQL query produce the same (correct) result; in contrast, a SQL query execution over a text database might produce answers that are not fully accurate or complete, for a number of reasons. To address these challenges, we study a family of select-project-join SQL queries over text databases, and characterize query processing strategies on their efficiency and—critically—on their result quality as well. We optimize the execution of SQL queries over text databases in a principled, cost-based manner, incorporating this tradeoff between efficiency and result quality in a user-specific fashion. Our large-scale experiments—over real data sets and multiple information extraction systems—show that our SQL query processing approach consistently picks appropriate execution strategies for the desired balance between efficiency and result quality.**

I. Introduction

Real-world applications frequently rely on the information in large collections of text documents such as news articles, reports, and email messages. Text often embeds valuable structured data, such as who recommends selling which stocks, who has been hired by which corporation, or the number of people affected by a disease outbreak. In this paper, we consider the problem of effectively answering SQL queries over a collection of text documents.

Applications that benefit from structured data embedded in text arise in a wide range of domains. In disease control, suppose an E. coli outbreak has just occurred in a remote country; an epidemiologist may then want to find the Web-accessible archive of a reputable newspaper in that country, to compile statistics on E. coli outbreaks. In business intelligence, a law firm may want to know if its competitor X has recently worked with a company Y, and if so, in which cases; toward this goal, the firm extracts evidence of such cases from the competitor's Web site, recent news articles, and court filings. In scientific data management, a hydrologist may hypothesize that soil erosion near Stevenson City has seriously affected the downstream water quality of the Columbia River, using reports of soil erosion that mention Stevenson City from a variety of sources.

Unfortunately, despite the pervasiveness of these applications, none of the current solutions for addressing the above information needs is fully satisfactory. One solution is to rely on keyword search to retrieve documents that are relevant to the task at hand, and then manually identify the (structured) data of interest in the documents, a tedious process. An alternative is an "extract-then-query" solution, which first converts the text collection into a structured database by applying information extraction (IE) techniques, and then poses SQL queries over the extracted database. Consider again the E. coli example above. Using a suitably trained IE system, our epidemiologist might turn to the (previously unseen) Web-accessible archive of a reputable newspaper from the target country to extract relevant tuples of a relation $DiseaseOutbreaks(DiseaseName, Location, Year)$, where a tuple $\langle d, \ell, y \rangle$ indicates that there was an outbreak of disease d in location ℓ in year y. Then, the epidemiologist may pose SQL queries over the relation.

This solution can address expressive information needs, but has two important limitations. First, it often wastes substantial time extracting useless information, because only a relatively small number of articles in a text database might be useful to process a query. This is likely the case for the above E. coli query and the newspaper archive. Furthermore, the actual "slice" of the relations that is needed to answer a query (e.g., the tuples with $DiseaseName$ equal to E. coli in our example) may also be relatively small. Then, to answer a query in a timely manner, materializing the entire relations and processing all database documents is undesirable.

Another important limitation of the above solution is that it might be too slow for *urgent* information needs. Information extraction is well known to be computationally expensive [1] (e.g., applying IE to even a moderate-size collection can easily take many hours for some IE systems). So the above solution would be too slow for an epidemiologist who must act fast to control an infectious disease, or for a stock analyst who must respond to the market in a timely manner. In such cases, reliable but not exhaustive query results might be appropriate, as long as they are returned fast. On the other hand, when the need is not urgent (e.g., as in the law firm and hydrologist examples described earlier), users may want to receive exhaustive query results, for which they may be

willing to wait a relatively long time. In general, users often have preferences regarding the quality and run-time efficiency expected from the querying process, which the above rigid solution cannot accommodate.

To address the above limitations, in this paper we propose viewing this as a *query optimization* problem. Accordingly, we define a space of execution plans—which includes the extract-then-query approach—and a cost model that captures user preferences; we also develop a way to estimate plan costs and, correspondingly, to select the best plan. We show that query processing can be decomposed into a sequence of basic steps: retrieving relevant text documents, extracting relations from the documents, cleaning the extracted data, and assembling the cleaned data into the final query answers. Our problem is then to consider the desired user-specified balance between quality and execution efficiency, and choose a strategy accordingly, in a principled, cost-based manner.

In the rest of the paper, we elaborate on our solution to process SQL queries over text databases, called SQOUT (for "SQL queries over unstructured text databases"). Specifically, our contributions are as follows:

- We establish that it is feasible to execute SQL queries over text on-the-fly, with IE systems (Section III).
- We show how IE systems, document retrieval strategies, and data cleaning operators—components often studied in isolation in the past—can be seamlessly integrated to form a space of execution plans for SQL queries over text databases (Section III).
- We develop a cost model that exposes the tradeoff between efficiency and result quality, and enables users to flexibly adjust their preferences (Section IV).
- We define the statistics necessary to estimate both execution efficiency and result quality, and show how to obtain such statistics efficiently and effectively from text databases (Sections IV and V).
- We evaluate our approach with extensive experiments over a real-world data set and using state-of-the-art IE systems. Our results demonstrate that our approach consistently picks execution strategies appropriate for the desired balance between efficiency and result quality (Section VI).

II. Related Work

The problem of information extraction from unstructured text has received significant attention (see [2] for a survey). This research has mostly focused on improving IE accuracy (e.g., [3, 4, 5, 6]). The related problem of "wrapper induction" from template-based Web pages has also been studied extensively (e.g., in [7]). In addition, recent work has considered information extraction from the entire Web (e.g., [8]). Recently, [9] introduced a novel technique to leverage existing structured databases for information extraction.

The extraction of information from text is computationally expensive, because of the complex text processing generally required. Approaches have been proposed for improving IE *efficiency*, to avoid processing *useless* documents that are not relevant to the extraction task at hand and, instead, focus on *promising* documents that are likely to contain information to be extracted. One such approach, QXtract [1], uses machine learning to derive keyword queries that identify documents rich in target information. For example, QXtract might derive keyword query [*foodborne AND pathogens*] to retrieve promising documents for a *DiseaseOutbreaks* relation.

Recent approaches have also looked into handling extracted data using probabilistic databases, which model data uncertainty by assigning probabilities to tuples. Specifically, [10] showed how to store the output from an extraction system based on Conditional Random Fields in a probabilistic database, after appropriately deriving a probability for each extracted tuple. Recently, [11] introduced a structured query processing system that extracts data from documents in an offline step and relies on probabilistic databases to process user queries. In contrast, our approach handles the extracted data uncertainty during query processing, by resolving extraction errors and data conflicts using data cleaning techniques (Section III); however, the probabilistic approach in [11] can also be applied towards the same goal, which we will investigate as part of our future work.

The "on-the-fly" extraction nature of our work is somewhat reminiscent of work on question answering (e.g., [12]), but this research focuses on natural language questions, not on SQL queries, and has not considered the efficiency-quality tradeoffs in depth. Other relevant work has focused on specialized scenarios or settings: examples include extraction over dynamic data [13] and combining multiple extraction programs using declarative programs (e.g., UIMA [14], GATE [15], Xlog [16]). Finally, other important extraction aspects besides optimization, such as extraction architecture [17] and schema discovery [18], have also been addressed in the literature.

Closest to this paper is the analysis in [19, 20], which considers (among others) the problem of optimizing the document retrieval strategy for a single IE system S. Our current work is related to [19, 20], but differs from it in several crucial aspects. First, [19, 20] has studied only a single-IE-system scenario; however, to answer practical SQL queries, we must often employ multiple IE systems and relations, which raises novel processing challenges that we will discuss and address in this paper. Second, [19, 20] assumes that S is perfect in that it produces all—and only—correct tuples; we remove this assumption, and develop a principled method to model extraction errors. Finally, unlike [19, 20], we do not optimize for a pre-specified target recall, but consider the goal of balancing recall, precision, and execution time in a flexible manner.

A preliminary, 3-page version of this paper appears in [21].

III. Problem Statement and Execution Plans

Consider a document collection D, which has been identified for a specific application (e.g., D might be the Web-accessible archive of a newspaper for our E. coli example of Section I). Then our goal is to answer SQL queries over D using IE systems. From the many types of IE systems (see

Section II), we focus on the common type that takes as input a document and produces as output tuples of the relation for which the system was trained. For instance, consider an IE system trained for a relation *Headquarters(Company, Location)*, where a tuple $\langle c, \ell \rangle$ indicates that *c* is a company whose headquarters are located in ℓ. Then, from the text snippet, "Redmond-based Microsoft announced today ..." of a document, this IE system may extract tuple ⟨Microsoft, Redmond⟩.

In this paper, we focus on the problem of processing SQL queries using such IE systems:

Problem Statement 1: Consider a text database D with a boolean search interface [22], and n "base" relations $R_1, \ldots, R_n$ defined over D. Each base relation R_i can be extracted from D using one or more IE systems. We assume that all base relations $R_1, \ldots, R_n$ share the same primary key K and no other attributes, so each relation might be regarded as contributing additional attributes of the entities identified by the key attributes in K. We define a view $V = (R_1 \bowtie \cdots \bowtie R_n)_K$ as the natural outerjoin of the base relations $R_1, \ldots, R_n$ over the K attributes. We consider SQL selection-projection queries over V with selection condition conjuncts of the form $A = t$, where A is a textual attribute and t is a constant. Then, given such a SQL selection-projection query Q over V, our goal is to identify an execution strategy for Q that meets the desired efficiency and result quality requirements.

Following this problem statement, consider an IE system trained to extract the *Headquarters* relation above, as well as another system trained for an *Executives(Company, CEO)* relation, where a tuple $\langle c, e \rangle$ indicates that person e is the CEO of company c. We can then define a view *CompanyInfo(Company, Location, CEO)* as the natural outerjoin of these two "base" relations and express queries such as:

```
Q1:  SELECT Company, CEO FROM CompanyInfo
     WHERE  Location = 'Redmond'
```

This problem setting is appropriate as a first step in our investigation of the general problem of answering SQL queries over text databases, and is already useful in practice (see Section VI-B). In Section VII, we discuss how to generalize our approach further.

Given a SQL query Q as described above, we now discuss the space of execution plans for Q. To evaluate Q over database D, we need to:

(1) Select an IE system E_i for each base relation R_i.
(2) Select a document retrieval strategy X_i for each E_i.
(3) Use each strategy X_i to retrieve from database D a set of documents P_i, then process P_i using IE system E_i to obtain a relation instance r_i.
(4) Clean the relations $r_1, \ldots, r_n$ by reconciling references and eliminating data inconsistencies.
(5) Execute Q over view $v = (r_1 \bowtie \cdots \bowtie r_n)_K$.

We now elaborate on these steps.

Step (1), Selecting IE Systems: Given a query Q, we first determine the base relations that are needed for Q, based on the attributes in the SELECT and WHERE clauses. This, in turn, identifies the IE systems that are relevant to Q. If more than one IE system is available for a certain base relation, then which system we choose will depend on their efficiency and quality as well as on the user preferences (see Section IV).

Step (2), Selecting Retrieval Strategies: For each IE system E, we select a way to retrieve the documents that E will process. (This is analogous to selecting an access path in an RDBMS.) We consider four representative strategies based on the existing literature [19, 20]:

Scan: We sequentially scan the collection and feed each document to E. This strategy produces the complete query results for E, but makes E process *all* documents.

Const: We find constants (if any) in query Q, then feed E only the documents that contain those constants. For example, the constant "Redmond" in query $Q1$ above can be used as a keyword query to retrieve only documents with this word for *Headquarters*, on the ground that documents without it could not contribute useful tuples for that base relation. *Const* thus avoids processing all documents, and its efficiency is determined by the *selectivity* of the constants in the query.

PromD: Given E, *PromD* employs the learning method of QXtract [1] to derive keyword queries (e.g., *[based AND shares]* for *Headquarters*) and feed E only "promising" documents that contain these keywords. Thus, like *Const*, *PromD* also avoids processing all documents. However, *PromD* may miss some answer tuples and is IE-system-dependent, in that the keyword queries that it derives—and thus the set of documents retrieved—depend on the IE system E.

PromC: This strategy combines *Const* and *PromD* by ANDing their queries, to retrieve documents that are promising and, at the same time, satisfy the predicates of the SQL query. For example, for $Q1$ we combine query [*based AND shares*] from *PromD* with query [*Redmond*] from *Const* to obtain query [*based AND shares AND Redmond*]. Similarly to *PromD*, *PromC* avoids processing all documents, but may miss some answer tuples and is IE-system-dependent.

The following example illustrates these strategies:

Example 3.1: Figure 1 shows a possible execution for query $Q1$. This execution employs two document retrieval strategies, PromC *for* Headquarters *and* Scan *for* Executives. PromC *issues queries such as [based AND shares AND Redmond] to the search interface of the database, to retrieve promising documents. After feeding each of these documents to the* Headquarters *IE system, we obtain tuples such as ⟨Microsoft, Redmond⟩. Note that ⟨Microsoft Corp., New York⟩ was (erroneously) extracted in this step by the (often-less-than-perfect) extraction system. To extract* Executives, Scan *retrieves all documents exhaustively, one at a time, and feeds them to the extraction system for this relation, to extract tuples such as ⟨Microsoft Corp., Bill Gates⟩.* □

When query Q has a selection condition associated with a base relation, then all four retrieval strategies are applicable

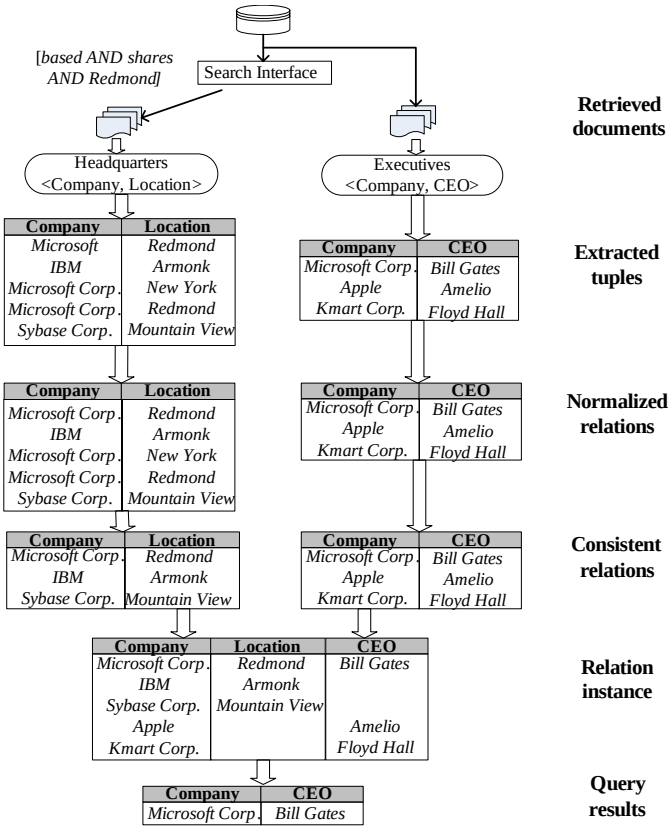

Fig. 1. Stages in the execution of query Q1.

for the relation (e.g., this is the case for *Headquarters* in query $Q1$). However, in such case *Const* is preferable to *Scan*: *Const* returns all documents that can contribute to the query result and never processes more documents than *Scan*. Similarly, *PromC* is preferable to *PromD*. Hence, in this case we will consider only *Const* and *PromC* for retrieving documents for the base relation. If, in contrast, a base relation has no associated selection condition, then only *Scan* and *PromD* apply.

Step (3), Retrieving and Processing Documents: Upon identifying the documents to process for each IE system E, we retrieve these documents from disk, feed them to E, and write the extracted tuples to an auxiliary database, for data cleaning and further query processing.

Step (4), Cleaning Extracted Data: The extracted tuples are often noisy: typographical errors commonly occur in text, entity references might be far from uniform, IE systems are error-prone, and text databases might sometimes include contradictions, which would be problematic even in the absence of extraction errors.

To address this problem, we first reconcile entity references. In principle, we can apply any existing reference reconciliation technique (variants of which are also known as record linkage, among other terms [2, 23]). For now, we rely on the domain-independent technique of [24][1], which is effective and can be implemented robustly within a RDBMS. We apply this technique to the values of the key attributes K of all our extracted relations collectively. For example, in Figure 1, we conclude that "Microsoft" and "Microsoft Corp." refer to the same company, in both *Headquarters* and *Executives*. We then pick an arbitrary canonical representation for the entity from among the associated values. For now, we choose the longest string in each group, so that "Microsoft Corp." represents "Microsoft."

We then resolve semantic inconsistencies due to extraction errors and inherent inconsistencies in the data. Rather than attempting to do this in a sophisticated way, we focus on simple majority arguments. Specifically, we group the extracted tuples in each relation (after reference reconciliation) by their key attributes and choose the most frequent value in each group, individually for each non-key attribute. The rationale is that text databases often exhibit substantial redundancy [3, 4] and errors are likely to be outnumbered by correct facts. Of course, alternate approaches could be used, especially in the presence of domain knowledge, and such approaches could be immediately substituted in our algorithm.

Step (5), Processing Query over Extracted View: As a final step, we execute the SQL query over the cleaned, materialized view generated after the above steps.

To summarize, we explore a space of candidate execution plans by seamlessly combining multiple IE systems along with appropriately chosen document retrieval strategies, and data cleaning techniques. Next, we discuss our cost model, to characterize query execution plans in terms of their efficiency and result quality.

IV. Efficiency-vs.-Quality Cost Model

To compare alternate execution strategies for a query Q over a database D, we will define the *goodness* of a query execution as a function of its efficiency and result quality. For this, we define *efficiency* as follows:

Definition 4.1: **[Efficiency]** *The **efficiency** of a query execution S over a text database D, $E(S, D)$, is the inverse of the execution time of S over D.* □

The goodness of an execution strategy is also a function of its result quality. So, we need to characterize the "ideal" result for Q over D, $Ideal(Q, D)$. This hypothetical ideal result consists of all the correct tuples for Q that could be derived from database D by "perfect" IE systems (i.e., IE systems with perfect precision and recall), using the *Scan* document retrieval strategy (i.e., by processing all database documents exhaustively), and by fully cleaning the data. Based on *Ideal*, we define *precision* and *recall* as follows:

Definition 4.2: **[Precision and Recall]** *Consider an execution strategy S for a query Q over a text database D, and let R be the results that S produces. We define the **precision***

[1] `http://pages.stern.nyu.edu/~panos/datacleaning`

*of S over D as $P(S,D) = \frac{|R \cap Ideal(Q,D)|}{|R|}$ and the **recall** of S over D as $R(S,D) = \frac{|R \cap Ideal(Q,D)|}{|Ideal(Q,D)|}$.* □

Of course, $|Ideal(Q,D)|$ is prohibitively expensive to compute for any large database D, since this "computation" would necessarily involve substantial human effort (e.g., no "perfect" IE systems exist). So we need to avoid computing *Ideal* when we characterize the goodness of an execution strategy. A key observation is that, to choose between strategies S_1 and S_2, with goodness values g_1 and g_2, respectively, we do not need to consider the exact values for g_1 and g_2. Instead, it suffices to inspect their ratio $\frac{g_1}{g_2}$ to decide which strategy is best. With this observation in mind, we define goodness—and quality, a metric based on which goodness is specified—so that the $|Ideal(Q,D)|$ constant in the denominators of both g_1 and g_2 "cancels out" when we consider goodness *ratios*. Specifically, we combine precision and recall into a single metric by computing their geometric mean, as follows[2]:

Definition 4.3: **[Quality]** *Consider an execution strategy S for a query Q over a text database D. We define the **quality** of S over D as $Q(S,D) = (P(S,D) \cdot R(S,D))^{1/2}$.* □

Our definition of quality is similar in spirit to the F_1-measure [22] used in information retrieval, yet our metric is easier to estimate in the realistic scenario where the *Ideal* results are not available, as we will see.

To finally define goodness, we note once again that, ultimately, the choice of the right balance between efficiency and quality is user-specific: sometimes users desire high-quality exhaustive query results, even if query execution takes a relatively long time; some other times, users are after some "quick and dirty" answers. We capture this desired efficiency-quality balance with a (user-specified) query processing parameter w, ranging from 0, to privilege efficiency, to 1, to privilege result quality. In turn, we use parameter w to characterize the overall *goodness* of a query execution, as follows:

Definition 4.4: **[Goodness]** *The **goodness** of a query execution S over a text database D for user-specified parameter w is $G_w(S,D) = Q(S,D)^w \cdot E(S,D)^{(1-w)}$.* □

The goodness function, as defined above, captures (a weighted version of) the result quality per execution time unit, and allows users to weigh efficiency and result quality appropriately. Several alternate query paradigms can also be used to reflect the user preferences. In particular, we could follow the relational model and define a `STOP-AFTER-K-TUPLES` paradigm, where the objective of an execution is to derive only K answer tuples as efficiently as possible. Alternatively, a `STOP-AFTER-TIME-T` paradigm could indicate that the query execution must finish within T time units and produce the highest quality results possible within that time frame. In this paper, we focus on finding query execution strategies with highest *goodness*, as in Definition 4.4; we will investigate the above alternate query paradigms—and others—in our future research work.

[2]Our definition of quality weighs precision and recall equally. This definition can be easily generalized, though, to allow for different (user-specified) weights for precision and recall.

Symbol	Description
$\|$*Docs(E, X, D)*$\|$	number of documents retrieved from D by retrieval strategy X for extraction system E
RTime(E, X, D)	average time to retrieve a document from D using X for extraction system E
ETime(E, X, D)	average time to run E on a document retrieved from D using X
$\|$*T(E, X, D)*$\|$	average number of tuples that E extracts from a document retrieved from D using X
$\|$*C(E, X, D)*$\|$	average number of *correct* tuples that E extracts from a document retrieved from D using X
$\|Join(\mathcal{E},D)\|$	cardinality of the join of relations $R_1,\ldots,R_n$, extracted as specified in $\mathcal{E} = \{(E_1,X_1),\ldots,(E_n,X_n)\}$, where E_i is an extraction system for R_i, with document retrieval strategy X_i $(i = 1,\ldots,n)$

TABLE I

DATABASE-SPECIFIC STATISTICS.

To estimate the properties of each candidate execution strategy, we use the statistics in Table I for an IE system E with an associated document retrieval strategy X (i.e., *Scan*, *PromD*, *Const*, or *PromC*) over a database D. (In Section V, we describe how we gather the Table I statistics.) Note that the behavior of X may be dependent on the choice of IE system E (e.g., as is the case for *PromD*; see Section III), so statistics such as $|$*Docs(E, X, D)*$|$ depend not only on X and D, but also on E (Table I).

Single-Relation Queries: Consider a query execution strategy S over a database D, and assume that S involves only one extraction system, E, with an associated retrieval strategy X. Our efficiency analysis ignores the post-extraction processing of Steps (4) and (5): the time required for these steps is directly proportional to the number of tuples extracted, and this time is (indirectly) accounted for by the analysis for Step (3), which considers the number of documents processed. Then, we estimate the execution time for S over D as:

$$|Docs(E,X,D)| \cdot (RTime(E,X,D) + ETime(E,X,D)) \quad (1)$$

The estimated efficiency of S, $E_{est}(S,D)$, then follows directly from this expression. We estimate precision as:

$$P_{est}(S,D) \quad = \quad \frac{|C(E,\ X,\ D)|}{|T(E,\ X,\ D)|} \quad (2)$$

To estimate recall, we face the challenge that $|Ideal(Q,D)|$ is unknown in the denominator of the definition of recall (Definition 4.2) and computing it would require a prohibitively large human effort[3]. We observe that the $|Ideal(Q,D)|$ factor affects the *absolute* goodness value (Definition 4.4) of the executions, but not their relative standing, as discussed earlier in this section. So we can safely replace recall with $R_{est}(S,D) = |C(E,\ X,\ D)| \cdot |Docs(E,\ X,\ D)|$: this value is the

[3]This problem is analogous to computing the exact recall for information retrieval techniques, an impossible proposition over large data sets.

estimated number of correct tuples extracted by E and X from D. Equivalently, using Equation (2),

$$R_{est}(S, D) = P_{est}(S, D) \cdot |T(E, X, D)| \cdot |Docs(E, X, D)| \quad (3)$$

Multiple-Relation Queries: We now focus on queries that join multiple base relations. Specifically, consider a query execution strategy S over a database D, involving extraction systems $E_1, \ldots, E_n$, with associated document retrieval strategies $X_1, \ldots, X_n$, respectively. We estimate the execution time for S as the sum of the time to generate each base relation, computed using Equation (1):

$$\sum_{i=1}^{n} |Docs(E_i, X_i, D)| \cdot (RTime(E_i, X_i, D) + ETime(E_i, X_i, D)) \quad (4)$$

Estimating precision for strategies that involve multiple extraction systems—and hence, extract multiple base relations—is challenging. In absence of additional information, we can assume independence for the extraction errors across the relations and define precision based on the values for the individual relations, as $P_{est}(S, D) = \prod_{i=1}^{n} \frac{|C(E_i, X_i, D)|}{|T(E_i, X_i, D)|}$. However, we observed experimentally that this assumption can result in underestimating the true precision, so we "boost" the single-relation precision before proceeding with the independence assumption and define:

$$P_{est}(S, D) = \prod_{i=1}^{n} \frac{2 \cdot P_i}{1 + P_i} \quad (5)$$

where $P_i = \frac{|C(E_i, X_i, D)|}{|T(E_i, X_i, D)|}$ and $n \geq 2$.

Regarding recall, we proceed as in the single-relation case (Equation (3)) and substitute the total number of correct tuples generated by S. Thus,

$$R_{est}(S, D) = P_{est}(S, D) \cdot |Join(\{(E_1, X_1), \ldots, (E_n, X_n)\}, D)| \quad (6)$$

where $|Join(\{(E_1, X_1), \ldots, (E_n, X_n)\}, D)|$ is the number of tuples in the (inner) join of the relations extracted from D by extraction systems $E_1, \ldots, E_n$, with respective document retrieval strategies $X_1, \ldots, X_n$. We estimate the join cardinality by appropriately scaling, in turn, the cardinality of the corresponding join computed over a document sample, as we describe in Section V-B.

By substituting E_{est}, P_{est}, and R_{est} for E, P, and R in Definitions 4.3 and 4.4, we can obtain a "proxy" goodness value whose computation does not involve knowledge of $|Ideal(Q, D)|$, as discussed. As we argued above, the strategy with highest "proxy" goodness also has highest actual goodness, so we can optimize the execution of a query effectively and without human intervention.

V. Deriving Database Statistics

We now discuss how to estimate the statistics in Table I, which requires that we resort to document sampling in a preprocessing step for SQOUT. We determine the sample size via the sequential sampling method in [25, 26] to achieve an appropriate confidence level on our statistics estimates. In our experiments, we target a 95% confidence level, which requires independent sample sizes of up to 5% of the collection size. In general, we can compute the statistics for *Scan* and *PromD* offline, while *Const* and *PromC*, which depend on query-specific constants, require some processing at query time.

A. Document Retrieval Statistics

Consider an IE system E and a database D. The document retrieval statistics on which we rely are:

Number of documents retrieved: *Scan* always retrieves all database documents, so |*Docs(E, Scan, D)*| = $|D|$. *PromD* retrieves only the documents that match its associated queries, so we derive |*Docs(E, PromD, D)*|—offline—by issuing the *PromD* queries associated with extraction system E to the database D and counting the unique documents among the query matches. Unlike *Scan* and *PromD*, *Const* and *PromC* retrieve documents in a query-specific manner, since these retrieval strategies rely on the constants in the SQL query. We can efficiently derive |*Docs(E, Const, D)*| at query-processing time by issuing the SQL constants as queries to the database (e.g., [*Redmond*] for our example SQL query Q1) and extracting the number of matches that the queries generate, without retrieving any documents.

To compute |*Docs(E, PromC, D)*|, we know (1) the number of matches—computed offline—for the *PromD* queries and (2) the number of matching documents—computed at query-processing time—for the *Const* queries. The number of documents that match the conjunction of each *PromD* query and each *Const* query is unknown, however, and issuing these queries to the database at query-processing time would be too expensive. Instead, we analyze—in an offline training step—whether database constants associated with each relation attribute tend to strongly co-occur with the *PromD* query keywords in the database documents.[4] If this is the case, then we assume that each *PromC* query, formed by the conjunction of a *PromD* query and a *Const* query, returns the minimum of the number of matches of each individual query. Otherwise, we estimate |*Docs(E, PromC, D)*|—by assuming independence—as |*Docs(E, Const, D)*| · |*Docs(E, PromD, D)*|/$|D|$.

Retrieval time: We estimate *RTime(E, Scan, D)*—offline—by computing the time required to retrieve each document in a random sample from D. We use this same estimate for *RTime(E, Const, D)*, since the *Const* queries tend to be short and hence relatively inexpensive to process. For *PromD*, the retrieval time is estimated—also offline—over a sample of the documents that match the *PromD* queries associated with E. We use this same estimate for *RTime(E, PromC, D)*, since the *PromC* queries tend to be only slightly longer than the *PromD* queries, and hence their retrieval-time behavior is similar.

B. Extraction Statistics

Consider an extraction system E with an associated document retrieval strategy X over a database D. To estimate

[4] This analysis relies on a paired t-test [27] and other statistical methods, which we do not discuss further because of space limitations.

the extraction statistics in Table I, we consider the (tuple-rich) documents that match the *PromD* queries for E separately from the rest of the documents: these two subsets of D are likely to exhibit substantially different values for the extraction statistics. Specifically, we perform stratified sampling [27] over D, with one stratum, P_D, corresponding to the *PromD* matches in D and the other stratum corresponding to the remainder of the documents in D. Figure 2 shows these strata for D. For *Scan*, the extraction statistics below are computed over both strata, namely P_D and $D - P_D$. For *PromD* and *PromC*, we compute the statistics over P_D only. Finally, for *Const* we use both P_D and $D - P_D$: conceptually, *Const* retrieves all of the *PromC* documents, which are included in P_D, plus some additional documents, which are in $D - P_D$ (see Figure 2). We compute the extraction statistics for *Const* based on |*Docs(E, PromC, D)*|, |*Docs(E, Const, D)*|, and the stratified sample statistics.

For instance, when deriving |*T(E, Scan, D)*| as described above, we retrieve documents until we reach a confidence of 95% in the estimated value. Then, if E extracted, on average, say t_D tuples from the $D - P_D$ stratum and t_{P_D} tuples from the P_D stratum, we compute |*T(E, Scan, D)*| as a weighted average of t_D and t_{P_D}, with weights $|D - P_D|$ and $|P_D|$, respectively. The other statistics are handled analogously.

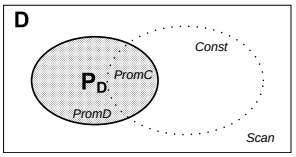

Fig. 2. Database strata for sampling.

Extraction time and number of extracted tuples: We estimate *ETime(E, X, D)* and |*T(E, X, D)*| by running E over the document sample(s) for X (see above).

Join cardinality: $|Join(\mathcal{E}, D)|$, for $\mathcal{E} = \{(E_1, X_1), \ldots, (E_n, X_n)\}$, is the cardinality of the join of the extracted relations $R_1, \ldots, R_n$, where R_i is extracted using extraction system E_i and document retrieval strategy X_i $(i = 1, \ldots, n)$. To estimate this cardinality, we handle the *Scan* and *PromD* cases offline: we first focus on the appropriate document sample for each E_i, X_i pair, as discussed above, then extract the relation tuples, $Sample(E_i, X_i, D)$, from each document sample, and finally join the extracted relations to determine the sample join cardinality j_s. Then, we produce the $|Join(\mathcal{E}, D)|$ estimate as [28]:

$$|Join(\mathcal{E}, D)| = j_s \cdot \prod_{i=1}^{n} \frac{|T(E_i, X_i, D)| \cdot |Docs(E_i, X_i, D)|}{|Sample(E_i, X_i, D)|}$$

For *Const* and *PromC*, we estimate the join cardinality online, from the corresponding join cardinality values j_s for *Scan* and *PromD*, respectively, as discussed above.

Verifying tuple correctness: To compute |*C(E, X, D)*|, we need to decide whether extracted tuples are correct, for which we could manually inspect the text database to make a decision. This manual inspection is, of course, tedious and prohibitively time-consuming, so we resort instead to a more automatic "verification" approach. Specifically, we first manually define a small number of "safe" natural-language patterns for each relation, such as "⟨LOCATION⟩-based ⟨ORGANIZATION⟩" for *Headquarters*. To decide whether an extracted tuple is correct or not, we instantiate the safe 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. Intuitively, we automate the process of deriving statistics by simulating a manual evaluation of tuples. For example, tuple ⟨Microsoft Corp., Redmond⟩ results in an instantiated pattern "Redmond-based Microsoft Corp." We consider the presence of an instantiated pattern in a database as strong evidence of tuple correctness, from which we derive a conservative approximation of the number of correct tuples extracted by an extraction system.

VI. Experimental Evaluation

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

A. Experimental Settings

Data and relations: We use a subset of the North American News Text Corpus[5], with 1995-6 articles from The New York Times, split into a *training* (135,438 documents from 1996) and a *test* database (137,893 documents from 1995). We define two "base" relations, *Headquarters(Company, Location)* and *Executives(Company, CEO)*, and a view *CompanyInfo(Company, Location, CEO)* over the base relations.

Queries: We use 33 SQL queries over *CompanyInfo*, defined manually to cover an interesting mix of selections (e.g., SELECT * FROM CompanyInfo WHERE Location = 'California') and projections (e.g., SELECT Company, CEO FROM CompanyInfo), including selections with many and with few or no database matches.

IE systems: We trained variations of our home-grown implementations of *DIPRE* [4] and Snowball [3] for *Headquarters* and *Executives*. We modified the original formulation of *DIPRE* to fit our newspaper database (e.g., we do not exploit URLs or HTML tags, but incorporate instead named-entity tags, such as ⟨ORGANIZATION⟩). For variety, we trained three versions of Snowball for each relation, obtained by privileging precision, recall, or a combination of both via the modified F_1-measure (see Section IV), and refer to them as *SB-P*, *SB-R*, and *SB-C*, respectively.

Document retrieval strategies: To define the *PromD* and *PromC* retrieval strategies (Section III), we use QXtract [1], as described in Section II, trained for each of the four extraction systems for each base relation.

[5] http://www.ldc.upenn.edu

SQOUT and baseline techniques: We compare our query processing approach, SQOUT, against 15 "baseline" techniques, each with a static choice of extraction system and document retrieval strategy. Specifically, we define four families of baseline techniques, corresponding to document retrieval strategies *Scan*, *PromD*, *Const*, and *PromC*. For each choice of retrieval strategy, we pick extraction systems (out of *DIPRE*, *SB-P*, *SB-R*, and *SB-C*) in four different ways, namely to privilege efficiency, precision, recall, or a combination of precision and recall. We make the choice of extraction system for each relation and baseline, once and for all, using the statistics of Section V, which are computed during SQOUT's preprocessing step. For instance, for the precision baseline with *PromD*, we pick an extraction system for *Headquarters* that maximizes our precision estimate $\frac{|C(E,\ PromD,\ D)|}{|T(E,\ PromD,\ D)|}$ for that relation. The *Const* (*PromC*) baselines use the same extraction systems selected for the corresponding *Scan* (*PromD*) baselines; also, *Const* and *PromC* "degenerate" to *PromD* for any relation with no selection condition in a query. The prefix of a baseline name denotes whether the baseline privileges extraction system efficiency (*B:E*), precision (*B:P*), recall (*B:R*), or a combination of precision and recall (*B:C*); the suffix of the name, in turn, denotes the retrieval strategy. For example, *B:P-PromD* is the precision-oriented baseline that uses *PromD* for document retrieval. We consider all combinations except for *B:E-Scan*, since *Scan* would not be a strategy of choice for an efficiency-oriented execution.

Evaluating our optimization approach: We first run all execution strategies (i.e., SQOUT and the 15 baseline techniques) for a query and then take the union of all the tuples produced collectively, eliminating duplicates. We then verify the correctness of each tuple via the automated template-based approach of Section V-B[6] and revisit tuples marked as correct to manually detect any suspicious tuples. (We found very few such cases.) Then, we manually check all extracted tuples that did not pass the automatic verification step. We did not perform this manual verification step for projection queries with no selection condition because their results were too large to be manageable. The final phase is to label each tuple in each query execution as correct or incorrect based on the above analysis; also, we mark as incorrect any tuple with a NULL attribute value[7] such that it has a correct complete counterpart, with no NULL attribute values, that was identified by any competing execution strategies. After labeling the results for each query execution, we calculate execution time, precision, recall—computed with respect to the pool of correct tuples that all alternative executions collectively produce for the query—and goodness.

Computing environment: All our experiments were conducted on a Dell Power Edge 2650 computer server (see Table II). We implemented our strategy in Java. We ran our experiments on an unloaded computer, restarting the Java Virtual Machine and flushing the file buffer and CPU caches before every single execution. We used the PostgreSQL[8] DBMS and, to provide a search interface to the text collections, we used Lucene[9]. Finally, for the data cleaning step (see Section III), we used a similarity threshold of 0.9 [24].

[6]If a tuple is the join of two base relation tuples, we require that each component base tuple be correct individually.

[7]Note that we take the outerjoin of the base relations (Section III).

CPU	Intel Xeon 2.4 GHz
RAM	4 GB
Operating System	Red Hat Enterprise Linux AS release 4 (2.6 EL)
Runtime Environment	Java 1.5.0_12 (Sun)

TABLE II
COMPUTING ENVIRONMENT FOR THE EXPERIMENTS.

B. Experimental Results

SQOUT vs. "Extract-then-query": As argued in the Introduction, the prevalent solution to process SQL queries over a text database is an "extract-then-query" approach, where all database documents are processed by the appropriate IE systems. To analyze the efficiency of this approach, which corresponds to the *Scan* retrieval strategy, we ran the 3 **-Scan* baselines, namely *B:C-Scan*, *B:P-Scan*, and *B:R-Scan*, for each of our 33 SQL queries (Section VI-A), for a total of 99 executions. Several of these executions demanded in excess of one hour to complete (single-relation executions are faster than join executions). These experiments confirm our initial observation: the extract-then-query approach is not appropriate for scenarios where efficiency is important; more generally, this approach does not adapt to user preferences on efficiency and result quality. In contrast, rather than processing all documents exhaustively, SQOUT extracts the database statistics of Table I in a preprocessing step, and then optimizes queries, to pick the best execution strategies for the desired efficiency-quality balance. We discuss SQOUT's preprocessing step later in this section, and now analyze the effectiveness of SQOUT's optimization approach.

SQOUT vs. Baselines: We vary the user-specified w parameter (Section IV) from 0 (to privilege efficiency) to 1 (to privilege result quality). We distinguish between the queries without any selection condition, to which we refer as *projection* queries, and the rest, to which we refer as *selection* queries, because the *Const* and *PromC* retrieval strategies do not apply to projection queries.

Figure 3 shows the average *goodness* (Figure 3(a)) and execution time (Figure 3(b)) of each technique over all selection queries, as a function of w. The values for SQOUT include the time required by the SQOUT optimization steps (Steps (1) and (2), Section III). SQOUT consistently has either the highest or close to the highest *goodness* for all values of w. When we privilege efficiency (w=0), the SQOUT *goodness* is better than that for all but two baseline techniques, namely *B:C-Const* and *B:P-Const*. For some queries, SQOUT

[8]`http://www.postgresql.org`
[9]`http://lucene.apache.org`

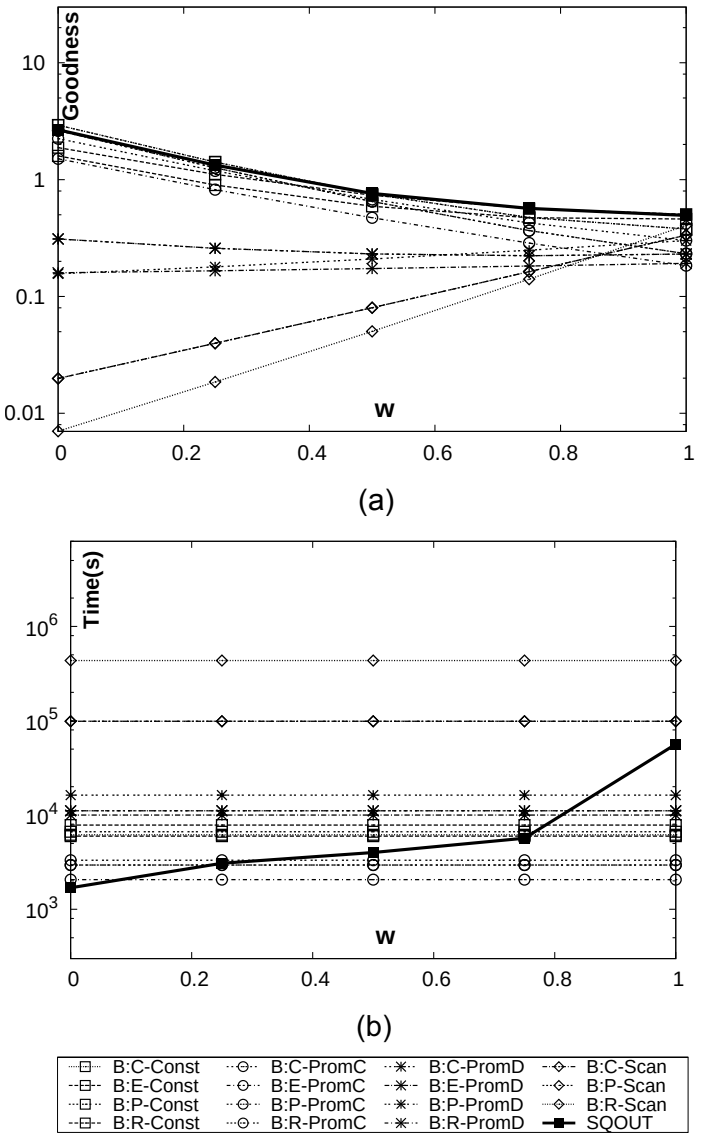

Fig. 3. (a) *Goodness* and (b) execution time over selection queries for varying w (log scale).

picks execution strategies that produce no tuples. Although the SQOUT strategies are the most efficient (see Figure 3(b)), their *goodness* value is zero (Definition 4.4). An example of one such query is SELECT Location FROM CompanyInfo WHERE Company = 'Applied Materials': SQOUT chooses *PromC* as the document retrieval strategy for *Headquarters*, which fails to extract any answer tuples. In contrast, *B:C-Const* and *B:P-Const* use *Const* and manage to generate answer tuples, using a longer execution. For some other queries (e.g., SELECT Company FROM CompanyInfo WHERE CEO = 'Larry Ellison'), SQOUT chooses *Const*, which leads to *goodness* values higher than those for the **-PromC* baselines (*PromC* generally retrieves fewer documents than *Const*, which in turn might lead to lower recall values). Therefore, the performance of SQOUT at $w = 0$ lies between those of *B:C-Const* and *B:P-Const* and those of the **-PromC* baselines. When we privilege result quality (w=1), SQOUT performs slightly worse than the best baseline, namely, *B:E-Const*. Finally, for projection queries SQOUT has the maximum *goodness* for all values of w. (We omit the figure because of space limitations.)

	Extraction System				Retrieval Strategy	
w	*DIPRE*	*SB-C*	*SB-P*	*SB-R*	*Const*	*PromC*
0	47.6	0	52.4	0	14.8	85.2
0.5	35.7	35.7	0	28.6	29.6	70.4
1.0	35.7	28.6	0	35.7	100	0

TABLE III

SQOUT'S CHOICE OF EXTRACTION SYSTEM AND RETRIEVAL STRATEGY, AS A PERCENTAGE OF ALL CASES, FOR SELECTION QUERIES.

Unlike our baseline techniques, SQOUT chooses document retrieval strategies and extraction systems for each query execution. To understand these choices, Table III focuses on selection queries for different w values. We show the fraction of times that SQOUT picks *Const* or *PromC* for base relations having an associated selection condition in the queries: for efficiency-oriented executions (w=0), SQOUT mostly picks *PromC*, whereas for quality-oriented executions (w=1), SQOUT picks *Const*, since *Const* cannot miss any relevant documents for a selection query. Table III also shows the choice of IE systems: when efficiency is privileged, SQOUT picks *DIPRE* and *SB-P*, which have lower processing times; in contrast, when quality is privileged, SQOUT progresses towards selecting *SB-C* or *SB-R* as the choice of extraction system.

Accuracy of SQOUT's Strategy Ranking: To complete our analysis, we measured the correlation between SQOUT's ranking of strategies and the (perfect) ranking generated using the actual goodness values; a positive correlation between these rankings is a good indicator of SQOUT's ability to avoid picking the worst strategies for a query and picking the best or close to the best strategies. We obtained consistently positive values of the Spearman correlation coefficient [27], which further supports our above conclusion that SQOUT robustly picks strategies with highest or close to highest goodness.

SQOUT's Preprocessing Step: SQOUT optimizes queries based on the statistics in Table I, which are computed during a preprocessing step. With highly unoptimized code, this step required 22.6 minutes in our experiments.[10] This step enables SQOUT to optimize SQL queries in a user-specific manner highly effectively, as shown above. SQOUT's preprocessing overhead makes it undesirable for one family of queries, namely for queries that both (1) have no constants (i.e., where *Const* and *PromC* are not applicable) *and* (2) specify

[10]SQOUT's preprocessing step relies on the sequential sampling approach of [25, 26] (Section V). Our current implementation of this preprocessing step is not optimized for efficiency. In fact, there are many opportunities for substantially reducing the running time of this step (e.g., reusing sample documents for both the single-relation and join statistics estimation), which we will explore in our future work.

that efficiency is not important (e.g., $w = 1$). In such case, an extract-then-query execution is preferable: SQOUT resorts to an extract-then-query execution, but also pays the extra cost of the preprocessing overhead. However, for all other scenarios, SQOUT is indeed attractive with respect to extract-then-query. For example, for $w = 0.5$ and query SELECT * FROM CompanyInfo WHERE CEO ='Eric Benhamou,' SQOUT's run time, including the preprocessing step, is 34% less than the time for the fastest extract-then-query approach, *B:P-Scan*. An interesting observation is that the execution for *B:P-Scan* generated no answer tuples for this query, whereas the execution picked by SQOUT generated an answer tuple. This observation underscores the benefits of considering both the result quality as well as the execution time when selecting a query execution.

In conclusion, our estimation techniques appropriately capture the factors involved in selecting a good query execution strategy and thus produce an ordering of alternate strategies that is often close to the perfect one. Overall, we showed that our proposed optimization approach consistently picks desirable query execution strategies for the user-specified tradeoff between execution efficiency and result quality. SQOUT, as expected, privileges efficiency for low values of w and result quality for high values of w. We have also extensively evaluated the *accuracy of the efficiency and quality estimates* on which SQOUT relies (Section IV). In a nutshell, the SQOUT estimates are generally close to the actual values, but the absolute errors of these estimates are often non-negligible; importantly, however, the relative rankings of the strategies according to the SQOUT efficiency and quality estimates are positively correlated—as measured using Spearman correlation tests—with the correct ranking in all cases, which explains the high goodness of the SQOUT executions. Due to space limitations, we omit further details.

VII. Conclusion and Future Work

This paper presented a principled, effective query optimization approach for simple SQL queries over text databases. Our approach relies on information extraction systems to discover the structured data that is "buried" in natural-language text, and considers both efficiency as well as query result quality when choosing an appropriate SQL query execution strategy over a text database. Many interesting research problems remain open in this area, and we plan to study them in our future work. One such problem is how to handle a richer family of SQL queries, beyond the simple selection-projection-join queries that we considered in this paper. Another challenging problem for future work is to explore the synergy between this paper's online extraction approach and an approach that exploits already extracted information (e.g., during earlier querying), to strike the right balance between offline computation—for persistent information needs and for the static portions of the text databases—and online extraction—for dynamic database contents and information needs.

Acknowledgments: The second author is supported by NSF CAREER grant IIS-0347903 and a Sloan fellowship. The remaining authors are supported by a generous gift from the Data Management, Exploration, and Mining Group, Microsoft Research.

References

[1] E. Agichtein and L. Gravano, "Querying text databases for efficient information extraction," in *ICDE*, 2003.

[2] A. McCallum, "Information extraction: Distilling structured data from unstructured text," *ACM Queue*, vol. 3, no. 9, 2005.

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

[4] S. Brin, "Extracting patterns and relations from the world wide web," in *WebDB*, 1998.

[5] M. Pasca, D. Lin, J. Bigham, A. Lifchits, and A. Jain, "Names and similarities on the web: Fact extraction in the fast lane," in *ACL*, 2006.

[6] E. Riloff, "Automatically constructing a dictionary for information extraction tasks," in *IAAI*, 1993.

[7] N. Kushmerick, D. Weld, and R. Doorenbos, "Wrapper induction for information extraction," in *IJCAI*, 1997.

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

[10] R. Gupta and S. Sarawagi, "Curating probabilistic databases from information extraction models," in *VLDB*, 2006.

[11] M. J. Cafarella, C. Re, D. Suciu, O. Etzioni, and M. Banko, "Structured querying of web text: A technical challenge," in *CIDR*, 2007.

[12] C. Kwok, O. Etzioni, and D. Weld, "Scaling question answering to the web," *ACM Transactions on Information Systems*, 2001.

[13] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan, "Efficient information extraction over evolving text data," in *ICDE*, 2008.

[14] D. Ferrucci and A. Lally, "UIMA: An architectural approach to unstructured information processing in the corporate research environment," in *Nat. Lang. Eng.*, 2004.

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

[16] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan, "Declarative information extraction using datalog with embedded extraction predicates," in *VLDB*, 2007.

[17] E. Chu, A. Baid, T. Chen, A. Doan, and J. Naughton, "A relational approach to incrementally extracting and querying structure in unstructured data," in *VLDB*, 2007.

[18] M. Cafarella, D. Suciu, and O. Etzioni, "Navigating extracted data with schema discovery," in *WWW*, 2007.

[19] 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.

[20] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano, "Towards a query optimizer for text-centric tasks," *ACM Transactions on Database Systems*, vol. 32, no. 4, Nov. 2007.

[21] A. Jain, A. Doan, and L. Gravano, "SQL queries over unstructured text databases (*"poster" paper)*," in *ICDE*, 2007.

[22] G. A. Salton and M. J. McGill, *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[23] I. P. Fellegi and A. B. Sunter, "A theory for record linkage," *Journal of the American Statistical Association*, vol. 64, no. 328, Dec. 1969.

[24] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava, "Text joins in an RDBMS for web data integration," in *WWW*, 2003.

[25] P. J. Haas and A. N. Swami, "Sequential sampling procedures for query size estimation," in *SIGMOD*, 1992.

[26] Y. Ling and W. Sun, "An evaluation of sampling-based size estimation methods for selections in database systems," in *ICDE*, 1995.

[27] R. Hogg and A. Craig, *Introduction to Mathematical Statistics*. Macmillan, 1995.

[28] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami, "Fixed-precision estimation of join selectivity," in *PODS*, 1993.