ABSTRACT

We study a parsing-based semantics for keyword search over databases that relies on parsing the search query using a grammar. The parsing-based semantics is often used to override the traditional "bag-of-words" semantics in web search and enterprise search scenarios. Compared to the "bag-of-words" semantics, the parsing based semantics is richer and more customizable. While a formalism for parsing-based semantics for keyword search has been proposed in prior work and ad-hoc implementations exist, the problem of designing efficient algorithms to support the semantics is largely unstudied. In this paper, we present a suite of efficient algorithms and auxiliary indexes for this problem. Our algorithms work for a broad class of grammars used in practice, and cover a variety of database matching functions (set- and substring-containment, approximate and exact equality) and scoring functions to filter and rank different parses. We formally analyze the running time complexity of our algorithms and provide a thorough empirical evaluation over real-world data to show that our algorithms scale well with the size of the database and grammar.

1. INTRODUCTION

Over the past few years, internet search technology has progressed far beyond the traditional bag-of-words retrieval framework. In particular, for some web search queries, parsing-based semantics is employed to "parse" the query using some rules or grammar and some underlying database of facts, and this parse influences the search output. As a simple example, for the query seattle to redmond, typical search engines use rules to identify that the query matches (location) to (location) since seattle and redmond are both present in a location table. This parse, once identified, may then be consumed by a map application which displays a map with directions alongside the search results.

There are several advantages to the parsing-based approach: (1) The parse of a query provides clues to possible user intentions behind the query. For instance, a query (business) customer support is presumably looking for customer support contact information for a business. A better understanding of the user intentions allows the search system to customize the output to a query [12, 28], e.g., show a map of driving directions or display contact information. (2) A parsing-based approach allows the query to contain non-content terms (e.g., from, to), combine data and schema terms (e.g., best buy phone), and relate content both inside and outside of "join paths" (e.g., godfather brando — matching a single row in a movie-actor relation versus new york to scottsdale — matching multiple rows in a location relation); such queries are hard to support in keyword search over databases that interprets all search terms as content arising from some row in the database.

Thus, parsing-based semantics allows internet search engines to override the bag-of-words retrieval framework and provide internet users with better or enhanced search results. In fact, prior work has shown that search results where such enhanced information is provided have an order of magnitude higher user satisfaction rate than those without [28]. To the best of our knowledge, however, the implementations of these semantics are ad-hoc, and there has been no formal study of the efficiency of the parsing-based semantics for internet keyword search.

Parsing keyword search queries is equally relevant in enterprise keyword search over databases [10]. Previous work at IBM on the Avatar project for enterprise search [26, 38] used heuristics to parse keyword search queries to a grammar. More recently, Fagin et al. [15] provided a formal basis for the problem of answering queries using a (particular instance of) parsing-based semantics, and showed that the problem admits a polynomial time (in input and output size) solution. While this result contains key insights into the theoretical complexity of the problem, Fagin et al. [15] did not propose efficient algorithms or indexes for the problem. For efficient keyword search, indexing techniques that avoid repeated scans of the entire input database to answer a query are critical. The central goal of this paper is to develop efficient indexing and query answering techniques to support parsing-based keyword search.

Our techniques are general; in particular, we study how the following dimensions impact the efficiency of our approach. (For our illustrations throughout, we use the Tables 1 and 2 as the database over which search is performed.)

1. Grammar: A central component of query parsing is the grammar that specifies how queries are parsed. We consider a rich class of grammars that can be loosely characterized as a regular expression over database concepts (columns) and special keywords (e.g., to, from). An example "pattern" is (Product) vs (Product), which can be used to parse the query quiet comfort vs quiet point, given electronic product data in Table 2. Note that the "matches" quiet comfort and quiet point: for this query arise from two different rows P1 and P4. While this is desirable in some cases, it is sometimes useful to constrain matches to arise from the same row. Our patterns support relational constraints to express such constraints: In the pattern, (Store<sup>1</sup> Location<sup>1</sup>) phone,
the common superscript 1 indicates that matches for Store and Location should arise from the same row. The pattern would parse the query "bose tucson phone", but not the query "bose glendale phone", given data in Table 1a. This pattern captures the intention “phone number of a store at a given location” better than the pattern without relational constraints.

2. Matching: To parse a query, we need to determine if a particular phrase in a query matches a value in the database. Given that keyword queries are typically short and incomplete, requiring exact matches would not be robust. We therefore focus on set containment as a matching function to determine matches, which is widely used in traditional keyword search as well. We also consider and present results for other well-known matching functions such as approximate equality and substring containment.

3. Scoring: Again, to be able to handle short and imprecise queries, we work with a relaxed definition of a parse that can skip over and “cover” only a subset of query terms. With this relaxed definition, often a query can be parsed in a multitude of ways. In this setting, it is important to be able to capture preferences for one parse over another. A natural preference is to prefer parses that cover more query terms to parses that cover less. We consider a general class of scoring functions to express such preferences, and this class includes well-known sequential models such as CRFs and HMMs [34].

Designing efficient algorithms for the general class of grammars, matching, and scoring functions that we consider is challenging. In practice, a database can contain 100s of tables and grammar might be a disjunction of 1000s of patterns. To illustrate some challenges, consider an approach for this problem based on constructing an “expanded” grammar. For example, we could expand the rule ⟨Product⟩ vs ⟨Product⟩ by replacing each ⟨Product⟩ with all possible values from the Product column in the database (we temporarily ignore issues arising from non-equality matching). A traditional finite state automaton can be built for this expanded grammar and used to parse queries. In fact—as we demonstrate in the full version of this paper—the space required for this approach can be formally shown to be impractically large. Moreover, prior work [33] has empirically verified that such approaches perform poorly in practice.

In developing efficient techniques to parse keyword queries, we need to address the following questions:

- How do we represent and store the grammar, and how is this representation consulted while parsing? The grammar may be very large. How do we avoid parsing each of the patterns in the grammar individually, and how may we enable parsing while sharing “partial” parses between patterns? How do we maintain these partial parses (potentially a large number) compactly?
- Which data structures or indexes do we need to avoid repeated scans of the entire database? What is the most efficient way of matching, with or without relational constraints for various matching functions? In some queries, there may be noise tokens or misspelled words. How do we eliminate the blowup that occurs with considering all subsets of keywords while matching? At what stage of parsing should we perform matching?
- How do we adapt our technique to be efficient for different scoring functions (i.e., those that score or rank complete parses—such as CRF and HMM) and different matching functions (i.e., equality or approximate equality and string containment)? Is there a classification of matching or scoring functions for which we can be more efficient? How do we design our system to be able to seamlessly support more patterns or different matching functions? And how can we enable all this while guaranteeing that we are returning the best parses for each query?

Our Solution: The indexing solution that we propose is based on a general extensible architecture of three modules that we call filtering, stitching, and matching. Our solution uses a non-deterministic finite state automaton (NFA) over database concepts. For a given query, the matching module identifies phrases in the query that match with a value in the database and the stitching module uses the input from the matching module to traverse the NFA and identify parses with a high score. The filtering module uses various heuristics to prune the NFA to minimize the work done by the matching and stitching modules. In addition to efficient algorithms for these modules, one of our contributions is the architecture itself: the clean separation and the precise interface that we develop between the matching and stitching modules enables us to relatively easily extend our framework to support a new matching function given an indexing “oracle” for the function.

Efficiency and Quality: The effectiveness of any search system hinges on two aspects (a) the efficiency of the search system, which is our focus, and (b) the quality of the retrieved search results. Broadly, quality refers to how good the system is in “understanding” user queries and returning relevant results. In a parsing-based system, the choice of the grammar used to parse queries, the choice of the scoring function used to rank queries, and the actions performed once the parse is identified, all impact the quality of the output results. These choices in turn depend on the target application. Configuring the system for best quality for a specific application is not a goal of this paper. There exists a large body of work on related problems [1, 33, 34] that can be leveraged for this purpose and this continues to be an active area of research. The results in this paper are application-independent and orthogonal to the work on improving quality, and we believe most techniques for configuring a parsing-based search system for quality would benefit from our study of performance related aspects.

Related problems: We briefly comment on two related problems studied in prior work. (A comprehensive coverage of related work is in Section 5.) There has been a lot of recent work on query intent classification [29, 24, 5]. The goal of this work is to identify if a search query belongs to a particular category or not (e.g., navigational vs informational, travel vs non-travel). The output of a intent classification system is a categorical value that can be used, for example, to direct the query to an appropriate search vertical. In contrast, the output of a query parsing system is more detailed and the kinds of challenges and issues in the two problems are fundamentally different. We note that query classification and query parsing can complement one another since query parsing is relevant and useful even after we narrow down the search vertical using query classification. The second related problem is text segmentation [34]. In segmentation, the goal is to assign labels to each token in the input string. We need additional terminology to compare segmentation with our work and we defer a full comparison until Section 5. Briefly, we can view our query parsing framework as bridging the spectrum between pure segmentation and bag-of-words keyword search and it therefore inherits the benefits of both technologies. We note that our techniques can also be used to improve the performance of pure segmentation, since we support state-of-the-art segmentation models such as CRFs.

Outline of the paper: In Section 2, we formally present the semantics of parsing-based keyword search and the problem of answering queries under this semantics. We present our indexing techniques in Section 3 and evaluate them empirically in Section 4. We cover related work in Section 5 before concluding (Section 6).
Table 1: StoreInfo and Address tables of sample database

<table>
<thead>
<tr>
<th>Store</th>
<th>Location</th>
<th>Phone</th>
<th>Address</th>
<th>Lat</th>
<th>Long.</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Circuit City</td>
<td>Scottsdale</td>
<td>123-456-789</td>
<td>A1</td>
<td>392 Havelock Road Singapore</td>
</tr>
<tr>
<td>S2</td>
<td>Circuit City</td>
<td>Tucson</td>
<td>912-345-678</td>
<td>A2</td>
<td>1 South Capitol Avenue Indianapolis</td>
</tr>
<tr>
<td>S3</td>
<td>Fry’s</td>
<td>Phoenix</td>
<td>891-234-567</td>
<td>A3</td>
<td>392 South Capitol Avenue Indianapolis</td>
</tr>
<tr>
<td>S4</td>
<td>Fry’s</td>
<td>Glendale</td>
<td>891-234-567</td>
<td>A4</td>
<td>66 Quai Charles de Gaulle Lyon</td>
</tr>
<tr>
<td>S5</td>
<td>Bose</td>
<td>Tucson</td>
<td>789-123-456</td>
<td>A5</td>
<td>Ave. Bergieres Case Postale Lausanne</td>
</tr>
<tr>
<td>S6</td>
<td>Target</td>
<td>Denver</td>
<td>678-912-345</td>
<td>A6</td>
<td>2 Vas. Alexandrou Ave. Athens Greece</td>
</tr>
<tr>
<td>S7</td>
<td>Walmart</td>
<td>Santa Fe</td>
<td>567-912-345</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(a) StoreInfo

(b) Address

Table 2: ProductInfo table of sample database

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Possible user intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(Store) phone</td>
</tr>
<tr>
<td>2</td>
<td>((Store)((Address))(Address))</td>
</tr>
<tr>
<td>3</td>
<td>(Product)+</td>
</tr>
<tr>
<td>4</td>
<td>(Store)(Category)</td>
</tr>
<tr>
<td>5</td>
<td>(Product) price</td>
</tr>
<tr>
<td>6</td>
<td>(Company\Product) price</td>
</tr>
<tr>
<td>7</td>
<td>(Company\Category)</td>
</tr>
<tr>
<td>8</td>
<td>(Store\Location) phone</td>
</tr>
</tbody>
</table>

Table 3: Regular expression patterns

2. PROBLEM FORMULATION

In our system, the preprocessing input consists of a database over which keyword search is to be performed, and a grammar that specifies the parsing-based semantics of keyword search. Our goal is to design algorithms and index structures to support efficient parsing of keyword search queries over the database for the semantics specified by the grammar. Throughout, we use \( \mathcal{D} \) and \( \mathcal{G} \) to denote the input database and grammar, respectively. The database consists of a collection of tables with text columns. For simplicity, we assume that column names are unique. For a column \( C \), we use Table(\( C \)) to denote the (unique) table that column \( C \) is part of.

The grammar \( \mathcal{G} \) is a regular expression over an alphabet of symbols that we call non-terminals. Informally, a non-terminal represents the unit of “matching” against the database while parsing queries. The simplest non-terminals are of the form \( \langle C \rangle \), where \( C \) is a column in the database and \( f \) denotes a matching function used to define the notion of a match. A matching function has the signature \( \text{string} \times \text{string} \rightarrow \{\text{true}, \text{false}\} \). Example matching functions include exact and approximate string equality and string containment (defined subsequently). More generally, a non-terminal is of the form \( \langle C_1 \alpha_1 f_1 \alpha_2 \ldots C_n \alpha_n f_n \rangle \), where \( C_i \) is a column and \( f_i \) a matching function; the \( \alpha \) values are symbols called relational variables that encode relational constraints. Informally, the matches for all columns \( C_i \) with the same relational variable should come from the same record in the database. For relational constraints to be meaningful, we require that Table(\( C_i \)) = Table(\( C_j \)) whenever \( \alpha_i = \alpha_j \). We refer to non-terminals involving a single column as basic non-terminals and those involving relational constraints relational non-terminals. In the following, we drop the matching function subscripts when the functions are clear from context or not relevant to the discussion.

Example 1. For our sample database, an example of a basic non-terminal is \( \langle \text{Store} \rangle \) and examples of relational nonterminals are \( \langle \text{Company} \rangle \langle \text{Product} \rangle \) and \( \langle \text{Location} \rangle \). The nonterminal \( \langle \text{Store} \rangle \langle \text{Address} \rangle \) would not be valid since \( \text{Store} \) and \( \text{Address} \) belong to different tables. An example of a grammar is the regular expression \( \langle \text{Product} \rangle + \langle \text{Store} \rangle ? \langle \text{Category} \rangle \).

There are two reasons for introducing relational non-terminals: First, it represents a precise way of specifying relational constraints. Second, as we discuss in Section 3.2, the set of columns in a relational nonterminal form a natural unit, since columns connected by a relational constraint need to be collectively “matched” against the database.

We do not explicitly model keyword constants such as phone and price in Table 3. We can incorporate a keyword constant by constructing a dummy single column table with a single row containing the constant.

2.1 Parsing-based Semantics

The parsing-based semantics of keyword search involves identifying zero or more parses of a given query. We associate a score with each parse, computed using a scoring function. The output consists of parses suitably ranked and filtered using their scores.

For the purposes of defining the semantics, we view all strings as a sequence of tokens. In our examples, we use whitespace as delimiters to define tokens, but our techniques are independent of the tokenization scheme. Given a token sequence \( q = t_1, \ldots, t_n \) and \( 1 \leq a \leq b \leq n \), the token interval \( q[a, b] \) is defined to be the subsequence \( t_a, \ldots, t_b \). When the original sequence \( q \) is clear from context, we use interval \( [a, b] \) as a shorthand for \( q[a, b] \). Given intervals \( I_1 = [a_1, b_1] \) and \( I_2 = [a_2, b_2] \), we say \( I_1 < I_2 \) if \( b_1 < a_2 \). Given \( I = [a, b] \), we define \( I.\text{start} \triangleq a \) and \( I.\text{end} \triangleq b \).

Nonterminal matches and Parses

For the rest of this section, assume a query \( q = t_1, \ldots, t_n \). We next formally define the notion of a match for a non-terminal. A token interval \( [a, b] \) is a match for a basic non-terminal \( \langle C \rangle \) if database \( D \) contains a value \( v \) in column \( C \) such that \( f(q[a, b], v) = \text{true} \). The following example introduces a couple of matching functions and illustrates matching.

\[ f(q[a, b], v) = \text{true} \text{ if } q[a, b] \text{ matches } v \]
Table 4: Example queries and their best parse (corresponding to the least value of noise)

<table>
<thead>
<tr>
<th>Query</th>
<th>Parse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 circuit city phone</td>
<td>1(0)</td>
</tr>
<tr>
<td>Q2 circuit city outlet near havelock road</td>
<td>2(2)</td>
</tr>
<tr>
<td>Q3 from indianapolis tp qual charles de guale</td>
<td>2(2)</td>
</tr>
<tr>
<td>Q4 bose ATH-ANC7B price</td>
<td>5(1)</td>
</tr>
<tr>
<td>Q5 audio technica quiet comfort</td>
<td>3(2)</td>
</tr>
<tr>
<td>Q6 quiet comfort ATH-ANC7B</td>
<td>3(0)</td>
</tr>
<tr>
<td>Q7 1 south capitol avenue 392 havelock road</td>
<td>2(0)</td>
</tr>
<tr>
<td>Q8 bose electronics headphones ATH-ANC7B price</td>
<td>7(2)</td>
</tr>
</tbody>
</table>

Example 2. Given two token sequences q and v, the set containment function \( f_q(q, v) = 1 \) if every token in q occurs in v, and false otherwise. The equality function \( f_q(q, v) = \text{true} \) if q and v are identical. Let q denote query Q1 in Table 4. The intervals [1, 1] (circuit), [2, 2] (city) and [1, 2] (circuit city) are all matches for nonterminal \( \langle \text{Product} \rangle \). However, nonterminal \( \langle \text{Store} \rangle \) has only one matching interval, [1, 1].

A match for a (relational) nonterminal \( \langle C_{1}^{\alpha_{1}} \ldots C_{m}^{\alpha_{m}} \rangle \) is a sequence of intervals \( I_1 < \ldots < I_m \) such that there exist rows \( r_1, \ldots, r_m \) in database D satisfying \( r_i = r_j \) whenever \( \alpha_i = \alpha_j \) and for all \( 1 \leq j \leq m, f_j(q[I_j], r_j[C_j]) = \text{true} \). Here, \( r_j[C_j] \) refers to the projection of row \( r_j \) on column \( C_j \). Note that this definition generalizes the definition of a match for basic nonterminals.

Example 3. Consider the nonterminal \( \langle \text{Product} \rangle \) and let \( f_C \) be the matching function for both columns. The common relational variable 1 implies that a match for the two columns should come from the same record of ProductInfo. Let q denote query Q5 (Table 4). The interval sequence [1, 2], [3, 3] is a match for q since there exists record P4 such that \( q[1, 2] \) (i.e., audio technica) is contained in P4[Company] and \( q[3, 3] \) (quiet) is contained in P4[Product]. On the other hand, [1, 2], [3, 4] is not a match: although \( q[3, 4] \) (quiet comfort) is contained in the Product column, the match involves two records, P1 and P4, violating the relational constraint.

Given a match \( M = \langle I_1, \ldots, I_m \rangle \) for a nonterminal \( \langle C_{1}^{\alpha_{1}} \ldots C_{m}^{\alpha_{m}} \rangle \), we define the extent of match \( M \), denoted Extent(M), to be the interval \( [I_1.start, I_m.end] \). Note that Extent(M) is the smallest interval containing the intervals \( I_1, \ldots, I_m \). In this case, we define the above property to be true for matches of basic nonterminals; however, for such matches \( M \), Extent(M) is trivially the single interval in M.

Definition 1. (parse) A Parse of a query q using a grammar G is defined as a sequence of nonterminals \( N_1 \ldots N_m \) and matches \( M_1, \ldots, M_m \) corresponding to the nonterminals such that:
1. The sequence \( N_1 \ldots N_m \) belongs to the language defined by G.
2. Extent(\( M_1 \)) < \ldots < Extent(\( M_m \))

We denote this parse \( \langle N_1 : M_1, \ldots, N_m : M_m \rangle \).

Example 4. Consider query Q6 in Table 4 and let G be the disjunction of regular expression patterns in Table 3. Assume \( f_C \) to be the matching function everywhere. A Parse for this query is: \( \langle \langle \text{Product} \rangle : [1, 2], \langle \text{Product} \rangle : [3, 4] \rangle \). The sequence \( \langle \text{Product} \rangle \) (Product) arises from the pattern \( \langle \text{Product} \rangle + \). Similarly, a Parse for query Q3 is \( \langle \langle \text{Address} \rangle : [2, 2], \langle \text{Address} \rangle : [4, 7] \rangle \). As this example suggests, a Parse need not be “perfect” and there could be tokens (e.g., from, tp) in the query not covered by the Parse.

Scoring Function

There could be several Pares for a given query and it is useful in a search system to capture preferences for some Pares over others. Such preferences can be used to suitably filter or rank Pares that are returned to the user. In this paper, we consider scoring functions to express such preferences. A scoring function assigns a numeric score to a Parse, with a higher score indicating greater preference. We denote the score returned by a scoring function \( F \) for query q and Parse \( P \) using \( F(q, P) \).

While the scores of Pares can be used in a variety of ways to filter and rank them, for presentational simplicity, in the rest of the paper, we restrict ourselves to the problem of identifying the Pare with the highest score. Our techniques can be generalized to other ranking and filtering settings.

For an arbitrary scoring function, we cannot do better than enumerating and scoring all Pares to identify the best one. We identify and study two general classes of scoring functions called linear and coverage-based which admit more efficient approaches.

Definition 2. Consider a Parse \( P = N_1 : M_1, \ldots, N_m : M_m \) for a query q. A scoring function \( F(q, P) \) is linear if it can be expressed as \( \sum_i g(q, N_i, M_i) \) for some function g. As a generalization, the function \( F(q, P) \) is k-th order linear if it can be expressed as \( \sum_i g(q, N_{i-k}, \ldots, N_{i-1}, N_i, M_i) \). The overall score of a linear function is a sum of “local” contributions for each nonterminal in the Parse, where the local contribution for a nonterminal depends only on the match for the nonterminal.

For k-th order linear function the local contribution depends also on the previous k nonterminals in the Parse. We note that standard sequential models such as HMMs and CRFs [34] are first-order linear models.

Given a query q = \( t_1, \ldots, t_n \) and Parse P, we say a token is covered if it is contained in some interval in some match in P. For Definition 3 below, let Cov(q, i, P) denote the coverage indicator function: \( \text{Cov}(q, i, P) = 1 \) if \( t_i \) is covered by P, and false otherwise.

Definition 3. Consider a Parse \( P = N_1 : M_1, \ldots, N_m : M_m \) for a query q and let Cov(q, i, P) denote the coverage indicator function. A scoring function \( F(q, P) \) is a coverage-based if it is of the form \( \sum_i \text{Cov}(q, i, P) \cdot g(q, i) \), where g is some positive function.

The overall score of a coverage-based function (hereafter, simply a coverage function) is the sum of positive individual token contributions, considering only tokens that are covered by the Parse. We can show that every coverage function is a linear function.

Example 5. A simple coverage function is the number of tokens covered in a Parse. In the rest of the examples, coverage function implicitly means this simple function. With this function, the score for the \( \langle \text{Address} \rangle : [2, 2], \langle \text{Address} \rangle : [4, 7] \) Parse of query Q3 in Table 4 is 5. It is also useful to consider the noise of a Parse, defined to be the number of tokens not covered by the Parse. Table 4 provides (within parenthesis) the noise value of the best Parse for each query.

Thus, we address the following problem in this paper:

**Problem 1.** Given a grammar G, a database D and a scoring function F, design algorithms and index structures for the following task at query time: On being given query q, determine the Parse P such that \( F(q, P) \) is maximized.
3. ALGORITHMS

This section describes our indexing and query answering algorithms for the parsing-based keyword search problem introduced in Section 2. In particular, we present algorithms for efficiently identifying the best parse for a given query string; the details of indexing structures that we build at preprocessing time is embedded in the description of the parsing algorithms. For the remainder of this section, assume a database $D$, a grammar $G$, a linear scoring function $F$, and a query string $q = t_1, \ldots, t_n$. (Our techniques work for $k$-th order linear functions with minor modifications.) We begin with an overview of our algorithms before presenting the details.

3.1 Overview

The overall parsing algorithm is structured around two broad modules called matching and stitching; the stitching module is the main query processing module that invokes matching from within it. Conceptually, we may view the matching module as our inter-face to the database $D$, while the stitching module is our interface to the grammar $G$. The scoring function $F$ impacts our use of both matching and stitching, since we may use the nature of $F$ to share computation.

The matching module identifies matches for nonterminals in grammar $G$. All access of data in $D$ is confined to the matching module, which contains indexes for efficient data access. Some of the logic of scoring parses is “pushed” into the matching step and the set of the matches returned by the matching module is not an exhaustive enumeration. On the other hand, the functionality provided by the stitching module is similar to classical regular expression parsing: It uses an NFA $Q$ that represents (a part of) grammar $G$, and traverses $Q$ based on the matches identified by the matching step. Again, the stitching module uses the scoring function to prune traversals guaranteed not to produce the best parse.

A third module, called filtering is invoked in the beginning before control passes on to stitching. Informally, this step starts with the “original” NFA $Q_{orig}$ corresponding to the entire grammar $G$ and uses the input query $q$ to produce a pruned NFA $Q$, which is input to the stitching step. The pruned NFA $Q$ has the property that any parse of $q$ in $Q_{orig}$ is also a valid parse in $Q$. The filtering module is optional for correctness but important for efficiency, especially when grammar $G$ is large.

Figure 1 illustrates the overview of our algorithm. Everything inside the dotted rectangle indicates query time computation while everything outside indicates indexes and data structures built at preprocessing time. The rest of this section describes each of these three modules.

3.2 Matching

This section presents the matching module. The matching module is accessed from the stitching module for matching portions of the query to a specific nonterminal.

The basic interface presented by the matching module to the stitching module for a specific nonterminal $N$ is the function $\text{FindMatches}(q, N)$. Essentially, this function returns matches $M$ for a nonterminal $N$ that start at $a$ and end before $b$, for all pairs of locations $[a, b]$. However, this step can be very expensive since the number of such matches $M$ can be exponential in $n$. Instead, we design $\text{FindMatches}$ to return at most $\binom{n}{k}$ matches: for all matches $M$ that have the same $\text{Extent}(M)$, it returns the match with the largest $g(q, N, M)$.

Before we introduce the second interface $\text{FindMaxMatches}$, we will first define an important concept. If the matching function satisfies a natural property termed monotonicity, the matching module will be much more efficient (as we will see subsequently).

Definition 4. Let $u$ and $v = xuy$ be token sequences such that $u$ is a subsequence of $v$. A matching function $f$ is monotonic if $f(u, v) = \text{true}$ when $f(v, w) = \text{true}$ for any token sequence $w$.

Example 6. The containment function $f_c$ is monotonic, since if $v$ is contained in $w$, any subsequence $u$ of $v$ is also contained in $w$. The subsequence function is another example: if $v$ is a subsequence of $w$, then any subsequence of $v$ is also a subsequence of $w$.

When the nonterminal is basic, and when the matching function is monotonic, then the matching module additionally supports an interface $\text{FindMaxMatches}(q, N)$. This interface provides at most $n$ matches $M$, ordered by their start positions. Each of these matches is maximal, i.e., they are not contained in any other matches of $N$. It is easy to show that when the matching function for a basic nonterminal is monotonic, there can be at most one maximal match beginning at each location $a$.

In the next section, we will see how $\text{FindMatches}$ and $\text{FindMaxMatches}$ are called upon by the stitching module as needed.

Basic Nonterminal

We now discuss matching for a basic nonterminal $N = \langle \text{IND} \rangle$. Recall that the match for $N$ is an interval $[a, b]$ such that $q[a, b]$ = $\text{true}$ for some value $v$ in column $C$. Natural matching functions $f$ include $f_e$ (equality), $f_c$ (set containment), $f_{subs}$ (substring containment [15]), $f_{jacc}$ (approximate equality using jaccard), and $f_{edit}$ (approximate equality using edit). These are all well-known functions that have been used in similar problem settings such as record matching [14] and segmentation [34].

For most of these functions $f$, even the simpler problem of verifying for a fixed sequence $u$, whether there exists a value $v$ in $C$ such that $f(u, v) = \text{true}$ is nontrivial when the number of distinct values in $C$ is large and is the subject of prior research [20]. In the following, we refer to any algorithm for this verification problem as an oracle for $f$. Given an oracle for $f$, a straightforward algorithm for $\text{FindMatches}$ is to invoke the oracle for each of the $\binom{n}{k}$ intervals. We can often do better: For $f_e$, the Aho Corasick algorithm provides a linear algorithm; prior work has studied designing efficient algorithms for $f_{jacc}$ [7] and $f_{edit}$ [36]. We leverage this prior work for matching algorithms for $f_{subs}$, $f_{jacc}$, and $f_{edit}$.

We do not know of any prior matching algorithm for $f_c$, and we present an algorithm for this case (Algorithm 1) — in fact, this algorithm generalizes to any other monotonic function $f$ by replacing the calls to the containment oracle with calls to an oracle for $f$. This algorithm uses $O(n)$ set containment oracle calls to identify all maximal matching intervals and therefore provides
Algorithm 1: Computing maximal matches for $f_1$

1: function FINDMAXMATCHES($q, N$)
2:   Input: $q = t_1, \ldots, t_n$; search query; $N = \{C_f\}$
3:   maxMatches ← ∅
4:   $l ← 1, r ← 1$
5:   while $r \leq n$ do
6:     if CONTAINED($q[l, r], C_1$) then
7:         UPDATE(maxMatches, $[l, r]$)
8:     else
9:         $l ← l + 1$
10:   end if
11: end while
12: return maxMatches
13: end function

an implementation of FINDMAXMATCHES. To find all matches (FINDMATCHES), we simply consider all $\binom{n}{2}$ intervals and return those contained in some maximal match.

The algorithm maintains two “pointers” $l$ and $r$. If the current interval $[l, r]$ is a matching interval, we increment $r$, otherwise, increment $l$. When we find a matching match $[l, r]$, we add it to the set of maximal matches maxMatches and remove any previously added interval $[l, r-1]$ from maxMatches. For any maximal interval $[a, b]$, we can show that $l$ does not cross $a$ until $r$ reaches $b$ and $r$ does not cross $b$ until $l$ reaches $a$, implying that we correctly identify $[a, b]$ as a maximal matching interval.

Example 7. Consider $q = 1$ south capital ave, 392 havelock road and matching involving nonterminal $\langle Address \rangle$ from Table 1b. Since each of the intervals, $[1, 1]$, $[1, 2]$, $[1, 4]$, $[2, 4]$ are contained in Address, the algorithm reaches the state $l = 1, r = 5$ with a single maximal match $[1, 4]$. The subsequent intervals $[1, 5]$, $[2, 5]$, $[5, 4], [4, 5]$ are not matches, resulting in the state $l = 5, r = 5$. We find another sequence of matching intervals $[5, 5], [5, 6], [6, 7]$ resulting in the second maximal match $[5, 7]$ at which point the algorithm terminates.

For the set containment oracle, we use the implementation proposed in [2]. We can further exploit the fact that subsequent calls to the oracle differ by exactly one token to improve the efficiency of the oracle; a similar optimization has been proposed by [9] in the context of segmentation.

Relational Nonterminal

We now discuss matching for a relational nonterminal $N = \{C_1, a_1, \ldots, C_m, a_m\}$. Note that relational nonterminals allow us to express some natural patterns, e.g., the pattern $\langle Company^1 Product^1 \rangle$ (information about a specific product sold by a company) or the pattern $\langle Movie^1 Theater^1 \rangle$ (information about movie showtimes at a given theater).

For simplicity, we assume that all columns share the same relational variable, i.e., $\alpha_1 = \ldots = \alpha_m = \alpha$, meaning that all columns $C_i$ belong to the same table (called $T$) of the database. With this assumption, a match for $N$ is a sequence of intervals $[a_1, b_1] < \ldots < [a_m, b_m]$ such that $[a_1, b_1]$ matches $r(C_1)$ ($i \in [1, m]$) for some row $r \in T$. We discuss details of handling nonterminals with multiple relational variables in the full version of the paper. Unlike for basic nonterminal matching, we do not know of prior work that can be directly used for matching relational nonterminals.

To be able to enforce relational constraints, we assume that oracles for column level match functions $f_i$ return sorted record identifiers (rid-lists). The naïve algorithm enumerates every possible interval sequence $M = [a_1, b_1] < \ldots < [a_m, b_m]$, invokes the column oracle for $f_i$ with sequence $q[a_i, b_i]$, and merges the rid-lists returned by the column oracles. The sequence $M$ is a valid match if there is a common record $r$ common to all rid-lists. This algorithm can be expensive since there are $\Theta(n^2k)$ possible interval sequences. We next present a more efficient algorithm that generalizes Algorithm 1 for the case where all matching functions are monotonic.

Let $m = 1$ (i.e., $N$ is a basic nonterminal) and consider the following algorithm to enumerate all maximal matches of $N$. We operate in $n$ steps and at step $l$ we generate maximal matches starting at position $l$ (if one exists). We have at hand all maximal matches found so far that contain $l$. Let us call this set $\text{Overlap}(l)$. Since the interval we are seeking is maximal, it must contain at least one point outside each of the above intervals; that is, the new maximal interval starting at $l$ (if one exists) must hit the complements of each interval in $\text{Overlap}(l)$. Let $r$ denote the rightmost point of any interval in $\text{Overlap}(l)$. Then we can show that $[l, r+1]$ is the unique minimal hitting interval for $\text{Overlap}(l)$, implying that any maximal interval starting at $l$ should contain $[l, r+1]$. Therefore we can identify this interval by “growing” the interval $[l, r+1]$ to the right. We can show that the algorithm sketched above is identical to Algorithm 1.

The above intuition extends for $m > 1$. As in the case of $m = 1$, we construct maximal matches in the order of their start positions. These start positions are $m$-dimensional and ordered lexicographically. We seek a minimal interval sequence with the current start position $\langle a_1, \ldots, a_m \rangle$ that hits the complements of each maximal match found so far overlapping $\langle a_1, \ldots, a_m \rangle$. We illustrate an example in Figure 2 for the case $m = 2$. There are 3 maximal interval sequences $\langle I_1, I_2 \rangle$, $\langle I_2, I_3 \rangle$, $\langle I_3, I_4 \rangle$ that overlap with the start position $\langle a, b \rangle$ (also marked in the figure). Unlike the single dimensional case, we cannot identify a unique end position that is outside all the above maximal matches. Thus, we can construct the hitting interval sequence starting at $\langle a_1, \ldots, a_m \rangle$ in multiple ways. The last location corresponding to $a_i$ can be set either to $a_i$ itself or to $I_i.end + 1$ where the sequence $\langle I_1, \ldots, I_m \rangle \in \text{Overlap}(\langle a_1, \ldots, a_m \rangle)$. In Figure 2, the first dimension of the end position of the hitting set is $a$ or one of the tokens in $R$, while the second dimension is $b$ or one of the tokens in $S$. Across the dimensions, we consider all combinations and find a minimal hitting set which is then grown in each dimension. We formalize the complexity of the above algorithm by the following theorem.

**Theorem 1 (Hitting Set Algorithm).** Algorithm 2 finds all $m$-dimensional maximal matches in $O(n^mm^{-1}c)$ oracle calls, where $c$ is the maximum number of maximal matches containing a token.

Typically $c$ is a small constant, thus the complexity is dominated by $n^m$. We show that unfortunately, for relational matching, we
cannot avoid being exponential in the dimensionality through the following result.

**Theorem 2** (Lower Bound). In the worst case, finding a relational match for a nonterminal \( N = (C_1, \ldots, C_k) \) needs \( \Omega(n^m) \) oracle calls.

### 3.3 Stitching

The input to the stitching module is an NFA \( Q \) and the query \( q \). As described in Section 3.1, the NFA \( Q \) is either the original NFA \( Q_{\text{orig}} \) corresponding to \( G \) or a pruned NFA produced by the filtering module as we describe in Section 3.4. For now, we assume that \( Q = Q_{\text{orig}} \). In other words, filtering is not used.

We describe how we generate the NFA \( Q \) from the input grammar \( G \), noting that this step happens only once, that too at preprocessing time. The NFA \( Q \) that we generate has the following three properties: (1) the transition edges of the NFA are nonterminals in \( G \), (2) the set of nonterminal sequences accepted by \( Q \) is exactly the set of nonterminal sequences in \( G \), and (3) there are no \( e \)-transitions in \( Q \). We can construct such an NFA using standard algorithms [22], while ensuring that the number of states in \( Q \) is linear in the size of \( G \) and the number of transitions, quadratic in the size of \( G \). Figure 3 shows part of the NFA corresponding to patterns 2 and 3 in Table 3. We first present a general algorithm for stitching that works for any linear function; subsequently, we present a more efficient specialized algorithm that can be used when the grammar \( G \) satisfies some conditions.

**Algorithm 2** Relational Nonterminal Matching Algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input: ( q = t_1, \ldots, t_n ): search query; ( N = (C_1, \ldots, C_k) )</td>
</tr>
<tr>
<td>2</td>
<td>maxMatches ( \leftarrow \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>for start locations ( a_1, a_2, \ldots, a_k ) in lexicographic order do</td>
</tr>
<tr>
<td>4</td>
<td>while true do</td>
</tr>
<tr>
<td>5</td>
<td>if all hitting set candidates for start location are analyzed then</td>
</tr>
<tr>
<td>6</td>
<td>break</td>
</tr>
<tr>
<td>7</td>
<td>for ( i \in 1, \ldots, k ) do</td>
</tr>
<tr>
<td>8</td>
<td>( b_i \leftarrow \text{end location for } j \text{th hitting set candidate} )</td>
</tr>
<tr>
<td>9</td>
<td>end for</td>
</tr>
<tr>
<td>10</td>
<td>end for</td>
</tr>
<tr>
<td>11</td>
<td>cand ( \leftarrow ([a_1, b_1], \ldots, [a_k, b_k]) )</td>
</tr>
<tr>
<td>12</td>
<td>if ( \text{CONTAINED}(q, \text{cand}), (C_1, \ldots, C_k) ) then</td>
</tr>
<tr>
<td>13</td>
<td>for ( i \in 1, \ldots, k ) do</td>
</tr>
<tr>
<td>14</td>
<td>( \text{double } b_i \text{ while } \text{CONTAINED}(q, \text{cand}), (C_1, \ldots, C_k) )</td>
</tr>
<tr>
<td>15</td>
<td>end for</td>
</tr>
<tr>
<td>16</td>
<td>( \text{UPDATE}(\text{maxMatches, cand}) )</td>
</tr>
<tr>
<td>17</td>
<td>end if</td>
</tr>
<tr>
<td>18</td>
<td>( j \leftarrow j + 1 )</td>
</tr>
<tr>
<td>19</td>
<td>end while</td>
</tr>
<tr>
<td>20</td>
<td>end for</td>
</tr>
<tr>
<td>21</td>
<td>Return maxMatches</td>
</tr>
</tbody>
</table>

Figure 3: NFA for patterns 2 and 3 in Table 3

**Algorithm 3** Parse using a linear scoring function

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure ( \text{PARSE}(q, Q) ) ( \triangleright q = t_1, \ldots, t_n ): search query ( \triangleright Q: e )-free NFA</td>
</tr>
<tr>
<td>2</td>
<td>( \text{bmScore} \leftarrow \text{COMPUTE BEST MATCH SCORES}(q, Q) )</td>
</tr>
<tr>
<td>3</td>
<td>for ( p = 1 ) to ( n ) do</td>
</tr>
<tr>
<td>4</td>
<td>for all ( s ) in ( Q.\text{States} ) do</td>
</tr>
<tr>
<td>5</td>
<td>( \text{bScore}[s, p] \leftarrow -\infty )</td>
</tr>
<tr>
<td>6</td>
<td>for all ( (s', p), (s, M) ) in ( Q.\text{Edges} ) do</td>
</tr>
<tr>
<td>7</td>
<td>( \text{for } p' = 1 ) to ( p - 1 ) do</td>
</tr>
<tr>
<td>8</td>
<td>if ( \text{bScore}[s', p'] + \text{bmScore}[N, p', p] ) ( \triangleright ) ( \text{score} \leftarrow \text{bScore}[s', p'] + \text{bmScore}[N, p', p] )</td>
</tr>
<tr>
<td>9</td>
<td>if ( \text{bScore}[s, p] &lt; \text{score} )</td>
</tr>
<tr>
<td>10</td>
<td>( \text{bScore}[s, p] \leftarrow \text{score} )</td>
</tr>
<tr>
<td>11</td>
<td>endfor</td>
</tr>
<tr>
<td>12</td>
<td>endfor</td>
</tr>
<tr>
<td>13</td>
<td>endfor</td>
</tr>
<tr>
<td>14</td>
<td>endfor</td>
</tr>
<tr>
<td>15</td>
<td>( \text{overallBestScore} \leftarrow -\infty )</td>
</tr>
<tr>
<td>16</td>
<td>for all ( s ) in ( Q.\text{AcceptStates} ) do</td>
</tr>
<tr>
<td>17</td>
<td>if ( \text{bScore}[s, n] &gt; \text{overallBestScore} )</td>
</tr>
<tr>
<td>18</td>
<td>( \text{overallBestScore} \leftarrow \text{bScore}[s, n] )</td>
</tr>
<tr>
<td>19</td>
<td>endfor</td>
</tr>
<tr>
<td>20</td>
<td>return ( \text{overallBestScore} ) ( \triangleright ) and its parse</td>
</tr>
<tr>
<td>21</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

**General Algorithm**

The general stitching algorithm (Algorithm 3) uses dynamic programming to compute, for every state \( s \) in NFA \( Q \) and every position \( p \in [1, n] \) in the query \( q \), the score of the best parse for the prefix \( t_1, \ldots, t_p \) that ends in state \( s \); we denote this score \( \text{bScore}[s, p] \). We can show that the best parse for \( t_1, \ldots, t_p \) that ends in state \( s \) involves an optimal parse for some \( t_1, \ldots, t_{p'} \) \( (p' < p) \) that ends in a state \( s' \) along with a match for a nonterminal \( N \) that can be used to transition from \( s' \) to \( s \) in NFA \( Q \). Formally:

\[
\text{bScore}[s, p] = \max_{s', p', N, M} \text{bScore}[s', p'] + g(q, N, M) \tag{1}
\]

where we range over all \( s', p', N, M \) such that \( p' < p, (s', N, s) \in Q.\text{Edges} \), \( M \) is a match for \( N \), and \( \text{Extent}(M).\text{Start} = p' \) and \( \text{Extent}(M).\text{End} < p \). Recall that \( g \) is the “local” function of the linear function \( F \) (see Definition 2).

We incorporate two improvements over the naive dynamic programming algorithm suggested by Equation 1. First, we compute the value of the second term in Equation 1 for every possible value of \( N, p', p \). We can show that this improves the efficiency of dynamic programming since the second term does not depend on \( s' \). Formally, we compute:

\[
\text{bmScore}[N, p', p] = \max_M \text{match of } N[p', r < p] g(q, N, M) \tag{2}
\]

We note that \( \text{bmScore}[N, p', p] \) is the best score of a match for nonterminal \( N \) that starts at position \( p' \) and ends before \( p \). This score is computed in \( \text{COMPUTE BEST MATCH SCORES} \) (Algorithm 4). We can now rewrite Equation 1 as:

\[
\text{bScore}[s, p] = \max_{s', p', N} \text{bScore}[s', p'] + \text{bmScore}[N, p', p] \tag{3}
\]

where we range over \( s', p', N \) as before. The relation formalized in Equation 3 forms the basis for Algorithm 3; see Steps 5-12.

Second, to compute \( \text{bmScore}[N, p', p] \), we need to iterate over all matches \( M \) for nonterminal \( N \) that start at \( p' \) and end before \( p \). This step can be expensive since the number of matches for a nonterminal can be exponential. As described in Section 3.2, for this, we push some score-based filtering into the matching step. Instead of returning all matches for \( N \), \( \text{FIND MATCHES} \) returns at most \( \binom{n}{2} \) matches that dominate all others.
Algorithm 4 Compute best matches for each segment

1: function COMPUTEBESTMATCHSCORES(q, Q) ▷ Matching Module
2: for each nonterminal N in Q do
3: \( M \leftarrow \text{FINDMATCHES}(q, N) \)
4: for \( p' = 1 \) to \( n \) do
5: \( \text{bestScore} \leftarrow -\infty \)
6: for \( p = p' + 1 \) to \( n \) do
7: if \( \exists M \in M : \text{Extent}(M) = [p', p] \)
8: \( \text{bestScore} \leftarrow \max(\text{bestScore}, q(q, N, M)) \)
9: \( \text{bmScore}[N, p', p] \leftarrow \text{bestScore} \)
10: endfor
11: endfor
12: end for
13: return \( \text{bmScore} \)
14: end function

Time and space complexity: The time complexity of Algorithm 3 depends on the matching step (Step 3, Algorithm 4). We showed in Section 3.2 that the time complexity of each \text{FINDMATCHES} call is \( O(\text{output size}) = O(n^2) \), ignoring a one-time processing cost, which depends on the matching function and the kind of nonterminal (basic vs relational) being matched. The time complexity of dynamic programming (Steps 3-14) in Algorithm 3 is \( O(En^2) \) where \( E = |Q, \text{Edges}| \) is the number of edges in NFA \( Q \). Within COMPUTEBESTMATCHSCORES we perform \( O(n^2) \) work for each nonterminal \( N \) and the number of nonterminals is bounded by \( E \). Thus, the overall time complexity of Algorithm 3 is \( O(En^2) \).

The space complexity is \( O(Sn) \), where \( S = |Q, \text{States}| \) is the number of states in NFA \( Q \), since we use \( O(n) \) space for each state in \( Q \).

Algorithm for coverage function

We present a more efficient stitching algorithm for the coverage scoring function when the matching functions for all the columns are monotonic (see Definition 4) and all nonterminals are basic (nonrelational). The ideas presented here are applicable even when only a subset of columns use monotonic matching functions and even when only a subset of nonterminals are basic. These details are omitted due to space considerations.

The more efficient stitching algorithm (that we call \text{PARSECM}) is presented in Algorithm 5. This algorithm too uses dynamic programming and for every position \( p \in [1, n] \) and state \( s \), computes the best parse of \( t_1, \ldots, t_p \) that ends in state \( s \). To simplify our presentation, we use the noise function to determine the best parse instead of the coverage scoring function. (Our techniques apply to the general variant.) Recall from Example 5 that noise is defined to be the number of tokens not covered by a parse, and the best parse is one with minimum noise. We use \( \text{noise}[s, p] \) to denote the minimum noise of a parse ending at state \( s \) and position \( p \).

Figure 4 illustrates the two possible scenarios for the best parse ending at state \( s \) and position \( p \) depending on whether the token \( t_p \) is covered by the parse (case (a)) or not (case (b)). For case (a), as illustrated in Figure 4, assume that we transition from state \( s' \) to state \( s \) using a match \( M \) for nonterminal \( N \). Since we consider only basic nonterminals, match \( M \) is a single interval of the form \([p', p]\). For this case, \( \text{noise}[s, p] = \text{noise}[s', p'] \), since there are no uncovered tokens in the interval \([p', p]\). For case (b), the token \( t_p \) is not covered, and therefore \( \text{noise}[s, p] = \text{noise}[s, p - 1] + 1 \). In summary:

\[
\text{noise}[s, p] = \min\left\{\begin{array}{ll}
\min_{s', p'} \text{noise}(s', p') & \text{Case (a)} \\
1 + \text{noise}(s, p - 1) & \text{Case (b)}
\end{array}\right.
\]

where we range over all \( s', p' \), \( N \) such that \( p' < p, (s', s, N) \in Q, \text{Edges} \), and there exists a match \( M = [p', p] \) for nonterminal \( N \).

We can compute Case (a) of Equation (4) by computing for each edge \((s', N, s) \in Q, \text{Edges}\) the value:

\[
\min_{p' : (p', p) \text{is a match for } N} \text{noise}(s', p')
\]

and taking the overall minimum across all edges. When the matching function for \( N \) is monotonic, the set of \( p' \) values in Expression (5) happens to be contiguous. Let \( \ell_N(p) \) denote the smallest position \( p \) such that \([p, p]\) is a match for \( N \). (We define \( \ell_N(p) = -\infty \) if \( p \) is not contained in any match for \( N \).) From monotonicity, it follows that for every \( \ell_N(p) \leq p' < p \), the interval \([p', p]\) is a match for \( N \); further, \([p', p]\) is not a match for any \( p' \) outside of this range. The steps 11-14 of Algorithm 5 implement the computation of Expression (5).

The \( \ell_N(p) \) values for all \( p \in [1, n] \) can be computed in \( O(n) \) time using the maximal matches of \( N \) provided the maximal matches are sorted by their start positions. Recall that a maximal match of a basic nonterminal \( N \) is a match that is not contained in any other match of \( N \). This computation relies on the observation that \( \ell_N(p) \) is the earliest start position of a maximal match of \( N \) containing \( p \) (if one exists). It follows that Step 11 of Algorithm 5 can be implemented in \( O(1) \) amortized time.

The number of positions \( p' \in \ell_N(p), p \) can be \( \Theta(n) \) and therefore Step 12 of Algorithm 5 can take \( \Theta(n) \) time as stated. We can
exploit the monotonicity of the matching function to implement
this step in amortized $O(1)$ time. In particular, we can show that
the monotonicity of the matching function implies that $\ell_N(p_1) \leq \ell_N(p_2)$ whenever $p_1 \leq p_2$. Algorithm 6 describes how this property
of $\ell_N(p)$ can be used to compute Step 12 for all values of $p \in [1, n]$ in $O(n)$ time, implying that each step takes $O(1)$ amortized
time. Intuitively, this algorithm maintains a list $(i, v(i))$ such that
$i$ increases and $v(i)$ decreases as we traverse the list from be-
ingning to end.

Time and space complexity: The time complexity of Algorithm 5
depends on the time complexity of the finding the maximal matches
(Step 3). As before, we can show that each FINDMAXMATCHES
call takes $O($output size$)$ time, ignoring a one-time processing cost
(described in Section 3.2). Since no two maximal matches can be
contained in one another, the number of maximal matches for
a nonterminal is bounded by $n$. This implies that the time com-
plexity of Steps 2-3 is $O(En)$ since the number of nonterminals is
bounded by $E$. As argued earlier, each of the inner steps 7-14 can
be implemented in $O(1)$ amortized time and therefore the overall
time complexity of Algorithm 5 is $O(En)$. The space complexity,
as for the general algorithm, is $O(Sn)$.

3.4 Filtering

The now present the details of the filtering step. The filtering step
takes as input the original NFA $Q_{orig}$ corresponding to grammar $G$,
the query $q$ and returns a pruned NFA $Q$ that has a subset of states
and edges of $Q_{orig}$. NFA $Q$ has the property that any valid parse of
$q$ in $Q_{orig}$ is also a valid parse in $Q$. Therefore the output produced
by the stitching module is the same for both $Q_{orig}$ and $Q$. However,
the goal of the filtering step is to make $Q$ as small as possible so
that the stitching and matching steps are more efficient.

To implement the filtering step, in a preprocessing step, we com-
pute for every column $C$ referenced in $G$, the set of tokens con-
tained in $C$. The data structure that we use to store this information
can have false positives (incorrectly identify a token as occurring in
a column) but no false negatives (all true occurrences must be iden-
tified); for example, a bloom filter can be used for this purpose.

To describe the filtering step, we begin by assuming $Q_{orig}$ has
only basic nonterminals. Algorithm 7 presents the filtering step.
The filtering step identifies a set of active states in $Q_{orig}$. Initially
(at step 0) only the start state in $Q_{orig}$ is active. At step $i$, we con-
sider token $t_i$. A new state $s$ becomes active if there exists an edge
$(s', (C), s)$ such that $s'$ is active at end of step $i - 1$ and column $C$
contains token $t_i$. Once a state becomes active, it remains active in
all future steps. After we process all the input tokens, we remove
any state that cannot reach an active accepting state using an path
of active states from the list of active nodes. The output NFA $Q$ is
simply the subgraph of $Q_{orig}$ induced by the active states. We can
show that the overall time complexity of this algorithm is $O(En)$. To
deal relational nonterminals, we replace every edge $(s', N, s)$
where $N = \langle C_1, \ldots, C_m \rangle$ with a sequence of $m - 1$ (new) states
$s_1, s_2, \ldots, s_{m-1}$ and $m$ edges $(s', (C_1), s_1), (s_1, (C_2), s_2), \ldots,
(s_{m-1}, (C_m), s)$, and run Algorithm 7 over the resulting NFA. We
collapse the expanded edges back before returning the output NFA
$Q$. It is easy to see that $Q$ does not miss any valid parses that $q$
matches in $Q_{orig}$.

4. EXPERIMENTS

As noted earlier in the paper, we know of no prior work that
proposes efficient algorithms for parsing-based keyword search.
Therefore, the goal of our experimental evaluation is to study the
overall performance of our parsing infrastructure as a function of
the number of patterns, the data size, the scoring function and the
query length. We use a (score) threshold variant of the problem,
not the best-parse variant presented, so as to be able to study the
performance of our algorithms as a function of the threshold.

Implementation: Our implementation is in-memory. Here, we re-
port performance for the containment matching function and cov-
verage score function (specifically, the noise function variant); we
include other results in the full version of the paper. As discussed
in Section 3.2, our implementation of the containment oracle is
based on the algorithm proposed in [2]; we also use the optimiza-
tion that exploits the overlap between successive oracle calls pro-
cessed in [9]. We build one set containment index per table across
all columns but invoke containment operations on the subset of
columns needed for parsing.

Data and Experimental Setup: We use two data sets for our
evaluation. The first dataset contains information about products
compiled by a large comparative shopping engine. There are 200
tables in the data set, each containing information about one prod-
uct category. The ith product category table Category$i$ contains
product name as well as other attributes relevant to the category
as columns. The total number of rows across all product category
tables is around 1 million. The other data set is a single table of
about 2 million proprietary addresses that we will refer to as Ad-
dress.

The patterns that we use are of the form (Category$i$, columns)
(Address columns) intended to represent queries looking for spe-
cific products near a given address. We enforce relational con-
strains on the product columns as well as on the address columns separately (there is no joint relational constraint). Thus, there are two relational nonterminals in each pattern, one corresponding to Category, and one corresponding to Address. Patterns are generated from each table Category_i by selecting different subsets of columns from Category_i (for the first relational nonterminal) followed by a subset from Address (for the second). This method of pattern generation lets us control the number and size of the patterns. For a given set of patterns, the overall grammar is the disjunction of all the patterns. Queries are generated for a given pattern by choosing tokens at random from the columns in a pattern while respecting relational constraints. We also add random noise to the query strings by injecting spurious tokens.

Our default setup utilizes tables Category_i, for i ∈ {1 . . . 100}. The patterns are all combinations of two columns from each Category_i table followed by two columns from Address — this corresponds to a total of 5796 patterns. The average query length is 5.2 tokens and the noise threshold of the parsing is set to 0.2 (i.e., 20% of the parse can be noise.) Each experimental setting is averaged over 10000 random queries. Finally, our experiments are conducted on a workstation running the Enterprise edition of Windows Server 2008 with a dual core AMD Opteron 2.19 GHz processor and 16GB of RAM.

Overall Performance and Varying Number of Patterns: We first consider the performance of our parsing algorithm as the number of patterns input increases. We increase the number of patterns by varying the number of product categories as well as the number of columns used to define a pattern. Figure 5(a) shows the results of our study. The X-axis shows the number of patterns and the Y-axis reports the average parsing time per query in milliseconds. The parsing time is divided among the three parts of the algorithm — filter, matching and stitching.

We observe from Figure 5(a) that even with 10000 patterns and data size containing a total of 3 million rows (over all tables), the total parsing time is less than 7ms per query.

Next, we observe that generally, the parsing time is dominated by the matching step. However, when the number of patterns increases, the filtering phase also begins consuming substantial time. The stitching time is consistently small throughout although the absolute value of the stitching time also increases with the number of patterns. This behavior is expected — the filtering step has to effectively consider all the input patterns for a given query. However the output of the filtering step is typically a much smaller number of patterns implying that the NFA size for stitching is typically small. Matching is the only step that accesses the database; thus, even though matching happens only over a small subset of non-terminals, it is relatively expensive.

Data Size: We next study the parsing performance as a function of the data size. We vary the data size by considering different subsets of the Address table. Figure 5(b) reports the result of the study. The X-axis reports the number of rows in the Address table and the Y-axis, the parsing time. As noted earlier, the number of patterns in our default setup is 5796 (corresponding to 100 product categories). The only part of the parsing that is sensitive to the data size is the matching step that issues set containment lookups. We find that the cost increases only sub-linearly with data size. This is because our containment checks are optimized for incremental lookups, do not require the full result to be materialized and use an output-sensitive index.

Varying Noise And Query Length: We now study the parsing performance as a function of the noise threshold and query length. Figures 6(a) and (b) show the results. The X-axis respectively shows the noise (as a fraction of the query length) and the average query length. The Y-axis shows the average parse time. As expected, the parsing time does increase with both the noise as well as the query length. The absolute value of each of the steps increases with the above parameters.

Matching Algorithms: From the above discussion, we see that the matching step consumes a significant fraction of the overall parsing time. In fact when the number of patterns is not very large, the cost of matching dominates the cost of parsing. This is expected because matching is the step where data access takes place and is therefore sensitive to the data size. We now compare the performance of our matching algorithm with a brute-force alternative that basically considers all interval sequences consistent with the token-level containment filter discussed in Section 3.4. We consider matching against the Address table with one relational non-terminal containing up to four columns with a relational constraint (⟨street^1 city^1 state^1 zip^1⟩). We vary the number k of columns in the relational non-terminal. For a given k, the patterns are different sets of columns of size k. For each column set, we generate query strings by choosing tokens at random from each column. The overall query length is fixed at 6 tokens.

Figure 6(c) shows the plot with the X-axis capturing the number of columns and the Y-axis showing the matching time (in ms). We find that the hitting set based algorithm yields a significant speedup of up to a factor of 4 over the bruteforce alternative even for relatively short queries of 6 tokens. We note that up to a point, the matching time increases with the number of columns but when the number of columns increases from 3 to 4, the matching time decreases for both algorithms. This is an artifact of the query generation process. Since the query length is fixed and since we choose at least one token from each column per query, a larger number of columns corresponds to a smaller range of tokens per column which can lead to more efficient matching as demonstrated in Figure 6(c).
5. RELATED WORK

The fields of string segmentation for information extraction [34] and keyword search [10] are the ones most closely related to our work. We first describe how a parsing-based keyword search approach (such as ours or the framework described in Fagin et al. [15]) contrasts with these two. Next, we describe detailed differences between our work and specific related papers.

Segmentation, Keyword Search and Parsing: String segmentation has goals similar to keyword search, in that the query is to be segmented into portions that match the database. However, the approach taken is different: segmentation relies primarily on local (typically token level) signals (e.g., is the word capitalized, does it follow a punctuation symbol), as well as dictionary signals (e.g., does the word match the database). These signals are weighted appropriately through a machine-learned HMM or CRF model.

On the other hand, keyword search treats a query as a bag of words (and therefore loses any ordering information), but relies on relational signals to (implicitly) interpret the query and retrieve results. Thus, the keyword search system correctly interprets the query gray transactions using relational signals to infer that the query derives from one record containing the (author) gray and (the book) transactions, while a segmentation approach would not be able to interpret this query correctly.

We can view a parsing-based keyword search system as bridging the spectrum between segmentation and bag-of-words keyword search and therefore inherits the benefits of both approaches. We can configure a parsing-based search system as a pure segmentation system or as a pure bag-of-words system, or anywhere in between. This allows us to explore tradeoffs in structured vs unstructured interpretations of query, exploiting local vs relational signals, and sophisticated vs simpler configuration alternatives. Note that our techniques can also be used to improve the performance of pure segmentation systems. Also, we hypothesize that relational signals could be a useful feature for segmentation quality, but this hypothesis requires detailed study and empirical validation and is outside the scope of this paper.

Our related work can be placed into 8 categories. We cover each of them in turn.

Parsing-based Keyword Search: Fagin et al. [15] introduced the problem of parsing-based keyword search. Given a grammar, a database of documents and a query, their goal is to return the most specific parse (i.e., the parse that matches the smallest number of documents). They show that this problem is intractable; however, they find that the problem of returning all relevant parses has polynomial complexity in the input and output. Subsequently, in [16], they add rewrite rules, and show that under certain conditions, finding all rewritten parses is decidable. In our work, we focus instead on efficiency, i.e., how do we design algorithms and index structures to efficiently support a general parsing-based keyword search.

We differ from prior work [15] in our use of a scoring function to express parse preference. In [15], ideas from query containment are used to define a partial ordering over parses. There are two advantages to the scoring function approach compared to the containment approach: (1) Containment based ordering is shown to be NP-hard [15], while scoring function ordering is tractable for many interesting classes of functions that we consider. (2) Scoring functions impose a total ordering over the parses unlike containment, so, e.g., the top-k parses are unambiguously defined.

Dictionary-based Segmentation: Chandel et al. [9] consider the problem of string segmentation while sharing lookups on the structured data. However, their work does not parse with a grammar or support relational constraints. However, we do use their algorithms to improve the efficiency of our containment lookups (Section 4).

Another recent work focuses on query segmentation using CRFs [37]. They use a notion similar to maximal matches to establish matches that also appear in the database. However, note that this work also does not consider relational constraints, and does not support a grammar.

There has also been some work on annotating web queries [35]. Once again, this work does not consider relational constraints or a grammar, and restricts matches to be tokens from a single table. However, note that some of their probabilistic techniques could be utilized to infer patterns.

General Keyword Search: There has been a large body of work on Keyword Search in databases over the last decade. (See [10] for a tutorial on the subject.) In particular, some popular database systems that support keyword search include DBXplorer [4], SPARK [30], BANKS [6] and Discover [23]. These systems focus on generating a connected subgraph of tuples, where the smaller the subgraph, the better it is ranked. However, these approaches do not possess the fine-grained control over user intent that parsing with a grammar allows. Also, to the best of our knowledge, none of these approaches allow special keywords (e.g., from, to) or noise tokens in the query.

Web Search: There has been some work recently on query segmentation by using click logs and query logs [31, 21] — however, these approaches do not leverage the rich structured information in the database, and do not consider relational constraints.

Query Expansion and Approximate Match: Recent work [11] has also looked at expanding on the notion of a “match” between a portion of the search query and a concept in underlying data (such as synonyms, hypernyms and hyponyms). Other papers tackle the problem using rewrite rules [16, 17], query substitutions [25], query cleaning [32], and efficient approximate entity extraction [27, 36]. These papers are orthogonal to our work (which focuses on efficiency of the basic parsing infrastructure) and can be used to potentially identify user intent even better.

Query Classification and Understanding User Intent: There has been a lot of recent work on query intent classification [29, 24, 5]. The goal of this work is to identify if a search query belongs to a particular category or not (e.g., navigational vs informational, travel vs nontravel). The output of a intent classification system is a categorical value that can be used, for example, to direct the query.
to an appropriate search vertical. In contrast, the output of a query parsing system is more detailed and the kinds of challenges and issues studied by the two problems are fundamentally different. We note that query classification and query parsing can complement one another since query parsing is relevant and useful even after we narrow down the search vertical using query classification.

XPath Streaming: There has been some work on identifying matches from a given set of XPaths [18, 13]. Most prior work in this area builds an automata and then “streams” the XML data through the automata. However, this body of work focusses on exactly matching the given XPath, does not have any underlying data or relational constraints, and therefore requires different techniques. Similar work has addressed the problem of retrieving and indexing regular expressions [8]. Once again, this work does not address the problem of matching against an auxiliary database in addition to a grammar.

Frequent Itemsets: Finally, we note that the relational matching problem is related to the well-known problem of finding frequent itemsets [19, 3]. While our overall solution is similar, we use information specific to our problem to carefully choose hitting sets and obtain an algorithm that under reasonable assumptions is worst-case optimal.

6. CONCLUSIONS
In this paper, we considered the problem of parsing a query according to a set of regular expression patterns using structured data. Our parsing semantics allows (1) a number of matching functions, including set containment, approximate match and equality, (2) relational constraints and (3) linear and coverage scoring functions, such as HMM, CRF and additive noise. We designed a parsing infrastructure around two modules: matching where data access is performed, and stitching along an NFA. We also developed a filtering module which prunes a large number of states from the NFA. We provide efficient algorithms for each of these modules. Our algorithms scale well with both data size and the size of the regular expression.

7. REFERENCES