An Algebraic Approach to Rule-Based Information Extraction

Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan

IBM Almaden Research Center
San Jose, CA, USA
[frreiss,rsriram,rajase,huaiyu,shiv]@us.ibm.com

***Abstract*— Traditional approaches to rule-based information extraction (IE) have primarily been based on regular expression grammars. However, these grammar-based systems have difficulty scaling to large data sets and large numbers of rules. Inspired by traditional database research, we propose an algebraic approach to rule-based IE that addresses these scalability issues through query optimization. The operators of our algebra are motivated by our experience in building several rule-based extraction programs over diverse data sets. We present the operators of our algebra and propose several optimization strategies motivated by the text-specific characteristics of our operators. Finally we validate the potential benefits of our approach by extensive experiments over real-world blog data.**

I. Introduction

Search and business intelligence applications are increasingly relying on the wealth of structured information that can be extracted from text [1], [2]. Information of interest to such applications ranges from mentions of entities and relationships (e.g., persons, phone numbers, addresses, etc.) [3], [4] to significantly more complex information such as reviews, opinions, and sentiments.

The area of rule-based information extraction (IE) has developed several rule languages and frameworks [5], [6], [7] for building such information extraction programs (called *annotators*). Since extraction is viewed as a sequential operation over text, such rule languages and their implementations are predominantly based on the theory of grammars and finite-state automata. However, there is a significant issue with the scalability of such approaches, particularly as the complexity of the annotators and the size of the document collections increase. To illustrate these issues, let us consider the following relatively complex extraction task of identifying certain kinds of reviews from blogs.

Example 1 (Extracting informal reviews from Blogs): Consider the task of extracting, from blogs, informal reviews of live performances by music bands. Figure 1 shows the high-level organization of an annotator that captures the domain knowledge needed to accomplish this task. The two individual modules ReviewInstance and ConcertInstance identify specific snippets of text in a blog. The ReviewInstance module identifies snippets that indicate portions of a concert review – e.g., "show was great", "liked the opening bands" and "Kurt Ralske played guitar". Similarly, the ConcertInstance module identifies occurrences of bands or performers – e.g., "performance by the local funk band Saaraba" and "went to the Switchfoot concert at the Roxy". The output from the ReviewInstance module is fed into the ReviewGroup module to identify contiguous blocks of text containing ReviewInstance snippets. Finally, a ConcertInstance snippet is associated with one or more ReviewGroups to obtain individual BandReviews.

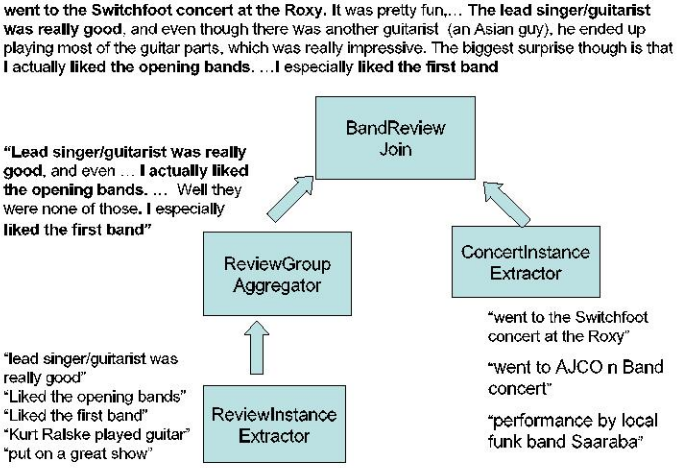

Fig. 1. Extraction of informal band reviews

A. The grammar approach

In a traditional rule-based IE system, the annotator described in Example 1 would be specified as a series of cascading grammars. To illustrate, let us consider a particular rule in the ReviewInstance module that is informally described as follows:

BandMember *followed within 30 characters by* Instrument

Fig. 2. Example extraction rule for ReviewInstance

A translation of this specification into a cascading grammar yields:

$$\begin{array}{lll} \text{ReviewInstance} & \leftarrow & \text{BandMember } .\{0,30\} \text{ Instrument} \quad (R_1) \\ \text{BandMember} & \leftarrow & \text{RegularExpression } (\texttt{[A-Z]\textbackslash w+(\textbackslash s+[A-Z]\textbackslash w+)*}) \quad (R_2) \\ \text{Instrument} & \leftarrow & \text{RegularExpression } (d_1|d_2|\ldots|d_n) \quad (R_3) \end{array}$$

Fig. 3. Cascading grammar rules

The top-level grammar rule R_1 expresses the requirement that the pattern BandMember and Instrument appear within

30 characters of each other. Executing R_1 invokes rules R_2 and R_3 which in turn identify BandMember and Instrument instances. In the interest of readability, we have include a simpler version of the fairly complex regular expression that we used for identifying BandMember instances [1]. For identifying Instrument instances, an exhaustive *dictionary* of instrument names is used. However, the actual implementation of a dictionary in a grammar-based system is via a regular expression expressed as a union of all the entries in the dictionary as shown in rule R_3.

The most popular and well-understood standard for cascading grammars is the Common Pattern Specification Language (CPSL) [5]. Using such a CPSL-like language, at the IBM Almaden Research Center, we have built a large number of annotators over several diverse data sets [8]. Our experiments indicate that a significant drawback of the cascading grammar implementations is their enormous execution time. For example, even after extensive performance tuning, the total running time for the annotator shown in Figure 1 *over 4.5 million blog entries is approximately eight hours*. Clearly such high execution times are a bottleneck in the widespread use of information extraction techniques.

1) Scalability Considerations: A careful analysis of our annotator revealed that the primary reason for such high execution times is the cost associated with the actual evaluation of each grammar rule. As an anecdotal data point, when executing only the 3 grammar rules listed in Figure 3 over 480K blog entries, the CPU cost of regular expression evaluation dominated all other costs (IO cost of reading in documents, generating output matches, etc.), accounting for more than 90% of the overall running time. Such high CPU cost is a consequence of the fact that for a grammar rule to be evaluated over a document, potentially every character in that document must be examined. As the number of rules increases, the associated CPU cost per document continues to grow, resulting in a large execution time over the entire collection.

Obviously, one approach to address this scalability problem is that of employing more hardware, distributing the document collection over a large number of processing nodes, and executing the annotators in parallel. However, our goal is to achieve scalability by improving the efficiency of the processing operations performed by the annotator.

To this end, we draw inspiration from the approach pioneered by relational database systems. By viewing data manipulation procedures as operators in an algebra, database query execution engines are able to consider equivalent but potentially faster execution plans for a given user query. *In this paper, we ask, and answer in the affirmative, the question of whether a similar approach, of treating annotator rules as queries in a formal algebraic framework, can be used to address the scalability problem described above.*

[1]The particular task that necessitated the extraction of such band reviews concerned the identification of new bands. This precluded the usage of any existing dictionary containing the names of current bands.

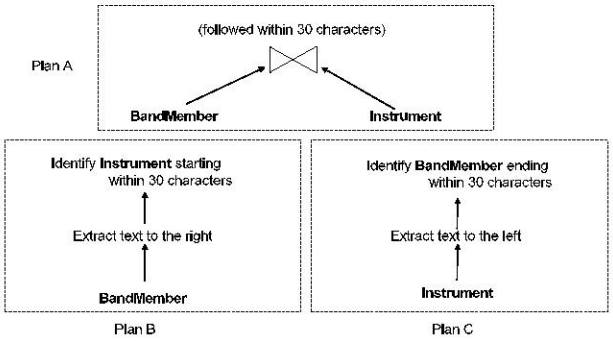

Fig. 4. 3 execution strategies for a CandidateReview Rule

B. An algebraic approach

We propose a simple data model and associated operator algebra for representing the text manipulation tasks that are performed by an annotator. There are two important aspects to our proposal: (i) since our focus is on information extraction tasks that only require the examination of a single document at any given time[2], our data model and algebra only represent intra-document operations; (ii) since the core operations of an annotator involve the generation or examination of contiguous regions of text, the fundamental concept in our algebra is that of a *span*, a region of text within a document identified by its "begin" and "end" positions.

A formal definition of a span, the algebra, and the associated optimization techniques form the subject of Sections II and III. However, to provide an intuition for how alternative execution plans can be developed for an annotator, let us revisit the cascading grammar rules listed in Figure 3. Three alternative plans for this rule are shown in Figure 4.

On each document, Plan A first evaluates BandMember and Instrument *independently* and then determines whether pairs of these annotations are within 30 characters of each other. Clearly, identifying BandMember and Instrument that are not within 30 characters of the other is wasted computation. Plans B and C exploit this very fact. Plan B first identifies all instances of BandMember. For each such instance, it identifies Instrument instances within a limited neighborhood to the right. On the other hand, Plan C first evaluates all instances of Instrument and for each instance evaluates BandMember in the "left" neighborhood. The running times for Plans A , B and C on 480K blog entries are 422 seconds, 391 seconds and 31 seconds respectively. The dramatic difference in running time for Plan C is due to the high selectivity of Instrument as compared with the low selectivity of BandMember. This optimization strategy (referred to as restricted span extraction), along with other text-specific optimization strategies are described in detail in Section III.

[2]For instance, a task such as collection-level entity resolution that involves examining the text of multiple documents is beyond the purview of this work.

Contributions

In this paper we propose a departure from traditional grammar-based information extraction systems to address the computational issues that have precluded scalable rule-based information extraction. Specifically, we claim the following contributions:

- An algebraic approach to rule-based information extraction
- An algebra consisting of span and text-specific operators based on our experiences with building information extraction modules over a wide range of data-sets
- Three text-specific optimization strategies that emerge from a holistic view of an information extraction task
- Extensive experiments over 4.5 million blog entries to evaluate the benefits of this algebraic approach and comparisons against an implementation of a popular grammar-based specification.

TABLE I

OPERATOR ALGEBRA FOR INFORMATION EXTRACTION

Operator class	Operators
Relational operators	σ, π, $\times$, $\cup$, $\cap$, ...
Span extraction operators	$\mathcal{E}_{re}$, $\mathcal{E}_d$
Span aggregation operators	Ω_o, Ω_c, β

TABLE II

SPAN PREDICATES

Predicate	Explanation
$s_1 \preceq_d s_2$	s_1 and s_2 do not overlap, s_1 precedes s_2, and there are at most d characters between the end of s_1 and the beginning of s_2
$s_1 \simeq s_2$	the spans overlap
$s_1 \subset s_2$	s_1 is strictly contained within s_2
$s_1 = s_2$	spans are identical

II. ALGEBRA AND DATA MODEL

In this section, we describe a simple object-relational data model for representing annotations over a given document. We then present a set of logical operators over this model and demonstrate that complex rule-based annotators (including the ones described in Section I) can be expresed as compositions of these operators.

A. Data and Execution Model

Our algebra is designed to extract annotations from a single document at a time, and we define the algebra's semantics in terms of the current document being analyzed. We model the current document as a string called doctext.

Each annotator finds regions of doctext that satisfy a set of rules and marks each region with an object called a *span*. A span is simply an ordered pair $\langle begin, end \rangle$ that denotes the region of doctext from position *begin* to position *end*.

To make the examples in this section easier to understand, we also include the text of the span's region in our notation. For example, if doctext was the string "`Information extraction`", $\langle 13, 22 \rangle$: "extraction" would denote the range from characters from positions 13 to 22 of the document.

Our algebra operates over a simple relational data model with three data types: span, tuple, and relation. In our data model, a *tuple* is an finite sequence of w spans $\langle s_1, ..., s_w \rangle$; we call w the *width* of the tuple. A *relation* is a multiset of tuples, with the constraint that every tuple in the relation must be of the same width. Each operator in our algebra takes zero or more relations as input and produces a single output relation.

B. Local Annotation Database.

Our algebra runs over a *local annotation database* consisting of the current document and a set of *annotation relations* that represent precomputed annotations. As part of the process of loading a document, the system computes a set of useful general-purpose annotations like Sentence, Paragraph, Noun, and Verb and inserts these annotations into the local annotation database. Since a local annotation database only deals with a single document, it generally fits entirely in main memory.

A collection of local annotation databases forms a *global annotation database*. To annotate all the documents in a global annotation database, our execution framework applies an algebra expression to every local annotation database separately. Execution proceeds as follows:

```
E ←{ algebra expression }
for localDB in globalDB do
begin
    1. { Read localDB into main memory }
    2. R ← E(localDB)
    3. { Add R to localDB }
    4. { Write modified localDB to disk }
end
```

To run multiple annotators in a single pass, step 2 in the above process can be repeated multiple times per document. In our experience, step 2 of this loop is by far the most time-consuming part of the annotation task, even for a very simple annotator running in isolation.

C. Operator Algebra

The set of operators in our algebra can be categorized broadly into *relational operators*, *span extraction operators*, and *span aggregation operators* as shown in Table I. Since our data model is a minimal extension to the relational model, all of the standard relational operators (select, project, join, etc.) apply without any change. The only addition is that we use some new selection predicates (Table II) that apply only to spans[3]. The rest of this section is devoted to a description of the span extraction and aggregation operators.

D. Span Extraction operators

Span extraction operators identify segments of text that match a particular input pattern and produce spans corresponding to each such text segment. Since text pattern matching is

[3]Note that several other predicates are possible [9] but we only list those that are used in this paper.

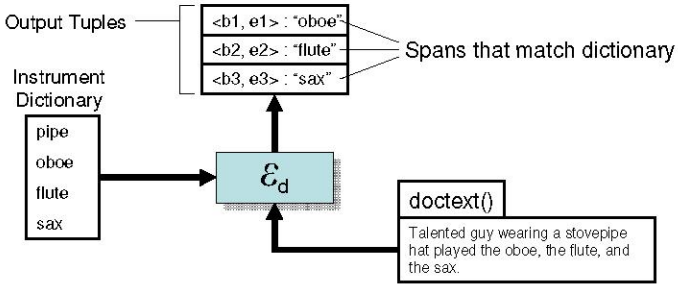

Fig. 5. Span extraction using the dictionary operator $\mathcal{E}_d$.

at the core of almost any information extraction task, these extraction operators are the workhorses of our algebra. Before delving into the details of specific extraction operators such as the regular expression matcher or the dictionary matcher, we describe their general form as follows.

Definition 1 (Span extraction operator): For a function f : $\langle \text{Pattern}, \text{String} \rangle \rightarrow \{\text{Span}\}$ that maps a string to a set of pattern matches within the string, the corresponding *span extraction operator* $\mathcal{E}_f(\text{Pattern})$ returns the maximal set of tuples $\{T_1, \ldots, T_n\}$, where each T_i consists of a span from $f(\text{Pattern}, \text{doctext}())$.

Our algebra incorporates two kinds of span extraction operators:

- *Standard regular expression matcher ($\mathcal{E}_{re}$).* Given a regular expression r, $\mathcal{E}_{re}(r)$ identifies all non-overlapping matches when r is evaluated from left to right over the text represented by s. The output of $\mathcal{E}_{re}(r)$ is the set of spans corresponding to these matches.
- *Dictionary matcher ($\mathcal{E}_d$).* Given a dictionary *dict* consisting of a set of words/phrases, the dictionary matcher $\mathcal{E}_d(dict)$ produces an output span for each occurrence of some entry in *dict* within the current document text.

In the presence of a generic regular expression operator, a separate dictionary operator may appear to be redundant. However, there are three reasons for including such an operator. First, most regular expression engines only produce non-overlapping matches whereas the dictionary operator produces all possible matches for each dictionary entry. Second, regular expressions operate at the character level whereas dictionaries are at the level of tokens (i.e, words and phrases). Especially for dictionaries that can run into thousands of entries, the use of a character-level regular expression matching engine is very expensive. Finally, dictionaries automatically enforce the semantics of word boundaries, i.e., dictionary matches only include complete words and phrases. For example, as shown in Figure 5, even though "pipe" is an entry in the dictionary, the string "pipe" in the sentence is not a match as its part of a larger word.

E. Span Aggregation operators

Span aggregation operators take in a set of input spans and produce a set of output spans by performing certain aggregate operations over their entire input. While the precise details are

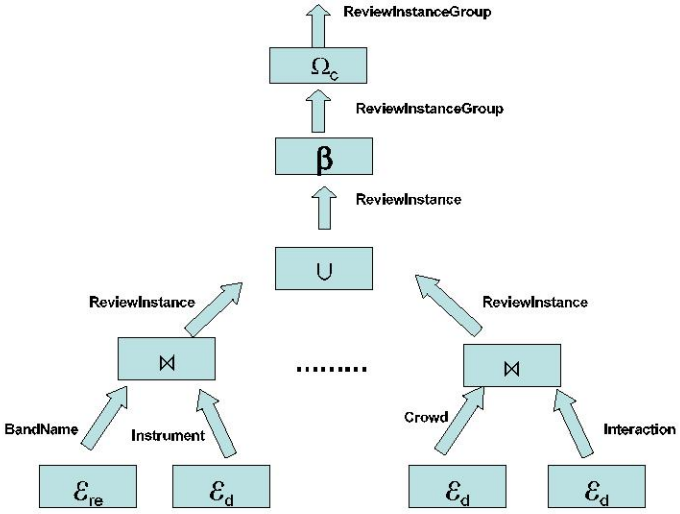

Fig. 6. Operator graph for ReviewGroup

different for the individual operators, the input and output of every span aggregation operator is a single-column relation of the form $R(a)$ where $R.a$ is of type Span. Below we describe three span aggregation operators: *containment consolidatation*, *overlap consolidation*, and *block*.

1) Consolidate: The consolidate operators are motivated by our observation that when multiple extraction patterns are used to identify the same concept, two different patterns often produce matches over the same or overlapping pieces of text. To resolve such "duplicate" matches, we introduce two kinds of consolidation operations.

- *Containment consolidation (Ω_c)* is used to discard annotation spans that are wholly contained within other annotation spans. Specifically, given a set of input spans, Ω_c produces as output only those spans in the input that are not contained within another. It is easy to show that containment consolidation can be expressed using relational operators by applying the correct span predicate.However, since it has been our experience that containment consolidation is a common operation in several extraction tasks, we have chosen to retain it as a first class operator in our algebra.
- *Overlap consolidation (Ω_o)* is used to produce new spans by merging overlapping spans. Given a set of spans as input, Ω_o produces a set of non-overlapping spans generated by repeatedly merging all possible spans in the input. In the interest of brevity, we do not discuss overlap consolidation in this paper but merely note that an expression for Ω_o in terms of relational operators requires a recursive fixed-point computation.

2) Block: The block operator (β) identifies a large span of text enclosing a set of input spans such that no two successive spans are more than a specified distance apart. Intuitively, the goal is to identify regions of text where input spans occur with enough regularity. For example, as shown in Figure 6, ReviewGroup is constructed by using the block operator to

identify regions of text containing regular occurrences of ReviewInstance.

The block operator takes in two user-defined parameters – a *distance constraint* and a *count constraint*. The distance constraint controls the regularity with which input spans must occur within the block and the count constraint specifies a minimum number of such input spans that must be contained within the block.

Formally, let the single-column relation $R(a)$, where $R.a$ is of type Span, be the input to a block operator β with distance constraint d and count constraint n. A span (b, e) is produced as output by this block operator if there exists a set of input spans $\rho((b, e)) \subseteq R.a$ such that:

- *B1.* No two spans in $\rho((b, e))$ are overlapping
- *B2.* Each span in $\rho((b, e))$ is contained within (b, e)
- *B3.* $|\rho((b, e))| \geq n$
- *B4.* Any two successive spans in $\rho((b, e))$ are separated by at most d characters[4]
- *B5.* $\exists\, (b, e_1) \in \rho((b, e))$ and $(b_1, e) \in \rho((b, e))$

The output of the block operator $\beta(n, d, R)$ is the set of all such spans that satisfy conditions B1..B5. Condition B5 ensures that every span output by the block operator begins and ends with one of the input spans.

III. Execution and Optimization

One of the primary reasons for developing an algebraic approach to information extraction is the opportunity to develop a principled *annotation optimizer* similar to database query optimizers. Indeed, since our data model and algebra build upon the standard relational model, all of the well-known strategies for generating alternative plans in the relational model (e.g., pushing down selections, re-ordering joins, etc.) are directly applicable. However, significantly more transformations can be performed by exploiting the semantics of the text-specific operators introduced in Section II. In this section, we present a suite of such techniques.

A. Observations

There are three important observations that guide the design of the techniques presented in this section:

- *(O1) Document-at-a-time processing.* In keeping with the per-document nature of information extraction, our algebra operates on a single document at a time. As a result, the individual per-document relations that our operators produce and consume are generally quite small and are often completely empty.
- *(O2) CPU-intensive text operations.* The core text processing operations of our algebra are the span extraction operators $\mathcal{E}_{re}$ and $\mathcal{E}_d$. In the absence of any index structures, these operators require the examination of each character or token in a document, resulting in significant CPU cost that often dominates the overall running time of an annotator.
- *(O3) Span properties.* A span is merely a special instance of the general mathematical object called an interval. Therefore, spans obey all of the natural properties of interval algebra [9] and these properties yield powerful transformation rules.

In this section, we describe three techniques for transforming annotator execution plans, inspired by these observations. The common thread underlying all our techniques is one of avoiding or reducing the effect of O2 by exploiting observations O1 and O3.

B. Shared Dictionary Matching (SDM)

Dictionary matching is a fairly expensive operation that involves tokenizing the current document's text and looking for all occurrences of the set of words and phrases listed in a specified dictionary. However, dictionaries are also fairly powerful information extraction primitives and therefore used quite often. For example, Figure 6 shows that ReviewInstance is computed as a union of spans produced by multiple subqueries. In our complete annotator, there were 39 such subqueries with a total of 69 instances of $\mathcal{E}_d$ involving 33 distinct dictionaries. The naive implementation of separately evaluating each instance of $\mathcal{E}_d$ proves to be extremely expensive. Even when documents are tokenized at the very beginning of the processing pipeline, an entire pass over these tokens for each $\mathcal{E}_d$ operator requires thousands of probes into the dictionary data structures.

To improve this performance, we propose a technique, called *shared dictionary matching (SDM)*, in which each dictionary is evaluated exactly once and the matches are used repeatedly as required. To implement SDM, we use two physical operators:

- A *DictEval* that produces a set of matching spans given a dictionary and a tokenized document
- A *Tee* operator that duplicates its input stream so that it can be fed into more than one operator further in the processing pipeline

The above version of SDM is effective in avoiding redundant computation when the same dictionary is used as part of multiple extraction patterns (e.g., the Instrument dictionary is used 9 times in the band review annotator shown in Figure 1). However, each distinct dictionary still required one complete pass over the tokens. Therefore, we extended SDM to use:

- a *MultiDictEval* operator that simultaneously produces matches for multiple dictionaries using a single scan over the tokens
- a modified version of the *Tee* operator that can forward a different set of dictionary matches over each of its output streams.

The use of SDM in the band review annotator improved document throughput by a factor of 3 in our experiments (See Section IV).

[4]Distance between non-overlapping spans is computed from the end of the first span to the beginning of the second span

$$BandMember(Blog, B) = \mathcal{E}_{re}(\text{“regexp”}, Blog)$$
$$Instrument(Blog, I) = \mathcal{E}_{d}(\text{“dictionary”}, Blog)$$
$$ConcertInstance = BandMember \bowtie_{B \preceq_{30} I} Instrument$$

Fig. 7. Algebraic expression for Plan A in Figure 4

C. Conditional evaluation (CE)

The idea in *conditional evaluation (CE)* is to avoid ev uating an entire subquery over a particular document if is possible to infer that that document is not going to yi any output annotations. For instance, consider the last st in the BandReview annotator in which ConcertInstance a ReviewBlock are joined together. If the subquery correspondi to ConcertInstance is evaluated first on each document, t evaluation of BandReview can be avoided on documents which there are no instances of the former. Note that the key this technique is the fact that the entire computation proce one document at a time, providing a natural granularity at which to implement such conditional evaluation. The symmetric transformation of evaluating ReviewBlock and conditionally evaluating ConcertInstance is also possible. The problem of choosing between these two conditional evaluation transformations requires appropriate statistics and cost estimation and we address some of these issues in Section III-E.

D. Restricted span extraction (RSE)

Both SDM and CE attempt to either reduce or eliminate work at the document level. In contrast, our third technique, called *restricted span extraction (RSE)*, operates at the subdocument level. The idea behind RSE is to restrict the evaluation of the expensive span extraction operators to some carefully chosen region(s) of text (as opposed to the entire document).

To illustrate this approach, let us revisit Plan A from Figure 4, expressed in our algebra as indicated in Figure 7. Notice that the join condition in Equation 3 involves a span predicate to enforce the requirement that Instrument must begin within 30 characters of the end of BandMember. Consider a particular BandMember instance b with a span $(10, 20)$. As per the join condition, we know that for an Instrument span to join with b, it must begin somewhere in the range $(21, 51)$. Let us further assume that the maximum length of any entry in the Instrument dictionary is 15 characters. It is easy to see that any Instrument instance that may potentially join with b can be identified by running the dictionary extractor in Equation 2 only over the span $(21, 66)$ (see below for why we must actually use $(20, 66)$). Thus, by examining only a portion of the document, the potential Instrument instances that join with b can be computed.

RSE optimization is a generalization of the technique illustrated by the above example. RSE is applicable for expressions, such as the one shown in Figure 8, involving a span-based join predicate p with one of the inputs to p computed using the dictionary operator. Similar expressions involving $\mathcal{E}_{re}$ instead of $\mathcal{E}_{d}$ are also amenable to RSE.

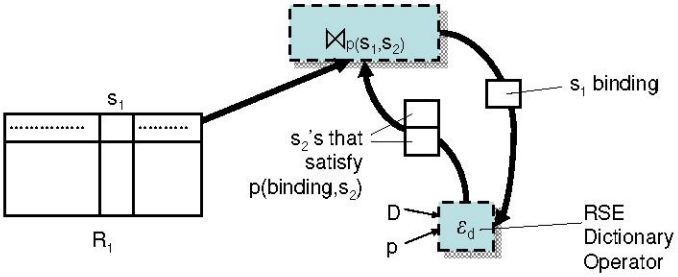

Fig. 8. Applying RSE

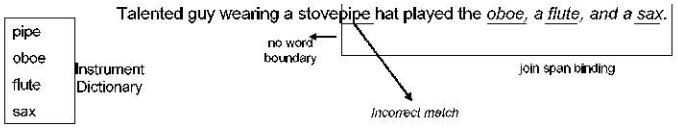

Fig. 9. Word boundary issue in dictionary matching

We have implemented versions of our extraction operators that accept *bindings* for all but one of the unbound variables in a given join predicate p. These *RSE extraction operators* compute the pattern matches that satisfy p for a given set of bindings, and they do so without examining the entire document. In our earlier example, we provided the intuition for how our RSE dictionary extraction operator works for the predicate $\preceq_d$. Our RSE implementation supports bindings for all the predicates listed in Table II.

When implementing an RSE extraction operator, two important issues must be considered:

- As mentioned earlier, dictionary matches enforce word boundaries, i.e., only match complete words or phrases. When restricting the execution of the dictionary extractor to a particular window of text, it is possible that spurious matches are returned at the two end-points of the window. Figure 9 illustrates how a spurious match for “pipe” can be produced for the same piece of text that was shown in Figure 5. To avoid this problem, we must examine one extra character at each end of the span to check for word boundaries. Thus, for our earlier example, the correct join span binding is $(20, 66)$.
- The design of an RSE regular expression extractor must take into account the left-to-right matching semantics of the regular expression operator. Typically, regular expression matches are evaluated in left-to-right order over the entire document. By evaluating a regular expression over an arbitrary window within this text, it may not be possible to precisely compute the set of matches in this window that would have been produced by evaluating over the entire document. Therefore, whenever $\mathcal{E}_{re}$ is involved, we adopt the conservative approach of using join span bindings to only compute the *end*-offset and always evaluating the regular expression from the very beginning of the document.

Due to lack of space, we omit further details of our RSE extraction operators.

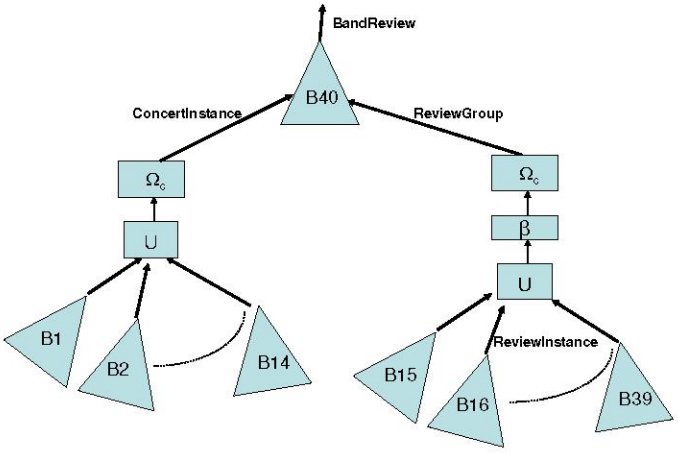

Fig. 10. Plan enumeration.

E. Annotation Plan Optimizer

In this section, we present the high-level design of an annotation plan optimizer based on the algebra and optimization techniques presented in this paper. We describe the space of execution plans that are considered and briefly sketch the algorithm for choosing an execution plan.

Given an operator graph for an annotator in terms of our algebra, the first step is to identify subgraphs that exclusively contain the operators σ, π, $\times$, $\mathcal{E}_d$, and $\mathcal{E}_{re}$ (i.e., a Select-Project-Join (SPJ) block extended to include the span extraction operators). In the case of the band review annotator, there are 40 such subgraphs as shown in Figure 10. We optimize each subgraph independently, but first we apply a topological sort to determine the order in which to process the subgraphs. The sort order ensures that the query plans for a given subgraph's inputs are computed before we optimize the subgraph itself.

Within each subgraph, we independently enumerate a space of possible plans by

- all possible join orders including ones that involve cross-products[5]
- standard transformations such as pushing down selections and projections to the extent possible, and
- additional plans generated by the application of the CE and RSE techniques introduced in Sections III-C and III-D.

In the simplest case, each subgraph would be treated independently, the least cost plan would be picked for each, and combined to produce the final plan.

However, with the SDM optimization, the cost of evaluating dictionaries is now amortized across subgraphs and must be carefully accounted for. There are two important considerations. First, sharing of dictionary computations is possible only between dictionary operators that are completely evaluated over a document, not when an optimization such as RSE has been applied to restrict the evaluation to a smaller span. Second, we can view the cost of executing dictionary matches as consisting of two parts: a certain fixed cost associated with tokenization and a variable cost associated with the actual matches produced by each operator. Given these considerations, we adopt an approach similar to the one used to handle *interesting orders* [10]. Essentially, for each subgraph B, we compute two optimal plans along with their associated costs:

- One plan under the assumption that tokenization cost can be amortized with another dictionary evaluation elsewhere in the query
- Another under the assumption that there is no dictionary evaluation in the rest of the query.

Once these pair of plans have been computed, a global pass over all the blocks is used to pick one of the two plans for each block and build the overall execution plan.

[5] Since the execution works one document at a time, the potential blow up in result size that is traditionally the problem with cross-product plans is not an issue in our scenario.

IV. Experiments

The goal of the experimental study is two-fold : (i) validate the performance benefits obtained by using an algebraic approach to information extraction (ii) understand and contrast the different optimization techniques presented in Section III.

A. Experimental Setup

The document corpus used in the experiments is a collection of 4.5 million web logs (5.1GB of data) crawled from http://www.blogspot.com. We used two annotators that identify informal reviews from these blogs (a) BandReview as shown in Figure 1 and (b) RestaurantReview, which identifies informal reviews of restaurants. Note that even though the two annotators are similar in spirit they have very different operator-graphs (omitted here for lack of space). All the experiments were run single-threaded on an IBM xSeries server with two 3.6GHz Intel Xeon CPUs.

B. Scalability Benefits of Using an Algebraic Approach

The first set of experiments compare the performance times between the grammar-based implementation and our proposed approach.

We started from a set of extraction rules such as the one shown in Figure 2, based on manual examination of blogs containing restaurant and band reviews. We then constructed three implementations of these extraction rules:

- GRAMMAR: A hand-optimized grammar-based implementation
- ALGEBRA$_{Baseline}$: Baseline obtained by manually constructing an algebra expression for each extraction rule
- ALGEBRA$_{Optimized}$: Plan obtained by applying the optimization algorithm presented in Section III-E over ALGEBRA$_{Baseline}$.

The execution times for BandReview and RestaurantReview are shown in Figures 11 from which we make the following observations:

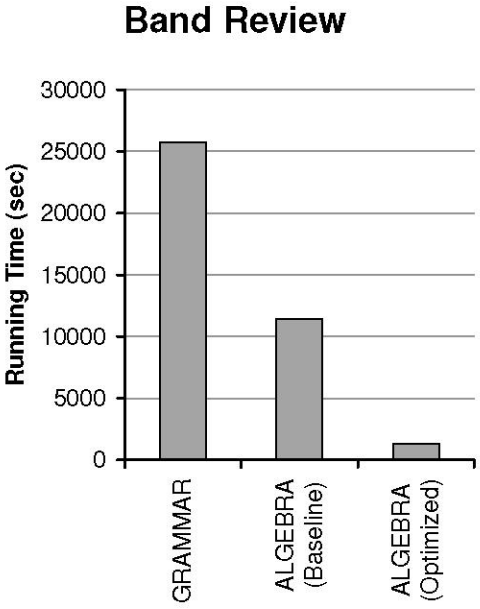

Fig. 11. Running time of the BandReview annotator

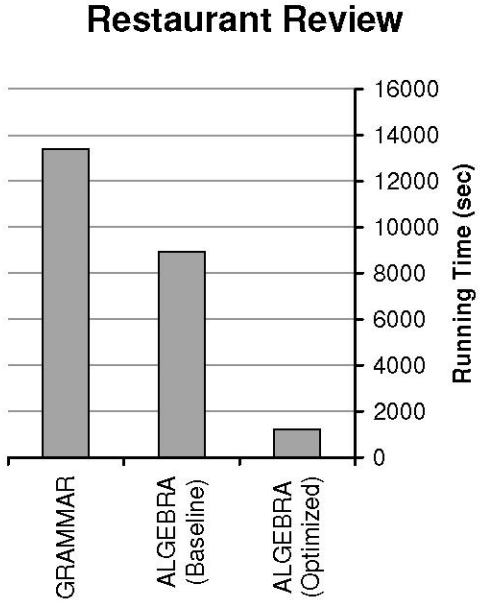

Fig. 12. Running time of the RestaurantReview annotator

- There is a two-fold improvement simply in moving from a grammar-based plan to an algebra-based plan.
- Applying the transformations discussed in Section III-E results in an order of magnitude improvement over ALGEBRA$_{Baseline}$.

Despite the fact that ALGEBRA$_{Baseline}$ and GRAMMAR represent the same extraction rules, there is still a significant improvement in running time. This is explained by the fact that every rule in a cascading grammar is evaluated over the *complete* text of the document. On the other hand operations in an algebra work only over the input annotations and consequently the running time depends primarily on the size of the input annotations. From this experiment it is clear that the value of an algebraic approach is unquestionable. The exact same information extraction task (BandReview) which took about eight hours in an optimized grammar-based implementation now runs in just under 30 minutes.

C. Analyzing the Various Transformations

In Section III, we discussed four categories of algebraic transformations: (i) Traditional (well-known relational optimization techniques such as join-reordering) (ii) Shared Dictionary Matching (SDM) (iii) Conditional Evaluation (CE) and (iv) Restricted Span Extraction (RSE). To understand the individual transformations and study their interactions with each other we ran multiple versions of BandReview. Each version applies a restricted combination of transformations and in total we experimented with seven combinations. Four combinations were obtained directly by applying each transformation individually. Two more were obtained by combining traditional with each of SDM and (RSE + CE) and the last one obtained by applying all transformations.

Figure 13 shows the relative improvement of each combination with respect to ALGEBRA$_{Baseline}$ based on which we observe:

- Individual transformations provide speedups that are dramatically different from each other. For example, traditional transformations provides no speedup and RSE a small 20%. On the other-hand CE and SDM give significantly greater speedups (a factor of 2 and 3 respectively). The improvement for CE is due to the gains obtained by pruning of an entire subtree (e.g., in Figure 1 the absence of a ConcertInstance enables the pruning of ReviewGroup and ReviewInstance. SDM shows even greater gains due to the fact that 33 dictionaries share computations.
- Combining a traditional transformation such as join-reordering with RSE and CE provides significantly larger speedups than using any of them separately. This is due to the fact that join reordering enables a larger number of applications of RSE transformations.
- As expected applying all four transformations provides a significant improvement over all other combinations.

D. Clean Semantics

While the primary motivation for our algebraic approach was to address problems of scalability, our approach has another significant advantage over cascading grammars. To illustrate, consider the following example that illustrates a common problem in complex information tasks, namely, *overlapping annotations*.

Example 2 (Overlapping annotations): Figure 14 shows two snippets of text drawn from real world blog entries. Snippet 1 has one instance of BandMember and two instances of Instrument while Snippet 2 has one instance of Instrument

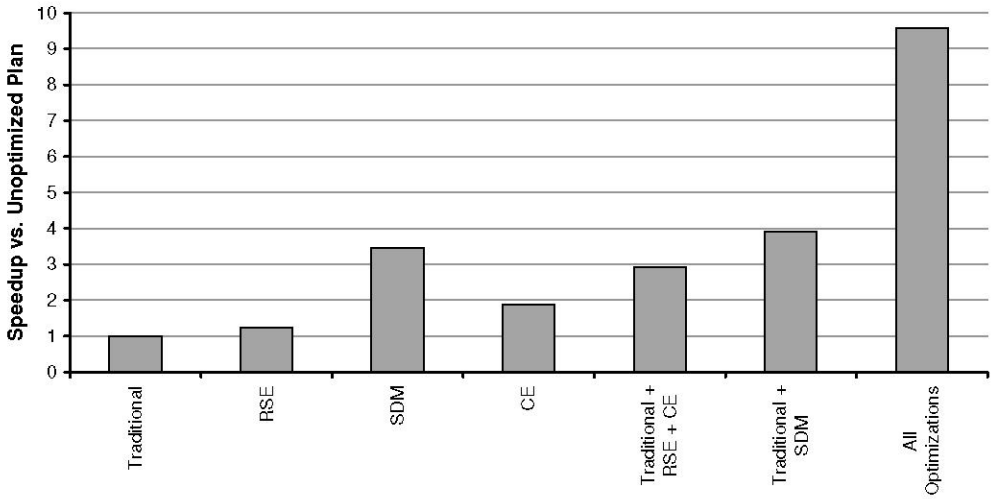

Fig. 13. Relative performance improvements from the initial plan for the BandReview annotator as different classes of optimizations are applied

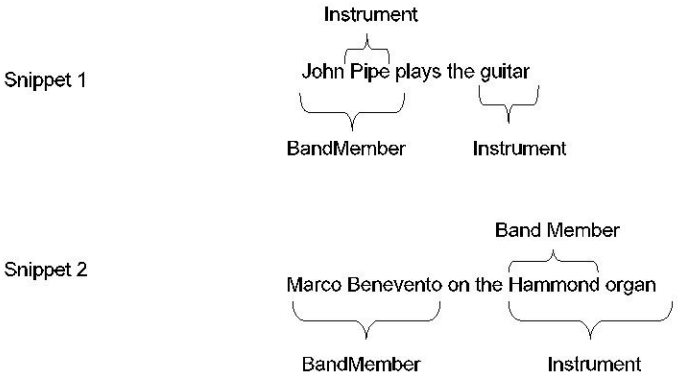

Fig. 14. Overlapping Annotations

and two instance of BandMember. Notice that both snippets have overlapping annotations. The text fragments "Pipe" in Snippet 1 and "Hammond" in Snippet 2 have both been identified as part of BandMember as well as Instrument.

There are two main reasons why annotations overlap: (a) individual rules are run independently, and (b) rules may make mistakes (in the sense that the author of that rule did not intend to capture a particular text snippet even though the snippet turned out to be a match). Since the input to a grammar must be a sequence of tokens, overlapping annotations must necessarily be disambiguated, i.e., "Pipe" must either be an Instrument or a part of BandMember and a similar choice must be made for "Hammond". In contrast, our algebraic approach considers the cross-product of BandMember and Instrument instances, thereby eliminating the need for such disambiguation.

In a grammar-based system, one of several ad hoc disambiguation strategies are employed. Two popular strategies are:(a) retain the annotation that starts earlier (e.g., BandMember for *John Pipe*), and (b) a priori, impose global tie-breaking rules (e.g., BandMember dominates Instrument). Using (a), the choice in the Snippet 2 is unclear since both annotations start at the beginning of *Hammond*. Using (b) and assuming BandMember dominates, Snippet 2 will not be identified by the cascading grammar in Figure 3. On the other hand, with the choice of Instrument dominating, Snippet 1 will not be identified.

1) Experimental Verification: To appreciate the true effects of such disambiguation, we ran two experiments using the rules from Figure 3 on 4.5 million blogs. When Instrument was chosen to be the dominant annotation, 6931 instances of ReviewInstance were identified. On the other hand, reversing the dominance resulted in only 5483 instances. Thus, with only three rules arranged into a 2-level cascading grammar, the number of resulting annotations varies dramatically depending on the choice of disambiguation. For extraction tasks with more rules, the situation can only become progressively worse. By considering the cross-product of BandMember and Instrument instances, our algebraic approach eliminates the need for such disambiguation.

V. Related Work

Information Extraction is a mature area of research that has received widespread attention in the NLP, AI, web and database communities (recent tutorials [11], [12], [13] provide details of prior work). Both rule-based approaches [5] and machine learning based approaches have been proposed [14], [15], [16]. A majority of this work is targeted towards improving the quality of results.

A. Frameworks for IE

Realizing the importance of sharing annotators, the NLP community has developed several software architectures, such as GATE [6], ATLAS [17], and UIMA [7]. The focus of these architectures is to provide a framework where annotators developed by different providers can be integrated and executed in a workflow. The mechanics of each individual annotators is opaque to the framework.

B. Optimizing IE

Recently, there has been work on improving the efficiency of information extraction tasks [18], [19], [20] in specific settings. For instance, [18], [20] provide techniques for scaling up the named-entity recognition whereas [19] proposes a technique to prune documents. In an alternative parallel effort, [21] considers the problem of efficiently composing multiple extraction modules into a larger program. In [21], basic extraction modules are viewed as IE predicates and users define whether these IE predicates satisfy certain properties. If so, specific optimization techniques become applicable. On the other hand, we propose an algebra that includes text-specific operations and present optimization techniques that exploit properties of these new operators.

C. Interval operations in other Domains

Interval operations are common in temporal databases [22] and extensive research in both data model and query language has culminated in temporal extensions to SQL. More recently, [23] proposed an algebra for querying biological datasets, another scenario where interval operations are common. While interval operations proposed in these contexts are applicable as span predicates, the span extraction and aggregation operations are specific to IE.

VI. Conclusions

In this paper we address the important problem of scalability of information extraction systems. We explain the limitations of existing grammar-based systems for scaling to current-day data sets and complex information extraction tasks. We propose a novel algebraic approach to this problem using lessons drawn from relational databases. Based on the observation that a large fraction of the time is spent in low-level text operations, the proposed algebra consists of several text-specific operators which enable significant optimization. We present a design for a plan optimizer. Finally, we validate our approach by implementing two complex annotators and running them on a real-world multi-gigabyte data set. The algebraic approach improves document throughput by a *factor of almost 20* relative to a state-of-the-art, heavily-tuned grammar-based implementation.

References

[1] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan, "DBLife: A community information management platform for the database research community," (demo). In CIDR-07, 2007.

[2] M. Theobald, R. Schenkel, and G. Weikum, "The topx dbir engine," in *SIGMOD*, 2007.

[3] H. Cunningham, "Information extraction - a user guide," University of Sheffield, Tech. Rep. CS-97-02, 1997. [Online]. Available: citeseer.ist.psu.edu/article/cunningham97information.html

[4] K. Nanda, "Combining lexical, syntactic and semantic features with maximum entropy models for extracting relations," in *ACL*, 2004.

[5] D. E. Appelt and B. Onyshkevych, "The common pattern specification language," in *TIPSTER workshop*, 1998.

[6] H. Cunningham, "GATE, a General Architecture for Text Engineering," *Computers and the Humanities*, 2002.

[7] D. Ferrucci and A. Lally, "UIMA: An architectural approach to unstructured information pro cessing in the corp orate research environment," *Nat. Lang. Eng.*, 2004.

[8] "Project Avatar," http://www.almaden.ibm.com/cs/projects/avatar/.

[9] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[10] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979.

[11] E. Agichtein and S. Sarawagi, "Scalable information extraction and integration," KDD, 2006.

[12] W. Cohen and A. McCallum, "Information extraction from the world wide web," KDD, 2003.

[13] A. Doan, R. Ramakrishnan, and S. Vaithyanathan, "Managing information extraction: State of the art and research directions," SIGMOD, 2006.

[14] D. Freitag, "Multistrategy learning for information extraction," in *ICML*, 1998.

[15] J. Lafferty, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *ICML*, 2001. [Online]. Available: citeseer.ist.psu.edu/article/lafferty01conditional.html

[16] F. Peng and A. McCallum, "Accurate information extraction from research papers using conditional random fields," in *HLT-NAACL*, 2004.

[17] S. Bird *et al.*, "ATLAS: A flexible and extensible architecture for linguistic annotation," in *LREC*, 2000.

[18] A. Chandel, P. C. Nagesh, and S. Sarawagi, "Efficient batch top-k search for dictionary-based entity recognition," in *ICDE*, 2006.

[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] G. Ramakrishnan, S. Balakrishnan, and S. Joshi, "Entity annotation based on inverse index operations," in *EMNLP*, 2006.

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

[22] R. T. Snodgrass, *Developing Time-Oriented Database Applications in SQL*. McGraw-Hill, 2000.

[23] S. Tata and J. Patel, "Piqa: An algebra for declaratively querying protein data sets," in *SSDBM*, 2003.