SystemT: A System for Declarative Information Extraction

Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan,
Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu
IBM Almaden Research Center,
http://www.almaden.ibm.com/cs/projects/avatar/

ABSTRACT

As applications within and outside the enterprise encounter increasing volumes of unstructured data, there has been renewed interest in the area of information extraction (IE) – the discipline concerned with extracting structured information from unstructured text. Classical IE techniques developed by the NLP community were based on cascading grammars and regular expressions. However, due to the inherent limitations of grammar-based extraction, these techniques are unable to: (i) scale to large data sets, and (ii) support the expressivity requirements of complex information tasks. At the IBM Almaden Research Center, we are developing SystemT, an IE system that addresses these limitations by adopting an algebraic approach. By leveraging well-understood database concepts such as declarative queries and cost-based optimization, SystemT enables scalable execution of complex information extraction tasks. In this paper, we motivate the SystemT approach to information extraction. We describe our extraction algebra and demonstrate the effectiveness of our optimization techniques in providing orders of magnitude reduction in the running time of complex extraction tasks.

1. INTRODUCTION

Enterprise applications for compliance, business intelligence, and search are encountering increasing volumes of unstructured text in the form of emails, customer call records, and intranet/extranet Web pages. Since unstructured text, in its raw form, has limited value, there is significant interest in extracting structured information from these documents – for example, extracting entities like persons and organizations, extracting relationships amongst such entities, detecting types of customer problems, etc.

Historically, the area of information extraction (IE) was studied by the Natural Language Processing community [1, 3, 5]. Both *knowledge engineering* approaches [2] and *machine learning* based approaches [6, 8, 9] have been proposed for building *annotators* that extract structured information from text. In the knowledge engineering approach, annotators consist of carefully crafted sets of *rules* for each task. In early IE systems, text was viewed as an input sequence of symbols and rules were specified as regular expressions over the lexical features of the symbols. The Common Pattern Specification Language (CPSL) developed in the context of the TIPSTER project [2] emerged as a popular language for expressing such extraction rules.

Numerous rule-based extraction systems were built by the NLP community in the 80's and early 90's, based on the formalism of cascading grammars and the theory of finite-state automata. These systems primarily targeted two classes of extraction tasks: *entity extraction* (identifying instances of persons, organizations, locations, etc.) and *relationship/link extraction* (identifying relationships between pairs of such entities). However, emerging applications, both within the enterprise (e.g., corporate governance and Business Intelligence), and on the Web (e.g., extracting reviews and opinions from blogs and discussion forums), are creating challenges of complex information extraction from large document collections. Due to inherent fundamental limitations, traditional grammar-based approaches are unable to support these new demands.

At the IBM Almaden Research Center, we are developing SystemT, an information extraction system that applies classical database ideas to overcome the limitations of grammar-based extraction. SystemT is used in several IBM products, such as Lotus Notes and eDiscovery Analyzer, to extract complex entities from enterprise documents and emails. SystemT is currently available for download [13]. In this paper, we describe the architecture, operators, query language, and optimization techniques that form the core of SystemT.

1.1 Limitations of grammar-based approaches

To motivate SystemT, we use the following example to highlight the limitations of grammar-based extraction with regard to performance and expressive power.

EXAMPLE 1 (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",*

SIGMOD Record Special Issue
.

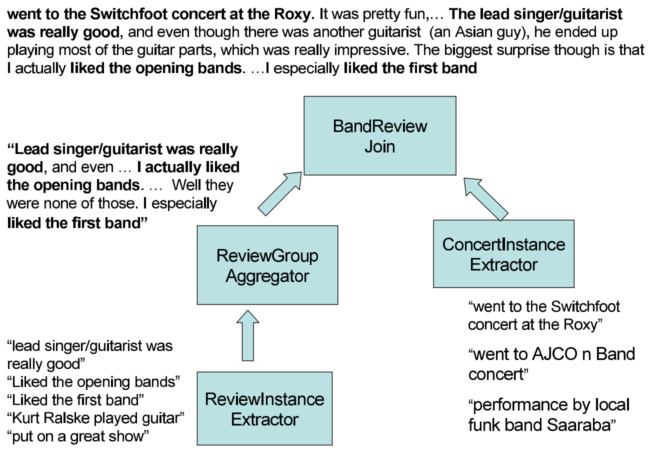
went to the Switchfoot concert at the Roxy. It was pretty fun,… **The lead singer/guitarist was really good**, and even though there was another guitarist (an Asian guy), he ended up playing most of the guitar parts, which was really impressive. The biggest surprise though is that I actually **liked the opening bands**. …I especially **liked the first band**
BandReview Join
"Lead singer/guitarist was really good, and even … **I actually liked the opening bands**. … Well they were none of those. I especially **liked the first band"**
ReviewGroup Aggregator
ConcertInstance Extractor
"went to the Switchfoot concert at the Roxy"
"went to AJCO n Band concert"
"performance by local funk band Saaraba"
"lead singer/guitarist was really good"
"Liked the opening bands"
"Liked the first band"
"Kurt Ralske played guitar"
"put on a great show"
ReviewInstance Extractor

Figure 1: Extraction of informal band reviews

"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*.*

In a traditional grammar-based IE system, the annotator described in Example 1 would be specified as a complex series of cascading grammars. For example, consider a rule in the ReviewInstance module described informally as: BandMember *followed within 30 characters by* Instrument. A translation of this specification into a cascading grammar yields:

ReviewInstance	$\longleftarrow$	BandMember `.{0,30}` Instrument	(R_1)
BandMember	$\longleftarrow$	RegularExpression (`[A-Z]\w+(\s+[A-Z]\w+)*`)	(R_2)
Instrument	$\longleftarrow$	RegularExpression ($d_1 \vert d_2 \vert \dots \vert d_n$)	(R_3)

Figure 2: 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[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.

Using a custom implementation of a CPSL-based cascading grammar system (similar to the implementation in JAPE [4]), we implemented the annotator shown in Figure 1. However, despite extensive performance tuning, the total running time *over 4.5 million blog entries was approximately seven hours* (see Figure 7) on an IBM xSeries server with two 3.6GHz Intel Xeon CPUs. 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 2 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 a consequence, for complex extraction tasks, the total execution time over large document collections becomes enormous.

[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 and required the use of a fairly complex regular expression. In the interest of readability, we have include a simpler version of the actual expression that we used for BandMember.

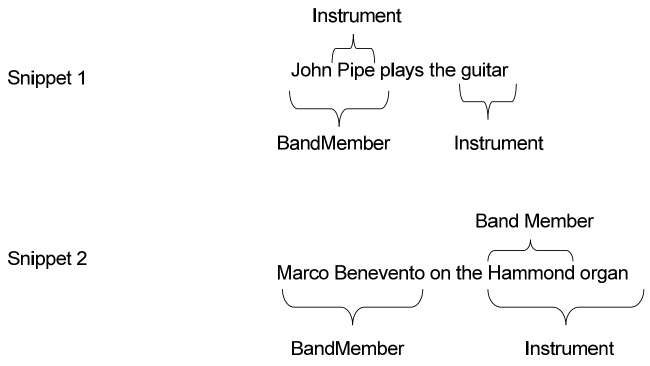

Figure 3: Overlapping Annotations

To address this scalability problem, in SystemT, we draw inspiration from the approach pioneered by relational databases. We develop an algebraic view of extraction in which rules are composed of individual extraction operators (Sec. 3) and employ cost-based optimization techniques to choose faster execution plans (Sec. 5).

Besides scalability, our algebraic approach addresses another fundamental issue in the way grammar-based systems handle *overlapping annotations*, a common phenomenon in complex extraction tasks. To illustrate, consider the following example:

Example 2 (Overlapping annotations). *Figure 3 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 and two instances 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*.*

Annotations overlap because individual rules are run independently and these 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

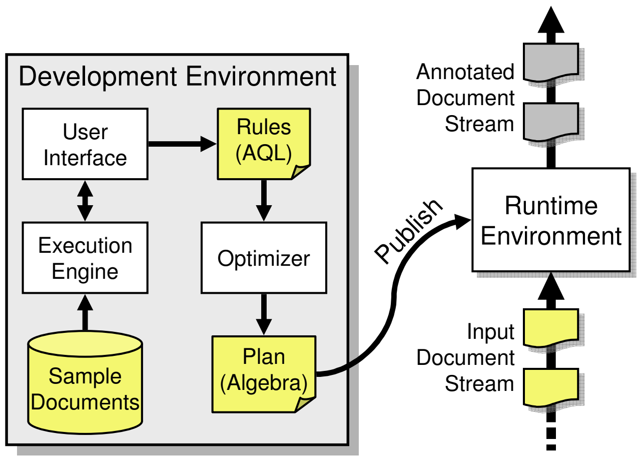

Figure 4: System Architecture

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 before being fed as input to the next higher level of a cascading grammar. For example, "Pipe" must either be an Instrument or a part of BandMember, and a similar choice must be made for "Hammond". Typically, one of several ad hoc disambiguation strategies are employed. Two such 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 2. On the other hand, with the choice of Instrument dominating, Snippet 1 will not be identified.

To appreciate the true effects of such ad hoc disambiguation, we ran two experiments using the rules from Figure 2 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.

On the other hand, by imposing no such requirements for input sequencing, SystemT eliminates the need for such forced disambiguation.

2. THE SYSTEMT ARCHITECTURE

Figure 4 illustrates the major components of SystemT. The *Development Environment* supports the iterative process that is involved in constructing and refining the rules for an extraction task. The rules are specified in a language called AQL (Annotation Query Language) as described in Section 4. The development environment provides facilities for compiling the rules into an algebra expression and for visualizing the results of executing the rules over a corpus of representative documents.

Once a developer is satisfied with the results that her rules produce on these documents, she can *publish* her annotator. Publishing an annotator is a two-step process. First, the AQL rules are fed into the *Optimizer*, which compiles them into an optimized algebraic expression. Then, the *Runtime Environment* instantiates the corresponding physical operators.

The Runtime Environment ("Runtime" for short) is typically embedded inside the processing pipeline of a text analytics application. The Runtime receives a continuous stream of documents, annotates each document, and outputs the annotations for further application-specific processing. The source of this document stream depends on the overall application. In Lotus Notes, for example, email messages are fed to the Runtime Environment as the user opens them in her mail client. In other applications, the document stream could come from a web crawler, an incoming message stream, or an offline document archive.

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 1: Operators in the SystemT algebra

3. OPERATORS AND ALGEBRA

The Optimizer and Runtime components of SystemT are based on an operator algebra that we have specifically developed for information extraction. In this section, we briefly summarize the salient aspects of our algebra and refer the reader to [10] for a more in-depth description.

3.1 Data and Execution Model

Since our algebra is designed to extract annotations from a single document at a time, we define its semantics in terms of the current document being analyzed. The current document is modeled as a string called doctext.

Our algebra operates over a simple relational data model with three data types: *span*, *tuple*, and *relation*. A span is an ordered pair $\langle begin, end\rangle$ that denotes the region of doctext from position *begin* to position *end*. A *tuple* is a 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 must be of the same width. Each operator in our algebra takes zero or more relations as input and produces a single relation as output.

3.2 Algebra Operators

Based on their functionality, the set of operators in our algebra fall into the following three categories (Table 3.2):

Relational Operators

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 main addition is that we use a few new selection predicates applicable only to spans [10].

Span Extraction Operators:

A span extraction operator identifies segments of text that match a particular input pattern and produces spans corresponding to each such text segment. Our algebra incorporates two kinds of span extraction operators:

- *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. The operator takes several optional parameters such as the maximum number of tokens that a match may span over.
- *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 doctext.

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. The precise details of how the aggregation is computed is different for the individual operators. Below, we describe two example aggregation operators:

- *Block.* The block operator is used to identify regions of text where input spans occur with enough regularity. For instance, in Example 1 (Figure 1), 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 *separation constraint* and a *length constraint*. The separation constraint controls the regularity with which input spans must occur within the block and the length constraint specifies minimum and maximum number of such input spans that must be contained within the block.
- *Consolidate.* The consolidate operator is 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. For instance, "liked the opening bands" and "liked the opening" are overlapping ReviewInstance occurrences identified by two different extraction patterns. To resolve such "duplicate" matches, we define several consolidation functions that define how overlapping spans need to be handled. Example consolidation functions in our algebra include

```
-- Define a dictionary of instrument names
create dictionary Instrument as ('flute', 'guitar', ... );

-- Use a regular expression to find names of band members
create view BandMember as
extract regex /[A-Z]\w+(\s+[A-Z]\w+)*/
    on 1 to 3 tokens of D.text
    as name
from Document D;

-- A single ReviewInstance rule. Finds instances of
-- BandMember followed within 30 characters by an
-- instrument name.
create view ReviewInstance as
select CombineSpans(B.name, I.inst) as instance
from
    BandMember B,
    (extract dictionary 'Instrument' on D.text as inst
     from Document D) I
where
    Follows(B.name, I.inst, 0, 30)
consolidate on CombineSpans(B.name, I.inst);
```

Figure 5: The ReviewInstance rules from Figure 2, expressed in AQL.

- *Containment consolidation*: discard annotation spans that are wholly contained within other annotation spans.
- *Overlap consolidation*: produce new spans by merging overlapping spans.

4. DECLARATIVE RULE LANGUAGE

The annotator developer's interactions with SystemT occur through the annotation rule language called AQL. AQL is a declarative language that combines the familiar syntax of SQL with the full expressive power of our text-specific operator algebra. AQL is specifically geared towards SystemT's document-at-a-time execution model. A built-in view called Document models the current document. The rule developer uses extraction primitives and text-specific predicates to build up a collection of higher-level views, eventually producing structured output annotations.

Figure 5 shows an example AQL rule from the ReviewInstance annotator. The first two statements define low-level *features* — dictionary and regular expression matches — that serve as inputs to the rule. The third statement defines the rule itself as a join with text-specific join predicates.

In addition to expressing complex low-level patterns in a declarative way, AQL also provides a compact and declarative way to define complex high-level entities. Consider the top-level "BandReview Join" rule of our annotator for informal concert reviews (see Figure 1). In English, this rule translates to:

> *Find all instances of ConcertInstance, followed within 0-30 characters by a block of 3 to 10 ReviewInstance annotations. Successive ReviewInstance annotations must be within 100 characters of each other. For each such ConcertIn-*

stance annotation, create a new output annotation that starts at the beginning of the ConcertInstance annotation and runs to the end of the last ReviewInstance annotation. Handle overlapping matches by removing any overall match that is completely contained within another match.

Such a complex pattern is nearly impossible to express in previous-generation languages, but AQL can express it quite succinctly:

```
create view BandReview as
select
    CI.instance as concert,
    CombineSpans(CI.instance, RI.instblock) as review
from
    ConcertInstance CI,
    (
        extract blocks
            with count between 3 and 10
            and separation between 0 and 100 characters
            on I.instance as instblock
        from ReviewInstance I
    ) RI
where
    Follows(CI.instance, RI.instblock, 0, 30)
consolidate on CombineSpans(CI.instance, RI.instblock)
    using 'ContainedWithin';
```

5. OPTIMIZER

Compared to traditional relational query optimization [11], the optimization problem in SystemT is distinct in several important ways. The cost of a relational database query, invariably, consists mostly of I/O and join costs. In contrast, the running time of extraction rules is dominated by the CPU cost of operations like regular expression matching and dictionary evaluation (in our experience, a typical naive execution plan will spend 90 percent or more of its time on these operators). Indeed, since SystemT's Runtime processes documents one-at-a-time, executing complex operations on each document, the time spent on I/O is generally insignificant compared to the time spent processing a document. Also, as a result of this document-at-a-time execution model, one can view the SystemT Runtime as evaluating the same operator graph over multiple separate "database instances", one per document. Thus, the goal of the SystemT optimizer is to find a plan that minimizes the *expected* cost over all these instances.

We have developed an Optimizer for SystemT that is designed specifically for the unique challenges of document-at-a-time execution and expensive extraction primitives. Note that the optimization approach adopted in SystemT, of reducing the cost of the CPU-intensive extraction operators, is complementary to the related work on optimizing extraction workflows [12]. The latter treats the individual extraction operations as black boxes and the focus is on the optimization of higher level workflows involving multiple extractors.

In the following sections, we describe the two phases in the SystemT optimization process: rule rewriting and cost-based optimization. The rewrite component of the Optimizer applies text-specific query rewrites to reduce the costs of extraction primitives. Then the cost-based component chooses join orders and methods that minimize the costs of primitive operations by taking advantage of document-at-a-time execution.

5.1 Rule Rewriting

Rule rewrites in SystemT directly improve the performance of the expensive regular expression and dictionary operators by applying transformations that do not affect semantics but almost always lead to a more efficient execution plan.

To reduce the cost of regular expressions, we have developed a technique, called *regular expression strength reduction*, that was motivated by the following observation: a major reason that regular expression evaluation is so expensive is that most regular expression engines support the full expressive power of the POSIX regular expression standard. However, for certain restricted classes of regular expressions, it is possible to build specialized engines that offer a significant performance improvement. For example, a regular expression that reduces to finding a finite set of strings can be executed far more efficiently by using a string matching engine. SystemT implements several such specialized engines. Strength reduction works by analyzing each regular expression in an AQL statement and choosing the fastest engine that can execute the expression. This technique cuts the running times of certain rules by an order of magnitude.

Another form of rule rewriting that is used in SystemT is called *shared dictionary matching*. This rewrite uses a version of the Dictionary operator that can evaluate many dictionaries at once in a single pass. Dictionary evaluation consists of three steps: identifying token boundaries, looking up each token in a hash table, and generating information about dictionary matches. By sharing the first two of these steps among many dictionaries, shared dictionary matching improves dictionary performance significantly, especially for rule sets involving a large number of small dictionaries.

5.2 Cost-Based Optimization

The rule rewrites described in the previous section work by directly reducing the overhead of expensive extraction operations. While these techniques are beneficial, SystemT achieves far more impressive performance improvements when the system can avoid executing these expensive operations at all. SystemT implements special text-specific join operators that take advantage of document boundaries and the sequential nature of text to avoid evaluating expensive extraction operations on some or all of the input text.

Figure 6 shows an example of one such operator, the *conditional evaluation join* operator, CEJoin. CEJoin, a physical implementation of the logical join operator $\bowtie$, takes advantage of SystemT's document-at-a-time execution to avoid executing extraction operators located below it in the operator graph. If the outer (left-hand) argument of the join produces no output tuples for a

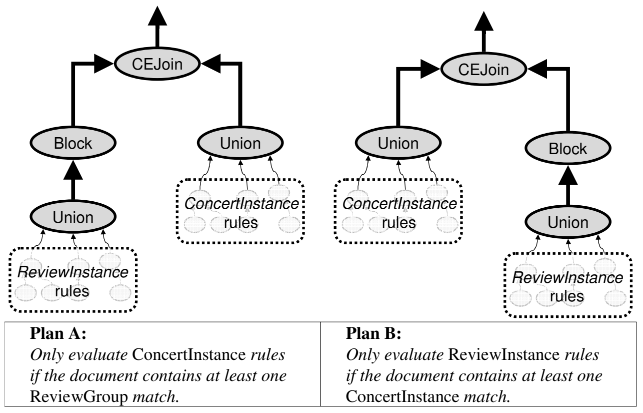
CEJoin
Block
Union
Union
ReviewInstance rules
ConcertInstance rules
CEJoin
Union
Block
ConcertInstance rules
Union
ReviewInstance rules
Plan A:
Only evaluate ConcertInstance rules if the document contains at least one ReviewGroup match.
Plan B:
Only evaluate ReviewInstance rules if the document contains at least one ConcertInstance match.

Figure 6: Two alternative plans that use Conditional Evaluation to evaluate the top-level join from the annotator in Figure 1.

given document, then the operator does not evaluate its inner (right-hand) argument. Depending on the expected cost of the two subtrees, as well as the probability that each subtree will produce zero output tuples on a given document, swapping the outer and inner operands of CEJoin can change overall execution times by orders of magnitude.

5.3 Experimental Results

To test the effectiveness of our optimization framework, we have performed numerous experiments over multiple data sets. In this section, we present sample results from experiments involving the annotator described in Example 1. We constructed three different implementations of this annotator:

- *GRAMMAR*: A hand-tuned grammar-based implementation.
- $ALGEBRA_{Baseline}$: A baseline algebraic plan constructed by directly translating each rule to the algebra.
- $ALGEBRA_{Optimized}$: A plan obtained by applying the text-specific optimizations described in this section.

To compare performance, we used a document corpus consisting of a collection of 4.5 million blogs (5.1GB of data) crawled from `http://www.blogspot.com`. All the experiments were run single-threaded on an IBM xSeries server with two 3.6GHz Intel Xeon CPUs.

Figure 7 shows the execution times for the three different implementations of the annotator. Simply moving from a grammar-based implementation to a naive algebraic plan led to a two-fold improvement in performance, primarily due to the fact that an algebraic implementation makes fewer passes over the text of the document. However, the application of text-specific optimizations (such as conditional evaluation) led to an additional *order of magnitude* increase in performance, for an overall improvement of close to 20 times. The net

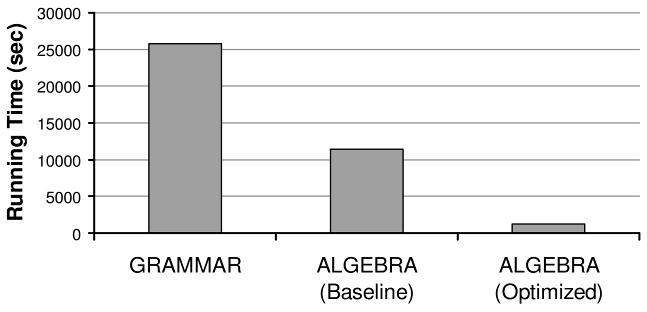

Figure 7: Results of an experiment that compares the running times of three different annotator implementations over a 4.5 million document corpus of blogs. SystemT's text-specific optimizations led to a speedup of 10x versus a naive algebraic implementation and 20x versus a hand-tuned grammar implementation.

result of this experiment is that the exact same information extraction task (BandReview) which took about seven hours in an optimized grammar-based implementation now runs in under 30 minutes using SystemT.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented SystemT, a declarative information extraction system that represents a paradigm shift in the way rule-based information extraction systems are built. Using declarative queries built upon a powerful operator algebra and employing effective cost-based optimization techniques, SystemT is able to support complex extraction tasks and scale to large document collections – well beyond the capabilities of traditional grammar-based extraction systems.

We are continuing to actively develop and enhance SystemT. Some of the areas that we are actively working on include:

- *Enhanced cost-based optimization:* Modern relational engines use statistics about the distribution of values in a database table to better estimate execution cost. In a similar fashion, we are looking to enhance SystemT's optimizer with more accurate cost estimates. However, while most database query optimization work focuses on join and I/O costs, much of the cost in SystemT is highly concentrated in the extraction primitives. Thus, accurate costing of plans in SystemT requires extending the state of the art in cost modeling to deal with the unique characteristics of text. For example, changing a single character in a regular expression can change the expression's running time or number of matches by an order of magnitude. Similarly, to cost plans involving conditional evaluation, the system needs to estimate the probability that an AQL sub-expression will produce zero results on a given document.
- *Indexing techniques:* While using the SystemT Development environment, a fixed corpus of docu-

ments is used to iteratively refine and craft a rule set for an extraction task. During this development process, rule writers need to repeatedly execute their rule sets over the same document collection and execution speed becomes critical to enable effective development. With this in mind, we are working on novel index structures and algorithms to reduce the time spent in computing regular expression and dictionary matches. For instance, we are looking at indexing techniques that will allow SystemT to completely avoid executing a regular expression on a document if it can be deduced that no matches will result.

- *Language extensions:* We are continuing to work on several extension and improvements to the AQL rule language. Based on feedback from the numerous deployments of SystemT within IBM product and research groups, we are extending the language with support for features like user-defined functions and recursion. In addition, we are working towards enabling support within SystemT for part-of-speech tagging and shallow parsing.

- *Distributed computing platforms:* There is increasing interest within enterprises to use distributed computing platforms to process and analyze large volumes of unstructured and semi-structured data such as email archives, Web server logs, query logs, etc. Since information extraction is an essential component of these analyses, we are actively working to embed SystemT into the popular Hadoop platform [7]. Our goal is to enable applications to easily exploit the full power of a large cluster when performing expensive extraction tasks.

7. REFERENCES

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

[2] D. E. Appelt and B. Onyshkevych. The common pattern specification language. In *TIPSTER workshop*, 1998.

[3] W. Cohen and A. McCallum. Information extraction from the World Wide Web. KDD, 2003.

[4] H. Cunningham, D. Maynard, and V. Tablan. JAPE: a java annotation patterns engine. Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, 2000.

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

[6] D. Freitag. Multistrategy learning for information extraction. In *ICML*, 1998.

[7] Hadoop. http://hadoop.apache.org/.

[8] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

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

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

[11] 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*, pages 23–34, 1979.

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

[13] System Text for Information Extraction. `http://www.alphaworks.ibm.com/tech/systemt`.