SystemT: an Algebraic Approach to Declarative Information Extraction

Laura Chiticariu **Rajasekar Krishnamurthy** **Yunyao Li**
Sriram Raghavan **Frederick R. Reiss** **Shivakumar Vaithyanathan**
IBM Research – Almaden
San Jose, CA, USA
{chiti,sekar,yunyaoli,rsriram,frreiss,vaithyan}@us.ibm.com

Abstract

As information extraction (IE) becomes more central to enterprise applications, rule-based IE engines have become increasingly important. In this paper, we describe SystemT, a rule-based IE system whose basic design removes the expressivity and performance limitations of current systems based on cascading grammars. SystemT uses a declarative rule language, AQL, and an optimizer that generates high-performance algebraic execution plans for AQL rules. We compare SystemT's approach against cascading grammars, both theoretically and with a thorough experimental evaluation. Our results show that SystemT can deliver result quality comparable to the state-of-the-art and an order of magnitude higher annotation throughput.

1 Introduction

In recent years, enterprises have seen the emergence of important text analytics applications like compliance and data redaction. This increase, combined with the inclusion of text into traditional applications like Business Intelligence, has dramatically increased the use of information extraction (IE) within the enterprise. While the traditional requirement of extraction quality remains critical, enterprise applications also demand efficiency, transparency, customizability and maintainability. In recent years, these systemic requirements have led to renewed interest in rule-based IE systems (Doan et al., 2008; SAP, 2010; IBM, 2010; SAS, 2010).

Until recently, rule-based IE systems (Cunningham et al., 2000; Boguraev, 2003; Drozdzynski et al., 2004) were predominantly based on the cascading grammar formalism exemplified by the Common Pattern Specification Language (CPSL) specification (Appelt and Onyshkevych, 1998). In CPSL, the input text is viewed as a sequence of annotations, and extraction rules are written as pattern/action rules over the lexical features of these annotations. In a single phase of the grammar, a set of rules are evaluated in a left-to-right fashion over the input annotations. Multiple grammar phases are cascaded together, with the evaluation proceeding in a bottom-up fashion.

As demonstrated by prior work (Grishman and Sundheim, 1996), grammar-based IE systems can be effective in many scenarios. However, these systems suffer from two severe drawbacks. First, the expressivity of CPSL falls short when used for complex IE tasks over increasingly pervasive informal text (emails, blogs, discussion forums etc.). To address this limitation, grammar-based IE systems resort to significant amounts of user-defined code in the rules, combined with pre- and post-processing stages beyond the scope of CPSL (Cunningham et al., 2010). Second, the rigid evaluation order imposed in these systems has significant performance implications.

Three decades ago, the database community faced similar expressivity and efficiency challenges in accessing structured information. The community addressed these problems by introducing a relational algebra formalism and an associated declarative query language SQL. The groundbreaking work on System R (Chamberlin et al., 1981) demonstrated how the expressivity of SQL can be efficiently realized in practice by means of a *query optimizer* that translates an SQL query into an optimized query execution plan.

Borrowing ideas from the database community, we have developed SystemT, a declarative IE system based on an algebraic framework, to address both expressivity and performance issues. In SystemT, extraction rules are expressed in a declarative language called AQL. At compilation time,

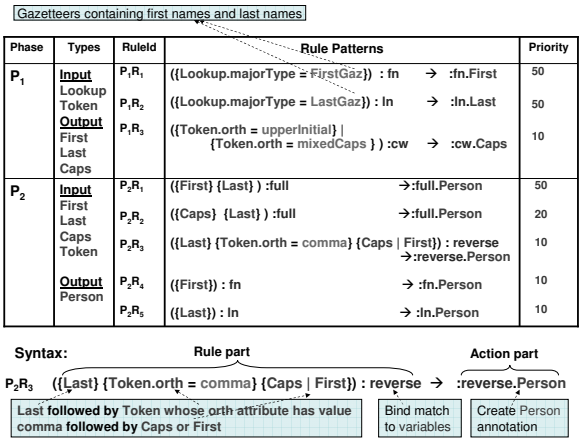

Figure 1: Cascading grammar for identifying Person names

SystemT translates AQL statements into an algebraic expression called an *operator graph* that implements the semantics of the statements. The SystemT optimizer then picks a fast execution plan from many logically equivalent plans. SystemT is currently deployed in a multitude of real-world applications and commercial products[1].

We formally demonstrate the superiority of AQL and SystemT in terms of both expressivity and efficiency (Section 4). Specifically, we show that 1) the expressivity of AQL is a strict superset of CPSL grammars not using external functions and 2) the search space explored by the SystemT optimizer includes operator graphs corresponding to efficient finite state transducer implementations. Finally, we present an extensive experimental evaluation that validates that high-quality annotators can be developed with SystemT, and that their runtime performance is an order of magnitude better when compared to annotators developed with a state-of-the-art grammar-based IE system (Section 5).

2 Grammar-based Systems and CPSL

A cascading grammar consists of a sequence of phases, each of which consists of one or more rules. Each phase applies its rules from left to right over an input sequence of annotations and generates an output sequence of annotations that the next phase consumes. Most cascading grammar systems today adhere to the CPSL standard.

Fig. 1 shows a sample CPSL grammar that identifies person names from text in two phases. The first phase, P_1, operates over the results of the tokenizer and gazetteer (input types *Token* and *Lookup*, respectively) to identify words that may be part of a person name. The second phase, P_2, identifies complete names using the results of phase P_1.

[1] A trial version is available at *http://www.alphaworks.ibm.com/tech/systemt*

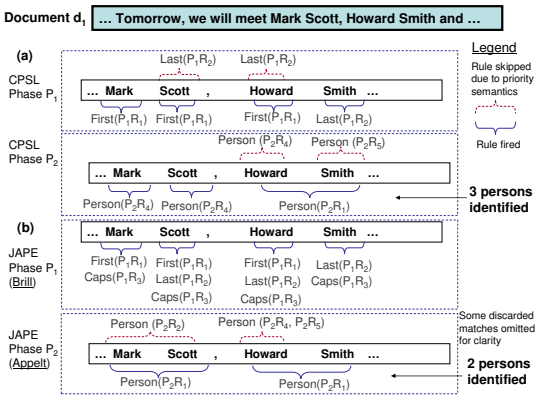

Figure 2: Sample output of CPSL and JAPE

Applying the above grammar to document d_1 (Fig. 2), one would expect that to match "Mark Scott" and "Howard Smith" as *Person*. However, as shown in Fig. 2(a), the grammar actually finds three *Person* annotations, instead of two. CPSL has several limitations that lead to such discrepancies:

L1. Lossy sequencing. In a CPSL grammar, each phase operates on a sequence of annotations from left to right. If the input annotations to a phase may overlap with each other, the CPSL engine must drop some of them to create a non-overlapping sequence. For instance, in phase P_1 (Fig. 2(a)), "Scott" has both a *Lookup* and a *Token* annotation. The system has made an arbitrary choice to retain the *Lookup* annotation and discard the *Token* annotation. Consequently, no *Caps* annotations are output by phase P_1.

L2. Rigid matching priority. CPSL specifies that, for each input annotation, only one rule can actually match. When multiple rules match at the same start position, the following tie-breaker conditions are applied (in order): (a) the rule matching the most annotations in the input stream; (b) the rule with highest priority; and (c) the rule declared earlier in the grammar. This rigid matching priority can lead to mistakes. For instance, as illustrated in Fig. 2(a), phase P_1 only identifies "Scott" as a *First*. Matching priority causes the grammar to skip the corresponding match for "Scott" as a *Last*. Consequently, phase P_2 fails to identify "Mark Scott" as one single *Person*.

L3. Limited expressivity in rule patterns. It is not possible to express rules that compare annotations overlapping with each other. E.g., "Identify words that are both capitalized and present in the

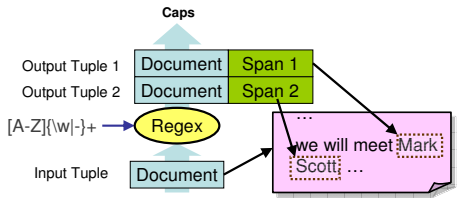

Figure 3: Regular Expression Extraction Operator

FirstGaz gazetteer" or "Identify *Person* annotations that occur within an *EmailAddress*".

Extensions to CPSL

In order to address the above limitations, several extensions to CPSL have been proposed in JAPE, AFst and XTDL (Cunningham et al., 2000; Boguraev, 2003; Drozdzynski et al., 2004). The extensions are summarized as below, where each solution S_i corresponds to limitation L_i.

- **S1**. Grammar rules are allowed to operate on graphs of input annotations in JAPE and AFst.
- **S2**. JAPE introduces more matching regimes besides the CPSL's matching priority and thus allows more flexibility when multiple rules match at the same starting position.
- **S3**. The rule part of a pattern has been expanded to allow more expressivity in JAPE, AFst and XTDL.

Fig. 2(b) illustrates how the above extensions help in identifying the correct matches '*Mark Scott*' and '*Howard Smith*' in JAPE. Phase P_1 uses a matching regime (denoted by *Brill*) that allows multiple rules to match at the same starting position, and phase P_2 uses CPSL's matching priority, *Appelt*.

3 SystemT

SystemT is a declarative IE system based on an algebraic framework. In SystemT, developers write rules in a language called AQL. The system then generates a graph of *operators* that implement the semantics of the AQL rules. This decoupling allows for greater rule expressivity, because the rule language is not constrained by the need to compile to a finite state transducer. Likewise, the decoupled approach leads to greater flexibility in choosing an efficient execution strategy, because many possible operator graphs may exist for the same AQL annotator.

In the rest of the section, we describe the parts of SystemT, starting with the algebraic formalism behind SystemT's operators.

3.1 Algebraic Foundation of SystemT

SystemT executes IE rules using graphs of operators. The formal definition of these operators takes the form of an algebra that is similar to the relational algebra, but with extensions for text processing.

The algebra operates over a simple relational data model with three data types: span, tuple, and relation. In this data model, a *span* is a region of text within a document identified by its "begin" and "end" positions; a *tuple* is a fixed-size list of spans. A *relation* is a multiset of tuples, where every tuple in the relation must be of the same size. Each *operator* in our algebra implements a single basic atomic IE operation, producing and consuming sets of tuples.

Fig. 3 illustrates the regular expression extraction operator in the algebra, which performs character-level regular expression matching. Overall, the algebra contains 12 different operators, a full description of which can be found in (Reiss et al., 2008). The following four operators are necessary to understand the examples in this paper:

- The **Extract** operator ($\mathcal{E}$) performs character-level operations such as regular expression and dictionary matching over text, creating a tuple for each match.
- The **Select** operator (σ) takes as input a set of tuples and a predicate to apply to the tuples. It outputs all tuples that satisfy the predicate.
- The **Join** operator ($\bowtie$) takes as input two sets of tuples and a predicate to apply to pairs of tuples from the input sets. It outputs all pairs of input tuples that satisfy the predicate.
- The **consolidate** operator (Ω) takes as input a set of tuples and the index of a particular column in those tuples. It removes selected overlapping spans from the indicated column, according to the specified policy.

3.2 AQL

Extraction rules in SystemT are written in AQL, a declarative relational language similar in syntax to the database language SQL. We chose SQL as a basis for our language due to its expressivity and its familiarity. The expressivity of SQL, which consists of first-order logic predicates over sets of tuples, is well-documented and well-understood (Codd, 1990). As SQL is the pri-

```
create view Caps as
extract regex /[A-Z](\w|-)+/ on D.text as name from Document D;

create view Last as
extract dictionary LastGaz on D.text as name from Document D;

create view CapsLast as
select  CombineSpans(C.name, L.name) as name
from    Caps C, Last L
where   FollowsTok(C.name, L.name, 0, 0);
...
create view PersonAll as
    (select R.name from FirstLast R) union all ...
                    ... union all (select R.name from CapsLast R);

create view Person as select * from PersonAll R
consolidate on R.name using 'ContainedWithin';

output view Person;
```

Figure 4: *Person* annotator as AQL query

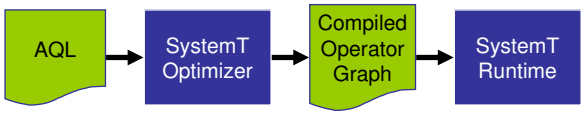

Figure 5: The compilation process in SystemT

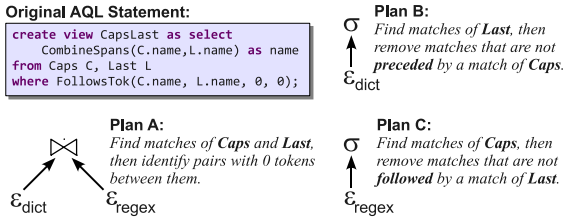

Figure 6: Execution strategies for the *CapsLast* rule in Fig. 4

mary interface to most relational database systems, the language's syntax and semantics are common knowledge among enterprise application programmers. Similar to SQL terminology, we call a collection of AQL rules an AQL *query*.

Fig. 4 shows portions of an AQL query. As can be seen, the basic building block of AQL is a *view*: A logical description of a set of tuples in terms of either the document text (denoted by a special view called `Document`) or the contents of other views. Every SystemT annotator consists of at least one view. The *output view* statement indicates that the tuples in a view are part of the final results of the annotator.

Fig. 4 also illustrates three of the basic constructs that can be used to define a view.

- The **`extract`** statement specifies basic character-level extraction primitives to be applied directly to a tuple.
- The **`select`** statement is similar to the SQL `select` statement but it contains an additional `consolidate on` clause, along with an extensive collection of text-specific predicates.
- The **`union all`** statement merges the outputs of one or more `select` or `extract` statements.

To keep rules compact, AQL also provides a shorthand *sequence pattern* notation similar to the syntax of CPSL. For example, the `CapsLast` view in Figure 4 could have been written as:

```
create view CapsLast as
extract pattern <C.name> <L.name>
from Caps C, Last L;
```

Internally, SystemT translates each of these *extract pattern* statements into one or more *select* and *extract* statements.

SystemT has built-in multilingual support including tokenization, part of speech and gazetteer matching for over 20 languages using LanguageWare (IBM, 2010). Rule developers can utilize the multilingual support via AQL without having to configure or manage any additional resources. In addition, AQL allows user-defined functions to be used in a restricted context in order to support operations such as validation (e.g. for extracted credit card numbers), or normalization (e.g., compute abbreviations of multi-token organization candidates that are useful in generating additional candidates). More details on AQL can be found in the AQL manual (SystemT, 2010).

3.3 Optimizer and Operator Graph

Grammar-based IE engines place rigid restrictions on the order in which rules can be executed. Due to the semantics of the CPSL standard, systems that implement the standard must use a finite state transducer that evaluates each level of the cascade with one or more left to right passes over the entire token stream.

In contrast, SystemT places no explicit constraints on the order of rule evaluation, nor does it require that intermediate results of an annotator collapse to a fixed-size sequence. As shown in Fig. 5, the SystemT engine does not execute AQL directly; instead, the SystemT *optimizer* compiles AQL into a graph of operators. By tying a collection of operators together by their inputs and outputs, the system can implement a wide variety of different execution strategies. Different execution strategies are associated with different evaluation costs. The optimizer chooses the execution strategy with the lowest estimated evaluation cost.

Fig. 6 presents three possible execution strategies for the *CapsLast* rule in Fig. 4. If the opti-

mizer estimates that the evaluation cost of *Last* is much lower than that of *Caps*, then it can determine that Plan C has the lowest evaluation cost among the three, because Plan C only evaluates *Caps* in the "left" neighborhood for each instance of *Last*. More details of our algorithms for enumerating plans can be found in (Reiss et al., 2008).

The optimizer in SystemT chooses the best execution plan from a large number of different algebra graphs available to it. Many of these graphs implement strategies that a transducer could not express: such as evaluating rules from right to left, sharing work across different rules, or selectively skipping rule evaluations. Within this large search space, there generally exists an execution strategy that implements the rule semantics far more efficiently than the fastest transducer could. We refer the reader to (Reiss et al., 2008) for a detailed description of the types of plan the optimizer considers, as well as an experimental analysis of the performance benefits of different parts of this search space.

Several parallel efforts have been made recently to improve the efficiency of IE tasks by optimizing low-level feature extraction (Ramakrishnan et al., 2006; Ramakrishnan et al., 2008; Chandel et al., 2006) or by reordering operations at a macroscopic level (Ipeirotis et al., 2006; Shen et al., 2007; Jain et al., 2009). However, to the best of our knowledge, SystemT is the only IE system in which the optimizer generates a full end-to-end plan, beginning with low-level extraction primitives and ending with the final output tuples.

3.4 Deployment Scenarios

SystemT is designed to be usable in various deployment scenarios. It can be used as a standalone system with its own development and runtime environment. Furthermore, SystemT exposes a generic Java API that enables the integration of its runtime environment with other applications. For example, a specific instantiation of this API allows SystemT annotators to be seamlessly embedded in applications using the UIMA analytics framework (UIMA, 2010).

4 Grammar vs. Algebra

Having described both the traditional cascading grammar approach and the declarative approach used in SystemT, we now compare the two in terms of expressivity and performance.

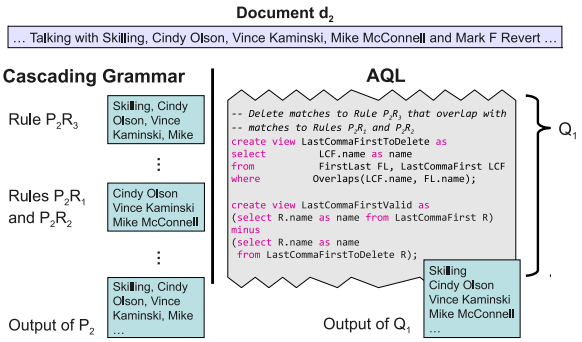

Figure 7: Supporting Complex Rule Interactions

4.1 Expressivity

In Section 2, we described three expressivity limitations of CPSL grammars: Lossy sequencing, rigid matching priority, and limited expressivity in rule patterns. As we noted, cascading grammar systems extend the CPSL specification in various ways to provide workarounds for these limitations.

In SystemT, the basic design of the AQL language eliminates these three problems without the need for any special workaround. The key design difference is that AQL views operate over sets of tuples, not sequences of tokens. The input or output tuples of a view can contain spans that overlap in arbitrary ways, so the lossy sequencing problem never occurs. The annotator will retain these overlapping spans across any number of views until a view definition explicitly removes the overlap. Likewise, the tuples that a given view produces are in no way constrained by the outputs of other, unrelated views, so the rigid matching priority problem never occurs. Finally, the *select* statement in AQL allows arbitrary predicates over the cross-product of its input tuple sets, eliminating the limited expressivity in rule patterns problem.

Beyond eliminating the major limitations of CPSL grammars, AQL provides a number of other information extraction operations that even extended CPSL cannot express without custom code.

Complex rule interactions. Consider an example document from the Enron corpus (Minkov et al., 2005), shown in Fig. 7, which contains a list of person names. Because the first person in the list ('*Skilling*') is referred to by only a last name, rule P_2R_3 in Fig. 1 incorrectly identifies '*Skilling, Cindy*' as a person. Consequently, the output of phase P_2 of the cascading grammar contains several mistakes as shown in the figure. This problem occurs because CPSL only evaluates rules over the input sequence in a strict left-to-right fashion.

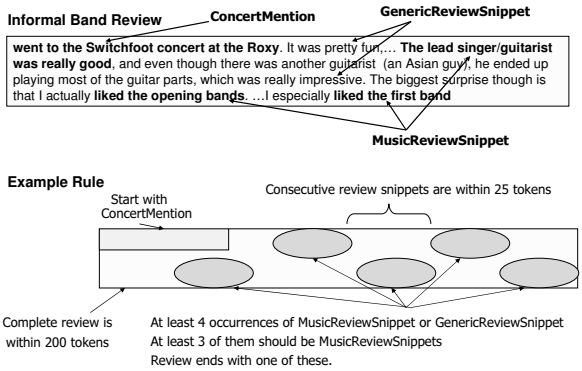

Figure 8: Extracting informal band reviews from web logs

On the other hand, the AQL query Q_1 shown in the figure applies the following condition: "Always discard matches to Rule P_2R_3 if they overlap with matches to rules P_2R_1 or P_2R_2" (even if the match to Rule P_2R_3 starts earlier). Applying this rule ensures that the person names in the list are identified correctly. Obtaining the same effect in grammar-based systems would require the use of custom code (as recommended by (Cunningham et al., 2010)).

Counting and Aggregation. Complex extraction tasks sometimes require operations such as counting and aggregation that go beyond the expressivity of regular languages, and thus can be expressed in CPSL only using external functions. One such task is that of identifying informal concert reviews embedded within blog entries. Fig. 8 describes, by example, how these reviews consist of reference to a live concert followed by several review snippets, some specific to musical performances and others that are more general review expressions. An example rule to identify informal reviews is also shown in the figure. Notice how implementing this rule requires *c*ounting the number of *MusicReviewSnippet* and *GenericReviewSnippet* annotations within a region of text and *a*ggregating this occurrence count across the two review types. While this rule can be written in AQL, it can only be approximated in CPSL grammars.

Character-Level Regular Expression CPSL cannot specify character-level regular expressions that span multiple tokens. In contrast, the *extract regex* statement in AQL fully supports these expressions.

We have described above several cases where AQL can express concepts that can only be expressed through external functions in a cascading grammar. These examples naturally raise the question of whether similar cases exist where a cascading grammar can express patterns that cannot be expressed in AQL.

It turns out that we can make a strong statement that such examples do not exist. In the absence of an escape to arbitrary procedural code, AQL is strictly more expressive than a CPSL grammar. To state this relationship formally, we first introduce the following definitions.

We refer to a grammar conforming to the CPSL specification as a *CPSL grammar*. When a CPSL grammar contains no external functions, we refer to it as a *Code-free CPSL grammar*. Finally, we refer to a grammar that conforms to one of the CPSL, JAPE, AFst and XTDL specifications as an *expanded CPSL grammar*.

Ambiguous Grammar Specification An *expanded CPSL grammar* may be under-specified in some cases. For example, a single rule containing the disjunction operator (|) may match a given region of text in multiple ways. Consider the evaluation of Rule P_2R_3 over the text fragment "Scott, Howard" from document d_1 (Fig. 1). If "Howard" is identified both as *Caps* and *First*, then there are two evaluations for Rule P_2R_3 over this text fragment. Since the system has to arbitrarily choose one evaluation, the results of the grammar can be non-deterministic (as pointed out in (Cunningham et al., 2010)). We refer to a grammar G as an *ambiguous grammar specification* for a document collection $\mathcal{D}$ if the system makes an arbitrary choice while evaluating G over $\mathcal{D}$.

Definition 1 (*UnambigEquiv*) *A query Q is UnambigEquiv to a cascading grammar $\mathcal{G}$ if and only if for every document collection $\mathcal{D}$, where G is not an ambiguous grammar specification for $\mathcal{D}$, the results of the grammar invocation and the query evaluation are identical.*

We now formally compare the expressivity of AQL and expanded CPSL grammars. The detailed proof is omitted due to space limitations.

Theorem 1 *The class of extraction tasks expressible as AQL queries is a strict superset of that expressible through expanded code-free CPSL grammars. Specifically,*
(a) Every expanded code-free CPSL grammar can be expressed as an UnambigEquiv AQL query.
(b) AQL supports information extraction operations that cannot be expressed in expanded code-free CPSL grammars.

Proof Outline: *(a)* A single CPSL grammar can be expressed in AQL as follows. First, each rule

r in the grammar is translated into a set of AQL statements. If r does not contain the disjunct (|) operator, then it is translated into a single AQL *select* statement. Otherwise, a set of AQL statements are generated, one for each disjunct operator in rule r, and the results merged using *union all* statements. Then, a *union all* statement is used to combine the results of individual rules in the grammar phase. Finally, the AQL statements for multiple phases are combined in the same order as the cascading grammar specification.

The main extensions to CPSL supported by expanded CPSL grammars (listed in Sec. 2) are handled as follows. AQL queries operate on graphs on annotations just like expanded CPSL grammars. In addition, AQL supports different matching regimes through consolidation operators, span predicates through selection predicates and coreferences through join operators.

(b) Example operations supported in AQL that cannot be expressed in expanded code-free CPSL grammars include (i) character-level regular expressions spanning multiple tokens, (ii) counting the number of annotations occurring within a given bounded window and (iii) deleting annotations if they overlap with other annotations starting later in the document. □

4.2 Performance

For the annotators we test in our experiments (See Section 5), the SystemT optimizer is able to choose algebraic plans that are faster than a comparable transducer-based implementation. The question arises as to whether there are other annotators for which the traditional transducer approach is superior. That is, for a given annotator, might there exist a finite state transducer that is combinatorially faster than any possible algebra graph? It turns out that this scenario is not possible, as the theorem below shows.

Definition 2 (Token-Based FST) *A* token-based finite state transducer *(FST) is a nondeterministic finite state machine in which state transitions are triggered by predicates on tokens. A token-based FST is* acyclic *if its state graph does not contain any cycles and has exactly one "accept" state.*

Definition 3 (Thompson's Algorithm) Thompson's algorithm *is a common strategy for evaluating a token-based FST (based on (Thompson, 1968)). This algorithm processes the input tokens from left to right, keeping track of the set of states that are currently active.*

Theorem 2 *For any acyclic token-based finite state transducer T, there exists an UnambigEquiv operator graph G, such that evaluating G has the same computational complexity as evaluating T with Thompson's algorithm starting from each token position in the input document.*

Proof Outline: The proof constructs G by structural induction over the transducer T. The base case converts transitions out of the start state into *Extract* operators. The inductive case adds a *Select* operator to G for each of the remaining state transitions, with each selection predicate being the same as the predicate that drives the corresponding state transition. For each state transition predicate that T would evaluate when processing a given document, G performs a constant amount of work on a single tuple. □

5 Experimental Evaluation

In this section we present an extensive comparison study between SystemT and implementations of expanded CPSL grammar in terms of quality, runtime performance and resource requirements.

Tasks We chose two tasks for our evaluation:

- NER : named-entity recognition for Person, Organization, Location, Address, PhoneNumber, EmailAddress, URL and DateTime.
- BandReview : identify informal reviews in blogs (Fig. 8).

We chose NER primarily because named-entity recognition is a well-studied problem and standard datasets are available for evaluation. For this task we use GATE and ANNIE for comparison[3]. We chose BandReview to conduct performance evaluation for a more complex extraction task.

Datasets. For quality evaluation, we use:

- *EnronMeetings* (Minkov et al., 2005): collection of emails with meeting information from the Enron corpus[4] with Person labeled data;
- *ACE* (NIST, 2005): collection of newswire reports and broadcast news/conversations with Person, Organization, Location labeled data[5].

Table 1 lists the datasets used for performance evaluation. The size of $Finance_L$ is purposely

[3]To the best of our knowledge, ANNIE (Cunningham et al., 2002) is the only publicly available NER library implemented in a grammar-based system (JAPE in GATE).

[4]http://www.cs.cmu.edu/ enron/

[5]Only entities of type NAM have been considered.

Table 1: Datasets for performance evaluation.

Dataset	*Description of the Content*	*Number of documents*	*Document size*	
			range	*average*
$Enron_x$	Emails randomly sampled from the Enron corpus of average size xKB $(0.5 < x < 100)$[2]	1000	xKB $+/-$ 10%	xKB
WebCrawl	Small to medium size web pages representing company news, with HTML tags removed	1931	68b - 388.6KB	8.8KB
$Finance_M$	Medium size financial regulatory filings	100	240KB - 0.9MB	401KB
$Finance_L$	Large size financial regulatory filings	30	1MB - 3.4MB	1.54MB

Table 2: Quality of Person on test datasets.

	Precision (%) (Exact/Partial)	*Recall (%) (Exact/Partial)*	*F1 measure (%) (Exact/Partial)*
EnronMeetings			
ANNIE	57.05/76.84	48.59/65.46	52.48/70.69
T-NE	**88.41/92.99**	**82.39/86.65**	**85.29/89.71**
Minkov	81.1/NA	74.9/NA	77.9/NA
ACE			
ANNIE	39.41/78.15	30.39/60.27	34.32/68.06
T-NE	**93.90/95.82**	**90.90/92.76**	**92.38/94.27**

small because GATE takes a significant amount of time processing large documents (see Sec. 5.2).

Set Up. The experiments were run on a server with two 2.4 GHz 4-core Intel Xeon CPUs and 64GB of memory. We use GATE 5.1 (build 3431) and two configurations for ANNIE: 1) the *default* configuration, and 2) an *optimized* configuration where the Ontotext Japec Transducer[6] replaces the default NE transducer for optimized performance. We refer to these configurations as ANNIE and ANNIE-Optimized, respectively.

5.1 Quality Evaluation

The goal of our quality evaluation is two-fold: to validate that annotators can be built in SystemT with quality comparable to those built in a grammar-based system; and to ensure a fair performance comparison between SystemT and GATE by verifying that the annotators used in the study are comparable.

Table 2 shows results of our comparison study for Person annotators. We report the classical (*exact*) precision, recall, and $F1$ measures that credit only exact matches, and corresponding *partial* measures that credit partial matches in a fashion similar to (NIST, 2005). As can be seen, T-NE produced results of significantly higher quality than ANNIE on both datasets, for the same Person extraction task. In fact, on *EnronMeetings*, the $F1$ measure of T-NE is 7.4% higher than the best published result (Minkov et al., 2005). Similar results can be observed for Organization and Location on *ACE* (exact numbers omitted in interest of space).

[6]http://www.ontotext.com/gate/japec.html

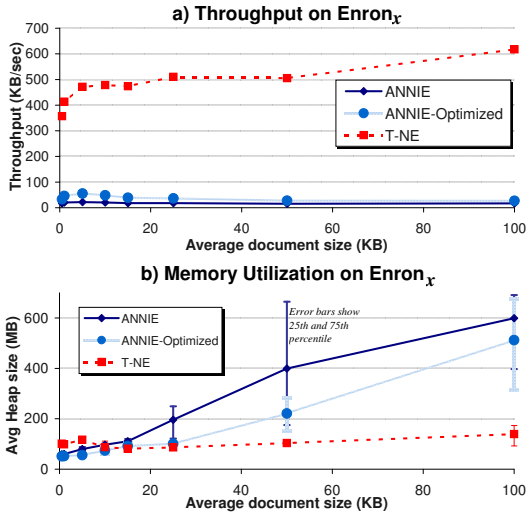

Figure 9: Throughput (a) and memory consumption (b) comparisons on $Enron_x$ datasets.

Clearly, considering the large gap between ANNIE's $F1$ and partial $F1$ measures on both datasets, ANNIE's quality can be improved via dataset-specific tuning as demonstrated in (Maynard et al., 2003). However, dataset-specific tuning for ANNIE is beyond the scope of this paper. Based on the experimental results above and our previous formal comparison in Sec. 4, we believe it is reasonable to conclude that annotators can be built in SystemT of quality at least comparable to those built in a grammar-based system.

5.2 Performance Evaluation

We now focus our attention on the throughput and memory behavior of SystemT, and draw a comparison with GATE. For this purpose, we have configured both ANNIE and T-NE to identify only the same eight types of entities listed for NER task.

Throughput. Fig. 9(a) plots the throughput of the two systems on multiple $Enron_x$ datasets with average document sizes of between 0.5KB and 100KB. For this experiment, both systems ran with a maximum Java heap size of 1GB.

As shown in Fig. 9(a), even though the throughput of ANNIE-Optimized (using the optimized trans-

Table 3: Throughput and mean heap size.

Dataset	ANNIE		ANNIE-Optimized		T-NE	
	Throughput (KB/s)	*Memory (MB)*	*Throughput (KB/s)*	*Memory (MB)*	*Throughput (KB/s)*	*Memory (MB)*
WebCrawl	23.9	212.6	42.8	201.8	**498.9**	**77.2**
$Finance_M$	18.82	715.1	26.3	601.8	**703.5**	**143.7**
$Finance_L$	19.2	2586.2	21.1	2683.5	**954.5**	**189.6**

ducer) increases two-fold compared to ANNIE under default configuration, T-NE is between 8 and 24 times faster compared to ANNIE-Optimized. For both systems, throughput varied with document size. For T-NE, the relatively low throughput on very small document sizes (less than 1KB) is due to fixed overhead in setting up operators to process a document. As document size increases, the overhead becomes less noticeable.

We have observed similar trends on the rest of the test collections. Table 3 shows that T-NE is at least an order of magnitude faster than ANNIE-Optimized across all datasets. In particular, on $Finance_L$ T-NE's throughput remains high, whereas the performance of both ANNIE and ANNIE-Optimized degraded significantly.

To ascertain whether the difference in performance in the two systems is due to low-level components such as dictionary evaluation, we performed detailed profiling of the systems. The profiling revealed that 8.2%, 16.2% and respectively 14.2% of the execution time was spent on average on low-level components in the case of ANNIE, ANNIE-Optimized and T-NE, respectively, thus leading us to conclude that the observed differences are due to SystemT's efficient use of resources at a macroscopic level.

Memory utilization. In theory, grammar based systems can stream tuples through each stage for minimal memory consumption, whereas SystemT operator graphs may need to materialize intermediate results for the full document at certain points to evaluate the constraints in the original AQL. The goal of this study is to evaluate whether this potential problem does occur in practice.

In this experiment we ran both systems with a maximum heap size of 2GB, and used the Java garbage collector's built-in telemetry to measure the total quantity of live objects in the heap over time while annotating the different test corpora. Fig. 9(b) plots the minimum, maximum, and mean heap sizes with the $Enron_x$ datasets. On small documents of size up to 15KB, memory consumption is dominated by the fixed size of the data structures used (e.g., dictionaries, FST/operator graph), and is comparable for both systems. As documents get larger, memory consumption increases for both systems. However, the increase is much smaller for T-NE compared to that for both ANNIE and ANNIE-Optimized. A similar trend can be observed on the other datasets as shown in Table 3. In particular, for $Finance_L$, both ANNIE and ANNIE-Optimized required 8GB of Java heap size to achieve reasonable throughput[7], in contrast to T-NE which utilized at most 300MB out of the 2GB of maximum Java heap size allocation.

SystemT requires much less memory than GATE in general due to its runtime, which monitors data dependencies between operators and clears out low-level results when they are no longer needed. Although a streaming CPSL implementation is theoretically possible, in practice mechanisms that allow an escape to custom code make it difficult to decide when an intermediate result will no longer be used, hence GATE keeps most intermediate data in memory until it is done analyzing the current document.

The BandReview Task. We conclude by briefly discussing our experience with the BandReview task from Fig. 8. We built two versions of this annotator, one in AQL, and the other using expanded CPSL grammar. The grammar implementation processed a 4.5GB collection of 1.05 million blogs in 5.6 hours and output 280 reviews. In contrast, the SystemT version (85 AQL statements) extracted 323 reviews in only 10 minutes!

6 Conclusion

In this paper, we described SystemT, a declarative IE system based on an algebraic framework. We presented both formal and empirical arguments for the benefits of our approach to IE. Our extensive experimental results show that high-quality annotators can be built using SystemT, with an order of magnitude throughput improvement compared to state-of-the-art grammar-based systems. Going forward, SystemT opens up several new areas of research, including implementing better optimization strategies and augmenting the algebra with additional operators to support advanced features such as coreference resolution.

[7] GATE ran out of memory when using less than 5GB of Java heap size, and thrashed when run with 5GB to 7GB

References

Douglas E. Appelt and Boyan Onyshkevych. 1998. The common pattern specification language. In *TIPSTER workshop*.

Branimir Boguraev. 2003. Annotation-based finite state processing in a large-scale nlp arhitecture. In *RANLP*, pages 61–80.

D. D. Chamberlin, A. M. Gilbert, and Robert A. Yost. 1981. A history of System R and SQL/data system. In *vldb*.

Amit Chandel, P. C. Nagesh, and Sunita Sarawagi. 2006. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*.

E. F. Codd. 1990. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

H. Cunningham, D. Maynard, and V. Tablan. 2000. JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, November.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, pages 168 – 175.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Marin Dimitrov, Mike Dowman, Niraj Aswani, Ian Roberts, Yaoyong Li, and Adam Funk. 2010. Developing language processing components with gate version 5 (a user guide).

AnHai Doan, Luis Gravano, Raghu Ramakrishnan, and Shivakumar Vaithyanathan. 2008. Special issue on managing information extraction. *SIGMOD Record*, 37(4).

Witold Drozdzynski, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. 2004. Shallow processing with unification and typed feature structures — foundations and applications. *Künstliche Intelligenz*, 1:17–23.

Ralph Grishman and Beth Sundheim. 1996. Message understanding conference - 6: A brief history. In *COLING*, pages 466–471.

IBM. 2010. IBM LanguageWare.

P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. 2006. To search or to crawl?: towards a query optimizer for text-centric tasks. In *SIGMOD*.

Alpa Jain, Panagiotis G. Ipeirotis, AnHai Doan, and Luis Gravano. 2009. Join optimization of information extraction output: Quality matters! In *ICDE*.

Diana Maynard, Kalina Bontcheva, and Hamish Cunningham. 2003. Towards a semantic extraction of named entities. In *Recent Advances in Natural Language Processing*.

Einat Minkov, Richard C. Wang, and William W. Cohen. 2005. Extracting personal names from emails: Applying named entity recognition to informal text. In *HLT/EMNLP*.

NIST. 2005. The ACE evaluation plan.

Ganesh Ramakrishnan, Sreeram Balakrishnan, and Sachindra Joshi. 2006. Entity annotation based on inverse index operations. In *EMNLP*.

Ganesh Ramakrishnan, Sachindra Joshi, Sanjeet Khaitan, and Sreeram Balakrishnan. 2008. Optimization issues in inverted index-based entity annotation. In *InfoScale*.

Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan. 2008. An algebraic approach to rule-based information extraction. In *ICDE*, pages 933–942.

SAP. 2010. Inxight ThingFinder.

SAS. 2010. Text Mining with SAS Text Miner.

Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. 2007. Declarative information extraction using datalog with embedded extraction predicates. In *vldb*.

SystemT. 2010. AQL Manual. http://www.alphaworks.ibm.com/tech/systemt.

Ken Thompson. 1968. Regular expression search algorithm. pages 419–422.

UIMA. 2010. Unstructured Information Management Architecture. http://uima.apache.org.