SystemT: A Declarative Information Extraction System

Yunyao Li
IBM Research - Almaden
650 Harry Road
San Jose, CA 95120
yunyaoli@us.ibm.com

Frederick R. Reiss
IBM Research - Almaden
650 Harry Road
San Jose, CA 95120
frreiss@us.ibm.com

Laura Chiticariu
IBM Research - Almaden
650 Harry Road
San Jose, CA 95120
chiti@us.ibm.com

Abstract

Emerging text-intensive enterprise applications such as social analytics and semantic search pose new challenges of scalability and usability to Information Extraction (IE) systems. This paper presents SystemT, a declarative IE system that addresses these challenges and has been deployed in a wide range of enterprise applications. SystemT facilitates the development of high quality complex annotators by providing a highly expressive language and an advanced development environment. It also includes a cost-based optimizer and a high-performance, flexible runtime with minimum memory footprint. We present SystemT as a useful resource that is freely available, and as an opportunity to promote research in building scalable and usable IE systems.

1 Introduction

Information extraction (IE) refers to the extraction of structured information from text documents. In recent years, text analytics have become the driving force for many emerging enterprise applications such as compliance and data redaction. In addition, the inclusion of text has also been increasingly important for many traditional enterprise applications such as business intelligence. Not surprisingly, the use of information extraction has dramatically increased within the enterprise over the years. While the traditional requirement of extraction quality remains critical, enterprise applications pose several two challenges to IE systems:

1. *Scalability*: Enterprise applications operate over large volumes of data, often orders of magnitude larger than classical IE corpora. An IE system should be able to operate at those scales without compromising its execution efficiency or memory consumption.
2. *Usability*: Building an accurate IE system is an inherently labor intensive process. Therefore, the usability of an enterprise IE system in terms of ease of development and maintenance is crucial for ensuring healthy product cycle and timely handling of customer complains.

Traditionally, IE systems have been built from individual extraction components consisting of rules or machine learning models. These individual components are then connected procedurally in a programming language such as C++, Perl or Java. Such procedural logic towards IE cannot meet the increasing scalability and usability requirements in the enterprise (Doan et al., 2006; Chiticariu et al., 2010a).

Three decades ago, the database community faced similar scalability and expressivity challenges in accessing structured information. The community addressed these problems by introducing a relational algebra formalism and an associated declarative query language SQL. Borrowing ideas from the database community, several systems (Doan and others, 2008; Bohannon and others, 2008; Jain et al., 2009; Krishnamurthy et al., 2008; Wang et al., 2010) have been built in recent years taking an alternative declarative approach to information extraction. Instead of using procedural logic to implement the extraction task, declarative IE systems separate the description of *what* to extract from *how* to extract it, allowing the IE developer to build complex extrac-

Proceedings of the ACL-HLT 2011 System Demonstrations, pages 109–114,
Portland, Oregon, USA, 21 June 2011.

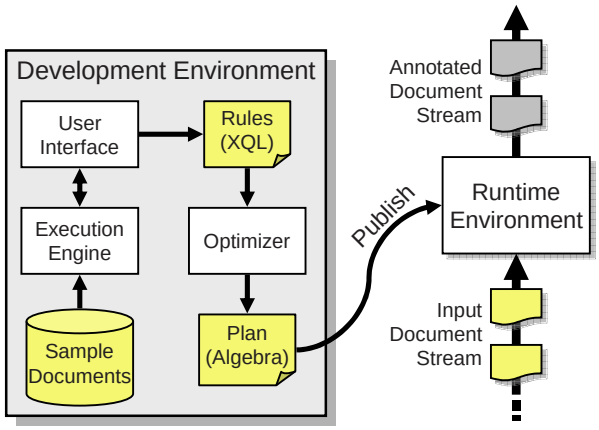

Figure 1: Overview of SystemT

```
create view Phone as
extract regex /\d{3}-\d{4}/ on D.text as number
from Document D;

create view Person as
extract dictionary 'firstNames.dict' on D.text as name
from Document D;

create view PersonPhoneAll as
select CombineSpans(P.name, Ph.number) as match
from Person P, Phone Ph
where FollowsTok(P.name, Ph.number, 0, 5);

create view PersonPhone as
select R.name as name
from PersonPhoneAll R
consolidate on R.name;

output view PersonPhone;
```

Figure 2: An AQL program for a *PersonPhone* task.

tion programs without worrying about performance considerations.

In this demonstration, we showcase one such declarative IE system called SystemT, designed to address the scalability and usability challenges. We illustrate how SystemT, currently deployed in a multitude of real-world applications and commercial products, can be used to develop and maintain IE annotators for enterprise applications. A free version of SystemT is available at *http://www.alphaworks.ibm.com/tech/systemt*.

2 Overview of SystemT

Figure 1 depicts the architecture of SystemT. The system consists of two major components: the Development Environment and the Runtime Environment. The SystemT Development Environment supports the iterative process of constructing and refining rules for information extraction. The rules are specified in a declarative language called AQL (F.Reiss et al., 2008). The Development Environment provides facilities for executing rules over a given corpus of representative documents and visualizing the results of the execution. 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, given an AQL annotator, there can be many possible graphs of *operators*, or execution plans, each of which faithfully implements the semantics of the annotator. Some of the execution plans are much more efficient than others. The SystemT Optimizer explores the space of the possible execution plans to choose the most efficient one. This execution plan is then given to the SystemT Runtime to instantiate the corresponding physical operators. Once the physical operators are instantiated, the SystemT Runtime feeds one document at a time through the graph of physical operators and outputs a stream of annotated documents.

The decoupling of the Development and Runtime environments is essential for the flexibility of the system. It facilitates the incorporating of various sophisticated tools to enable annotator development without sacrificing runtime performance. Furthermore, the separation permits the SystemT Runtime to be embedded into larger applications with minimum memory footprint. Next, we discuss individual components of SystemT in more details (Sections 3 – 6), and summarize our experience with the system in a variety of enterprise applications (Section 7).

3 The Extraction Language

In SystemT, developers express an information extraction program using a language called AQL. AQL is a declarative relational language similar in syntax to the database language SQL, which was chosen as a basis for our language due to its expressivity and familiarity. An AQL program (or an AQL annotator) consists of a set of AQL rules.

In this section, we describe the AQL language and its underlying algebraic operators. In Section 4, we explain how the SystemT optimizer explores a large space of possible execution plans for an AQL annotator and chooses one that is most efficient.

3.1 AQL

Figure 2 illustrates a (very) simplistic annotator of relationships between persons and their phone number. At a high-level, the annotator identifies person names using a simple dictionary of first names, and phone numbers using a regular expression. It then identifies pairs of *Person* and *Phone* annotations, where the latter follows the

former within 0 to 5 tokens, and marks the corresponding region of text as a *PersonPhoneAll* annotation. The final output *PersonPhone* is constructed by removing overlapping *PersonPhoneAll* annotations.

AQL operates over a simple relational data model with three data types: span, tuple, and view. In this data model, a *span* is a region of text within a document identified by its "begin" and "end" positions, while a *tuple* is a list of spans of fixed size. A *view* is a set of tuples. As can be seen from Figure 2, each AQL rule defines a view. As such, a view is the basic building block in AQL: it consists of a logical description of a set of tuples in terms of the document text, or the content of other views. The input to the annotator is a special view called `Document` containing a single tuple with the document text. The AQL annotator tags some views as *output views*, which specify the annotation types that are the final results of the annotator.

The example in Figure 2 illustrates two of the basic constructs of AQL. The `extract` statement specifies basic character-level extraction primitives, such as regular expressions or dictionaries (i.e., gazetteers), that are applied directly to the document, or a region thereof. The `select` statement is similar to the corresponding SQL statement, but contains an additional `consolidate on` clause for resolving overlapping annotations, along with an extensive collection of text-specific predicates.

To keep rules compact, AQL also allows a shorthand *pattern* notation similar to the syntax of the CPSL grammar standard (Appelt and Onyshkevych, 1998). For example, the `PersonPhoneAll` view in Figure 2 can also be expressed as shown below. Internally, SystemT translates each of these *extract pattern* statements into one or more *select* and *extract* statements.

```
create view PersonPhoneAll as
extract pattern
<P.name> <Token>{0,5} <Ph.number>
from Person P, Phone Ph;
```

SystemT has built-in multilingual support including tokenization, part of speech and gazetteer matching for over 20 languages using IBM LanguageWare. Annotator developers can utilize the multilingual support via AQL without having to configure or manage any additional resources. In addition, AQL allows user-defined functions in a restricted context in order to support operations such as validation or normalization. More details on AQL can be found in the AQL manual (Chiticariu et al., 2010b).

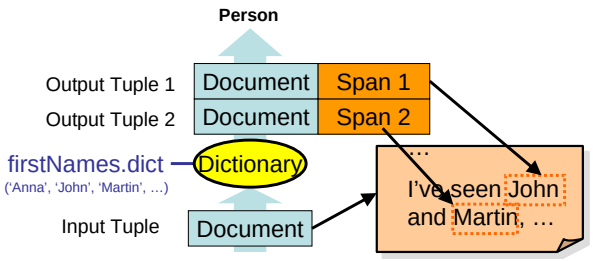

Figure 3: Dictionary Extraction Operator

3.2 Algebraic Operators in SystemT

SystemT executes AQL rules using graphs of operators. These operators are based on an algebraic formalism that is similar to the relational algebra formalism, but with extensions for text processing. Each *operator* in the algebra implements a single basic atomic IE operation, producing and consuming sets of tuples (i.e., views).

Fig. 3 illustrates the dictionary extraction operator in the algebra, which performs character-level dictionary matching. A full description of the 12 different operators of the algebra can be found in (F.Reiss et al., 2008). Three of the operators are listed below.

- The **Extract** operator ($\mathcal{E}$) performs character-level operations such as regular expression and dictionary matching over text, producing one tuple for each match.
- The **Select** operator (σ) takes as input a set of tuples and a predicate to apply to the tuples, and 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. It outputs all pairs satisfying the predicate.

Other operators include **PartOfSpeech** for part-of-speech detection, **Consolidate** for removing overlapping annotations, **Block** and **Group** for grouping together similar annotations occurring within close proximity to each other, as well as expressing more general types of aggregation, **Sort** for sorting, and **Union** and **Minus** for expressing set union and set difference, respectively.

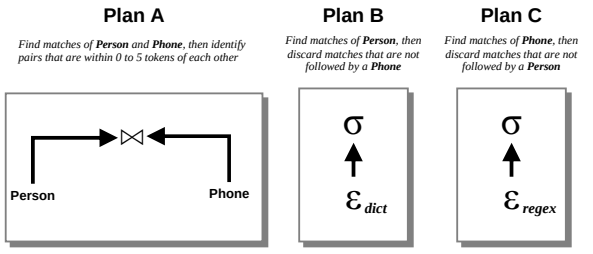

Figure 4: Execution strategies for *PersonPhoneAll* in Fig. 2

4 The Optimizer

Grammar-based IE engines such as (Boguraev, 2003; Cunningham et al., 2000) place rigid restrictions on the order in which rules can be executed. Such systems that implement the CPSL standard or extensions of it must use a finite state transducer to evaluate each level of the cascade with one or more left to right passes over the entire input token stream. In contrast, SystemT uses a declarative approach based on rules that specify *what* patterns to extract, as opposed to *how* to extract them. In a declarative IE system such as SystemT the specification of an annotator is completely separate from its implementation. In particular, the system does not place 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. 1, the SystemT engine does not execute AQL directly; instead, the SystemT Optimizer compiles AQL into a graph of operators. Given a collection of AQL views, the optimizer generates a large number of different operator graphs, all of which faithfully implement the semantics of the original views. Even though these graphs always produce the same results, the execution strategies that they represent can have very different performance characteristics. The optimizer incorporates a *cost model* which, given an operator graph, estimates the CPU time required to execute the graph over an average document in the corpus. This cost model allows the optimizer to estimate the cost of each potential execution strategy and to choose the one with the fastest predicted running time.

Fig. 4 presents three possible execution strategies for the *PersonPhoneAll* rule in Fig. 2. If the optimizer estimates that the evaluation cost of *Person* is much lower than that of *Phone*, then it can determine that Plan B has the lowest evaluation cost among the three, because Plan B only evaluates *Phone* in the "right" neighborhood for each instance of *Person*. More details of our algorithms for enumerating plans can be found in (F.Reiss et al., 2008).

The optimizer in SystemT chooses the best execution plan from a large number of different algebra graphs available. Depending on the execution plan generated by the optimizer, SystemT may evaluate views out of order, or it may skip evaluating some views entirely. It may share work among views or combine multiple equivalent views together. Even within the context of a single view, the system can choose among several different execution strategies without affecting the semantics of the annotator. This decoupling is possible because of the declarative approach in SystemT, where the AQL rules specify only what patterns to extract and not how to extract them. Notice that many of these strategies cannot be implemented using a transducer. In fact, we have formally proven that within this large search space, there generally exists an execution strategy that implements the rule semantics far more efficiently than the fastest transducer could (Chiticariu et al., 2010b). This approach also allows for greater rule expressivity, because the rule language is not constrained by the need to compile to a finite state transducer, as in traditional CPSL-based systems.

5 The Runtime

The SystemT Runtime is a compact, small memory footprint, high-performance Java-based runtime engine designed to be embedded in a larger system. The runtime engine works in two steps. First, it instantiates the physical operators in the compiled operator graph generated by the optimizer. Second, once the first step has been completed, the runtime feeds documents through the operator graph one at a time, producing annotations.

SystemT exposes a generic Java API for the integration of its runtime environment with other applications. Furthermore, SystemT provides two specific instantiations of the Java API: a *UIMA API* and a *Jaql function* that allow the SystemT runtime to be seamlessly embedded in applications using the UIMA analytics framework (UIMA, 2010), or deployed in a Hadoop-based environment. The latter

allows SystemT to be embedded as a Map job in a map-reduce framework, thus enabling the system to scale up and process large volumes of documents in parallel.

5.1 Memory Consumption

Managing memory consumption is very important in information extraction systems. Extracting structured information from unstructured text requires generating and traversing large in-memory data structures, and the size of these structures determines how large a document the system can process with a given amount of memory.

Conventional rule-based IE systems cannot garbage-collect their main-memory data structures because the custom code embedded inside rules can change these structures in arbitrary ways. As a result, the memory footprint of the rule engine grows continuously throughout processing a given document.

In SystemT, the AQL view definitions clearly specify the data dependencies between rules. When generating an execution plan for an AQL annotator, the optimizer generates information about when it is safe to discard a given set of intermediate results. The SystemT Runtime uses this information to implement garbage collection based on reference-counting. This garbage collection significantly reduces the system's peak memory consumption, allowing SystemT to handle much larger documents than conventional IE systems.

6 The Development Environment

The SystemT Development Environment assists a developer in the iterative process of developing, testing, debugging and refining AQL rules. Besides standard editor features present in any well-respected IDE for programming languages such as syntax highlighting, the Development Environment also provides facilities for visualizing the results of executing the rules over a sample document collection as well as explaining in detail the *provenance* of any output annotation as the sequence of rules that have been applied in generating that output.

7 Evaluation

As discussed in Section 1, our goal in building SystemT was to address the scalability and usability challenges posed by enterprise applications. As such, our evaluation focuses on these two dimensions.

Application Type	Type of Platform
brand management	server-side
business insights	server-side
client-side mashups	client-side
compliance	server-side
search (email, web, patent)	server-side
security	server-side
server-side mashups	server-side

Table 1: Types of applications using SystemT

7.1 Scalability

Table 1 presents a diverse set of enterprise applications currently using SystemT. SystemT has been deployed in both client-side applications with strict memory constraints, as well as on applications on the cloud, where it can process petabytes of data in parallel. The focus on scalability in the design of SystemT is essential for its flexible execution model. First of all, efficient execution plans are generated automatically by the SystemT Optimizer based on sample document collections. This ensures that the same annotator can be executed efficiently for different types of document collections. In fact, our previous experimental study shows that the execution plan generated by the SystemT optimizer can be 20 times or more faster than a manually constructed plan (F.Reiss et al., 2008). Furthermore, the Runtime Environment of SystemT results in compact memory footprint and allows SystemT to be embedded in applications with strict memory requirements as small as 10MB.

In our recent study over several document collections of different sizes, we found that for the same set of extraction tasks, the SystemT throughput is at least an order of magnitude higher than that of a state-of-the-art grammar-based IE system, with much lower memory footprint (Chiticariu et al., 2010b). The high throughput and low memory footprint of SystemT allows it to satisfy the scalability requirement of enterprise applications.

7.2 Usability

Table 2 lists different types of annotators built using SystemT for a wide range of domains. Most,

Domain	Sample Annotators Built
blog	`Sentiment`, `InformalReview`
email	`ConferenceCall`, `Signature`, `Agenda`, `DrivingDirection`, `PersonPhone`, `PersonAddress`, `PersonEmailAddress`
financial	`Merger`, `Acquisition`, `JointVenture`, `EarningsAnnouncement`, `AnalystEarningsEstimate`, `DirectorsOfficers`, `CorporateActions`
generic	`Person`, `Location`, `Organization`, `PhoneNumber`, `EmailAddress`, `URL`, `Time`, `Date`
healthcare	`Disease`, `Drug`, `ChemicalCompound`
web	`Homepage`, `Geography`, `Title`, `Heading`

Table 2: List of Sample Annotators Built Using SystemT for Different Domains

if not all, of these annotators are already deployed in commercial products. The emphasis on usability in the design of SystemT has been critical for its successful deployment in various domains. First of all, the declarative approach taken by SystemT allows developers to build complex annotators without worrying about performance. Secondly, the expressiveness of the AQL language has greatly eased the burden of annotator developers when building complex annotators, as complex semantics such as duplicate elimination and aggregation can be expressed in a concise fashion (Chiticariu et al., 2010b). Finally, the Development Environment further facilitates annotator development, where the clean semantics of AQL can be exploited to automatically construct explanations of incorrect results to help a developer in identifying specific parts of the annotator responsible for a given mistake. SystemT has been successfully used by enterprise application developers in building high quality complex annotators, without requiring extensive training or background in natural language processing.

8 Demonstration

This demonstration will present the core functionalities of SystemT. In particular, we shall demonstrate the iterative process of building and debugging an annotator in the Development Environment. We will then showcase the execution plan automatically generated by the Optimizer based on a sample document collection, and present the output of the Runtime Environment using the execution plan. In our demonstration we will first make use of a simple annotator, as the one shown in Fig. 2, to illustrate the main constructs of AQL. We will then showcase the generic state-of-the-art SystemT Named Entities Annotator Library (Chiticariu et al., 2010c) to illustrate the quality of annotators that can be built in our system.

References

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

B. Boguraev. 2003. Annotation-based finite state processing in a large-scale nlp arhitecture. In *RANLP*.

P. Bohannon et al. 2008. Purple SOX Extraction Management System. *SIGMOD Record*, 37(4):21–27.

L. Chiticariu, Y. Li, S. Raghavan, and F. Reiss. 2010a. Enterprise information extraction: Recent developments and open challenges. In *SIGMOD*.

Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick R. Reiss, and Shivakumar Vaithyanathan. 2010b. Systemt: an algebraic approach to declarative information extraction. ACL.

Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Frederick Reiss, and Shivakumar Vaithyanathan. 2010c. Domain adaptation of rule-based annotators for named-entity recognition tasks. EMNLP.

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.

A. Doan et al. 2008. Information extraction challenges in managing unstructured data. *SIGMOD Record*, 37(4):14–20.

A. Doan, R. Ramakrishnan, and S. Vaithyanathan. 2006. Managing Information Extraction: State of the Art and Research Directions. In *SIGMOD*.

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

A. Jain, P. Ipeirotis, and L. Gravano. 2009. Building query optimizers for information extraction: the sqout project. *SIGMOD Rec.*, 37:28–34.

R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. 2008. SystemT: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13.

D. Z. Wang, E. Michelakis, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein. 2010. Probabilistic declarative information extraction. In *ICDE*.