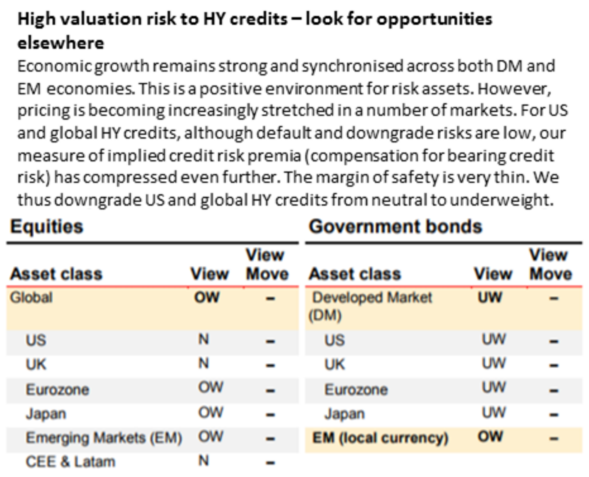
SystemT: Declarative Text Understanding for Enterprise

Laura Chiticariu[†] **Marina Danilevsky**[‡] **Yunyao Li**[‡] **Frederick R. Reiss**[‡] **Huaiyu Zhu**[‡]
[†]IBM Watson [‡]IBM Research - Almaden
650 Harry Rd
San Jose, CA 95120
`{chiti,mdanile,yunyaoli,frreiss,huaiyu}@us.ibm.com`

Abstract

The rise of enterprise applications over unstructured and semi-structured documents poses new challenges to text understanding systems across multiple dimensions. We present SystemT, a declarative text understanding system that addresses these challenges and has been deployed in a wide range of enterprise applications. We highlight the design considerations and decisions behind SystemT in addressing the needs of the enterprise setting. We also summarize the impact of SystemT on business and education.



High valuation risk to HY credits – look for opportunities elsewhere
Economic growth remains strong and synchronised across both DM and EM economies. This is a positive environment for risk assets. However, pricing is becoming increasingly stretched in a number of markets. For US and global HY credits, although default and downgrade risks are low, our measure of implied credit risk premia (compensation for bearing credit risk) has compressed even further. The margin of safety is very thin. We thus downgrade US and global HY credits from neutral to underweight.

Equities

Asset class	View	View Move
Global	OW	–
US	N	–
UK	N	–
Eurozone	OW	–
Japan	OW	–
Emerging Markets (EM)	OW	–
CEE & Latam	N	–

Government bonds

Asset class	View	View Move
Developed Market (DM)	UW	–
US	UW	–
UK	UW	–
Eurozone	UW	–
Japan	UW	–
EM (local currency)	OW	–

Figure 1: Fragment of investment report.

1 Introduction

With the proliferation of information in unstructured and semi-structured form, text understanding (TU) is becoming a fundamental building block in enterprise applications. Numerous tools, algorithms and APIs have been developed to address various text understanding sub-tasks, ranging from low-level text tasks (e.g., tokenization) and core natural language processing (e.g., syntactic and semantic parsing) to higher-level tasks such as document classification, entity and relation extraction, and sentiment analysis. Real world applications usually require several such components.

As an example, consider a *Financial Investment Research Analysis* application, which leverages financial market analyst reports such as the one in Fig. 1 to inform automatic trading and financial recommendations. Information is conveyed in both natural language (e.g., "*We thus downgrade US and global HY credits*") and tabular form, requiring both natural language understanding primitives (e.g., syntactic or semantic parsing) and document structure understanding primitives (e.g., table titles, row and column headers, and the association of table cells and headers). Furthermore, the business problem involves higher-level tasks such as financial entity extraction (e.g., equities, bonds, currencies) in natural language and tabular forms, and sentiment analysis for such entities.

While approaches for solving individual TU sub-tasks have proliferated, considerably less effort has been dedicated to developing systems that enable building end-to-end TU applications in a principled, systematic and replicable fashion. In the absence of such systems, building a TU application involves piecing together individual components in an *ad hoc* fashion, usually requiring custom code to address the impedance mismatch in data models between the different components, and to bridge gaps in functionality. Different implementations of the same application may yield vastly different runtime performance characteristics as well as, even more worryingly, different output semantics. For example, two developers may make disparate assumptions implementing even such a seemingly simple text operation as dictionary matching: Should dictionary terms match the input text only on token boundaries, or is matching allowed in the middle of a token? Which tokenization approach should be used? Is matching case sensitive or insensitive? In an enterprise environment, such *ad-hoc* approaches produce code repositories that are difficult to understand, maintain, reuse in new applications, or optimize for runtime performance.

This paper presents SystemT, an industrial-strength system for developing end-to-end TU applications in a declarative fashion. SystemT was developed at IBM Research and has been widely adopted inside and outside IBM (Sec. 4) Borrowing ideas from database systems, commonly used text operations are abstracted away as built-in operators with clean, well-specified semantics and highly optimized internal implementations, and exposed through a formal declarative language called AQL *(Annotation Query Language)*. Fig. 2 illustrates a fragment of AQL for extracting financial entities and associated sentiment from financial reports as in Fig. 1. The snippet illustrates several types of declarative TU structures (or rules) expressible in AQL, including sequential structures (*AssetClass*), semantic understanding structures (*RecommendationNU*), and table understanding structures (*RecommendationTable*). The rules leverage built-in text operators readily available in AQL, including dictionary matching (*AssetClassSuffixes*), core NLP operators such as Semantic Role Labeling (SRL) (*Verbs*, *Arguments*), and document structure operators such as table structure understanding (*AllCells*). Sec. 3 details the data model and semantics of AQL.

Architecture Overview. As illustrated in (Fig. 3), SystemT consists of two major components: the *Compiler* and the *Runtime*.

The *Compiler* takes as input a TU program specified in AQL and compiles it into an execution plan. AQL is a purely declarative language: the developer specifies *what* should be extracted, but not *how* to do it. The *Optimizer* computes the *how* automatically by enumerating multiple logical equivalent plans, and choosing a plan with the least estimated cost. Since each operator has well-specified semantics, the Optimizer can automatically determine when operators can be reordered, merged, or even discarded without affecting the output semantics, while significantly increasing the runtime performance of the TU program. The *Operator Graph* captures the execution plan generated by the *Optimizer* and is used by the *Runtime* to decide the actual sequence of operations.

The *Runtime* is a lightweight engine that loads the *Operator Graph* and then processes inputs

```
create dictionary AssetClassSuffixes
    with lemma_match
    as ('bond', 'credit', 'equity');

create view AssetClassSuffix as
    extract dictionary 'AssetClassSuffixes' on D.text as span
    from Document D;

create view AssetClass as
    extract pattern <G.span> <Token>{0,1} <S.span> as span
    from GeoRegion G, AssetClassSuffix S;

create dictionary RecommendVerbs as ('upgrade', 'downgrade');

create table PolarityAdj (polarity Text, adj Text)
   as values
   ('negative', 'underweight'),
   ('negative', 'UW'),
   ('neutral',  'neutral'),
   ('neutral',  'N'),
   ('positive', 'overweight'),
   ('positive', 'OW');


create view RecommendationNU as
    select E.span as entity, P.polarity
    from Verbs V, Arguments A, Contexts C, AssetClass E, PolarityAdj P
    where Equals (V.id, A.verbId)
      and Equals (V.id, C.verbId)
      and Contains (A.span, E.span)
      and MatchesDict(V.verbBase, 'RecommendVerbs')
      and Equals(C.type, 'to')
      and Equals(GetText(C.span), P.adj);


create view RecommendationTable as
    select C1.span as entity, P.polarity
    from AllCells C1, AllCells C2, PolarityAdj P
    where Equals(C1.colHeader,'Asset class')
      and Equals(C2.colHeader,'View')
      and Equals(C1.rowId, C2.rowId)
      and Equals(GetText(C2.span), P.adj);
```

Figure 2: Example AQL specifications expressing a sequential structure (*AssetClass*), a semantic structure (*RecommendationNU*) and a table structure (*RecommendationTable*). (Simplified for presentation.)

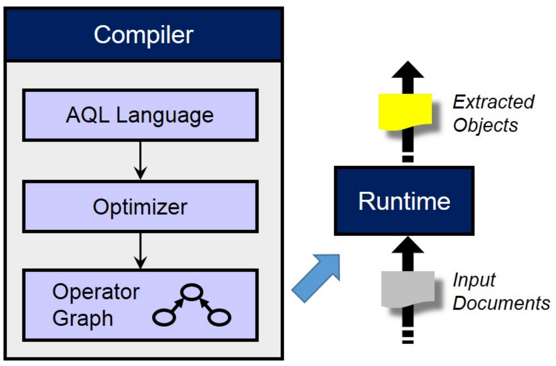


Figure 3: SystemT Overall Architecture

document-at-a-time (DAAT), with the text and metadata of each document acting like a separate "database" from the perspective of the AQL rules. This DAAT model speeds up join and aggregation operations within a document, as the entire document's data is in memory. It also simplifies scale-out processing and can be scaled up by utilizing multithreading and multiple processes, as well as various cluster and cloud environments.

Related Work. SystemT's declarative approach is a departure from other rule-based TU systems. Early systems (Cunningham et al., 2000; Boguraev, 2003; Drozdzynski et al., 2004) are based on the Common Pattern Specification Language (CPSL) (Appelt and Onyshkevych, 1998), a cascading grammar formalism where 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. Each grammar consists of a set of rules evaluated in a left-to-right fashion over the input annotations, with multiple grammars cascaded together and evaluated bottom-up.

As discussed in (Chiticariu et al., 2010), grammar-based systems suffer from two fundamental limitations: expressivity and performance. Formal studies of AQL semantics have shown that AQL is strictly more expressive than regular expression-based languages such as CPSL (Sec. 3.2.5). Furthermore, the rigid evaluation order imposed in grammar-based systems has significant runtime performance implications, as the system is effectively forced into a fixed pipeline execution strategy, leaving little opportunity for global optimization. While the expressivity of grammar-based systems has been extended in different ways, such as additional built-in constructs (Boguraev, 2003; Drozdzynski et al., 2004; Cunningham et al., 2000), or allowing a mix of sequential patterns and rules over dependency parse trees (Valenzuela-Escárcega et al., 2016), such extensions do not fundamentally address the inherent expressivity and performance limitations due to the intertwining of rule language semantics and execution strategy. In contrast, SystemT's declarative approach enables the optimizer to explore a variety of execution plans, resulting in orders of magnitude higher throughput and a lower memory footprint (Chiticariu et al., 2010).

2 Requirements for Enterprise TU

While the traditional accuracy requirement remains important, emerging enterprising applications introduce additional requirements for enterprise-scale TU systems.

Scalability. Compared to conventional TU task corpora, such as those available via the Linguistic Data Consortium, enterprise TU corpora comprise much larger volumes of data from a wider variety of sources, ranging from user-created content and public data, to proprietary data such as call-center logs. The TU system must scale for both documents (e.g., a single financial report runs in the tens of MBs) and document collections (e.g., 500 million new tweets posted daily on Twitter; terabytes of system logs produced hourly in a medium-size data center).

Expressivity. Enterprise TU must handle an ample variety of functionalities required by different enterprise applications, from known natural language processing tasks such as entity extraction to more practical challenges such as table structure understanding. Data may come from plain text, semi-structured text (e.g., HTML or XML), or the conversion of a binary format (PDF, Word) to text. Different degrees of noise may be present, from manually introduced noise such as typos and acronyms (e.g., in tweets) to systematic errors such as those resulting from format conversion.

Transparency. Enterprise TU must be transparent and enable ease of comprehension, debugging and enhancement, to avoid TU development becoming the bottleneck for building enterprise applications. Furthermore, as the underlying data or application requirements change, it must be easy to adapt existing TU programs in response.

Extensibility. Enterprise TU, no matter how well designed, might not provide all of the capabilities required by a real-world use case out-of-the-box. As such, it should be extensible to gracefully handle tasks that are not natively supported.

3 SystemT Highlights

We now describe the key design considerations and decisions behind SystemT and discuss how they help address the requirements in Sec. 2.

3.1 Preliminaries

Data model. AQL operates over a simple relational data model with three basic data structures: *field value*, *tuple*, and *view*. A tuple (aka an *annotation*) consists of a list of named and typed fields. The field values can be of the text-specific type *span*, representing a region of text within a document identified by its "begin" and "end" positions, or any of a collection of familiar data types such as *text*, *integer*, *scalar List*, and *null* (with 3-value logic, similar to SQL). A view is a set of tuples with the same schema (field name and field type).

Statements and modules. Each AQL statement defines a view as the result of some operations on one or more other views, which are defined by previous statements. A special view called *Document* is created automatically, which contains a single tuple with the document text. AQL code is organized in modules which provide a namespace for the views. The modules are compiled. At run time the modules are loaded on demand and executed.

3.2 Expressivity

AQL[1] is similar in syntax to the database language SQL, chosen for its expressivity and its familiarity to enterprise developers. It provides a number of TU constructs, including primitive extraction operators for finding parts of speech, matches of regular expressions and dictionaries, as well as set operators such as sequence, union, filter and consolidate. Each operator implements a single basic atomic operation, producing and consuming sets of tuples. AQL developers create TU programs by composing these operators together into sets of rules, or statements.

3.2.1 Basic Primitives

Three of the most basic operators of AQL include: *Extract* ($\mathcal{E}$) performs character level operations such as regular expression and dictionary matching over text, creating a tuple for each match. *Select* (τ) applies a predicate to a set of tuples, and outputs all those tuples that satisfy the predicate. *Join* ($\bowtie$) applies a predicate to pairs of tuples from two sets of input tuples, outputting all pairs that satisfy the predicate.

AQL also provides a sequence pattern notation, similar in its grammar-like syntax to that of CPSL (Appelt and Onyshkevych, 1998), which is translated internally into one or more select and extract statements. Other operators include *Detag* for removing HTML tags and retaining the locations of important HTML metadata such as section, lists and table markup, *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* and *Limit* for sorting and truncating output, and *Union* and *Minus* for expressing set union and set difference, respectively. Rules can also be easily customized to particular TU domains using external dictionary and table structures, which can be rapidly populated with relevant terms without the need to alter existing AQL code.

3.2.2 Advanced Primitives

SystemT has built-in multilingual support including tokenization, part of speech and lemmatization for over 20 languages. TU developers can utilize the multilingual support via AQL without having to configure or manage any additional resources. Language expansion is enabled as described in Sec. 3.4. SystemT also has advanced primitives for semantic role labeling (SRL), the task of labeling predicate-argument structure in sentences with shallow semantic information. Such advanced primitives enable the creation of cross-lingual TU programs (see, e.g., (Akbik et al., 2016).)

[1]AQL manual: `https://ibm.biz/BdZpjX`.

3.2.3 Extensions with Pre-Built Libraries

Corpora can introduce a variety of additional TU challenges, including having a high degree of noise (e.g., non-standard word forms or informal usage patterns), exposing data through non-free-text structures (e.g., tables), or existing in a difficult-to-digest format (e.g., PDF). We have extended the functionality of SystemT by creating pre-built libraries with advanced TU capabilities such as text normalization (Baldwin and Li, 2015), semantic table processing (Chen et al., 2017), and document format conversion.

3.2.4 An AQL Example

As discussed in Sec.1, Fig. 2 shows AQL snippets for extracting sentiment around financial assets from investment reports such as the one in Fig. 1. View *AssetClass* identifies financial entities using syntactic constructs: a geographical region followed within 0 to 1 tokens by a common suffix (e.g., ‘credits’, ‘equities’).

View *RecommendationNU* uses semantic parse primitives to identify recommendations expressed in natural language such as *‘We downgrade ... from neutral to underweight’*. In order to assign the correct polarity to *‘underweight’* and *‘neutral’*, SRL information exposed in views *Verbs*, *Arguments* and *Contexts* is joined with the view *AssetClass* and the AQL table *PolarityAdj*, mapping domain terminology (e.g., *‘UW’, ‘underweight’*) to sentiment polarities. The view uses several AQL built-in join and selection predicates (equality, span containment, dictionary matching.)

Finally, view *RecommendationTable* identifies recommendations present in tabular form. It leverages table semantic understanding primitives exposed through the view *AllCells*, which connects the span of each table cell with additional row and column metadata such as row and column ids and headers. *RecommendationTable* identifies all cells that appear in the same row under specific column headers, and assigns polarity using *PolarityAdj*.

3.2.5 Formal Analysis

Fagin *et al* (Fagin et al., 2013, 2016) formalized the semantics of relational-based extraction languages like AQL with the theoretical framework *document spanners*. The initial paper showed that a restricted version of document spanners are broadly equivalent in expressivity to regular-expression-based languages such as CPSL. The follow-on paper described how *consolidation*, a facility also supported by SystemT for addressing overlapping spans in intermediate results, extends the expressivity of the spanners framework.

3.3 Scalability

SystemT is architected to achieve scalability along three dimensions: (1) Cost-based Optimizer to select the most efficient execution plan for a given declarative specification and input documents; (2) Document-centric Runtime engine, scalable by trivial parallelization, leveraging the advances in parallel computing (e.g., via Hadoop/Spark); (3) Hardware acceleration for sped up computation.

3.3.1 Optimizer

Internally, the declarative AQL specification compiles into an algebra consisting of individual operators with well-specified semantics and properties that compose to form an execution plan. The SystemT Optimizer chooses an efficient execution plan among all possible equivalent execution plans for a given AQL specification. The Optimizer is inspired by relational query optimizers, which have been operational in commercial database systems for over 40 years (Astrahan et al., 1979), with one important difference: While SQL optimizers are designed to minimize I/O costs, the SystemT Optimizer focuses on minimizing CPU-intensive text operations. We briefly describe the two classes of optimization techniques used, and refer the reader to (Reiss et al., 2008) for a detailed description.

Cost-based optimizations are used to select an efficient join ordering for AQL statements that join two or more relations (as in Fig. 2). The search algorithm uses dynamic programming to build up larger sub-plan candidates from smaller sub-plans. The Optimizer uses a text-centric cost model to estimate the execution time of each operator based on the input size, selectivity of the join predicate, and choice of join algorithm.

Rewrite-based optimizations include plan rewrites known to always speed up the execution time, including (1) text-centric optimizations such as Shared Dictionary Matching and Shared Regex Matching, which group multiple dictionaries or regular expressions together to be executed in a single pass over the document; and (2) relational-style query rewrites, such as pushing down select and project operators.

3.3.2 Runtime

The SystemT Runtime engine is architected as a compact in-memory embeddable Java library (<2 MB). It is purely document-centric, leaving the storage of document collections and extracted artifacts to the embedding application. The engine exposes two low-level Java APIs: (1) *instantiate()* creates an *OperatorGraph instance*, an in-memory representation of the execution plan generated by the Optimizer, and (2) *execute()* takes as input an in-memory representation of the input document and returns an in-memory representation of the objects extracted from that document. The APIs are reentrant and thread safe. The *execute()* API is multithreaded (a single OperatorGraph instance can be used simultaneously by multiple threads, each annotating a different document.)

This document-centric in-memory design has provided the flexibility necessary to embed the SystemT Runtime in a variety of environments, including: (1) Big Data platforms (e.g., Hadoop, Spark) (2) Cloud platforms (e.g., Kubernetes), and (3) custom applications. We enable SystemT in Big Data and Cloud platforms by providing platform-specific APIs that mirror the low-level Java APIs, but are tailored to the platform's compute and data models. Custom applications not built on Big Data or Cloud platforms (e.g., a desktop email client, a travel app, or a compliance program) can embed the SystemT runtime in any way suitable for the application.

3.3.3 Hardware Acceleration

The separation of specification from implementation has the added advantage that new hardware can be relatively easy to take advantage of, with no changes in the declarative specification of the program. For example, recent work on hardware acceleration for low level text operators such as regular expressions (Atasu et al., 2013) can be leveraged by extending the Optimizer's search space and cost model to incorporate alternative hardware implementations of individual operators and associated cost model (Polig et al., 2018).

```
create function classifyPolarity (
 firstArg table(entity String,
                polarity String,
                numPolarity Integer) as locator)
return table( entity String, docLevelPolarity String)
external_name
'udfs/udf.jar:udf.ClassifyTableFn!eval'
language java
deterministic
called on null input;
```

(a) Declaring UDF Function

```
create view RecommendationAll as
    (select R.entity, R.polarity
       from RecommendationNU R)
    union all
    (select R.entity, R.polarity
       from RecommendationTable R);

create view RecommendationFeatures as
select GetText(E.entity) as entity, R.polarity,
       Count(E.polarity) as numPolarity
from RecommendationAll R
group by GetText(E.entity), R.polarity;

create view RecommendationDocLevel as
select T.entity, T.polarity
from classifyPolarity(RecommendationFeatures) T;
```

(b) Using UDF Function

Figure 4: Example UDF function: document-level sentiment classification

3.4 Extensibility

SystemT has been built with extensibility in mind. With a grey-box design, users are able to extend its capabilities via the following mechanisms.

User-Defined Functions (UDFs) can be defined to extend AQL by performing operations that are not supported natively by SystemT. Using a UDF requires three simple steps: (1) Implementing the function in Java or PMML; (2) Declaring it in AQL; and (3) Using it in AQL. Fig. 4 illustrates how to extend AQL to support a simple document-level sentiment classifier for asset classes using features extracted with sentiment extractors defined earlier in Sec. 3.2.4. The example illustrates several other AQL constructs, including unioning, grouping and counting annotations.

NLP Primitive API. Low-level language-dependent primitives such as tokenization, part-of-speech tagging, lemmatization or semantic role labeling are pluggable through an internal API, and automatically exposed to all AQL constructs. For example, the matching of *AssetClassSuffixes* dictionary on lemma form in Fig. 2 is enabled by the underlying tokenizer and lemmatizer for the given language. This has two advantages: (1) isolating language-dependent primitives from the rest of the system, without requiring changes to the AQL language itself; and (2) leveraging newer primitive models as they become available, without requiring changes to existing AQL programs.

3.5 Transparency and Machine Learning

A common criticism of pure machine learning (ML) systems in the enterprise world is that statistical models are opaque to the application using them, making the results difficult to explain or be quickly fixed (Chiticariu et al., 2013). SystemT addresses this challenge by using a declarative language for specifying the TU program. Since the results are produced by constructs with well-understood semantics, it is possible to automatically generate explanations of why a certain output was or was not produced.

At the same time, SystemT has the flexibility to leverage ML techniques in the context of its overall declarative framework in two dimensions. First, primitive APIs and the user-defined interface (Sec. 3.4) allow for plugging in low-level NLP primitives, as well as trained models for higher-level tasks such as entity extraction. The former makes NLP primitives available to all AQL constructs. The latter allows AQL specifications to provide features to the model, post-process the result of the model, or use the model as building block in solving a higher-level task. Second, ML techniques are leveraged to learn AQL programs, thereby generating a deterministic, transparent model in lieu of a probabilistic one. Such algorithms can be incorporated in the SystemT IDEs (Sec. 3.6) to speed up AQL development.

3.6 Integrated Development Environments

SystemT provides integrated development environments (IDEs) designed for a wide spectrum of users. The professional IDE allows expert developers to create complex TU programs in AQL (Li et al., 2012). The visual IDE enables novice users and non-programmers to construct drag-and-drop TU programs without learning AQL (Li et al., 2015). Both IDEs leverage ML and human-computer interaction techniques as those summarized in (Chiticariu et al., 2015) to support typical development life cycles of TU tasks (Fig. 5).

3.7 Empirical Evaluation

In addition to theoretical studies of AQL expressivity and runtime performance, we have also evaluated SystemT empirically on multiple TU tasks. We show that extractors built in AQL yield results of comparable quality to the best published results

Domain	Application
Compliance	Multilingual named entity extraction for document retention
	Element extraction and classification in legal documents (e.g. contract, regulations)
	Named entity extraction for document retention and regulation compliance
Email/Online Chat	Entity and event extraction in emails and online chat for AI assistant
	Named entity extraction in emails for search
Finance	Extracting financial information from public records to estimate the true cost of water
	Extracting company fundamentals (e.g. financial metrics and key personnel) from regulatory filings to create a knowledge base
General domain	Building text understanding programs for entity extraction and sentiment analysis
	Building text understanding programs
Life science	Extracting features (e.g. entities and relations) from life science literature to speed up the discovery of new drugs
Material science	Extracting entities and relations from natural language and tables in material science literature to speed up the discovery for new materials
Security and privacy	Personal information extraction and redaction for security and privacy
Social media	Sentiment analysis over social media for indepth understanding of social behavior
Travel	Extracting information and sentiment from online reviews to build AI assist for travel

Table 1: Partial list of SystemT applications

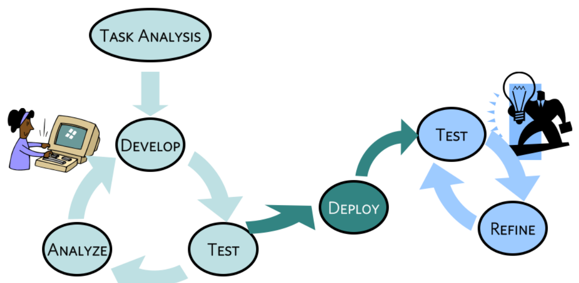


Figure 5: Development life cycles using SystemT

on several competition datasets, while achieving orders of magnitude speed-up in processing time, and requiring smaller memory utilization (Krishnamurthy et al., 2009; Chiticariu et al., 2010; Nagesh et al., 2012; Wang et al., 2017).

4 Impact of SystemT

Business Impact. Started as a research prototype, SystemT has been widely adopted within IBM and its clients. It is embedded in over 10 commercial product offerings and used in numerous internal and external projects[2] for a wide variety of enterprise applications, a small subset of which is highlighted in Table 1.

Research Impact. Various aspects of SystemT have been published in 40+ major research conferences and journals in diverse areas, including natural language processing, database systems, artificial intelligence and human-computer interaction. This is the first time that all aspects of the system, including design considerations and current functionality, are described in a single paper.

Education Impact. SystemT is available to teachers and students under a free academic license. We have developed a full graduate-level course on text understanding using SystemT, which has been taught in several universities. A version of this class has been made available[3] as a MOOC with 10,000+ students enrolled in less than 18 months.

5 Conclusion

In this paper, we discuss new challenges posed by enterprise applications to text understanding (TU) systems. We present SystemT, an industrial-strength system for developing end-to-end TU applications in a declarative fashion. We highlight the key design decisions and discuss how they help meet the needs of the enterprise setting. SystemT has been used to build enterprise applications in a wide range of domains, and is publicly available for commercial and academic usage.

References

Alan Akbik, Laura Chiticariu, Marina Danilevsky, Yonas Kbrom, Yunyao Li, and Huaiyu Zhu. 2016. Multilingual information extraction with PolyglotIE. In *COLING'16*.

Douglas E. Appelt and Boyan Onyshkevych. 1998. The common pattern specification language. In *Pro-*

[2] Example products exposing or embedding SystemT include: IBM BigInsights, IBM Streams, IBM Watson for Drug Discovery

[3] `https://cognitiveclass.ai/learn/text_analytics/`

ceedings of a Workshop held at Baltimore, Maryland, October 13-15, 1998, TIPSTER '98, pages 23–30.

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Mario Schkolnick, Patricia G. Selinger, Donald R. Slutz, H. Raymond Strong, Paolo Tiberio, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1979. System R: A relational data base management system. *IEEE Computer*, 12(5):42–48.

Kubilay Atasu, Raphael Polig, Christoph Hagleitner, and Frederick R. Reiss. 2013. Hardware-accelerated regular expression matching for high-throughput text analytics. In *FPL 2013*, pages 1–7.

Tyler Baldwin and Yunyao Li. 2015. An in-depth analysis of the effect of text normalization in social media. In *HLT-NAACL*, pages 420–429.

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

Xilun Chen, Laura Chiticariu, Marina Danilevsky, Alexandre Evfimievski, and Prithviraj Sen. 2017. A rectangle mining method for understanding the semantics of financial tables. In *International Conference on Document Analysis and Recognition (ICDAR)*.

Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick R. Reiss, and Shivakumar Vaithyanathan. 2010. Systemt: An algebraic approach to declarative information extraction. ACL '10, pages 128–137.

Laura Chiticariu, Yunyao Li, and Frederick Reiss. 2015. Transparent machine learning for information extraction: State-of-the-art and the future. In *EMNLP*.

Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. Rule-based information extraction is dead! long live rule-based information extraction systems! In *EMNLP 2013*, pages 827–832.

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.

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.

Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2013. Spanners: a formal framework for information extraction. In *PODS 2013*, pages 37–48.

Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2016. A relational framework for information extraction. *SIGMOD Rec.*, 44(4):5–16.

Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. 2009. Systemt: A system for declarative information extraction. *SIGMOD Rec.*, 37(4):7–13.

Yunyao Li, Laura Chiticariu, Huahai Yang, Frederick Reiss, and Arnaldo Carreno-fuentes. 2012. WizIE: A best practices guided development environment for information extraction. In *ACL 2012 System Demonstrations*, pages 109–114.

Yunyao Li, Elmer Kim, Marc A. Touchette, Ramiya Venkatachalam, and Hao Wang. 2015. VINERy: A visual ide for information extraction. *Proc. VLDB Endow.*, 8(12):1948–1951.

Ajay Nagesh, Ganesh Ramakrishnan, Laura Chiticariu, Rajasekar Krishnamurthy, Ankush Dharkar, and Pushpak Bhattacharyya. 2012. Towards efficient named-entity rule induction for customizability. EMNLP-CoNLL '12, pages 128–138.

Raphael Polig, Kubilay Atasu, Heiner Giefers, Christoph Hagleitner, Laura Chiticariu, Frederick Reiss, Huaiyu Zhu, and Peter Hofstee. 2018. A hardware compilation framework for text analytics queries. *J. Parallel Distrib. Comput.*, 111:260–272.

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

Marco Antonio Valenzuela-Escárcega, Gus Hahn-Powell, and Mihai Surdeanu. 2016. Odin's runes: A rule language for information extraction. In *LREC*. European Language Resources Association (ELRA).

Chenguang Wang, Doug Burdick, Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, and Huaiyu Zhu. 2017. Towards re-defining relation understanding in financial domain. In *DSMM'17*, pages 8:1–8:6.