Information Systems 53 (2015) 124–144

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

INDREX: In-database relation extraction

Torsten Kilias [a,*], Alexander Löser [b], Periklis Andritsos [c]

[a] Technische Universität Berlin, Strasse des 17. Juni Berlin, Germany
[b] Beuth Hochschule für Technik Berlin, Luxemburger Strasse 10 Berlin, Germany
[c] Université de Lausanne, Bâtiment Internef 1015, Lausanne, Switzerland

ARTICLE INFO

Article history:
Received 20 March 2014
Received in revised form
5 September 2014
Accepted 18 November 2014
Available online 10 December 2014

Keywords:
Iterative text mining in a RDBMS
Ad-hoc reports from text data
Information extraction

ABSTRACT

The management of text data has a long-standing history in the human mankind. A particular common task is extracting relations from text. Typically, the user performs this task with two separate systems, a relation extraction system and an SQL-based query engine for analytical tasks. During this iterative analytical workflow, the user must frequently ship data between these systems. Worse, the user must learn to manage both systems. Therefore, end users often desire a single system for both analytical and relation extraction tasks.

We propose INDREX, a system that provides a single and comprehensive view of the whole process combining both relation extraction and later exploitation with SQL. The system permits a data warehouse style extract-transform-load of generic relations extracted from text documents and can support additional text mining analysis libraries or systems. Once generic relations are loaded, the user can define SQL queries on the extracted relations to discover higher level semantics or to join them with other relational data.

For executing this powerful task, our system extends the SQL-based analytical capabilities of a columnar-based massively parallel query processing engine with a broad set of user-defined functions and a data model that supports this task. Our white-box approach permits INDREX to benefit from built-in query optimization and indexing techniques of the under-laying query execution engine.

Applications that support both text mining and analytical workflows leverage new analytical platforms based on the MapReduce framework and its open source Hadoop implementation. We compare our system against this base line. We measure execution times for common workflows and demonstrate orders of magnitude improvement in execution time using INDREX.

1. Introduction

From the earliest days of computers, the analysis of textual data has been a fundamental application that has driven research and development. As the Internet has become a mass medium, searching text data has become a daily activity for everyone, from children to research scientists. On the other hand, hundreds of millions of active users generate searchable content in online-forums, blogs and wikis, while news stories appear in a plethora of platforms, from traditional news outlets (e.g. Reuters) to social media (e.g. Twitter). Consider the following analysis demand by journalists:

While browsing historical news articles, an editor wants to perform research with respect to the sentiment of news stories and how it has evolved over time. She wants to initially extract names of journalists, dates and story titles, filter them by subject (e.g. health or economy) and associate (join) them with information that exists in external dictionaries. A next step may involve the grouping of the stories across time and subject. Finally, she

* Corresponding author.
E-mail addresses: torsten.kilias@tu-berlin.de (T. Kilias), aloeser@beuth-hochschule.de (A. Löser), periklis.andritsos@unil.ch (P. Andritsos).

http://dx.doi.org/10.1016/j.is.2014.11.006
0306-4379/

may want to identify the most often discussed reason for the sentiment of specific stories.

For executing the aforementioned query, the ideal system needs to spot words that may represent sentiment. It needs to verify these words with information about known sentiment orientation, such as with values from an in-house relational data. Next, the system extracts all sentences that contain additional information about such a sentiment as well as the news categories and persons involved. This information does not exist yet in the in-house relational data. Finally, the system loads all information into a SQL-based relational DBMS and executes aggregations and grouping statements.

Similar queries and demands might arise from sales departments (monitor and identify leads that soon will buy a car), human resources (identify professionals with capabilities in text mining), market research (monitor the effectiveness of a campaign), product development (incorporate feedback from customers into the development process) as well as from the medical domain (anamnesis).

1.1. Problem statement

Neither Web search engines nor data warehouse systems support such analytical query workflows over both textual data and relational data. Therefore, most content producers, mostly the ordinary end-consumers but also other kinds of commercial knowledge workers, are not able to drill down into that intrinsic "knowledge" yet. Having relational information from text data available, with low costs for extracting and organizing it, provides knowledge workers and decision makers in an organization with insights that have, until now, not existed.

Managing text data vs. relational data: Managing text data differs fundamentally from managing relational data, *i.e.* data stored in relational tables. The first difference is the *data model*: text data is represented as bag-of-words, sequences of lexico-syntactic expressions or as dependency trees, while relational data always represents multi-sets. This model heterogeneity evolved over many decades and resulted in many *different systems*, which is the second difference. Currently, mature systems exist for *either* managing relational data *or* for extracting information from text. As a result, the user must *ship data between various systems* for executing necessary transformations. Moreover, transforming textual data in a relational representation *requires glue and development time* to bind these different system landscapes seamlessly. Finally, domain specific information extraction is an iterative task. It requires *to continuously adopt extraction rules and accompanying semantics* by both the extraction system and the database system. A result from the above discussion is that many projects that combine textual data with existing relational data may likely fail and/or be infeasible.

Query execution model for in-database relation extraction. In this work we overcome these system and model barriers. We propose the INDREX system that enables users to describe relation extraction tasks across documents and relational data with SQL, for the first time. For executing this task, INDREX users issue queries written in SQL on top of loaded base tables. These queries transfer generic candidate relations into semantic meaningful relations that can be used further in OLAP or other SQL-based applications. Once generic relations are loaded, the user can define SQL queries on the extracted relations to extract higher level semantics or to join them with other relational data.

Design requirements for INDREX : The evolution of INDREX is based on four essential design requirements.

Design requirement 1: *INDREX should provide a single schema for loading relations from an Open Information Extraction (OIE) system.*

Open information extraction extracts generic relations and their arguments from sentences in natural language text. Ideally, INDREX should provide a schema that will permit loading and storing such relations from any OIE-system. Later, the user may join generic relations, for example extracted from news stories, with domain specific relational data, such as news categories, journalists, geographical locations or politicians.

Executing such joins in short time is extremely helpful for adopting relations from a domain independent OIE system to a particular domain. A particulary helpful join is the theta-join, which covers all kinds of non-equality joins, such as joins that use regular expression (regex) conditions, joins that use `LIKE` predicates or other text similarity joins. This leads to our second design requirement.

Design requirement 2: *INDREX should provide joins and other integration operations for adopting generic relations from an OIE system with potentially existing domain specific relational data.*

Generic relations often do not provide key attributes. Therefore, the user must often formulate complex and potentially expensive theta joins. Moreover, values of join attributes may be homonymous, synonymous or may entail other attributes. Therefore users need to iterate multiple times over the join statement until the result shows a sufficient precision and recall. Next, often users desire to learn about the variety of join predicates from analyzing large data. However, text-based information follows a Zipfian distribution; a large data set includes a significant higher variance in textual expressions for the relation extraction task that a small sample could often not provide. For fulfilling these requirements the query processor of INDREX must leverage query optimization and indexing techniques, such as pipeline/task parallelism, data parallelism or instruction level parallelism. This leads to our third design requirement.

Design requirement 3: *Text data follows a Zipfian distribution. INDREX should be able to iteratively process data with such distributions and should provide answers to queries over millions of documents within seconds.*

The lingua franca of end users for analytical tasks on structured data is the structured query language, namely SQL. Millions of users have been trained to express their query demands in this declarative language. In addition, many applications update and retrieve data in an automated fashion with one of the SQL dialects. In contrast, the community of natural language processing (NLP) has not managed to standardize a core language for writing information extractors. Rather, this community relies on academic initiatives, such as GATE [18], commercial languages like AQL [14] or DIAL [28], integration frameworks like UIMA [29], RUTA [36] or in-house extractors based on regular

expressions, R, Python or Java. Ordinary SQL developers will often not be willing to learn and understand the differences and flavors of these abstraction principles. Rather, INDREX must hide the variety of different extraction frameworks through a single SQL based query interface. Our last design requirement is the following:

Design requirement 4: *Empower end-users with SQL support for analytical tasks on text.*

1.2. Our contributions

The INDREX system implements the data model from our previous work in [35]. This paper extends these preliminary findings significantly in the following areas:

Data model for relation extraction and analytical extraction workflows: We propose a data model that permits transformation operators to store, retrieve and combine text data with relational data. Our work extends [35] with an implementation for column-based query engine.

Operator formalization: Finding the right abstraction for expressing relation extraction tasks from text with SQL is a difficult task. We asked experienced SQL users to express their information demands in pseudo-SQL. Our user study revealed many missing query types for processing text data on top of a SQL-based query processor. We structure them into local queries, joins with existing relational data and aggregation queries. For these query types we rigorously formalize our operator design. This formalization permits the application of our abstractions to many system architectures.

White-box design of missing operators in Hadoop and in Cloudera IMPALA. Applications that support both text mining and analytical workflows leverage new analytical platforms based on the MapReduce framework [20], for example IBMs SystemT on Hadoop [9]. As a proof of concept we implemented as a base line system the INDREX query capabilities for a Hadoop-based infrastructure where users can formulate analytical queries with Pig Latin [44]. In addition, we extended the SQL-based analytical capabilities of Cloudera IMPALA, a columnar-based massively parallel query processing engine with a broad set of user-defined functions. Our white-box approach permits INDREX to benefit from built-in query optimization and indexing techniques of each underlying query execution engine.

Benchmark proposal and extensive experiments: We are not aware of any benchmark for relation extraction and joining tasks within the same RDBMS. Therefore another contribution is the proposal of a suite of business-oriented queries for typical enterprise tasks across text and relational data. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates text-based decision support systems that execute queries with a high degree of complexity. We report on the feasibility of INDREX for both implementations, the Hadoop-based implementation and the Cloudera IMPALA based implementation.

The rest of this paper is organized as follows: in Section 2, we review the relation extraction stack, describe the iterative relation extraction process and review existing work on batch-based relation extraction. Section 3 briefly reviews our data model from our work in [35] and discusses our extensions. In Section 4, we unravel common SQL patterns for the relation extraction task and formalize operators. In Section 5, we define a query benchmark and report from our extensive experiments. Finally, in Section 6, we summarize our work. The appendix of this paper contains our benchmark queries.

2. Related work

We abstract the task of relation extraction as an iterative multi-label multi-class classification task. Given a set of document-specific, surface, syntactic, deep syntactic, corpus-specific and domain-specific features the classifier determines occurrences in text (such as sequences of consecutive and non-consecutive characters and tokens) that likely represent a relationship of a particular semantic type.

In this section, we present relevant work around algorithms for computing required features and interactivity for adopting these features to a domain. Finally, we present existing RDBMS techniques for executing this task.

2.1. Understanding relation extraction

For identifying natural language features and their interplay (aka. conditional dependencies) the complex task of relation extraction requires several base extraction functionalities that depend on each other: first, the software needs to recognize document specific structures. Here, we focus on document structures that include natural language sentences and paragraphs. From these common structures the software will extract shallow syntax, deep syntax [7] and open information extraction [25]. Given these syntactic structures the software can determine un-typed binary [25], and higher order [17,2], candidate relationships and arguments. Next, the system clusters likely synonymous relationship candidates with the help of corpus specific distributions into the so-called synsets [3,43,33]. Finally, these synset clusters are further adapted towards the target schema through appropriate human interactions. A system could implement these domain adaptation procedures through active learning [46], or through rule writing environments [14]. In both cases, the human requires to overview corpus-wide distributions to learn common signals for the target domain.

2.2. Iterative domain adaptation process

Discovering relationships is an iterative task that involves *lookup*, *learn* and *explore* activities. This simple abstraction was recognized and published first by Bloom in 1956 [10]. Later, different disciplines enriched this abstraction model. Authors of [45] refined *lookup* activities into navigational, informational and transactional ones. Furthermore, the work in the context of service search by [42] gives example operators for each activity. For instance, the author considers aggregation, comparison and integration as activities for *learn* and analysis, exclusion and transformation as activities for *explore*. Most recently, authors of [6] apply the original ideas of Bloom to the problem of exploratory data and text mining. Given our stack from Fig. 1, we abstract this process into a step of an initial sequence of document and language specific

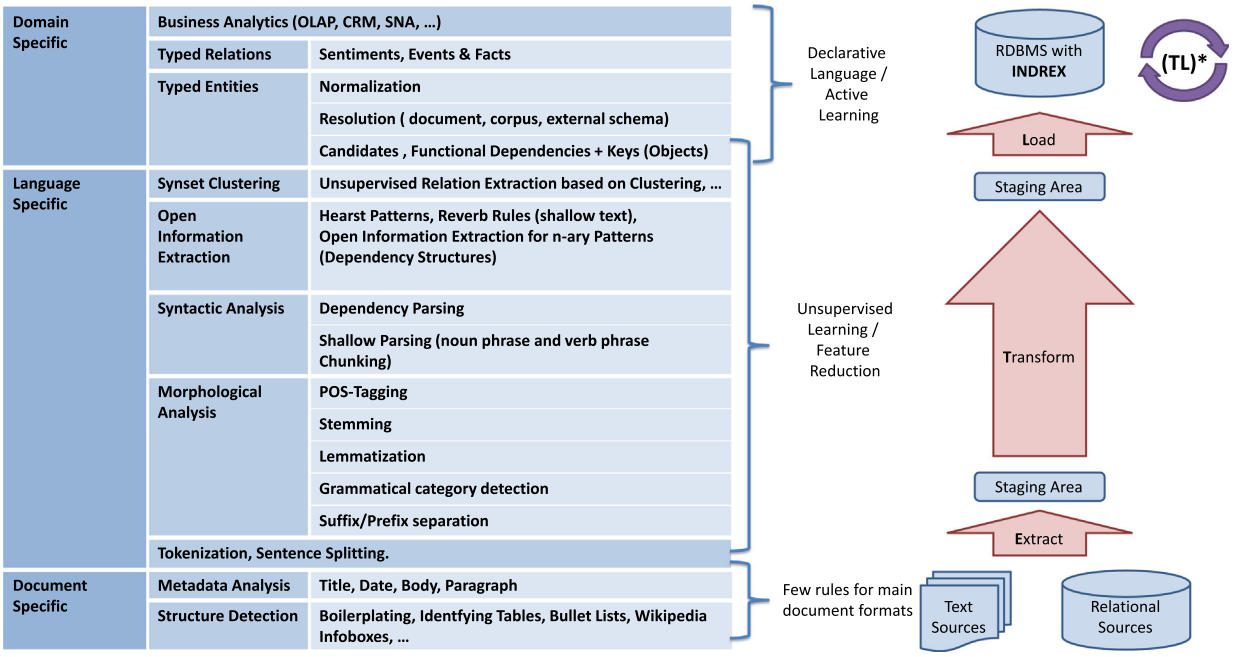

Fig. 1. Transformation steps for relation extraction. For the task of domain dependent relation extraction, we abstract document and language adaptation as a linear process and domain adaptation as an interactive and iterative process. This abstraction is true in most scenarios where non-NLP experts bring in domain knowledge on a fixed corpus. In scenarios where the goal is to improve NLP techniques, these domain dependent interactions may also trigger learning tasks on the syntactic and document specific layer; for example, authors of [41] consider human click behavior for retrieving only fact-rich documents and authors of [30] consider human click behavior for adapting a part-of-speech tagger.

transformations and an iterative process of domain specific abstraction transformations.

2.3. Declarative relation extraction

Declarative relation extraction enables domain experts to adapt existing and create new extraction rules. This way, similar to SQL and RDBMS technology, the system takes care of optimizing the declarative query. As a result, the domain expert can focus on the task of domain adaptation only. Moreover, some declarative languages have a similar syntax as SQL, therefore these languages often do not require an expensive training phase for most analysts.

Batch-based document-by-document extraction: Authors of *UIMA* [29] describe a software architecture for Unstructured Information Management. Similar to the staging area in a data warehouse, they define roles, interfaces and communications of large-grained components essential for transformation steps in natural language processing. Authors of [37] were among the first to recognize the power of declarative languages for NLP domain adoption tasks. They propose an SQL-like language called AQL that is based on the principle of a span, which is basically an occurrence of a string in a document with a single or multiple semantic meaningful labels. Humans can define AQL extractors through rules. An engine called System-T executes these rules. The AQL language provides user defined functions and predicates for comparison and combination of multiple spans. These functions and predicates enable the users to combine multiple basic extractors into a combined and complex extractor. Contrary to *INDREX*, System-T executes AQL queries per document only and in a batch mode. For joining extracted data with domain specific data the System-T materializes its output to the HDFS. Next, the user needs to define a join condition, such as in the data flow description language JAQL and the JAQL compiler reads the data from the HDFS, executes the join on top of the Apache Map/Reduce framework and persists the result in the HDFS again. Finally, the user may load the data from the HDFS into a RDBMS. This approach requires from the user to manage three systems, IBMs System-T, IBM's JAQL language and the RDBMS. Moreover, the user must wait minutes to hours until a map/reduce job is finished and the data is loaded in the RDBMS. Another standard system is GATE [18]. It provides a set of annotators, a language called JAPE that is based on cascading grammars for combining annotations and a batch-based, document oriented processing system. GATE runs on a remote 'cloud' platform or in a local java environment. Analogue to System-T, the GATE user must learn to manage different systems and needs to wait minutes to hours until a result of a query is visible.

INDREX overcomes the important limitations of batch-based implementations on top of map/reduce execution platforms by executing the domain specific information extraction task inside the database system while still leveraging the power of a declarative query language.

Optimizations for text-joins: The authors of SQOUT [34] assume existing extractors that create views where each view represents the textual content that describes tuples of a single relationship type in the text. The user can integrate these views with select–project–join queries, while the SQOUT system optimizes join processing. We consider the SQOUT system as an orthogonal optimization for a specific join scenario that might further speed up query processing for NLP tasks in a RDBMS. Authors of [23] propose another join optimization and join selection strategy, while authors of [13]

discuss optimization strategies for main memory databases and Hadoop clusters. Finally, authors of GRAFT [8] propose another set of scoring-based optimization strategies in the presence of an index.

All-in-one-system: Most similar to our work is the system proposed by the authors of [47]. Their work is based on the semi-structured data model and proposes a parse tree database where they hold dependency tagged sentences and a query language for selecting subtrees that likely indicate a relationship. For fast retrieval of relevant subtrees they use an additional key value index called *Lucene*. Hence, they still use a separate indexing system and a home-grown query parserm called PTQL, on top of the index and a standard RDBMS. In contrast, the abstraction in INDREX is based on the relational data model that allows us to leverage the full spectrum of existing RDBMS and data warehouse technology, including main memory and distributed databases with existing adaptors for data integration for a large number of text data sources. Moreover, the INDREX user does not need to learn a new query language but can continue with SQL. Finally, the INDREX user can utilize views and can use built-in technologies to grant rights for abstracting and protecting data inside the RDBMS.

3. Data model

The nature of human language allows us to use a wide vocabulary of words to express meanings. Consequently, any RDBMS that processes these words to extract meanings and relations needs to be *open for new words*, such as words from open classes like verbs, nouns, adjectives or adverbs. Moreover, the system should also permit the user (or an application) to assign new meanings to these words. Such an assignment is basically an interval query to a sequence of text. For instance, we could denote the interval of the characters from the last sentence with the type *document:sentence*. Hence, the system must represent these intervals and assigned types. Finally, the system must permit the user (or the application) to map intervals and assigned types to a relation, like the relation *PersonCareerAge(Person, Position, Age)*.

In the remainder of this section we review our data model from [35] that fulfils these requirements. Our model represents plain text as well as intervals, assigned types and relations that can be extracted from plain text, such as intervals in text that represent a relation argument or a set of arguments that may represent a relation. Moreover, the model is flexible enough to hold additional important structures for the relation extraction task, such as document structures, shallow and deep syntactic structures, and structures for resolving argument values across documents that may represent the same logical object.

3.1. Formalization

Our model uses the relational data model with three data types, namely *spans*, *annotations* and *relations*, which we explain below.

Character and segment spans: Each document, such as an email or a web page, consists of one or more strings. Each string represents textual data as a sequence of characters. We use the term *character span* to mark intervals of such a string:

Definition 1. A *character-based span* (*csp*) consists of the string ID, $\text{strID} \in S_{ID}$, together with the positions of the first character, b, and the last character, e. We denote a character-span as the 3-tuple $\text{csp} = (\text{strID}, b, e) \in \text{CSP}$, where

$$\text{CSP} = \{(\text{strID}, b, e) \,\|\, \text{strID}, b, e \in \mathbb{N} \text{ and } 0 \le b \le e < \text{length}(\text{strID})\}$$

In practice, a user-defined function may split the string into meaningful segments. One example is a tokenization function for the English language, which will spilt the string representing this sentence into tokens, such as {“One”, “example”, …}. Often, applications and users prefer to work on tokens, or other segments, instead of working on characters. Moreover, commercial vendors of text mining technology sometimes propose proprietary tokenization schemes to enforce a vendor lock-in with their customizers. INDREX permits such segment spans. We define an (optional) segment span as follows:

Definition 2. A *segment-span* (*ssp*) holds a complete and non-overlapping segmentation of the string, such as the output of a tokenization or sentence-splitting function on a character span. We denote a segment-span $\text{ssp} = (\text{segID}, b, e) \in \text{CSP}$ as a 3-tuple referring to a segment within a text with segID as the ID of the segmentation tool, b the position of first segment, e the position of the last segment and

$$\text{SSP} = \{(\text{segID}, b, e) | \text{segID}, b, e \in \mathbb{N},\ 0 \le b \le e\}$$

Depending on the information extraction task, we require either character- or segment-based spans, or both span types. Therefore, we combine character spans and optional segment spans into a span.

Definition 3. A span is a tuple $\text{sp} = (\text{csp}, \text{ssp}) \in \text{SP}$, with $\text{csp} \in \text{CSP} \wedge \text{ssp} \in \text{SSP}$.

Example “*Span*”: In Fig. 2 the character-span {26,0,6} represents the observation that in document with ID 26, keyword *Torsten* appears between character positions 0 and 6. At the same time the segmentation-span {1,0,0} represents the observation that the same string is at token position 0 for the tokenizer with the segID 1.

Annotation and meaning: An *annotation* assigns a *single meaning to a single span*. For instance, we can assign the meaning *syntax:determiner* to the span that represents the first word, ‘*The*’, in this sentence. In practice, we often require *assigning a single meaning to a set of spans*. This is helpful if these spans are distributed across the same document or are distributed across different documents. We use such annotations for representing n-ary relations, for assigning different string representations of the same logical object across documents or for assigning information from tree structures, such as dependency trees, to flat strings. The data type *annotation* permits these assignments and is defined as follows:

Definition 4. An annotation $\text{an} \in \text{AN} = \text{SP}^+ \times M$ is an n-tuple of spans with a meaning $m \in M$. SP^+ is the transfer

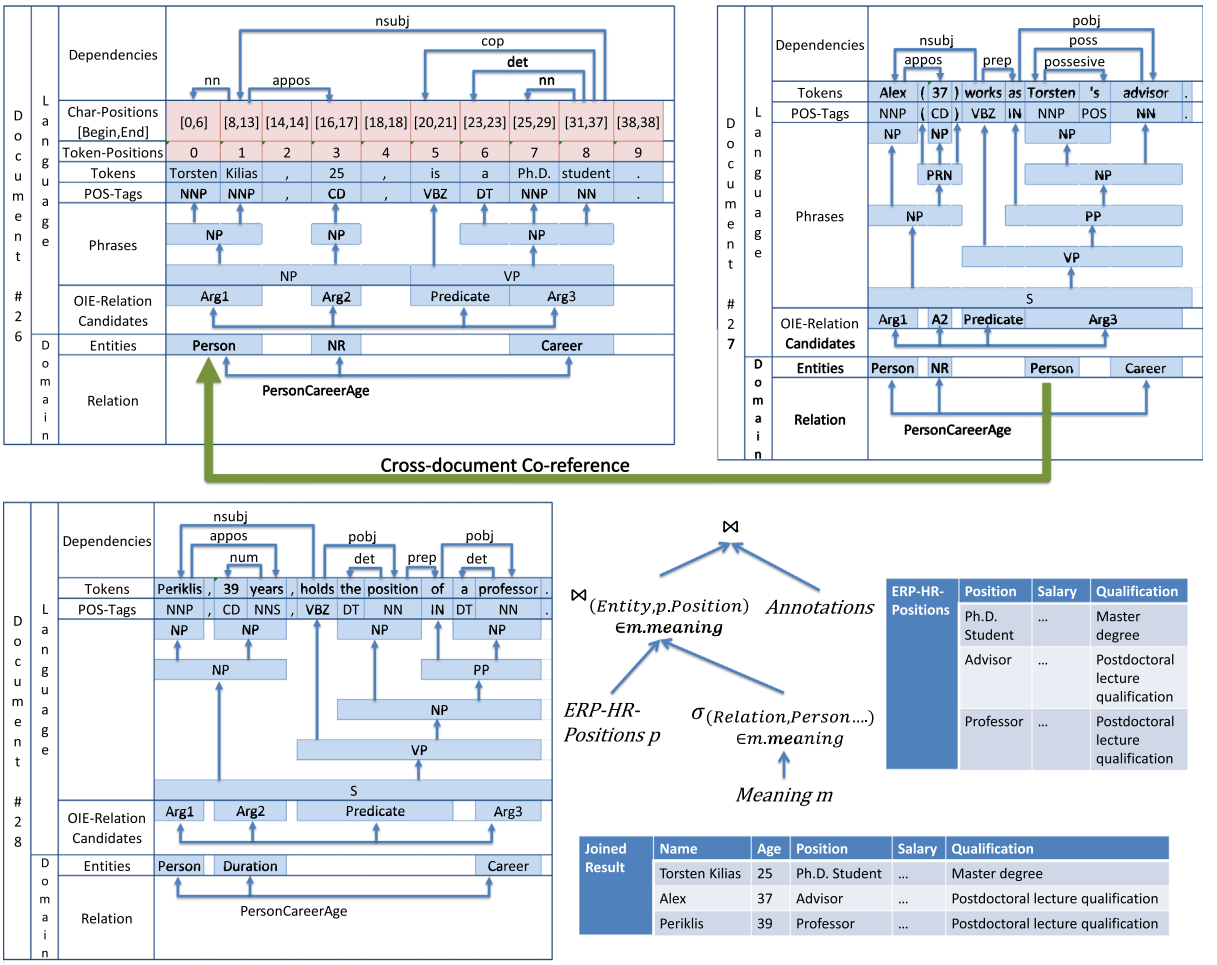

Fig. 2. Three documents with the most common language specific annotations, such as tokens, part-of-speech (POS), phrases, dependencies, constituencies (between the phrases), OIE-relationship candidates, entities and semantic relationships. Each horizontal bar represents an annotation with one span and each arrow an annotation with multiple spans. The semantic relationship `PersonCareerAge` is an annotation with three spans. The entities "Torsten Kilias" in document 26 and "Torsten" in document 27 are connected by a cross-document co-reference annotation. The query in the right-bottom corner shows a join between a Human Resource relation in a CRM system with the extracted semantic relationships from text.

of the Kleene Plus

$$\mathrm{SP}^{+} = \bigcup_{i=1}^{n} \mathrm{SP}_i = \mathrm{SP}_1 \cup \mathrm{SP}_2 \cup \mathrm{SP}_3 \cup \ldots$$

Example: "*Annotation*: *Assigning a single meaning to a single span*". In Fig. 2 the string *POS:NNP* denotes the value that a computational linguist would assign to the proper noun "Torsten". Another example is the character span of an entire Web page for which we can assign an annotation of the type *Web page*.

Example: "*Annotation*: *Assigning multiple meanings to a group of spans*'. An annotation can also consist of multiple spans. This group of spans might have different meanings. Consider in Fig. 2, the character-span {26,0,6} that is assigned to the annotation *POS:NNP*. This character-span is part of another character-span {26,0,13} which is assigned to the annotation *Phrases:NP* and both character spans are part of the annotation *Entities:Person*.

N-ary relation candidates and dependency trees: In practise, computational linguists use syntactic patterns for connecting multiple attribute value candidates of the same sentences into a relation candidate. One option for representing binary relation candidates are the so-called open information extraction patterns [24,25]. For instance, in the English language a common pattern for a binary relation is the pattern *NounPhrase follows VerbPhrase follows NounPhrase*. The basic assumption here is that in a particular sentence the first noun phrase likely expresses the subject, the second noun phrase expresses the object and the verb phrase expresses the predicate of the same sentence. In practice, English sentences follow a more complex structure. Therefore, computational linguists often use dependency trees for identifying relation candidates. These tree-based structures capture much more robust subjects, predicates, objects, adverbs and other syntax structures that likely represent arguments of a relation [2,17].

In our data model we use the following definition of a relation. This definition and the remaining definitions for the relational model and the relational algebra extend or refer to the definitions defined from Grefen and de By [32].

Definition 5. A domain A is a set of atomic values. The term atomic refers to the fact that each value in the

domain is indivisible as far as operators of the relational data model are concerned. We denote $\mathcal{A}$ as the set of all possible domains.

Definition 6. A relation schema $\mathcal{R}$ consists of a relation name and a list of attributes $(A_1, \ldots, A_n)$. Each attribute, A_i, is defined on a domain $\mathrm{dom}(A_i) \in \mathcal{A}$. The type of $\mathcal{R}$ is defined as $\mathrm{dom}(\mathcal{R}) = \mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n)$. A relation or relation instance R of relation schema $\mathcal{R}$ is a multi-set of elements in $\mathrm{dom}(\mathcal{R})$, i.e. a function $R : \mathrm{dom}(\mathcal{R}) \to \mathbb{N}$, where $\mathbb{N}$ denotes the domain of the natural numbers. The value of $R(x)$ is called the multiplicity of x in R.

Example "*Annotation*: *Relation candidates and dependencies*". In our model we represent a dependency between two edges in the dependency tree as an annotation with two spans. For example, in Fig. 2 two character spans {26,31,37} and {26,8,13} represent start and end points of the annotation *Dependency:nsubj* respectively. We can also represent n-ary relation candidates, for example in Fig. 2 the spans {26,0,13}, {26,16,17}, {26,25,37} represent potential attribute values and the span {26,20,23} represents the predicate for a 3-ary relation candidate for the relation schema *PersonPositionAge*(*Person*, *Position*, *Age*).

Relation and lineage: Efficiently mapping candidate relations manually to an existing database schema is an active area in computational linguistics. Therefore, assigning a semantic meaning to relation candidates, either manually by inspecting the relation candidate or in an automated fashion, is a core functionality in INDREX. The system permits the user or a system to attach a meaning to a span. For instance, the user can assign a meaningful type to a span that represents a candidate relation or the user can assign a meaningful type to the span that represents an attribute of this relation. For representing such meanings in our data model, INDREX implements the definition of Grefen and de By [32] for representing a relation in a RDBMS. The definition states that 'A multi-set relation is a multi-set of tuples' and represents that the defacto implementation state of a modern RDBMS. INDREX also keeps stores 'lineage information', *i.e.*, our model is able to store references to annotations and spans that the user or an application used to create the relation. For instance, a user or an application can use this information for selecting spans from 'trusted' base extractors, or for unraveling interesting combinations of extractors. The following definition follows these two requirements.

Definition 7. We denote a relation schema with a list of attributes $\{A_1, \ldots A_n\}$ and there exists $\alpha = \{\%i_1, \ldots, \%i_n\}$ such that for each $\%i \in \alpha$ holds $\mathrm{dom}(A_{\%i}) \subseteq \mathrm{SP}$ as a *span relation schema* $\mathcal{R}_{SP}[\alpha]$. In addition, we denote a relation schema with a list of attributes $\{A_1, \ldots A_n\}$ and for all $1 \le i \le n$ holds $\mathrm{dom}(A_i) \not\subseteq \mathrm{SP}$ as a *non-span relation schema*.

We now give details on loading data into base tables that represent our model.

3.2. Loading base tables

Document and language specific transformations: For more than 50 years the NLP community designs *domain independent extractors* with propriety languages. INDREX will not attempt to replace these efforts or to repackage these efforts to run inside the RDBMS. Rather, we designed INDREX to execute highly optimized document and language dependent extractors outside the RDBMS. For example, we apply language depended taggers from the Stanford CoreNLP pipeline[1] over these documents, including sentence taggers, shallow and deep syntax taggers and the Stanford 7-Class NER tagger. We extract these base annotations outside the RDBMS and load INDREX with these annotations.

Open information extraction: These document and language specific transformations are the base for Open Information Extraction frameworks, such as CLAUSIE [17] or REVERB [26]. These frameworks detect predicates, subjects and objects of a sentence. Moreover, systems such as CLAUSIE utilize syntax information for detecting binary, 3 or 4-ary relations, called clauses, between adverbs and predicates, subjects or objects. These relations are likely candidates for meaningful relations. For example, CLAUSIE will return for the sentence from Fig. 2 *Alex* (*37*) *works as Torsten's advisor.* Three clauses are (*Alex*) (*is*) (*37*), (*Alex*) (*works as*) (*Torsten's advisor*) and (*Torsten*) (*has*) (*advisor*). Note that open information extraction approaches, such as CLAUSIE, neither determine the meaning of these relations, nor detect the synonymous ones. Similarly, open information systems detect neither attribute types nor attribute value ranges.

INDREX stores sentences, subjects, predicates, objects and clauses from CLAUSIE in the model as annotations, spans and relations. Table 1 presents an example annotation after this loading phase. In the next section, we explain how an INDREX user may add domain semantics to these candidate relations or how a user might refine these candidate relations in the RDBMS.

4. Queries and functions

Our study in [35] revealed *three common query types* for extracting relations from text data: (1) *Local queries* create and refine initial candidate relations, often within the context of a single sentence. (2) *Join queries* augment text data with domain semantics, such as semantics from existing rules or semantics from existing relational data. (3) *Aggregation queries* enable the user to explore the distribution of corpus-wide semantics and permit her to refine candidate relations.

This section extends these findings significantly: first, for each query type we abstract and formalize common query patterns. Next, we list functionalities that a standard RDBMS must support. Finally, if the RDBMS does not support a functionality, we present our design for adding the functionality.

[1] nlp.stanford.edu/software/corenlp.shtml

Table 1
The table presents an example schema of our base tables after the loading phase. The table shows a string (such as a document) with ID=26. It contains the characters *Torsten* at chars 0–6 that are recognized as term *Torsten* with the lemma *Torsten*, which are a part of a named entity of the type person called *Torsten Kilias* starting from char 0 to char 13. Moreover, a dependency link, *nn*, exists for the phrase and annotates 'Torsten' and 'Kilias' as a compound noun phrase.

Schema	Annotation							
	Spans						Meaning	
	CharSpan			SegmentSpan			Key	Value
	StringID	Begin	End	SegmenterID	Begin	End		
Examples	26	0	6	1	0	0	Term	Torsten
							Lemma	Torsten
							POS	NNP
							Extractor	Stanford
	26	0	13	1	0	1	Entity.Type	Person
							Entity.Value	Torsten Kilias
							Extractor	Stanford
	26	8	13	1	1	1	Dependency	nn
	26	0	6	1	0	0	Extractor	Stanford

4.1. Local queries

Computational linguists frequently utilize regular expressions and other string functions for extracting relation arguments, values, names and types from a sentence or another document structure (see for example [25,33]). Local queries permit the user to execute these tasks. Thereby, *local* means that the query processor executes the task on a single sequence of characters, such as the entire document or a more fine granular structure, such as a sentence. The INDREX database execution engine references this sequence of characters as a string, while the computer linguist determines in her query the semantics (document, sentence, etc.) of the string.

Definition 8. We denote a relation expression E_{local} defined on the span relation schema $\mathcal{R}_{SP}[\alpha]$ as a *local query*, if the following condition holds:

$$\forall r \in E_{local} \forall \%i, \quad \%j \in \alpha: \\ \text{stringID}(r.\%i) = \text{stringID}(r.\%j)$$

White-box user defined functions permit optimizations by the query execution engine: INDREX supports local queries with predicates, scalar functions and table generating functions. We implemented these extensions as user defined functions in INDREX. Thereby, we chose the white-box model to permit the query optimizer introspecting our code and suggesting runtime optimizations.

4.1.1. Predicates

A common subtask for extracting relations is testing if a string appears in a set of strings or as a substring of another string. Most RDBMSs support this task with the IN or LIKE predicate. A user might write a query and apply the above-mentioned built-in or user defined predicates. Each predicate might return a value that is either *true* or *false*.

Definition 9. Let $\mathcal{R}$ be a relation schema. A *condition* θ on $\mathcal{R}$ is a mapping from $\text{dom}(\mathcal{R})$ to the Boolean domain $\mathcal{B} = \{\text{true}, \text{false}\}$, in short $\theta: \text{dom}(\mathcal{R}) \to \mathcal{B}$. We denote $\Theta(\mathcal{R})$ as the *set of all conditions* on $\mathcal{R}$.

The RDBMS builds from these predicates the *WHERE* clauses; see also the example query in Fig. 3. However, computational linguists require a much broader set of predicates for local queries that current extraction systems, such as AQL [15] or GATE [18], provide but that are not supported by an RDBMS. We now present detailed definitions of these predicate functions:

Detecting span proximity: We abstract span sequences in text data as intervals. For processing proximity queries on top of intervals, INDREX implements predicate functions from Allen's interval algebra [5]. These predicate functions are based on the following basic functions for retrieving the *Begin* or *End* of a character or segment span:

Definition 10. Let $\text{span} \in \text{SP}$ and $\text{span} = ((\text{str}_{ID}, \text{char}_{begin}, \text{char}_{end}), (\text{seg}_{ID}, \text{seg}_{begin}, \text{seg}_{end}))$. We define the following unary span functions:

- $\text{stringID}(\text{span}) = \text{str}_{ID}$
- $\text{charBegin}(\text{span}) = \text{char}_{Begin}$
- $\text{charEnd}(\text{span}) = \text{char}_{End}$
- $\text{segmenterID}(\text{span}) = \text{seg}_{ID}$
- $\text{segmentBegin}(\text{span}) = \text{seg}_{Begin}$
- $\text{segmentEnd}(\text{span}) = \text{seg}_{End}$
- $\text{charLength}(\text{span}) = \text{char}_{End} - \text{char}_{Begin} + 1$
- $\text{segmentLength}(\text{span}) = \text{seg}_{End} - \text{seg}_{Begin} + 1$

Given these basic functions we define functions for testing the proximity of two spans:

Definition 11. Let $\text{span}_1, \text{span}_2 \in \text{SP}$. We define the following binary span predicates:

- $\text{sameString}(\text{span}_1, \text{span}_2) \\ = \text{stringID}(\text{span}_1) = = \text{stringID}(\text{span}_2)$

- $\text{sameSegmenter}(\text{span}_1, \text{span}_2) \\ = \text{segmenterID}(\text{span}_1) \quad = = \text{segmenterID}(\text{span}_2)$

- charSpanBefore$(\text{span}_1, \text{span}_2)$
  $= \text{sameString}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charEnd}(\text{span}_1) < \text{charStart}(\text{span}_2)$

- charSpanStartsBefore$(\text{span}_1, \text{span}_2)$
  $= \text{sameString}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charStart}(\text{span}_1) < \text{charStart}(\text{span}_2)$

- charSpanEndsBefore$(\text{span}_1, \text{span}_2)$
  $= \text{sameString}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charEnd}(\text{span}_1) < \text{charEnd}(\text{span}_2)$

Definitions for segment spans are analogue.

Testing overlapping spans or span containment: Following Allens interval algebra two spans may partially overlap or may contain each other.

Definition 12. Let $\text{span}_1, \text{span}_2 \in \text{SP}$. The following predicates test if two spans overlap or contain each other:

- charSpanStartsWith$(\text{span}_1, \text{span}_2)$
  $= \text{sameString}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charStart}(\text{span}_1) = = \text{charStart}(\text{span}_2)$

- charSpanEndsWith$(\text{span}_1, \text{span}_2)$
  $= \text{sameString}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charEnd}(\text{span}_1) = = \text{charEnd}(\text{span}_2)$

- charSpanContains$(\text{span}_1, \text{span}_2)$
  $= (\text{charSpanStartsBefore}(\text{span}_1, \text{span}_2)$
  $\vee\ \text{charSpanStartsWith}(\text{span}_1, \text{span}_2))$
  $\wedge\ (\text{charSpanEndsBefore}(\text{span}_1, \text{span}_2)$
  $\vee\ \text{charSpanEndsWith}(\text{span}_1, \text{span}_2))$

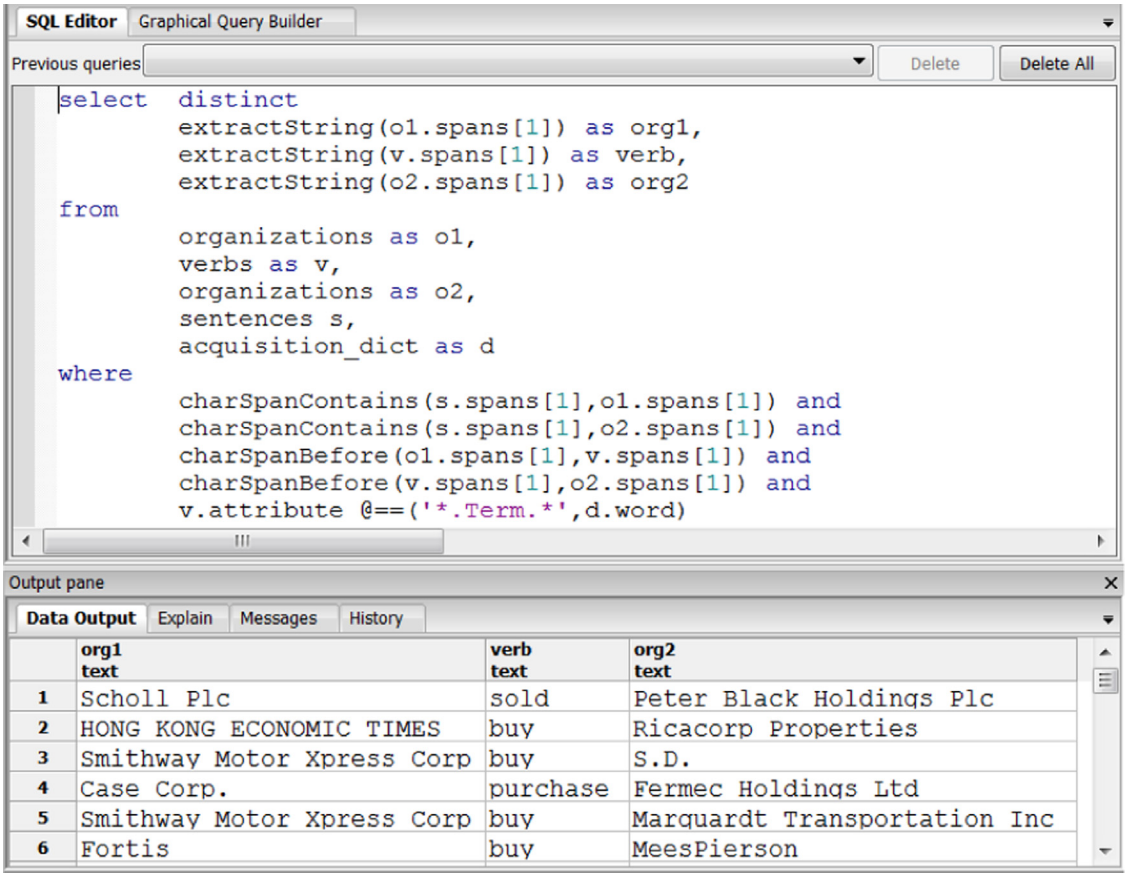

Fig. 3. This query represents Q15 from our benchmark. The query reads from tables `organizations`, `verbs` and `sentence`. We load these tables as base tables in INDREX (see Section 3). In addition, the query reads from a user defined dictionary table that contains words representing acquisitions. In its WHERE clause, the query tests if the same sentence span contains a span that represents an organization, another span that represents a verb and a third span that represents another organization. Moreover, the query tests if these spans follow each other. Finally, the query tests if the value for the span that represents the verb is also a value in the user defined dictionary. If these conditions are true, the query extracts the value of the span that represents organization 1, the verb and organization 2 and applies a DISTINCT operator. The figure shows the output of example results from a news corpora. The user might apply additional SQL statements to validate data.

- $\text{charSpanLeftOverlaps}(\text{span}_1, \text{span}_2)$
  $= \text{charSpanStartsBefore}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charSpanEndsBefore}(\text{span}_1, \text{span}_2)$
  $\wedge\ \text{charStart}(\text{span}_2) < \text{charEnd}(\text{span}_1)$

- $\text{charSpanOverlaps}(\text{span}_1, \text{span}_2)$
  $= \text{charSpanLeftOverlaps}(\text{span}_1, \text{span}_2)$
  $\vee\ \text{charSpanLeftOverlaps}(\text{span}_2, \text{span}_1)$

Definitions for segment spans are analogue.

Example: span predicates. Consider again the example query in Fig. 3. The query uses predicate function charSpanContains for testing if a span for an organization *o*1.*spans*[1] is contained in a span of a sentence *s*.*span*[1].

Local join: Local queries with two or more predicates require from the execution engine an additional test that the spans, that serve as input parameter to the predicate, are contained in the same character sequence. Such a character sequence might represent an entire Web page or a part of the Web page, such as a single sentence.

Definition 13. Let $\mathcal{R}_{SP}[\alpha]$ be a span relation schema and $\%i,\ \%j \in \alpha$. A *local condition* $\xi \in \Theta(\mathcal{R})$, such that

$$\forall r \in \text{dom}(\mathcal{R}):\quad \text{stringID}(r.\%i) \neq \text{stringID}(r.\%j) \Rightarrow \xi(r) = \text{false}.$$

We denote $\Xi(\mathcal{R})$ as *the set of all local conditions* on $\mathcal{R}$.

A special case of such a local query is the local join. This complex function tests if a span contains two other spans.

Definition 14. Let E_1 and E_2 be local relational expressions; E_1 defined on a span relation schema $\mathcal{E}_1$ and E_2 defined on a span relation schema $\mathcal{E}_2$. We denote a join $E_1 \bowtie_\xi E_2$ with $\xi \in \Xi(\mathcal{E}_1 \oplus \mathcal{E}_2)$ as a *local join*.

*Example: local join:*In Fig. 3 the query executes a local join that selects spans representing companies, spans representing headquarter locations and tests, if these two spans appear inside the same span that must represent a sentence. In addition, the query executes additional predicates to the local join to test if the two spans appear after each other.

4.1.2. Scalar functions

The user will apply a scalar function in INDREX in the SELECT clause of the query or as the input to a predicate. These functions return a value of non-Boolean types, such as span, annotation or relation. INDREX supports two common types of scalar functions:

Returning a character sequence from a span: Consider again the SELECT list in Fig. 3. In the query, the scalar function extractString takes a span as input and returns the substring for the span from the corresponding string.

Returning context for a span: Other commonly used scalar functions are leftSpanContext, rightSpanContext, betweenSpans and combineSpans. These functions create spans right or left of another span or the span between two other spans or the span that contains two other spans. These functions are common among text mining frameworks but not included in a RDBMS. Fig. 5 uses the function combineSpans to combine the span of an organization with span of the following comma.

Definition 15. CombineSpan is a scalar function on spans that creates a span that contains the input spans.

- $\text{combineCharSpans}(\text{span}_1, \text{span}_2)$
  $= \textit{if}\ \text{sameString}(\text{span}_1, \text{span}_2)\ \textit{then}$
  $\text{CharSpan}(\text{stringID}(\text{span}_1),$
  $\min(\text{charBegin}(\text{span}_1),$
  $\text{charBegin}(\text{span}_2)),$
  $\max(\text{charEnd}(\text{span}_1),$
  $\text{charEnd}(\text{span}_2)))$
  $\textit{else null}$

- $\text{combineSegmentSpans}(\text{span}_1, \text{span}_2)$
  $= \textit{if}\ \text{sameSegmenter}(\text{span}_1, \text{span}_2)\ \textit{then}$
  $\text{SegmentSpan}(\text{segmenterID}(\text{span}_1),$
  $\min(\text{segmentBegin}(\text{span}_1),$
  $\text{segmentBegin}(\text{span}_2)),$
  $\max(\text{segmentEnd}(\text{span}_1),$
  $\text{segmentEnd}(\text{span}_2)))$
  $\textit{else null}$

- $\text{combineSpans}(\text{span}_1, \text{span}_2) =$
  $\textit{if}\ \text{sameString}(\text{span}_1, \text{span}_2)\ \textit{then}$
  $\text{Span}(\text{combineCharSpans}(\text{span}_1, \text{span}_2),$
  $\text{combineSegmentSpans}(\text{span}_1, \text{span}_2))$
  $\textit{else null}$

4.1.3. User-defined table generating functions

Consolidation: Predicate and scalar functions already permit computational linguists creating extractors. These users may desire to merge potentially overlapping spans from these extractors and output a single span or a list of best matching spans. We call this operation *consolidation*. It requires from the query processor to group the set of input spans by some criteria, for example by the span ID of sentences or another structure. Next, for each group the function validates user defined criteria, such as span proximity or overlapping spans. The following definition formalizes this function.

Definition 16. Let E be a relational expression defined on $\mathcal{E}$. We denote $\Omega_{g,f}E$ a consolidate expression with the grouping function g and the aggregation function f:

$$\Omega_{g,f}E = \{(x,1) | x \in G\} \cup \{(x,0) | x \in \mathcal{F} \wedge x \notin G\}$$

where

$$G = \{f(E') \in \mathcal{F} \mid \exists E' \subseteq E : g(E', E)\}$$

A special case is the ContainedWithin function. It tests if a set of spans is contained within another span, applies on the group an aggregation function, such as ‘left longest spans wins’, and outputs the resulting span.

Definition 17. Let E a relational expression defined on a $\mathcal{R}_{SP}[\alpha]$ with $\%i \in \alpha$. ContainedWithin is a consolidate expression $\Omega_{g,f}E$ with the grouping function:

$$g(E',E) = |E'| == 1 \wedge e' \in E' \wedge \forall e \in E: \\ \text{charSpanContains}(e'.\%i, e.\%i) \Rightarrow e'.\%i = e.\%i$$

and the aggregation function $f(E') = E'$

Block: Users may combine multiple potentially non-overlapping nearby spans into a new span. We call this operation *block*.

Definition 18. Let E a relational expression defined on a $\mathcal{R}_{SP}[\alpha]$ with $\%i \in \alpha$ and $E' \subseteq E$. Block is a consolidate expression $\Omega_{g,f}E$ with the grouping function:

$$g(E',E) = \forall E'' \subseteq E: E' \subseteq E'' \wedge \text{isBlock}(E') \wedge \text{isBlock}(E'') \\ \Rightarrow E' = E''$$

where

$$\text{isBlock}(\{e_1, \ldots, e_n\}) \\ = \text{minPts} \leq n \leq \text{maxPts} \\ \wedge \forall 1 \leq i < j \leq n: (\text{charSpanBefore}(e_i.\%i, e_j.\%i) \\ \wedge \text{minDist} \leq \text{charSpanDistance}(e_i.\%i, e_j.\%i) \\ \text{charSpanDistance}(e_i.\%i, e_j.\%i) \leq \mathit{maxDist})$$

and the aggregation function

$$f(E') = \text{combineSpans}/(E')$$

Example: BLOCK function. Fig. 5 shows an example. The query takes as input sentences that include conjunctions of companies. The query transforms these conjunctions into lists of companies, one for each sentence. Another example, not shown here, is a query that test if the company identification number, the company name and the CEO may appear in the imprint of a page.

White-box user defined table generating function: We implemented these functions in INDREX based on the concept of user defined table generating functions (UDT). A UDT takes as input multiple spans and may outputs multiple spans (and thus may generate a table). Analogue to standard RDBMS, INDREX expects UDTs inside the *FROM* clause.

4.2. Joining text data with domain semantics

Common approaches for assigning domain semantics to text data are built-in joins, and special join cases such as dictionary lookups and regular expressions. In this subsection, we present their design in INDREX.

4.2.1. Built-in equi- or theta-joins

INDREX represents entities in textual data with the span relation schema (see Section 3). Therefore INDREX can leverage built-in (and often highly optimized) *joins* of the RDBMS for matching equal or similar string representations of entities in text data with entity representations from a domain specific relation schema. The task of the user is to formalize one or multiple join conditions between the two entity representations. The following definition formalizes this join.

Definition 19. Let E_1 and E_2 be relational expressions; E_1 defined on a span relation schema $\mathcal{E}_1$ and E_2 defined on a non-span relation schema $\mathcal{E}_2$. We denote a join $E_1 \bowtie_\theta E_2$ with $\theta \in \Theta(\mathcal{E}_1 \oplus \mathcal{E}_2)$ a *join between the two entity representations.*

Example: Complement existing structured data with information from text. Fig. 2 shows a table `ERP-HR-Position` that has a column `position`. For complementing this information, the user selects from the span relation schema extracted spans for the relationship type *Person-Career(Person, Career)*. Next, the user joins these spans with the condition *ERP.position = PersonCareer.Career* and retrieves a table that contains persons mentioned in text, their age, salary and qualifications.

Join results between a span relation schema and a domain specific relation schema may contain potentially false or incomplete join candidate results, caused by homonymous or synonymous words or by word entailment. The task of the user is to identify additional join conditions for minimizing false positives and for maximizing result completeness. An interesting research direction is supporting the user in these tasks [21,31].

4.2.2. Joins with external dictionaries

Standard text mining frameworks, such as GATE [18], provide dictionaries for assigning a meaning to words. Dictionaries are sets of multi-words that represent domain semantics. Common examples are pre-built Gazetteers, such as Domestic and Antarctic Names of US places[2] or job titles,[3] among many other resources. The text mining systems load the content from a file in main memory and match character sequences from text data to multi-words from the dictionary. If such a match is successful the system assigns the matched interval a particular meaning.

INDREX provides this functionality in the RDBMS. A user defined function expects the dictionary data already loaded in a table. INDREX supports matches between the span relational table and the dictionary table as equi-join, denoted as $@==$ or as a theta-join with an approximate string matching, denoted as $@\sim\ \sim$.

Example: *Disambiguating relationship names.* Consider a computational linguists applying a synset dictionaries [3,43,4] for resolving words representing acquisition.[4] Fig. 3 shows a join between these synsets and the predicate of a sentence to test for an acquisition event.

4.2.3. Regular expressions

Another common technique of computational linguists is the reuse of predefined pattern dictionaries for regular expressions. For instance, the system REVERB uses few regular expressions for identifying strings that likely represent the name of a relationship, such as V = verb *particle*?adv?, W = (noun|adj|adv|pron|det) or P = (prep|p *article*|inf. marker).

[2] http://geonames.usgs.gov/domestic/download_data.htm

[3] https://www.freebase.com/business/job_title?instances=

[4] The dictionary may contain words, such as `acquired by, acquisition by, acquisition of, acquired, purchased, bought, takeover of, agreed to buy, to acquire, sold, sold to, acquired in, to sell, acquisition in, acquires stake in, purchased by, bid for, bought by, sale of, buys interest in, bought out, completed took over, corporation in, merger with, purchase of, incorporated from, bought from, announced to purchase.`

Computational linguists combine these patterns into a single regular expression, such as $V|VP|VW*P$.

INDREX provides the user-defined-function called regex_lookup(span,regex_name) that takes as input the name of a regular expression and a span, executes the regular expression on the span and returns matches as sets of new annotations. This functionality enables reusing existing regular expressions and adapting existing regular expressions to specific domains.

4.3. Aggregations

INDREX permits computational linguists executing aggregations directly in the same system that permits the extraction from text. These users may learn distributions of words, syntax, semantics or other features. They may formalize these distributions into predicates and scalar functions and thereby increase the recall or precision of their existing extractors. INDREX provides built-in aggregation operators, such as MIN(), MAX(), SUM() or AVG() or the combination of grouping and aggregation called generalized projection. Too, INDREX supports built-in other user defined aggregates of the underlying RDBMS, such as OLAP extensions [48] like *CUBE* or *WINDOW* operators.

Definition 20. Let E_1 be basic relational expressions; E_1 defined on schema $\mathcal{E}_1$ with the attribute list $\{A_1,\ldots,A_n\}$, such that there exists $1 \le j \le n$ with $\mathrm{dom}(A_j) \subseteq S_{ID}$ or $\mathrm{dom}(A_j) \subseteq \mathrm{SP}$. We denote $\Gamma_{\alpha,f,\beta}E_1$ as a local aggregation expression, if $\alpha = \{\%i_1,\ldots,\%i_m\}$ and there exists $1 \le k \le m$, such that $\%i_k = j$.

Example: *Person-Age Extractor*: Consider the sentences in Fig. 2. A user can learn from the sentence the distribution of words between a person and a position in a sentence, for example, *holds the position of* (1 *times*), *works as* (1 *times*), and *is a* (1 *times*). This distribution shows likely synonym expressions for the relationship type *person-position*. Analogue, the user can unreveal that the attribute *age of a person* is expressed as an apposition of the argument of the semantic type *person* by commas, brackets and the keyword *years*. Finally, Fig. 4, which also applies span and consolidation operators, shows a distribution of age information for persons in the text.

5. Evaluation

Computationally intensive applications that support both text mining and analytical workflows often leverage analytical platforms based on the MapReduce framework. In this section, we describe INDREX on a MapReduce implementation and on a Impala implementation, our data setup and our evaluation methodology. Finally, we report on our experimental results.

5.1. Setup overview

Relation extraction is a computationally expensive task. Ideally, the task is executed on massively parallel execution system with multiple nodes and multiple cores on each node. The distributed storage system must support efficient access to relational data, to text data and to data represented by our span-based data model from Section 3. Finally, the system must support local and global joins, local queries and aggregations (see Section 4).

Table 2 compares these characteristics for systems GATE, SystemT, IMPALA and PTQL (see also Section 2 for details about these systems). PTQL, a commercial system, uses a centralized broker architecture. It stores sentences in a IR-engine and retrieves promising sentences matching lexical predicates for a query. A broker node 'joins' these sentences with domain data from a RDBMS. Authors show results on 13.000 medical abstracts. Because of the central broker design, PTQL does suffer from similar problems of broker based distributed database systems. Contrary to PTQL and analogue to INDREX, SystemT and GATE support local queries on text data. These systems can be executed as user defined functions in MapReduce environments that support massive parallel query execution of aggregations and joins.

We decided to re-implement the functionality of INDREX in a Hadoop-based MapReduce environment. Fig. 6 overviews our experimental setup. In the initial ETL (steps 1–3) we read textual data and perform extraction operations that return generic relations and an annotated corpus. Next, we load this data into the highly optimized distributed Parquet.io filesystem. We consider this implementation as baseline (steps 4.1, 5.1 and 6). Very recently, Cloudera proposed the *Impala* query processing engine for analytical query workloads in a data warehouse setting which is our second test system (steps 4.2, 5.2 and 6).

Both systems, Hadoop and Impala, are distributed data processing systems, but differ in implementation languages, query execution techniques and built-in optimizations. INDREX is designed to leverage and benefit from these features and optimizations. An interesting question is which built-in optimizations support most iterative relation extraction query workflows. We now describe our setup for analyzing this question.

5.2. Preparing and loading data

Data set: We chose the Reuters Corpus Volume 1 (RCV1) [39], which is a standard evaluation corpus in the information retrieval community. It contains information about news from 1997 and shows characteristic distributions for textual data.

We briefly summarize the main characteristics of this corpus from our previous work [11]: we inspected the distribution of 39 entity types and 70 relationship types across the documents in the corpus. We used a commercial extractor[5] and extracted roughly 860,000 entities and 1,800,000 relations. We observed a power law distribution: more than 65 relationship types are rare and only appear in a few hundred documents while only relations of the types *PersonCareer*, *PersonCommunication*, *Acquisitions* and *CompanyAffiliates* are distributed across 10,000 documents or more. Querying such typical distributions for text data requires optimizations for selective queries, such as index structures, parallel query execution or data partitioning schemes.

[5] opencalais.com

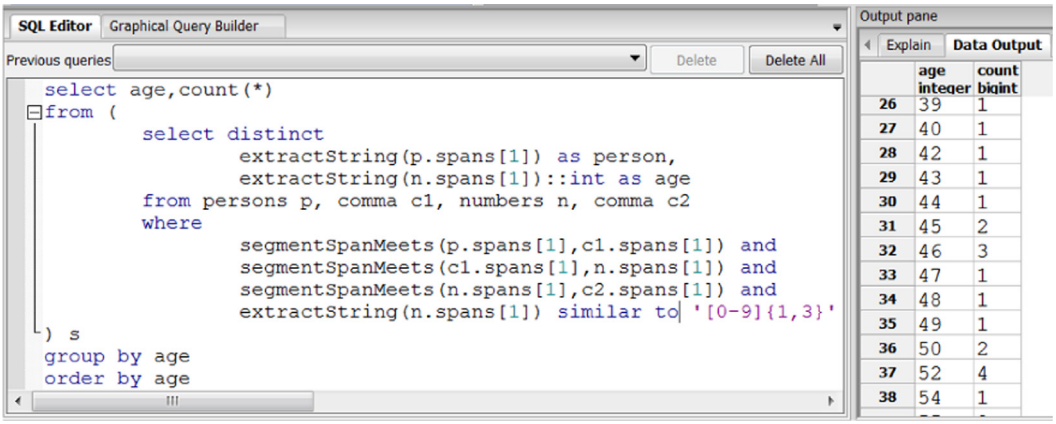

Fig. 4. The query shows a person-age extractor. In addition, the query provides a distribution for age information in our corpus. The outer SELECT statement computes the aggregation for each age observation, grouped and ordered by age. The nested SELECT statement projects two strings: the `person` and the `age` from conditions in the WHERE clause that implement the heuristic of a syntactic apposition, in this case represented by the pattern ⟨*person*⟩⟨*comma*⟩⟨*number*⟩⟨*comma*⟩. For this corpus, we observe that seven persons are either 32 or 52 years old, six persons are either 37, 38 or 46 years old and eight persons share an age between 29 and 54.

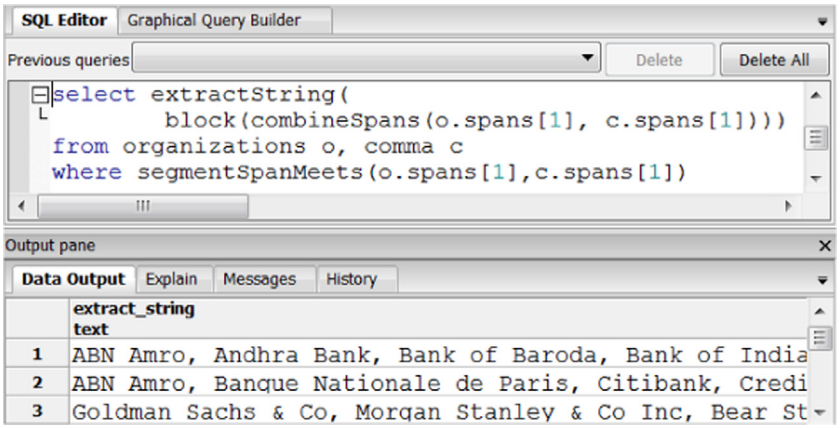

Fig. 5. This query represents Q17 from the benchmark and extracts conjunctions of organizations. The figure shows the first three conjunctions.

Base extractors and tables: In this work, we used the Stanford CoreNLP pipeline as base extractor. This complex extraction pipeline emulates language specific extractors from Fig. 1, including sentence recognition, tokenization, part-of-speech tagging, lemmatization, 7-class named-entity tagging, dependency tagging and co-reference resolution. Moreover, we used the Open-Information Extraction system CLAUSIE [17] that takes these annotations as input and outputs n-ary candidate arguments and candidate relations per sentence, but without attaching a meaningful relationship type (see also Section 3.2). Commercial relation extractors, such as the Open Calais extractor from above, use similar base extractors.

ETL on 800K annotated documents. Overall, the raw corpus has a size of ca. 2.5 GB. After annotating the corpus with the Stanford CoreNLP pipeline and CLAUSIE, the annotated corpus achieved a size of ca. 107 GB. Our relation extraction stack (see also Fig. 2) created more than 2500 annotations per document on average or roughly 2 billion annotations for our 800K documents. Overall, we could observe that linguistic base annotations increase the raw data by more than an order of magnitude.

We store this data on the Parquet.io file format [1]. This columnar file format stores data files in the Hadoop Distributed File System (HDFS) as its primary data storage medium and benefits from the built-in redundancy of the HDFS. Parquet allows compression schemes to be specified on a per-column level and uses various compression formats for files. For our scenario we observe a compression ratio of a nearly 10x factor: The data size in the Parquet.io file format is reduced from 107 GB to roughly 10 GB.

Overall, for base NLP transformations, compressing/encoding and loading data into Parquet.io, the ETL took on average 6 s per document or overall 7 h for the complete corpus of 800,000 documents on our cluster with 200 AMD cores with 2.4 GHz and 8 GB RAM per core.

5.3. Evaluated systems

Hadoop and Impala read annotations and text data from the Parquet.io storage layer (steps 3 and 4), execute queries and forward results to the user. The user might refine the queries or may adds additional views and rerun queries (iteration in step 6 to step 4 and 5).

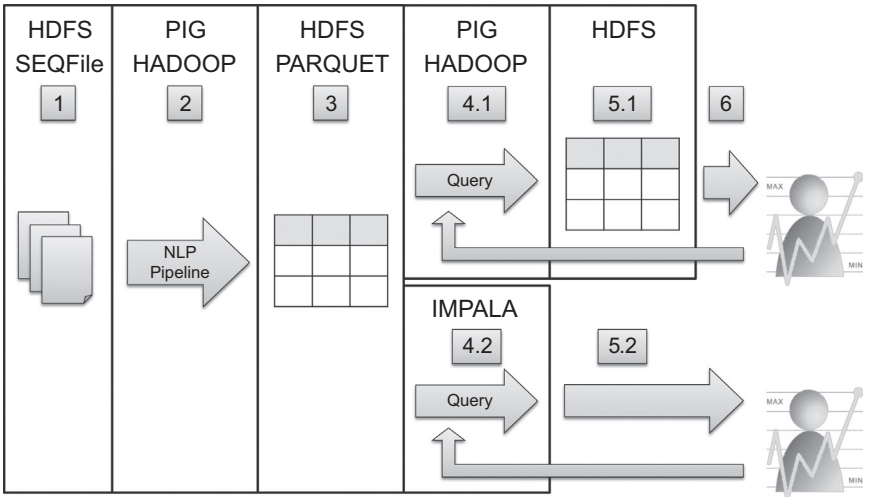

Fig. 6. This figure shows the initial ETL (steps 1–3) and two implementations that create iterative query workflows on loaded data. Both setups read their input from HDFS sequence files (step 1) and the same base linguistic operations (step 2). The result of preprocessing is written in highly compressed HDFS Parquet files (step 3). Our baseline creates iterative query workflows with MapReduce on Hadoop (step 4.1) and writes results into HDFS sequence files (step 5.1). The user analyses the result from the HDFS (step 6) and may refine the query. Our second system creates iterative query workflows in Impala (Step 4.2). The system supports ad-hoc queries and optimizes expensive disc accesses. The user inspects results (step 5.2) and may refine the query again. IMPALA uses caching and other techniques for optimizing such iterative query workflows (see also Section 5.3).

INDREX on Hadoop + Pig. We run Hadoop 2.3.0 and Pig 0.12 on a cluster with nine nodes. Each node consist of 24 AMD cores, each with 2.4 GHz, 256 GB RAM and 24 disks. Overall we had 9×24 cores available. The cluster runs the Ubuntu 12.0 operating system on all nodes as well as Cloudera's CDH5.1. Hadoop can leverage multiple cores, such as for processing local queries per core or for distributing the workload for global queries among multiple cores. We assigned Hadoop up to 200 cores for Map and 100 cores for Reduce tasks. Each task consumes up to 8 GB RAM. Reduce tasks start if 90 percent of the map tasks are complete.

For global queries, our INDREX implementation on Hadoop uses built-in statements and optimizations from the data flow description language Pig [44]. Pig was originally developed by Yahoo for analytical workflows and is now part of many MapReduce distributions and is made up of two components: The first is the language itself, which is called PigLatin. The language permits users to create data flows (similar to query execution plans in a RDBMS), including joins, group-by statements, projections, selections etc. However, PigLatin misses the text mining functions from Section 4. We implemented the INDREX functionality as user-defined functions that a PigLatin data flow can call. The second component is the runtime environment that parses, optimizes and compiles these data flows into sequences of MapReduce statements. For example, the language uses combiners for global aggregations and provides distributive and algebraic aggregation functions to reduce the size of intermediate results.

INDREX on Impala: We tested Cloudera's Impala version 1.2.3 on the same machine setup and can leverage up to 200 multiple cores too. Impala uses a streaming based query processing approach and retrieves main memory only on demand, such as for building in-memory hash tables to store intermediate results from an aggregation query. Profiling Impala revealed that the system did not retrieve more than 10 GB during query execution. The Impala query processor leverages various built-in optimizations, such as parallel scans, parallel pre-aggregations, rule-based or statistical join order optimizations, join distribution optimization techniques (broadcast vs. partitioning join) as well as code optimizations at query compile time with LLVM[6]. For example, Impala supports equi-joins with a hash-join implementation and theta joins with a cross-join implementation. We implemented our INDREX extensions as so-called macros, which are basically white box UDF implementations.

5.4. Methodology

Overview: We conduct experiments on the feasibility to show (1) if the built-in functionalities of the system plus our white box extensions permit the user to execute common query scenarios, and, (2) for which query type our INDREX implementation might leverage built-in optimizations from the system underneath.

27 queries simulate common tasks during the iterative relation extraction process: Our queries include typical operations, such as point and range selections at various attribute selectivities, local joins within the same sentence or document, joins with external domain data, queries using UDTs and UDAs as well as global aggregation queries across individual documents. A detailed list of all 27 queries is given in the Appendix.

Measurements. We ran each query ten times on each system, measured the execution time in milliseconds and selected the largest measurement (slowest query execution) per query. The next subsection reports on our results.

[6] llvm.org

5.5. Quantitative observations

Both systems support a wide variety of queries: We could formalize the required UDFs, UDAs and UDTs to support all of our queries in PigLatin. Impala cannot yet execute two of the queries that require UDTs, namely Q16 and Q17. These queries consolidate spans and require support for user-defined table generating functions (UDT).

Impala outperforms Hadoop/Pig by nearly two orders of magnitude for text mining query workflows: Fig. 7 shows query execution times for both systems over all queries. We observe a similar runtime for PigLatin across all queries. Each query in Pig reads tuples from disc, where here are tuples from the HDFS/Parquet file. Next, Pig always executes a full table scan over these tuples and, by design, always stores query results back to the distributed file system. Contrarily, for Impala we observe nearly two orders of magnitude faster query execution times since Impala conducts various optimizations that also benefit our iterative text mining workflows.

For understanding this impressive performance, we report details for each individual operation. Fig. 8 compares the aggregated runtime for individual operations, such as complex and local queries (including local joins), joins with external data, aggregations and selections.

Fig. 9 shows query execution runtime per document for each of our benchmark queries. We group queries by type, such as queries that just execute selection predicates, queries with aggregations, queries joining data from the same sentence or document and queries joining text with external resources. We will now give details for each operator category.

5.5.1. Selections

Low selectivity queries: Queries *Q2*, *Q3* and *Q4* select specific annotations. These queries are often the base for more advanced queries, such as joins or group-bys. Both systems, Pig and Impala, do not have a built-in index and fall back to a parallelized full table scan for selection queries. Impala uses a highly optimized C++ implementation for the scan of the Parquet.io files. Pig, instead, uses a slower Java implementation. Furthermore, while reading each tuple from the Parquet.io storage, Pig creates a tuple, which is a tree of Java objects. In our implementation Pig reads for each query the entire document and all of its annotations into the java virtual machine memory, analogue to systems like System-T or GATE. A document contains on average 2500 annotations, such that Pig creates a huge set of objects, a well-known problem of Java. For example, the slow-down of the garbage collector for many objects is such a performance issue. Fig. 6 shows that Pig needs an additional step, as it keeps the results into the HDFS, before the user could inspect it. The replication of the HDFS adds an additional overhead to the query runtime.

High selectivity queries: Query *Q1* returns a very large result set. Although Impala can leverage its parallel running pipelines, it needs to ship results from parallel pipelines to a single query client. Hence, the time this client needs for printing/storing the result becomes the bottleneck here.

Data intensive point or lookup queries: Query *Q5* retrieves all annotations for a particular sentence. In our corpus, we observe that the 'average sentence' contains between 40 and 100 tokens. For each token we assign the lexical information, the lemma and the part of speech annotation as well as dependency structures between tokens. Open information extraction approaches, such as CLAUSIE, often return tens of candidate relations, each with multiple arguments, for such a sentence. Because of the nature of open information extraction, many of these candidate relations may overlap or can be wrong. Therefore the number of annotations for an average English sentence may likely exceed 100 or more annotations.

For this type of point queries (also called lookup queries) Impala executes a scan over span tables that store lexical, syntactic, deep syntactic and relation candidate annotations for locating the specific stringID. *Q5* also enforces Impala to combine values from all 134 columns of our span table into a tuple. This operation is expensive because Impala must construct the tuple from separately stored columns.

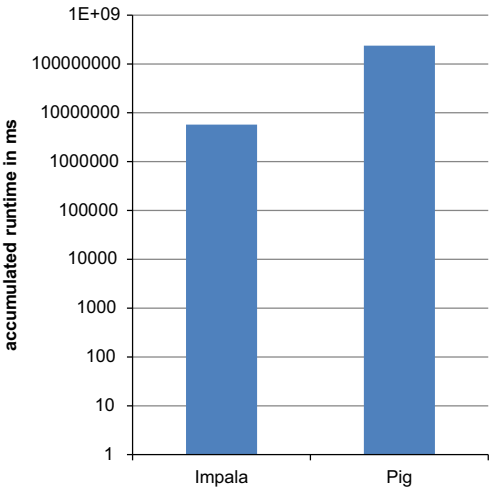

Fig. 7. Comparison of the total runtime for all benchmark queries between a Hadoop+Pig system and an Impala system on 800K documents. We observe that Impala executes queries nearly two orders of magnitude faster than Hadoop+Pig.

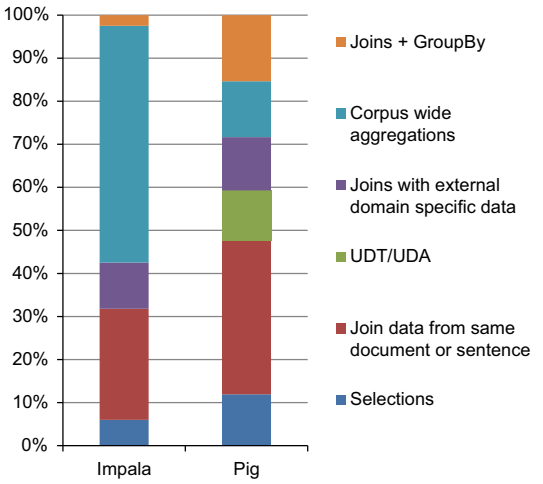

Fig. 8. Comparison of runtime percentages for different categories of operations for a Hadoop+Pig system and an Impala System on 800K documents.

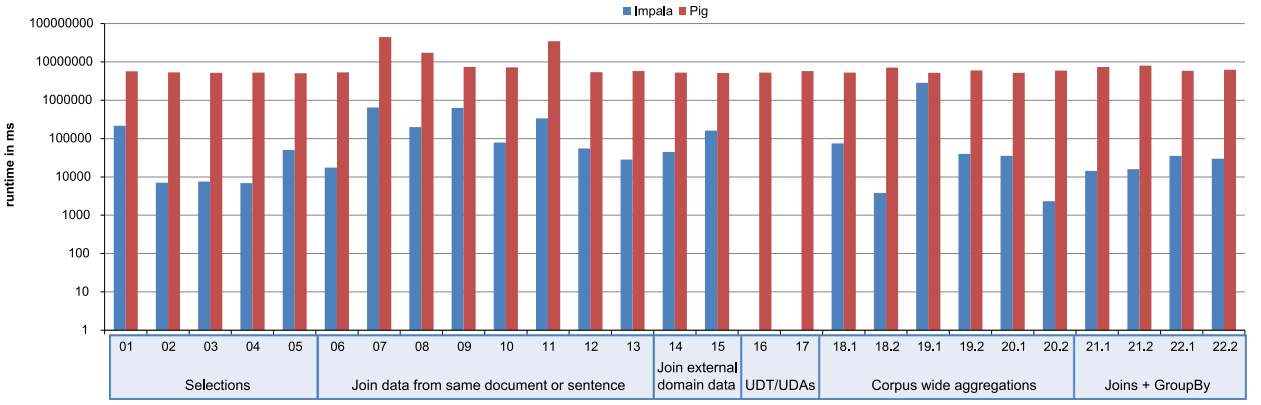

Fig. 9. This figure compares query execution time for each query in our benchmark. Impala does not provide UDTs (Q16 and Q17).

5.5.2. Joins

Local joins: At first, Impala selects the required annotations for the local join and redistributes them by a hash key to the join nodes. Afterwards, it evaluates the non-equi-join predicates for each bucket. Currently, Impala only supports one join thread per cluster node, which, for our cluster, creates 9 parallel join threads (since we are running on 9 cluster nodes). However, this redistribution brings some benefits to the query runtime: during the evaluation of the non-equi join predicates, Impala only needs to iterate over the tuples in the current bucket. The disadvantage of the redistribution, on the other hand, is a possible data transfer through network to another cluster node.

In contrast, Pig does not need a redistribution of the data, because it has already read the entire document. The disadvantage of our Pig Implementation is that it only supports simple nested loop joins. The nested loop joins compare, for each tuple of the right side, the complete left side. Currently, it does not support materialization for the right side of the join, such that it must iterate over all annotations and evaluate the selection.

Joins with external domain specific data: Queries Q14 and Q15 in our query set join external data with text data. Impala executes this join fully parallel as a broadcast join since the table that holds the user-defined dictionary is relatively small.

Pig reads the dictionary at the beginning of the Pig script into the memory and adds the containing tuples into a Bag. Next, it passes the Bag to each Map-Task that executes the UDF for the local part of the query. This enables Pig to evaluate the join between the annotations and the content of the dictionary for multiple documents in parallel.

5.5.3. Aggregations

Queries, Q18.1, Q18.2, Q19.1, Q19.2, Q20.1, Q20.2, Q21.1, Q21.2, Q22.1 and Q22.2, benefit from parallel pre-aggregations in Impala. Moreover, we observe that corpus-wide aggregations, such as Q18.2, Q19.2 or Q20.2, are executed faster than per-document or per-string aggregations, such as Q18.1, Q19.1 or Q20.1, in Impala.

A closer inspection revealed that again the size of the result set for per-document or per-string aggregations is often drastically larger than the size for a result set of a corpus-wide aggregation. Shipping and printing the result set to the client consumes most of the execution time in the Impala system.

Pig executes the per-document aggregations in the UDF for the local part of the query. We use a hash-based Group By implementation with accumulator-based aggregations. The global aggregations in Pig use the Reduce phase of the Map-Reduce framework. The Map phase executes for distributive aggregation functions, such as SUM, COUNT, MAX and MIN, the same UDF as for the per-document aggregation. Then, Pig applies a combiner to the already pre-aggregated documents. The combiner aggregates this pre-aggregated documents once more and reduces the size of the intermediate results. The framework then redistributes these intermediate results to the Reduce-Tasks, which apply the final aggregation. Additionally, the redistribution of the intermediate results begins when 90 percent of the Map-Task are complete. This reduces the time that the Reduce-Task adds on the total runtime of the query. Because of the reduction of the intermediate results and the early start of the Reduce-Tasks, the queries with global aggregations are only slightly slower than the queries with only per-document aggregations.

5.6. Qualitative results

After the presentation of results in the previous subsections, we turn to discuss INDREX's qualitative assessment.

Relation extraction and analytics in a single system: First and foremost we can see by now that our solution empowers users to query textual and non-textual data under the same framework due to the span model we presented. Our intuitive model allows querying on individual characters, tokens, sequences of tokens or characters, dependency tree structures and relational data. For example, a user (1) can seek sequences from a tree-based structure, such as the subject of a sentence, and (2) can match these sequences to shallow token sequences, such as named entity phrases. Additionally INDREX is able to load extracted annotation into a small number of base tables.

Declarative query processing: A recent study in [16] surveyed the landscape of information extraction (IE) technologies and identified that rule-based IE dominates the commercial world. The commercial world greatly values rule-based IE for its interpretability, which makes IE programs easier to adopt, understand, debug, and maintain in

the face of changing requirements. The user can express these rules in INDREX with declarative queries. One example is a wide range of local joins for executing proximity queries on the same sentence and other alternative document structures. Moreover, INDREX allows one to easily incorporate domain knowledge, which is essential for targeting specific business problems. One example is joins for integrating text data with existing relational data from the same system. INDREX also supports joins on external resources, such as dictionaries, joins using a regular expression, joins using built-in predicates, such as LIKE, or joins using a UDF as predicate function.

INDREX returns results within 10 s of seconds: In a business setting, the most significant costs of using information extraction are (a) the labor cost of developing or adapting extractors for a particular business problem, and (b) the throughput required by the system. INDREX executes declarative queries with fast built-in optimizations. Our extensive evaluations show that INDREX can return results for most queries in 10 s of seconds in a columnar database, such as IMPALA. Only a few queries run for more than a few minutes. Most operations on text data are executed embarrassingly parallel, such as local and join operations on a single sentence or operations on a single document. INDREX benefits from built-in optimizations of the columnar database for these operations, such as multi-core support for executing expensive operations on a single sentence, built-in partitioning schemas for fast data-locality aware execution and at-query-time-code-generation for optimizing built-in and user-defined functions. We could also observe that INDREX benefits from built-in data compression techniques of the Parquet.io file format, such as dictionary encoding, zigzag encoding, and RLE encoding of data. However, Impala cannot yet execute queries that require user-defined table generating functions, such as queries for consolidating overlapping spans or queries that split results.

Overall, we can recommend running INDREX on a columnar database with multi-core support and compression, such as Impala. Contrarily, state-of-the-art approaches execute text mining workloads in Hadoop-based environments nearly two orders of magnitude slower.

Additional observations: Similar to the Extract-Transform-Load paradigm of a data warehouse, we recommend executing *common* language and document specific transformations before loading. Often, existing implementations of these tasks are already optimized towards result quality and execution speed. However, we recommend executing domain specific transformations in INDREX. These transformations often require the user to integrate annotations from base tables with domain specific data in an iterative fashion. The design of our INDREX system follows these guidelines and explicitly supports the often practiced trial-and-error process of many users during relation extraction.

Optionally, the loader could pre-compute document specific aggregations, such as term or annotation frequencies, or local joins when loading a document 'on-the-fly' and before query time. As a result, the database system can focus its resources on other computations during query time.

Sharing or protecting data or rules: INDREX leverages built-in capabilities of the underlying system for sharing data about the extraction process among different users. Example resources are views that incorporate base tables, tables that contain domain specific data, such as from a customer resource management system, or tables that contain data from external resources, like Gazetteers. INDREX users can protect these 'assets' with the built-in security system of a RDBMS and can grant rights to other users.[7] Hence, INDREX users execute these tasks with the standard workbench of the underlying system and do not need specific clients for accessing or managing these resources.

5.7. Future work

This work contributes to the vision of a single integrated query system over textual and relational data representations. We consider our future work in the following two areas:

5.7.1. Learning hints for rewriting queries

Resolving homonyms, synonyms and entailment: Working with natural language means managing the never-ending variance in language, in particular when dealing with homonyms, synonyms or entailment. Often the user needs help from the system when formulating complex queries for resolving these cases. One example is the disambiguation of relation names. In [3] we actively proposed synonymous expressions for words that likely represent names of relations or attributes. Moreover, in [4] we proposed selectional restrictions for attribute types of a relation learned from large corpora of text data. Another direction is learning to join, which requires from the system to identify predicates for joining text data with relational data.

Repairing spans: Another interesting direction is repair operations on span data. Such an operation would inspect results from candidate extractors and automatically correct overlapping or incorrect span boundaries.

Learning hints: For learning such hints, computational linguists frequently apply classifiers based on features from sentences [38]. These data mining functions execute iterative algorithms and require global states. Another source for hints is click stream-based approaches. For example, Google's knowledge graph leverages click stream data for refining rule based and pre-computed extractors. In the case of INDREX, an OLAP-[19], search- or CRM-application might trigger a query refinement. In [40] we defined a preliminary set of such interactions. In our future work, we will explore how these interactions may improve the quality (in terms of precision and recall) of extractors in INDREX.

5.7.2. Hints-as-you-type

We could observe interaction times in 10 s of seconds with our setup and nearly two orders of magnitude speedup in contrast to a Hadoop-based baseline. Ideally, the user would receive instant query refinements and results as-she-types. This scenario again requires answering times in hundreds of milliseconds, or another speedup from the system of nearly two orders of magnitude.

In-memory databases: In-memory databases leverage multi-core technologies and a hybrid row-based and columnar-based layout on main memory hierarchies. Examples are SAP HANA [27] or MonetDB [12]. This database

[7] IMPALA uses sentry, a role-based, fine-grained authorization system.

technology might permit the necessary performance gains for our scenario. Extending these databases not only requires transforming the principles of INDREX to this new hardware layer. It also requires extending these databases with powerful classifiers for computing suggestions and hints. One option is ensemble classifiers that may leverage multi-core and shared memory architectures.

Columnar storage improvements: Main memory databases may hold large samples of compressed relational data and data from our span table. However, main memory databases are too costly for analyzing billions of pages. Distributed columnar databases are a young phenomenon and currently various storage layouts on top of the HDFS exist [22]. An interesting topic is extending INDREX towards a query optimizer that incorporates various columnar layout techniques and clustered index structures for achieving sub-second response times. This includes support for query predicates with a low selectivity, such as the dictionary lookup predicate in Fig. 3, or lookup queries on primary keys, such as the ID of a span, document, sentence or annotation, or local queries that often execute self-joins. Other extensions include support for queries that exploit the proximity of span containment in textual data. Extending distributed columnar databases with the ability to execute iterative algorithms, including managing effectively global states across multiple nodes, is another relevant topic.

6. Conclusion

We described INDREX, a single system for managing both textual and relational data. The system supports a data warehouse style extract-transform-load of generic relations extracted from text documents and supports additional text analysis libraries. The user can query these generic relations to extract higher level semantics and can join them with other relational data. We formalized extensions supporting these queries and presented our intuitive span data model. Our white-box-functions extend a Hadoop-based and an Impala-based execution engine and benefit from built-in optimizations. Our results demonstrate the effectiveness of INDREX on Impala on a wide set of queries. We report nearly two orders of magnitude faster execution times for the Impala-based system over the Hadoop-based system and could receive results in 10 s of seconds.

There is an enormous opportunity for researchers to make this base system even more principled, effective, and efficient. For example, one vision is instant hints for supporting the user when writing queries. In our future work, we will extend the INDREX language with new primitives (and corresponding optimizations), and will explore modern hardware.

Acknowledgments

Torsten Kilias and Alexander Löser receive funding from the Federal Ministry of Economic Affairs and Energy (BMWi) under grant agreements 01MD11014A ("MIA-Marktplatz für Informationen und Analysen") and 01MD15010B ("Smart Data Web").

Appendix A. Benchmark queries

We propose the following set of queries for typical operations during an iterative relation extraction task. We used these queries for evaluating the feasibility of a system for executing them and for measuring the efficiency of the system.

Selections: These queries scan and select annotations with various comparison predicates.

Q1. Select all tokens (high cardinality).
Q2. Select all tokens *acquire* (low cardinality).
Q3. Select all tokens that match the regular expression *acquir.**.
Q4. Select all tokens LIKE *%acquir%*.
Q5. Select all annotations with stringId = 1000.

*Joining data from the same document or sentence:*These queries execute joins, such as self-joins, on the same document or sentence. For instance, a query that falls into this category tests if two spans appear directly after each other. Another example is queries that test if two nodes in the dependency path are connected by a common edge.

Q6. Return all occurrences of the 3-token phrase *President Bill Clinton*.
Q7. Return all token *3-grams*, such as *wor*, *ord* for *word*.
Q8. Return all occurrences of a *verb* between two *noun phrases* within a *sentence*.
Q9. Return all documents with their *sentences* and *tokens*.
Q10. Return all sentences that contain an *organization* and a *person*.
Q11. Return all paths with *length of three* in dependency graphs.
Q12. Return all occurrences of the dependency path $cop\langle -nsubj -\rangle nn$.
Q13. Return all organizations that contain the source or target of a dependency with the label *nsubj*.

*Joins with external domain specific data:*These queries execute a join against text data and an external resource, such as a dictionary.

Q14. Join noun phrases, such as *Hurricane Katrina*, with a dictionary for weather phenomena.
Q15. Join the predicate verb phrase, such as *acquire* or *purchased* or *took over*, between two organizations with a dictionary for business acquisitions.

*UDT/UDAs:*These queries execute user defined table generating functions, such as testing and consolidating the containment of multiple phrases.

Q16. Return all noun phrases that do not contain any other noun phrases. This query emulates a ContainedWithin () function.
Q17. Extract all enumerations of organizations, such as … *IBM, SAP and Software AG*.

*Corpus wide aggregations:*These queries conduct aggregations, such as counting annotations, or grouping and counting annotations by type or text.

Q18. Count all tokens …
- Q18.1. Within a string, such as a document or a sentence.
- Q18.2. Across all documents in the corpus.

Count the occurrences of each term

Q19.
- Q19.1. Within a string, such as a document or a sentence.
- Q19.2. Across all documents in the corpus.

Count the occurrences of each named entity type.

Q20.
- Q20.1. Within a string, such as a document or a sentence.
- Q20.2. Across all documents in the corpus.

Joins + GroupBy: These queries extend the case from above and group results after the join.

Q21. Return two organizations that frequently occur together in a sentence.
- Q21.1. Group by sentences from the same document.
- Q21.2. Group by sentences across the corpus.

Which verbs frequently occur between two organizations?

Q22.
- Q22.1. Group by sentences from the same document.
- Q22.2. Group by sentences across the corpus.

Appendix B. Runtime details

Figs. 10–14 show details for each query category and 'drill down into' the measured aggregated runtime of Fig. 7 from Section 5.5. The figures compare the runtime share (shown in percent) and for each query category between a Hadoop+Pig system and an Impala System on 800K documents.

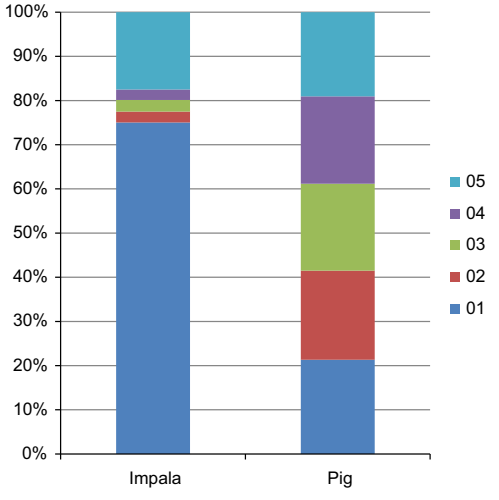

Fig. 10. Comparison of runtime percentages for different queries that process only selections.

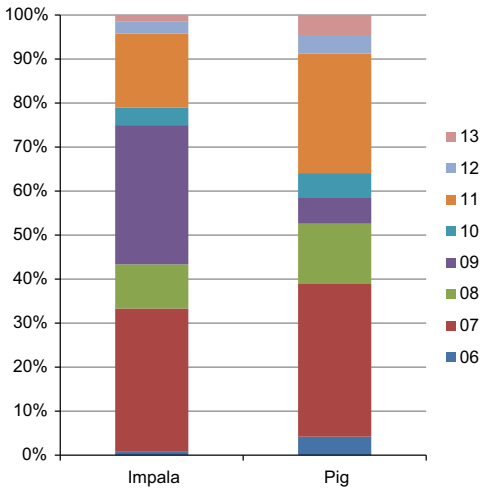

Fig. 11. Comparison of runtime percentages for different queries that process local joins.

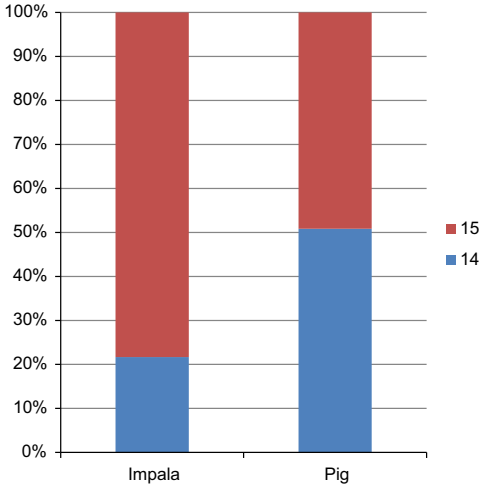

Fig. 12. Comparison of runtime percentages for different queries that process joins with external sources.

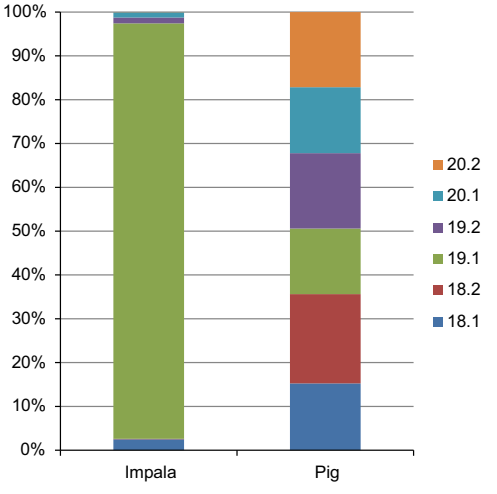

Fig. 13. Comparison of runtime percentages for different queries that process only aggregations.

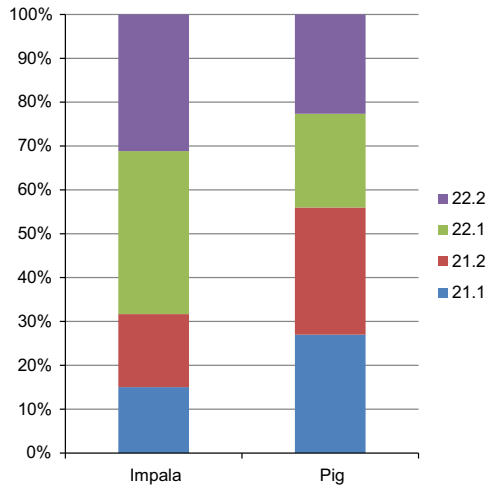

Fig. 14. Comparison of runtime percentages for different queries that process local joins with aggregations.

Appendix C. Feature Comparison

Table 2 shows a comparison of the available features for different relation extraction systems.

Table 2
This table shows a comparison of the available features for different relation extractions systems. We compare the two Prototypes mentioned in Section 5, INDREX on Impala and INDREX on Pig, such as SystemT [37], GATE [18] and PTQL [47] from Section 2.

Features	IMPALA	IMPALA/INDREX	PIG/INDREX	SystemT	GATE	PTQL
Supported annotations						
Token	No	Yes	Yes	Yes	Yes	Yes
POS	No	Yes	Yes	Yes	Yes	Yes
Lemma	No	Yes	Yes	Yes	Yes	Yes
Named entity	No	Yes	Yes	Yes	Yes	Yes
Phrases	No	Yes	Yes	Yes	Yes	Yes
Constituency parse	No	Yes	Yes	possible as UDF	UDF	Link grammar
Dependency parse	No	Yes	Yes	possible as UDF	UDF	No
Relations	No	Yes	Yes	Yes	Yes	No
Across-document annotations	No	Yes	Yes	No	No	No
Use existing base annotations during query	No	Yes	Yes	No	Yes	Yes
Generate base annotations during query	No	No, because of the lack of UDTs	Yes	Yes	Yes	No
Store new created annotations	No	Yes	Pig Latin	Pig Latin, JAQL	Yes	Yes
Local queries (per document)						
Regular expressions						
Char	Yes	Yes	Yes	Yes	Yes	No
Token	No	Only simple, without Kleene star	Yes	Yes	JAPE	No
Predicates and scalar functions on spans, e.g. before, contains or CombineSpans						
On char	No	Yes	Yes	Yes	No	No
On token	No	Yes	Yes	Yes	No	Only Before
On segment	No	Yes	Yes	No	No	No
Consolidate	No	No	Yes	Yes	No	No
Joining with existing semantics						
Dictionary lookup	No	File, table or column based	File, table or column based	File based	File based	Broker joins the query results
Global queries						
Group by						
Local	Yes	Yes	Yes	Yes	No	No
Global	Yes	Yes	Pig Latin	Pig Latin/ JAQL	No	No
Aggregations (SUM, COUNT, AVG, MIN, MAX)						
Local	Yes	Yes	Yes	Yes	No	No
Global	Yes	Yes	Pig Latin	Pig Latin/ JAQL	No	No
Across-document queries, e.g. follow web links	No	Yes	Pig Latin	Pig Latin/ JAQL	No	No

References

[1] Parquet.io homepage ⟨www.parquet.io⟩ (last visited 04/02/2014).
[2] A. Akbik, A. Löser, Kraken: N-ary facts in open information extraction, in: Joint Workshop on Automatic Knowledge Base Construction and Web-scale Knowledge Extraction, AKBC-WEKEX '12.
[3] A. Akbik, L. Visengeriyeva, P. Herger, H. Hemsen, A. Löser, Unsupervised discovery of relations and discriminative extraction patterns, in: COLING, 2012, pp. 17–32.
[4] A. Akbik, L. Visengeriyeva, J. Kirschnick, A. Löser, Effective selectional restrictions for unsupervised relation extraction, in: IJCNLP, 2013.
[5] J.F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (November (11)) (1983) 832–843.
[6] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M.J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, C. Zhang, Brainwash: a data system for feature engineering, in: CIDR, 2013.
[7] G. Attardi, F. dell'Orletta, M. Simi, A. Chanev, M. Ciaramita, Multilingual dependency parsing and domain adaptation using desr, in: EMNLP-CoNLL, 2007, pp. 1112–1118.
[8] N. Bales, A. Deutsch, V. Vassalos, Score-consistent algebraic optimization of full-text search queries with graft, in: SIGMOD Conference, 2011, pp. 769–780.
[9] K.S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M.Y. Eltabakh, C.-C. Kanne, F. Özcan, E.J. Shekita, Jaql: a scripting language for large scale semistructured data analysis, PVLDB 4 (12) (2011) 1272–1283.
[10] B. Bloom, Taxonomy of Educational Objectives: Handbook I: Cognitive Domain, Longmans, Green, New York, 1956.
[11] C. Boden, A. Löser, C. Nagel, S. Pieper, Factcrawl: a fact retrieval framework for full-text indices, in: WebDB, 2011.
[12] P.A. Boncz, M.L. Kersten, S. Manegold, Breaking the memory wall in monetdb, Commun. ACM 51 (12) (2008) 77–85.
[13] F. Chen, A. Doan, J. Yang, R. Ramakrishnan, Efficient information extraction over evolving text data, in: ICDE, 2008, pp. 943–952.
[14] L. Chiticariu, V. Chu, S. Dasgupta, T.W. Goetz, H. Ho, R. Krishnamurthy, A. Lang, Y. Li, B. Liu, S. Raghavan, F. Reiss, S. Vaithyanathan, H. Zhu, The systemt ide: an integrated development environment for information extraction rules, in: SIGMOD Conference, 2011, pp. 1291–1294.
[15] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F.R. Reiss, S. Vaithyanathan, System t: an algebraic approach to declarative information extraction, in: ACL, 2010.
[16] L. Chiticariu, Y. Li, F.R. Reiss, Rule-based information extraction is dead! Long live rule-based information extraction systems!, in: EMNLP, 2013, pp. 827–832.
[17] L.D. Corro, R. Gemulla, Clausie: clause-based open information extraction, in: WWW, 2013, pp. 355–366.
[18] H. Cunningham, Gate: a general architecture for text engineering, Comput. Human. 36 (2) (2002) 223–254.
[19] A. Cuzzocrea, S. Mansmann, Olap visualization: models, issues, and techniques, in: Encyclopedia of Data Warehousing and Mining 2009, pp. 1439–1446.
[20] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[21] G. Demartini, D.E. Difallah, P. Cudré-Mauroux, Large-scale linked data integration using probabilistic reasoning and crowdsourcing, VLDB J. 22 (5) (2013) 665–687.
[22] J. Dittrich, J.-A. Quiané-Ruiz, Efficient big data processing in hadoop mapreduce, PVLDB 5 (12) (2012) 2014–2015.
[23] A. El-Helw, M.H. Farid, I.F. Ilyas, Just-in-time information extraction using extraction views, in: SIGMOD Conference, 2012, pp. 613–616.
[24] O. Etzioni, M. Banko, S. Soderland, D.S. Weld, Open information extraction from the web, Commun. ACM 51 (12) (2008) 68–74.
[25] O. Etzioni, A. Fader, J. Christensen, S. Soderland, Mausam, Open information extraction: the second generation, in: IJCAI, 2011, pp. 3–10.
[26] A. Fader, S. Soderland, O. Etzioni, Identifying relations for open information extraction, in: EMNLP, 2011.
[27] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, J. Dees, The sap hana database—an architecture overview, IEEE Data Eng. Bull. 35 (1) (2012) 28–33.
[28] R. Feldman, Y. Regev, M. Gorodetsky, A modular information extraction system, Intell. Data Anal. 12 (January (1)) (2008) 51–71.
[29] D. Ferrucci, A. Lally, UIMA: an architectural approach to unstructured information processing in the corporate research environment, Nat. Lang. Eng. 10 (September (3–4)) (2004) 327–348.
[30] K. Ganchev, K. Hall, R.T. McDonald, S. Petrov, Using search-logs to improve query tagging, in: ACL, vol. 2, 2012, pp. 238–242.
[31] C. Gokhale, S. Das, A. Doan, J.F. Naughton, N. Rampalli, J. Shavlik, X. Zhu, Corleone: Hands-off crowdsourcing for entity matching, in: ACM SIGMOD, 2014.
[32] P. Grefen, R. de By, A multi-set extended relational algebra: a formal approach to a practical issue, in: Proceedings of 10th International Conference on Data Engineering, February 1994, pp. 80–88.
[33] M. A. Hearst, Automatic acquisition of hyponyms from large text corpora, in: COLING, 1992, pp. 539–545.
[34] A. Jain, A. Doan, L. Gravano, Optimizing SQL queries over text databases, in: ICDE, 2008, pp. 636–645.
[35] T. Kilias, A. Löser, P. Andritsos, Indrex: in-database distributional relation extraction, in: DOLAP, 2013, pp. 93–100.
[36] P. Kluegl, M. Toepfer, P.-D. Beck, G. Fette, F. Puppe, Uima ruta workbench: rule-based text annotation, in: 25th International Conference on Computational Linguistics: System Demonstrations, 2014.
[37] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, H. Zhu, SystemT: a system for declarative information extraction, SIGMOD Rec. 37 (March (4)) (2009) 7–13.
[38] H. Lee, A.X. Chang, Y. Peirsman, N. Chambers, M. Surdeanu, D. Jurafsky, Deterministic coreference resolution based on entity-centric, precision-ranked rules, Comput. Linguist. 39 (4) (2013) 885–916.
[39] D.D. Lewis, Y. Yang, T.G. Rose, F. Li, RCV1: a new benchmark collection for text categorization research, J. Mach. Learn. Res. 5 (December) (2004) 361–397.
[40] A. Löser, S. Arnold, T. Fiehn, The goolap fact retrieval framework, in: Business Intelligence, Springer, US, 2012, pp. 84–97.
[41] A. Löser, C. Nagel, S. Pieper, C. Boden, Beyond search: retrieving complete tuples from a text-database, Inf. Syst. Front. 15 (3) (2013) 311–329.
[42] G. Marchionini, Exploratory search: from finding to understanding, Commun. ACM 49 (April (4)) (2006) 41–46.
[43] N. Nakashole, G. Weikum, F. M. Suchanek, Patty: a taxonomy of relational patterns with semantic types, in: EMNLP-CoNLL, 2012, pp. 1135–1145.
[44] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: SIGMOD Conference, 2008, pp. 1099–1110.
[45] D.E. Rose, D. Levinson, Understanding user goals in web search, in: WWW, 2004, pp. 13–19.
[46] A. Sun, R. Grishman, Active learning for relation type extension with local and global data views, in: CIKM, 2012, pp. 1105–1112.
[47] L. Tari, P.H. Tu, J. Hakenberg, Y. Chen, T.C. Son, G. Gonzalez, C. Baral, Incremental information extraction using relational databases, IEEE Trans. Knowl. Data Eng. 24 (1) (2012) 86–99.
[48] H. Wang, C. Zaniolo, Using sql to build new aggregates and extenders for object-relational systems, in: VLDB, 2000, pp. 166–175.