IPTC EXTRA project
D1.1 Technical Approach

Version 1.0 - 10 March 2017

This deliverable provides a high-level description of the selected technical approach and the design of the proposed solution. It justifies the design of the proposed system architecture as well as the selected implementation technologies.

Overview

Given the foreseen large scale of classification rules (2M according to the EXTRA requirements[1] document), the exhaustive evaluation of an input document against all rules is expected to be impractical for very large rule sets, i.e. it would lead to execution times much larger than the one second mark that is defined by the requirements as acceptable. To this end, the proposed solution must be able to quickly select only a subset of candidate rules and evaluate a document only on this limited set. To this end, the main steps to be followed are the following:

1. Given a structured document, a set of candidate rules is selected from the rule index. To implement this step, it is crucial to use an indexing structure that supports accurate and fast retrieval of the appropriate rule subset.
2. The set of candidate rules is evaluated against the input document. In the ideal scenario, the list of candidate rules is small and all of them match the input document.
3. For all rules that match the input document (i.e. evaluate to TRUE), a default or custom relevance function is computed.

An overview of the architecture of the proposed solution is depicted in Figure 1. A brief description of the modules of EXTRA rules engine is presented in Table 1. Storage and access of rules (in their initial format as entered by the end users), and other data objects such as schemas, dictionaries and relevance algorithms, is implemented on top of a native JSON database (such as mongoDB), denoted as **DB**. This component supports the basic CRUD operations as defined in the requirements through an appropriate set of API methods. Functionalities such as validation of rules and documents are performed from the API that communicates directly with the DB component.

Regarding the classification of documents, the core component of EXTRA is the **Rules Index (RI)**. This is an indexing data structure that supports the efficient search/selection of rules based on input documents (using the facilities of ElasticSearch percolate as will be explained below). To perform the indexing of rules, the **Rules Mapper** component takes care of transforming the EXTRA rules, described in EXTRA Rule Language, into an internal representation that is usable

[1] https://iptc.org/download/workstream/extra/IPTC-EXTRA-TechnicalRequirements_v100_2017-01-30.pdf

by the underlying indexing framework (ES percolate). Also, RM is responsible for the decomposition of rules into simpler forms as will be described in the next section.
The actual document classification pipeline starts from the **Rules Selector (RS)** component, which analyzes the input document and transforms into an appropriate representation that can be used for efficient retrieval of the candidate rules from the RI. For example, assuming that Elastic Search is used as the underlying indexing framework, RS transforms input documents into percolate queries and requests for rules match the document from RI. Then, the **Rules Evaluator (RE)** iterates over the set of retrieved rules and calculates whether each of them truly matches the input document. As candidate rules can be parts of other rules (as a result of decomposition takes place in Rules Mapper) this module is responsible to identify the initial rules activated by the input document. Finally, the **Rules Ranker (RR)** calculates the score for each rule based on the specified relevance algorithm, that indicates how strongly the document matches each rule. The set of available relevance algorithms and their parameters are stored in the DB module, and the end user can select the appropriate algorithm during classification. These three modules, depicted within a green frame in Figure 1, are successive steps of the classification process and are exposed as a single API method.

Table 1: Modules of EXTRA rules classification engine

Module	**Description**
EXTRA API	This module exposes the functionality of EXTRA as a set of REST methods.
DB	This module is a data persistence layer, used to store rules, schemas, dictionaries and relevance algorithms. MongoDB will be used as the storage engine.
Rules Mapper	This is responsible for two operations: a) decomposition of EXTRA rules stored in DB into simpler rules and b) transformation of rules into queries expressed in ES Query Language that will be the internal representation of the rule classification engine.
Rules Index	This is the indexing structure offered by ES. This module is used for the indexing of rules expressed as ES queries.
Rules Selector	This is the module that analyzes input documents, transforms them into percolate queries and forwards them to Rules Index for the retrieval of candidate rules.
Rules Evaluator	This processes a set of candidate rules and evaluates whether they actually satisfy the input document. As candidate rules can be parts of other rules (as a result of the decomposition that takes place in Rules Mapper) this module is responsible to identify the initial rules activated by the input document.
Rules Ranker	Rules Ranker applies relevance algorithms (specified or default) in the set of valid rules returned by Rules Evaluator, and sorts them according to it.

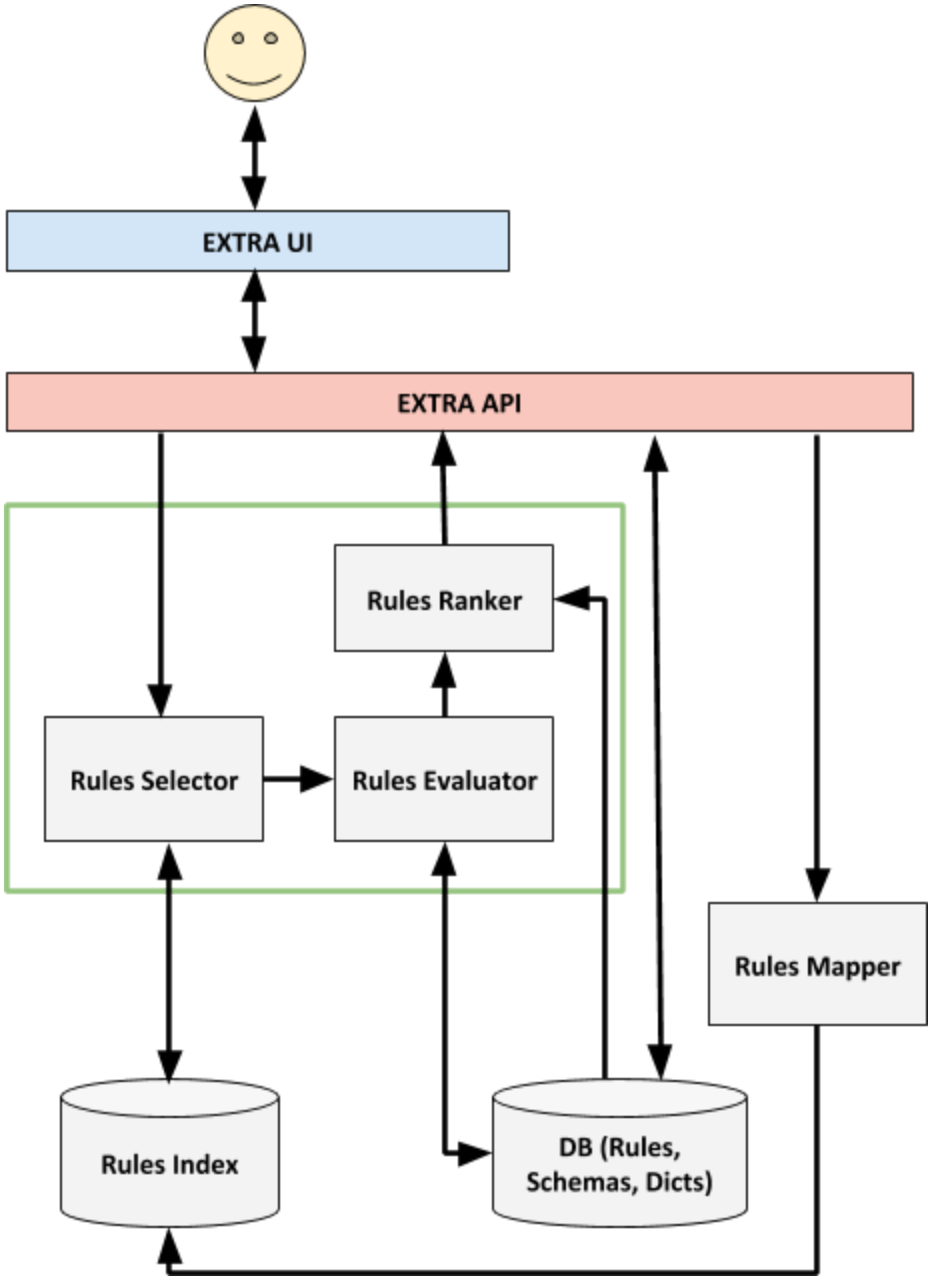

Figure 1: Overview of the core EXTRA rule classification engine. The red frame corresponds to the EXTRA API (further detailed in D1.2). Interaction can be performed either directly with the API or via the UI supported by the framework (blue frame). Green frame contains the modules that are responsible for the matching of documents to EXTRA rules. Note that this architecture is independent of the rule language used by EXTRA (more information on this is provided in D1.4).

At that point we briefly present three core actions of EXTRA Rule Engine and the modules associated with each action. The first action is the insertion of a new rule from the end user. The new rule that is inserted via a call to the corresponding API method is first stored into DB and then is forwarded to Rules Mapper. RM analyzes it and presumably decompose it to

multiple sub-rules. Each subpart (or the initial rule in case it remained intact) is transformed to ES queries. These queries are forwarded to Rules Index module. The second action is the validation of a newly created rule against a schema stored in DB. This validation is a required part of the creation process but the end user may want to validate a rule during editing or before submission. To this end, the specific schema is retrieved from DB and the rule is checked by the corresponding API method. Finally, the third and most important action in EXTRA system is the classification of a document by matching it to EXTRA rules. This action is performed by following the sequence of modules depicted inside the green frame in figure 1. Rules Selector analyzes input document, transform it to ES percolate query (or Flaxsearch Luwak query) and forwarded to to Rules Index. RI retrieves candidate rules that match the document. These rules are directed to Rules Evaluator, which retrieves the initial forms of the rules and evaluate whether are valid or not. The valid rules are ranked based in the specified relevance algorithm by Rules Ranker.

Rules Mapping, Indexing and Retrieval

Based on our initial assessment of relevant frameworks, we have identified two frameworks that are suitable to base our solution on: a) Elasticsearch Percolator[2] and b) Flaxsearch Luwak[3]. Both support the indexing of a large number of queries and the execution of searches against these queries based on an incoming document, i.e. the reverse problem setting of a full-text search engine. We have not further considered using UIMA[4] at this stage, since it could introduce additional complexity, which is not required given the requirements of the project. Given a query indexing mechanism in place, we envision that the basis of the EXTRA engine will be a component that will transform EXTRA rules[5] into queries (either Elasticsearch (ES) percolate queries or Luwak queries) that will be indexed and matched against incoming documents. Initially, ES has been selected as the indexing mechanism to base the EXTRA engine for the following reasons: a) more mature and widely used project, b) richer set of features, c) richer and more complete documentation, d) active development community and accompanying open-source projects. The only reason why Luwak might be preferable would be the lower performance of ES, in terms of response time to queries. However, as will be shown through our initial experimental study of ES, which will be presented below, ES exhibits promising performance and there do not seem to be obvious bottlenecks. In addition, we make the following observations:

- ES percolate queries and the corresponding API appear to be more expressive and easier to map to the EXTRA operators and required features.
- ES offers a richer set of facilities for indexing, query matching, highlighting and scaling through sharding.

---

[2] https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-percolate-query.html
[3] https://github.com/flaxsearch/luwak
[4] https://uima.apache.org/
[5] The language used to express EXTRA rules will be based on CQL and is further discussed in D1.4.

- Luwak features a “presearcher” feature, which is designed to quickly filter a large number of queries (rules) and ultimately is expected to lead to faster query search. The underlying mechanics and approach of Luwak might be possible to use in tandem with an ES index in case performance bottlenecks are identified at the indexing layer.

Having selected ES as the underlying rule indexing mechanism, considerable research and development effort will be necessary due to the following:

- Mapping of EXTRA rules to queries is not straightforward and several trade-offs need to be studied in order to reach an efficient solution that at the same time meets all functional EXTRA requirements.
- Performance may prove to be limited. Previous efforts of using ES percolate out-of-the-box are rather discouraging regarding its performance, reporting an average throughput rate of one document per second on a query set of 100K documents[6] (much below the 2M target set by EXTRA). However, as our experimental study will show, this does not seem to be an issue anymore (we have tested version 5.2).

Without any loss of generality, we adopt ES percolate queries to show how classification rules can be indexed and retrieved efficiently. In general, ES uses its own domain specific language to represent queries and supports two query types: a) boolean and b) span queries. The majority of boolean operators that constitute elements of EXTRA classification rules can be mapped into these two categories. Table 2 presents the boolean operators that must be implemented by EXTRA and the corresponding ES queries. From these operators, #1, #2 and #3 can be mapped to simple boolean queries. Nested boolean queries can be used for operators #8, #10, #12 and #13. Span queries can represent operators #5, #7, #9, #11, #14. Finally, operators #4, #6 and #15-#20 have no direct mapping with ES queries.

Table 2: Mapping of EXTRA rule operators to ES queries

#	Operator	Definition	ES Query
1	AND	Takes two or more statements. Category matches only if all the statements are true.	{"bool":{"must"}}
2	OR	Takes two or more statements. Category matches if at least one statement is true.	{"bool":{"should"}}
3	NOT	Used in combination with AND. Category matches if the statement does not appear in combination with statement under the AND.	{"bool":{"must_not"}}
4	MINIMUM	Combined with a number (e.g., MINIMUM_2). Takes one or more statements. Category matches if a minimum of x statements from the list appear in the text.	minimum_should_match parameter on a bool query
5	DISTANCE	Combined with a number. Takes two or more statements. Category matches if statements are within x number of words from each other.	span_near with slop parameter
6	MINIMUM OCCURRENCE	Combined with a number. Takes one or more statements. Category matches if statement appears x amount of times in the text.	

[6] http://underthehood.meltwater.com/blog/2015/09/29/supercharging-the-elasticsearch-percolator/

7	ORDER	Takes two or more statements. Category matches if statements appear in the text in the same order that they appear in the rule.	span query with in_order param set to "true"
8	SENTENCE	Takes two or more statements. Category matches if statements appear within the same sentence.	nested AND queries (sentences need to be indexed as sub-fields in ES)
9	NOT WITHIN DISTANCE	Combined with a number. Takes two or more statements. Category matches if the statements are not within x amount of words from each other.	span_not with dist param set
10	PARAGRAPH	Takes two or more statements. Category matches if all the statements occur in the same paragraph.	nested AND queries (paragraphs need to be indexed as sub-fields in ES)
11	NOT IN PHRASE	Takes two statements. Category matches if the first statement occurs outside of the second statement.	span_not
12	NOT IN SENTENCE	Takes two or more statements. Category matches if all the statements do appear in the same document, but not in the same sentence.	nested AND queries (sentences need to be indexed as sub-fields in ES)
13	NOT IN PARAGRAPH	Takes two or more statements. Category matches if all the statements do appear in the same document, but not in the same paragraph.	nested AND queries (paragraphs need to be indexed as sub-fields in ES)
14	ORDER AND DISTANCE	Combined with a number. Takes two or more statements. Category matches if both statements occur in the same order in which they are written in the rule and if both are within x amount of words to each other.	span_near with in_order param set to "true"
15	MAXIMUM OCCURRENCE	Combined with a number. Takes at least one statement. Category matches if the statement appears no more than x amount of times in the text.	
16	FROM START	Combined with a number. Takes one or more statements. Category matches if the statement appears within x amount of words from the start of the text.	
17	FROM END	Combined with a number. Takes one or more statements. Category matches if the statement appears within x amount of words from the end of the text.	
18	MAXIMUM SENTENCES	Combined with a number. Takes one or more statements. Category matches if the statements appear within the first x sentences.	
19	MAXIMUM PARAGRAPHS	Combined with a number. Takes one or more statements. Category matches if statement appears within the first x paragraphs.	
20	PARAGRAPH POSITION	Combined with a number. Takes one or more statements. Category matches if statement appears within the x-th paragraph.	

The operators that have no direct mapping with ES queries are handled by using a relaxation approach as follows: given an operator with no direct mapping, we perform a relaxation into a

simplified form using boolean operators. This is expected to fetch more candidate rules, some of which may not match the query. These will be then filtered out by the RE component, which evaluates each of the candidate rules. For example, the **MAXIMUM_SENTENCES_X** operator can be replaced by the **AND** operator. The relaxed version of such a rule requires the statements to co-appear in the whole document. In the evaluation step, the RE would uses the original version of the rule that requires the statements to appear within the first X sentences. In the same way, **MINIMUM_OCCURRENCE_N** and **MAXIMUM_OCCURRENCE_N** have to be evaluated a posteriori. As ES cannot perform queries that use the occurrences of a term, these two operators cannot be mapped to an ES query. To this end, we relax these operation by requesting only the appearance of a term. Evaluation of the number of occurrences is performed by the RE module.

Given the inability of ES to mix boolean with span queries, but also the fact that rules can be quite complex, we propose the incorporation of a decomposition step that simplifies the classification rules to less complex forms. For example the **PARAGRAPH** operator, that requires that the statements it connects appear in the same paragraph, is implemented in ES as nested **AND** queries, while paragraphs in the document need to be expressed as different sub-fields in ES. On the other hand **ORDER** operator, that demands two statements to appear in specific order is mapped to a span ES query[7]. Assuming that a rule combines these two operators e.g. two statements, #1 and #2, appear in the same document but statement #2 must appear before statement #1, then we must to split the rule into two parts and evaluate each of them separate. As those types of ES queries cannot be combined, the initial rule, expressed as

(**AND**,
    (**PARAGRAPH**,
        (statement #1),
        (statement #2)
    ),
    (**ORDER**,
        (statement #2),
        (statement #1)
)
)

must be splitted into two parts (one for the **PARAGRAPH** and one for the **ORDER** sub-statement) and each of them is indexed separately. As both statements must be true, due to the **AND** operator that combine them, to evaluate the whole rule as active both sub-statements must be returned by Rules Selector and subsequently evaluated to true by Rules Evaluator.

In case of pure boolean expressions a similar procedure takes place mainly for efficiency reason. The first step at this procedure is the transformation of the expression representing the rule into a **Sum-Of-Products (SOP) form**. Namely we rewrite the rules in order to apply OR operators to AND expressions. For example, the rule:

(**AND**,
    field : exp1,
    (**OR**,

[7] https://www.elastic.co/guide/en/elasticsearch/reference/current/span-queries.html

```
                field  : expr2, field : expr3
        )
    )
```

would be transformed into the following form:

```
    (OR,
        (AND,
            field : expr1, field : expr2
        ),
        (AND,
            field : expr1, field : expr3
        )
    )
```

We opt for this form as sub-parts of **OR** operators can be evaluated independently (in parallel), given that a sub-part of this form can trigger the whole rule. Note that this step can be avoided if the rules are directly written in the SOP form by the end users. We can then map the above SOP form into an ES query as a whole, or alternatively use a segmentation approach to split it in more simple "independent" parts (sub-rules). The simplest approach is to split the rule to the first OR level. For example in the above example we would end up with two rules: (**AND**, field: expr1, field: expr2) and (**AND**, field: expr1, field: expr3).

However, it is possible that the expansion of a rule into a full SOP form will result in quite complex rules, with subparts that exhibit significant overlap with each other. For this purpose we can split the initial rule into finer parts by using its Abstract Syntax Tree (AST). Logical operators are the inner nodes of the tree while leaf nodes correspond to terms. Using a depth-first traversal of AST, we split the input rule in the **OR** nodes, leaving intact the sub-trees under **AND** nodes[8]. To sum up, these three rule decomposition approaches (no decomposition, SOP-based and AST-based) can be used interchangeably depending on the complexity of the rule. The selected level of decomposition must be the result of a trade-off between rule complexity and response time (consumed on rules retrieval) as will described in the next section. A similar approach is applied in case of rules that contain operators mapped to span queries mixed with boolean operators. For example a **NOT_IN_PHRASE** operators inside an **AND** statement cannot be mapped directly to an ES query. For that reason the part of the rule under **NOT_IN_PHRASE** operator is treated as a new rule. The mapping of the simple query described above to an ES query is performed by Rules Mapper. The final ES query is depicted in Table 3.

Table 3: Example of ES query

```
{
        "query": {
                "bool" : {
                        "should" : [
                                {"must" : [
```

[8] Although the first approach is splitting of rules on OR operators, we may consider decomposition of rules on AND nodes in cases of too complex rules. As we will show in the next section, complexity affects response time. To this end, it may be necessary to break rules into finer parts even if this decomposition takes place between parts combined with AND. At these cases all the generated rules has to be valid to trigger the whole rule.

```
                                        {"term" : { "field" : "expr1" }},
                                                {"term" : { "field" : "expr2" }}
                                        ]},
                                        {"must" : [
                                        {"term" : { "field" : "expr1" }},
                                                {"term" : { "field" : "expr3" }}
                                        ]},
                                ]
                        }
                }
}
```

As stated by the requirements of EXTRA (req. 7.2.1), rules must be able to reference other rules. In that way the system can compose rules by combining existing rules using the predefined boolean operators of Table 1. For example given rules Rule1 and Rule2, the user of EXTRA can create a new rule Rule3 ← (**AND**, Rule1, Rule2). The composition may lead to more complex rules by concatenating rules, operators and new expressions of terms. The resulting rules are handled in the same way as any other rule in the system, i.e. decomposed and indexed in the Rules Index structure. Rules containing regular expressions (req. 7.2.5) will rely on the wildcard query capabilities of ES.

For the selection of a candidate set of rules the input document is used as a query. As Rules are represented in the form of ES queries, we use the percolate query facilities of ES. Percolate queries i.e. a special type of queries generated from documents, can be used to match rules stored in an index. In other words using this reverse search allows for indexing of rules (queries) and percolating an input document to find rules (queries) that will match it.

In a simple example let us assume that we have documents consisting of two fields: *title* and *body*. We want to match input documents with the stored rules. First we have to create an index for the rules by using the command of Table 4. This index is part of the Rules Index component. We define two mappings, one for the documents and one for the rules. At that point, we have to note that as EXTRA will be able to support multiple schemas, multiple document mappings can be inserted into queries index. Furthermore, data types of fields are defined appropriately. In the specific case, *title* and *body* are defined as of type *"text"*, which is a predefined type in ES. In a real case scenario, different types have to be defined to support a more elaborated analysis of fields, e.g. stemming.

Table 4: Creation of rules_index (REST call refers to the ES API and should not be confused with the EXTRA API)

```
PUT /rules_index
{
   "mappings": {
      "doctype": {
         "properties": {
            "title": {"type": "text"}
            "body": {"type": "text"}
         }
      },
      "queries": {
         "properties": {
            "query": {
               "type": "percolator"
```

```
          }
        }
      }
    }
}
```

To register a rule, we index it on the *rules_index* created before. For example, using the REST call in Table 5, we can register the rule of Table 2 in the created index. The method of Table 5 is called by the Rules Mapper, but the actual indexing takes place in the Rules Index. The id of the query is 1 as denoted by the ES REST call. Based on the description of query decomposition presented above, we can index the query of Table 5 as a whole or split it in two queries that are indexed separately. In the latter approach, we use a different id for each of the sub-queries but the association to the original query is maintained by adding an extra parentID field in the rules_index. Queries with the same parentID are considered as parts of the same query.

Table 5: Addition of a query (rule) to the index (REST call refers to the ES API and should not be confused with the EXTRA API)

```
PUT /queries_index/queries/1?refresh
{
        "query": {
                "bool" : {
                        "should" : [
                                {"must" : [
                                {"term" : { "field" : "expr1" }},
                                        {"term" : { "field" : "expr2" }}
                                ]},
                                {"must" : [
                                {"term" : { "field" : "expr1" }},
                                        {"term" : { "field" : "expr3" }}
                                ]},
                        ]
                }
        }
}
```

Using the command of Table 6, we can search among the stored queries for the subset of those that match the input document. In this example, input document is defined as "*doctype",* but this can be any of the defined document mappings, based on the document schema. This procedure takes place in Rules Selector component. To enable highlighting in the response, we can add the highlight option of Table 7 in the JSON structure of Table 6. Using this option, the input document, and more specifically, the fields title and body will be highlighted inline to indicate the terms of the document that match the queries. Highlighting will take place several times, one for each matched query.

Table 6: Searching for indexed queries given an input document (REST call refers to the ES API and should not be confused with the EXTRA API)

```
GET /rules_index/_search
{
   "query" : {
      "percolate" : {
```

```
            "field" : "query",
            "document_type" : "doctype",
            "document" : {
               "title" : "This is the title of the document"
               "body" : "This is the body of the document."
            }
         }
      }
}
```

Table 7: Enabling highlight in the response of percolate query

```
"highlight": {
     "fields": {
       "title": {},
       "body": {}
     }
}
```

An additional mechanism that we plan to use to address the scalability and response time requirements for EXTRA is *index partitioning*. To this end, we consider distinct categories of rules:

1. Rules with simple terms (e.g. such as those described in appendices A.1, A.2 of the requirements document)
2. Rules having stemming and capitalization declarations
3. Rules with POS tags
4. Rules with wildcards.

Each of these categories may be indexed in a different ES index[9]. As some of the above rule categories can be quite expensive in terms of resources (for instance, rules with wildcards), it would be a good idea to keep them in separate indices, with a size much more limited compared to a single index. As a result, matching to the queries of separate indices is expected to be much faster than an index that would contain all rules together. In short, separating different rules with different characteristics across different indices can speed-up the execution of the RS step for a large part of the queries. The intuition behind this choice is that the majority of rules will be simple and will be possible to retrieve quite fast. On the other hand, more complex queries require more time per query to be matched but their total number is lower.

Document Processing

Input documents to be classified by EXTRA consist of multiple fields. There is a set of predefined fields defined by EXTRA (cf. Table 8 for an example). However the user can also define his/her own schema, save it to the DB component and validate input documents and

[9] As span queries can be quite complex and require splitting of documents into logical blocks (e.g. sentences or paragraphs), it is likely that an extra index would be required to handle this type of queries efficiently. However, the final setting will be based on experimentation.

rules against it. In case that no other schema is defined, the predefined one is used. Not all fields are required, as depicted in Table 8.

Table 8: Illustration of an example EXTRA document


```
{
   "kicker": "The Debt Crisis",
   "summary": "Italy accounts for a third of the eurozone's nonperforming loans. But that hasn't stopped its banks from extending credit to loss-making companies.",
   "headline": "Italian Banks Continue to Lend to Stagnant Companies as Debt Pile Mounts",
   "lede_graph": "In Italy, where two decades of economic stagnation have created a long line of barely breathing companies, Feltrinelli, one of the country's largest booksellers, stands out.",
   "subsection": "Europe",
   "section": "World"
}
```


Table 9: Predefined fields of input documents

Field	**Description**
Body	The text beginning with the first word in the first paragraph to the last word in the final paragraph.
Byline	The author of the asset.
Dateline	The date and location where the reporting occurred.
Kicker	A short phrase that precedes the headline and designates a collection of stories, such as an ongoing column or series.
Headline	The title of the asset.
Lede Graph	The first paragraph.
Section	A label for site navigation that groups content topically.
Subsection	A label for site navigation that sits hierarchically under section.
Summary	An abstract of 1-2 sentences that either summarizes the content of the asset, or extracts 1-2 key sentence(s) to entice the reader to read more.
Type of Material	The structural template of the asset, e.g. news article, review, editorial, op-ed, slideshow, photo, video.

To support these fields we must add the fields as properties in the mappings of rules_index (Table 4). For example in Table 4 we use two properties body and title. These properties can be extended to support the fields of the input document of Table 9. To support user-defined schemas a new mapping has to be added in the index for each new schema. The type of each property/field is defined accordingly. For example, a field could be a simple type like text, keyword, date, long, double or boolean. It is often useful to index the same field in different ways for different purposes. For instance, a string field could be indexed as a text field for full-text search, and as a keyword field for sorting or aggregations. Alternatively, one could index a string field with the standard analyzer, the English analyzer, and the French analyzer. In addition, different filters can be defined for a field to support stemming and lower case

transformation. This mechanism can be used to support rules based on POS tags, stemming and capitalization.
To make this clear let us consider the case of a simple document containing only the field body. In the first case the field is analyzed only using tokenization and a lower-case filter. For the second version (stemming and capitalization), we create another representation of the document that stems the tokens of the body and creates another field named body_C that preserves capitalization. Finally, in case of POS tags, we index not only terms but also their part-of-speech tags. These different representations are submitted as independent queries to the RS and the results are aggregated to create a single list of candidate rules.
Of particular interest in this step is the support for *concepts* (lists of entities). Instead of implementing concepts as a separate construct, at this stage, we recommend that they are implemented using rules. For instance, a concept named “SOCCER_PLAYER” would be imported in the system as a rule with the name “soccer player” and it would consist of many entries (e.g. “Cristiano Ronaldo”, “Lionel Messi”, etc.) that are connected with the OR operator. In that case, a more complex rule could refer to the rule “soccer_player” as one of its components.

Experimental Evaluation of ES Percolate

In order to test the performance of ES Percolate as a means of fast retrieval of candidate rules for an input document, we performed a set of experiments by using a set of automatically generated rules based on the Reuters corpus (Reuters-21578), which consists of 21,578 articles. These articles consist of several fields, but in our experiment we used only two of them that contain the actual content of the articles, title and body. More specifically, we first extracted named entities from these two fields, using the Stanford NLP library, including person names, organizations and locations from each article. To generate synthetic rules, we kept only 8,350 articles that had more than four entities. Each entity usually consists of 1 up to 3 terms. For each of these articles, we created k-combinations (with k from 3 up to 5) among its entities using the AND operator to link them, and then used random subsets of two combinations using the OR operator to combine them into more complex expressions. We ended up with 972,696 unique queries that serve as rules. Each of them has 15 terms and is associated to 1.12 documents on average, while each document has about 135 associated rules on average.
To make the procedure more clear let us consider the following example. Given an article that contain 5 entities *e1, e2, e3, e4, e5* we create k combinations such as *(e1, e2, e3), (e1, e2, e4), … (e1, e3, e4, e5),* etc. Next, we combine them with the AND operator. For example *(e1, e2, e3)* becomes *e1 AND e2 AND e3*. Finally, by selecting two random combinations we create the rule (e1 AND e2 AND e3) OR (*e3 AND e4 AND e5*).
For deployment we used Elastic Search on a machine having Intel Core i7-3770K processor and 16GB of RAM. We indexed 972,696 such rules using the ES percolation mechanism, and then used the set of 8,350 articles as queries. To index the documents we had to transform the combinations of named entities described above to Elastic Search queries. First we have to

map each entity to an ES query. For example given that entity e1 is the name *"ronald reagan"* we create the following query using the *multi_match* operator.

```
{
        "multi_match" : {
                "query":    "ronald reagan",
                "fields": [ "title", "body" ],
                "operator" : "and"
        }
}
```

In our example, we seek for the entity *"ronald reagan"* to appear in any of the two fields of the document, title or body. Note that in the following stages of development these queries can be quite more complex with different terms for each field. Also we define the operator to be *and* as we need both terms of the entity to appear in the text. Next we combine each of the multi_match queries using boolean queries of ES. As we want to combine them by AND we use the *must* operator as follows:

```
{
        "bool" : {
                "must" : {  // multi_match query of e1 },
                "must": {   // multi_match query of e2 },
                ……
        }
}
```

Finally, for the OR operation we use *should* operator:

```
{
        "should" : [
                { // must clauses from the first combination of entities },
                { // must clauses from the second combination of entities}
        ]
}
```

Table 10 contains some basic performance statistics. Recall value (both micro and macro[10]) is quite high indicating that the majority of associated rules per article is retrieved successfully. It is noteworthy that when we initially measured performance using the predefined analyzers of ES, the recall value was below 70%. This was fixed by having rules and articles following the same processing steps in ES.

Precision is still quite low, with micro-precision being around 25%. This means that we retrieve four times more rules than the rules that are associated with an article. However, with a close

[10] Given an input document, recall is defined as the fraction of relevant rules retrieved from ES, compared to the whole set of relevant rules. Having a set of documents to be evaluated, macro-recall, is defined as the average value of recall values calculated for each individual document. On the other hand, in micro-average method, individual true positives and false negatives for different documents are summed up and the fraction corresponds to micro-recall, is calculated based on these aggregated values.

examination of some cases we can observe that usually the retrieved rules are somehow valid but we miss that association during the rule creation. There is a set of named entities that occur quite often in the articles of Reuters corpus used for evaluation. These entities usually are countries and persons such as United States, Soviet Union, Ronald Reagan etc. Some simple rules consisted of these frequent terms could be generated by many different articles. But due to randomness in the procedure of rules generation (as for each article we keep only a random sample of the possible combinations), these rules are associated only to a subset of input articles. As this effect concerns mainly the simpler rules we expect that it will be diminished in the later stages of development as more complex rules will be used. In any case, note that an imperfect value for precision is not an issue as the outcome of this retrieval step will be used by the next module of the system that will evaluate the candidate rules exhaustively.

Table 10: Performance of ES on set of 1M rules using 8K documents as queries.

Precision	26% (micro), 77.4% (macro)
Recall	96.7% (micro), 97.5% (macro)
Response time	270 msec

To investigate the effect of number of indexed rules on rule retrieval time, we conducted a similar experiment using indexes of different size. More specifically, we created six indices, consisting of 10k, 50k, 100k, 200k and 500k rules, by selecting random rules from the initial rule of 972,696 rules. For these indices we calculated the average response time, which is depicted in Figure 2. Surprisingly, response time seems to be higher for smaller indices. For example, for 10k indexed rules the average response time is 340 msecs, while for the whole index of 972,696 rules, the response time was around 270 msecs. Our assumption is that this effect has to do mainly with the built-in caching mechanism of ES. Its influence seems to be greater for larger indices, where there are many rules that match with many documents. Also note that for indices larger than 200k rules response time seems to be constant. In all cases, the measured response time is considered satisfactory, as its is far below the requirement of 1 second per rule.

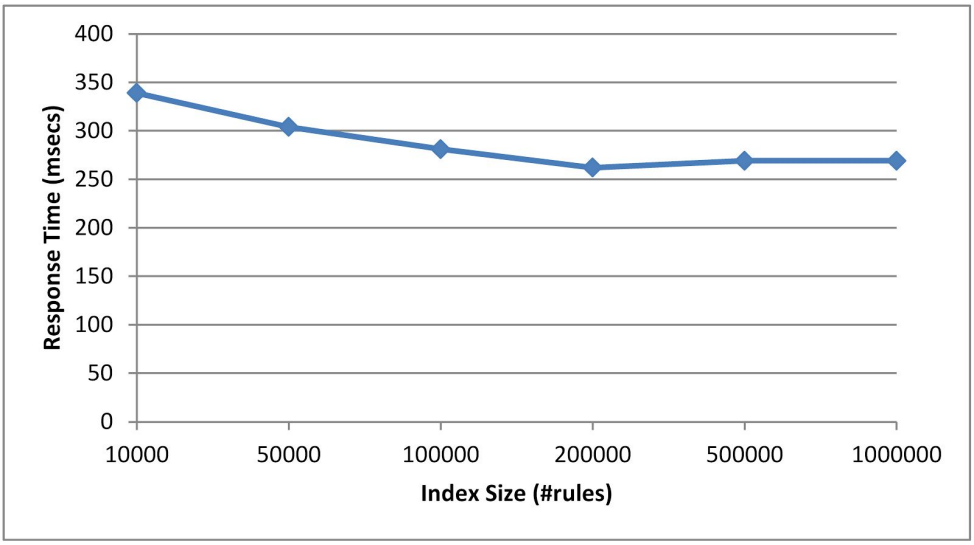

Figure 2: Average response time compared to number of indexed rules.

We also investigated the influence of rule complexity on response time. Complexity can be measured using several metrics. Given that a rule can be represented as an abstract syntax tree, other factors as tree depth and width can constitute measures of complexity. In our case, we chose to define complexity in a simpler way by counting the number of terms in a rule. More specifically, following a similar approach as described in the first paragraph of this section, but the outer OR combination may consist of more than two parts. Doing this we were able to create sets of rules having a varying number of terms, from 20 up to 100 terms combined with AND/OR, and we clustered these rules into five distinct groups (<20, 20-40, 40-60, 60-80, >100 terms). Each group, having 20k rules each, was indexed in a separate ES index. Response time is depicted in Figure 3. As shown in the figure, response time is higher than the initial set of rules, and is increased steadily as the number of terms increases. However this rise cannot be considered significant, as from 10 to 100 terms response time was increased by less than 15 milliseconds.

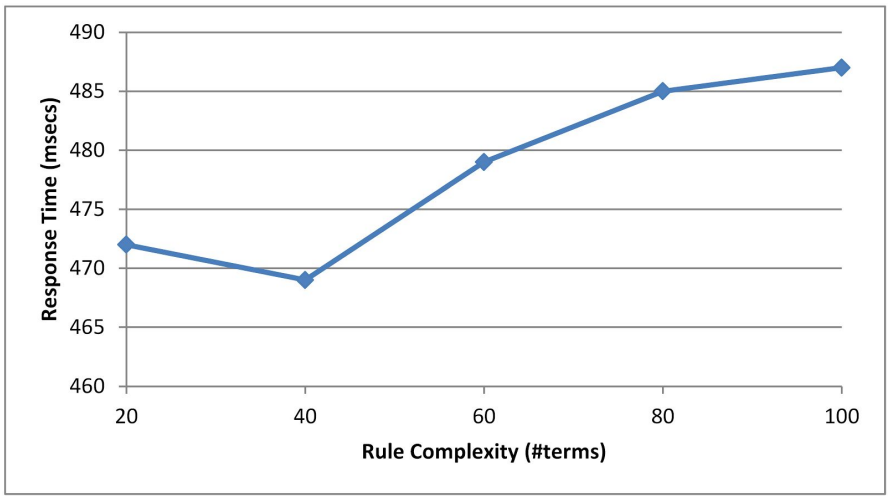

Figure 3: Average response time compared to number of terms in rules.

A conclusion that can be drawn from the latest experiment has to do with the affection operators in response time. As shown in Figure 3, the response time is higher compared to all cases of Figure 2. This has to do mainly with the fact that rules of that experiment consist of multiple OR clauses. These clauses are more time consuming compared to AND clauses as each subpart has to be tested and evaluated. On the other hand AND clauses are evaluated much quicker as one missing part is enough to stop examination of the remaining parts. As a result, we conclude that normalization / simplification of rules is an important part of EXTRA Rule Engine.
Finally, we investigated the impact of document length to response time. As documents may vary at length, and some of them may be quite long, we would like to know how this parameter influences performance. We grouped input documents based on the number of characters into 11 groups. For each group we generate box-plots of response time, depicted in Figure 4. As revealed by the figure, ignoring deviations and outliers, the mean value of response time increases as the document length increases.

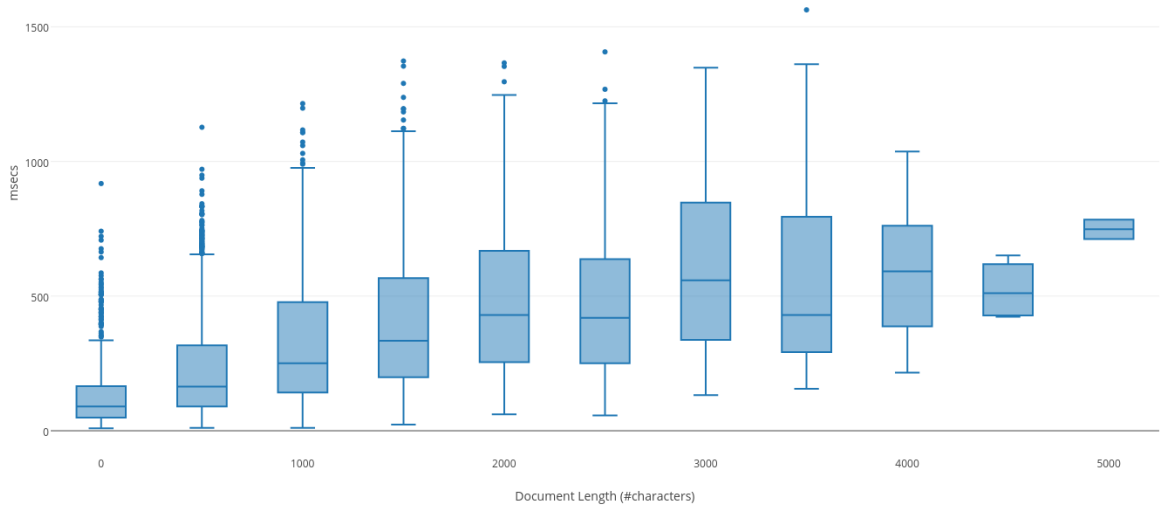

Figure 4: Response Time compared to document length

EXTRA User Interface

A simple and easy to use web-based user interface will be developed on top of the EXTRA API that will support the manual management of EXTRA rules. The basic User Stories described in Section 5 of the requirements document will be supported through this user interface, namely:

- Rule management (creation/selection/update/deletion/validation)
- Document classification
- Schema management (addition/update/deletion)
- Dictionary management (addition/update/deletion)
- Relevance algorithm management (addition/update/deletion)
- Hit highlighting

For most of the above features, standard UI elements will be used (lists, tree views, etc.) in accordance with best practices for presenting and editing tree-structured documents. In addition, intuitive widgets will be developed to make some of the tasks less burdensome for end users.

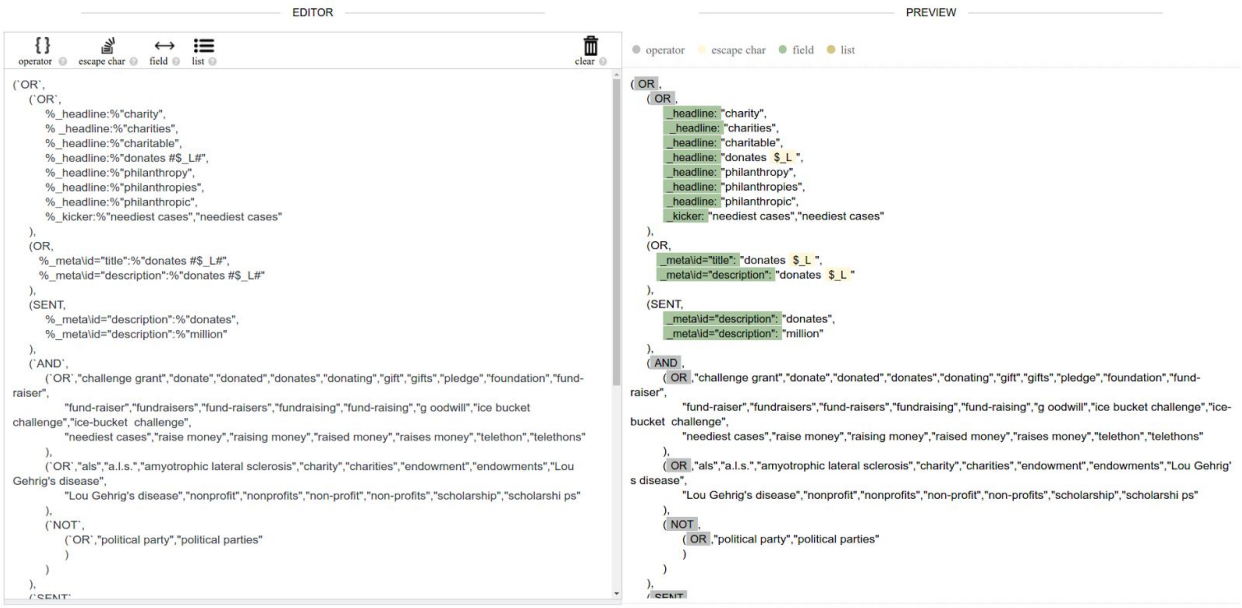

Figure 5: *Screenshot from UI developed by the team where queries are written into an editor on the left part of the screen, and parsed rules shown on the right side.*