RNI Elasticsearch Plugin Guide

Version 7.10.2.3 and includes RNI 7.35.1.c65.0

Publication date 2021-09-27

About Basis Technology

Verifying identity, understanding customers, anticipating world events, uncovering crime. For over twenty years, Basis Technology has provided the underlying analytical components enabling businesses and governments to tackle some of their toughest problems. Our Rosette™ text analytics platform employs a hybrid of classical machine learning and deep neural nets to extract meaningful information from unstructured data. Autopsy, our digital forensics platform, and Cyber Triage, our incident response tool, serve the needs of law enforcement, national security, and legal technologists with over 5,000 downloads every week. KonaSearch delivers natural language search of every field, object, and file in Salesforce all from the same index.

Company headquarters are in Somerville, Massachusetts, with branch offices in Washington, London, Tel Aviv, and Tokyo. For more information, visit www.basistech.com.

Basis Technology Corp.
1060 Broadway
Somerville, MA 02144
T 617.386.2000
F 617.386.2020
E support@rosette.com
http://support.rosette.com

Table of Contents

1. Introduction

RNI-Elasticsearch is an Elasticsearch [1] plugin for building fuzzy name retrieval and name matching applications for persons, locations, and organizations. It uses Rosette Name Indexer (RNI), implementing high-speed, scalable, cross-language, and cross-script searches with the Elasticsearch full-text search engine to store the names and search keys.

RNI performs searches across a large set of languages and writing scripts. Refer to Supported Text Domains for Name Indexing and Name Matching [67] for the complete list of supported languages. [2]

This guide describes how to use the RNI-Elasticsearch plugin and RNI features, and is not intended to be a complete guide to Elasticsearch.

1.1. Interpreting RNI Scores

Names are complex to match because of the large number of variations that occur within a language and across languages. RNI breaks a name into tokens and compares the matching tokens. RNI can identify variations between matching tokens including, but not limited to, typographical errors, phonetic spelling variations, transliteration differences, initials, and nicknames.

RNI scores range from 0 to 1. The higher the score, the greater the confidence that this a relevant match. A score of 1.0 indicates that the query name string and result name string are identical (including all name properties).

The match score is a relative indication of how similar the match is; it is not an absolute value. When comparing different name matches, the relative matches of the scores are more relevant than the actual score. Similar name matches in different languages may generate different match scores. To understand how RNI calculates the score, see Understanding Name Match Scores [16].

Scores less than 1.0 for similar names indicate the query name and index name vary with respect to one or more properties (such as language of origin) and/or one or more of the following:

Variation	Example(s)
Phonetic and/or spelling differences	*Nayif Hawatmeh* and *Nayif Hawatma*
Missing name components	*Mohammad Salah* and *Mohammad Abd El-Hamid Salah*
Rarity of a shared name component	Two English names that contain *Ditters* are more likely to match than two names that contain *Smith*
Initials	*John F. Kennedy* and *John Fitzgerald Kennedy*
Nicknames	*Bobby Holguin* and *Robert Holguin*
"Cousin" or cognate names	*Pedro Calzon* and *Peter Calzon*
Uppercase/Lowercase	*Rosa Elena PACHECO* and *Rosa Elena Pacheco*
Reordered name components	*Zedong Mao* and *Mao Zedong*
Variable Segmentation	*Henry Van Dick* and *Henri VanDick*, *Robert Smith* and *Robert JohnSmyth*
Corresponding name fields	For [Katherine][Anne][Cox], the similarity with [Katherine][Ann][Cox] is higher than the similarity with [Katherine Ann][Cox]
Truncation of name elements	For *Sawyer*, the similarity with *Sawy* is higher than the similarity with *Sawi*.

[1] Copyright by Elasticsearch BV. Dual licensed under Server Side Public License (SSPL version 1) and the Elastic License 2.0 (ELv2).
[2] The Java-only version of the plugin only supports English, French, German, Italian, Portuguese, and Spanish.

Scoring is commutative: the scores for two given names are always the same, regardless of which name is in the index and which name is in the query.

You can configure [18] RNI to customize how it scores different matching phenomena.

The score weighting associated with a token may vary depending on the token's characteristics, such as the frequency with which it appears in the language model (the more frequent, the lower the weighting).

2. Installing RNI-Elasticsearch

IMPORTANT
The RNI Elasticsearch plugin does not work with the AWS managed elastic service.

To use RNI-Elasticsearch you need the RNI Elasticsearch plugin, an RLP license file (`rlp-license.xml`) and Elasticsearch. [3]

NOTE
If you are using the Linux distribution of RNI-ES, note that glibc is required. The version of glibc that the native libraries are built against can be found in the filename of the distributed package.

1. If you do not already have it, install Elasticsearch.
   Download and unzip Elasticsearch-<version>.zip.

   

   IMPORTANT
   The version of Elasticsearch must match the first three digits of the version of the RNI-ES plugin. If your version of Elasticsearch does not match the plugin version, the plugin will not install.

   Example:

   - Elasticsearch version: 7.5.2
   - RNI-ES plugin version: 7.5.2.x where x is an integer

2. Install the plugin.
   Navigate to the `elasticsearch-<version>` root directory and run the **install** command.

[3] For RNI plugins that support earlier versions of Elasticsearch (such as 1.x.y), contact support@rosette.com.

On Unix, Linux and MacOS:

```
bin/elasticsearch-plugin install file:///path/to/rni-es-<version>.x.zip
```

On Windows:

```
bin\elasticsearch-plugin install file:///C:\path\to\rni-es-<version>.x.zip
```

NOTE

You must use the absolute file path to refer to the plugin zip file. For example, if the file is in the home directory of rniUser on macOS, the command would be:

```
bin/elasticsearch-plugin install file:///Users/rniUser/rni-es-<version>.x.zip
```

You may be prompted to grant permissions necessary for the plugin to function.
The plugin is now in `plugins/rni`.

NOTE

For Windows users, you must add

```
bin\elasticsearch-<version>\plugins\rni\bt_root\rlp\bin\*
```

to your PATH environment variable. In this case, you must replace * with the name of the subdirectory which contains platform-specific binary library files (for example, `amd64-w64-msvc120`).

Additionally, the RNI-Elasticsearch plugin cannot be installed into distributions of Elasticsearch found in the `C:\Program Files` directory.

3. Copy the RLP License (`rlp-license.xml`) to `plugins/rni/bt_root/rlp/rlp/licenses`. This license must be in place before you can use the RNI-Elasticsearch plugin.

NOTE

If your index contains complex mappings or searches, including many fields or nested fields, you may need to increase the heap size as described in the Elasticsearch documentation.

To start the Elasticsearch server, run:

```
bin/elasticsearch
```

NOTE
When starting Elasticsearch with the plugin you may see some non-fatal error messages. If a message follows the error stating that "Cluster health status changed from [RED] to [YELLOW]", the error can be ignored. This may occur when the `enableDynamicConfiguration` is set to `true`.

2.1. libpostal Data Directory

RNI uses libpostal to parse addresses; libpostal is a C library for parsing/normalizing street addresses around the world using statistical NLP and open data.

RNI packages libpostal data in `plugins/rni/bt_root/rlpnc/data/libpostal`. The data directory is relatively large (~2G). If you are certain that you won't be utilizing address matching of unfielded addresses, you can safely delete the libpostal data directory without impacting any other RNI-ES functionalities.

3. Prepare the Index

Elasticsearch provides real-time search and analytics for all kinds of data. The data is stored in documents, each having a set of fields, some of which are defined as search fields. An Elasticsearch index is a collection of these documents.

The RNI-Elasticsearch plugin uses an Elasticsearch index to store documents containing names, dates, addresses, or other fields to be matched.

Before using RNI to search for matches, you must create the index, define mappings, and load the index with documents.

1. Create an Index [4], or a searchable container for your documents.
2. Define a Mapping [5] for fields that contain person, location, organization, or identifier entity types. The type of a name field to be searched by RNI is `"rni_name"`. A mapping defines the data types of each of the searchable fields in a document. The mapping does not have to include every field in the document, just the searchable fields.
3. Index Documents [5] that contain one or more name fields along with other fields of interest. This step loads the documents into the index.
4. Test the RNI integration before continuing on.

Once you've completed the above steps, you are ready to query the index [9].

The following snippets use the cURL command-line tool to illustrate the Elasticsearch commands for running the plugin. You can also use Kibana, an open source dashboard for Elasticsearch.

3.1. Create an Index

An Elasticsearch index consists of one or more documents, and a document contains one or more fields. A name index is an indexed list of names.

The default port for running Elasticsearch locally is `localhost:9200`.

The following cURL statement creates an index named **rni-test**.

```
curl -XPUT 'http://localhost:9200/rni-test'
```

3.2. Define a Mapping

A mapping defines how a document, along with the fields it contains, is stored and indexed and sets the types of the search fields. For name search fields, set the **"type"** of the name fields to **"rni_name"**.

The following statement maps the **"primary_name"** and **"aka"** (also known as) fields in the document to the **"rni_name"** type in the **"rni-test"** index.

```
curl -XPUT 'http://localhost:9200/rni-test/_mapping' -H'Content-Type: application/json' -d '{
    "properties" : {
        "primary_name" : { "type" : "rni_name" },
        "aka" : { "type" : "rni_name" },
        "occupation" : { "type" : "text" }
    }
}'
```

3.3. Index Documents

This is the step where you add your data, or documents, to the index. A document is a JSON object containing one or more fields. Each field in a document is defined as a key-value pairs, where the key is the field and the value is the data.

Documents may include fields other than name fields.

```
curl -XPUT 'http://localhost:9200/rni-test/_doc/1' -H'Content-Type: application/json' -d '{
    "primary_name" : "Joe Schmoe",
    "aka" : "Bossman",
    "occupation" : "business owner"
}'
```

Name fields can include properties in addition to the name string (or "`data` " property). Properties are used when searching to optimize the search algorithms for the data. The "`entityType`" property is particularly important for name searching and customizations.

Property	Required	Description
`"data"`	✓	The name string.
`"language"`		ISO 639-3 Code for the language of use: the language of the document in which the name was found.
`"languageOfOrigin"`		ISO 639-3 Code for the language of origin of the name. For example, a name of Spanish origin (spa) may be found in an English (eng) document.
`"script"`		ISO 15924 code for the script.
`"entityType"`		Type of the name.
`"uid"`		Unique string identifier for the document.

Example:

```
curl -XPUT 'http://localhost:9200/rni-test/_doc/3' -H'Content-Type: application/json' -d '{
    "primary_name" : {
```

```
        "data" : "Joe Schmoe",
        "language" : "eng",
        "script" : "Latn",
        "entityType" : "PERSON"
    }
}'
```

TIP

When creating a large set of documents, use the Bulk Insert [7] for optimal performance.

3.3.1. Entity Types

RNI uses the `entityType` field to identify the type of name being matched and to optimize the algorithms used for matching. Where supported, stop words and override files are specific to an entity type.

IMPORTANT

The `entityType` should always be specified to utilize all RNI features when indexing and matching names. If you don't specify an `entityType`, the type `NONE` will be used and RNI may return less accurate results.

Entity Types

Type	Description	Features
PERSON	A human identified by name, nickname, or alias.	Values are tokenized and token pairs are compared. Stop words, overrides, frequency and gender models are supported.
LOCATION	A city, state, country, region or other location.	Values are tokenized and token pairs are compared. Stop words, overrides, and frequency models are supported.
ORGANIZATION	A corporation, institution, government agency, or other group of people defined by an established organizational structure.	Values are tokenized and token pairs are compared. Stop words, overrides, frequency models, and embeddings are supported.
IDENTIFIER IDENTIFIER:DRIVERS_LICENSE IDENTIFIER:LICENSE_PLATE IDENTIFIER:NATIONAL_ID_NUM	An alphanumeric identifier.	Values are not tokenized. The entire identifier is treated as a string. Scoring is primarily by string edit distance.

3.3.2. Fielded Names

You can process fielded names by separating the fields with "|". RNI assigns no explicit semantics to each field (such as given name or surname), but it does pay attention to the order of the fields when comparing two fielded names. RNI assigns lower scores to matches that cross field boundaries (e.g., the first field in name1 matches the second field in name2). Fields within a name can be empty.

When scoring a potential match between a name with fields and a name without fields, RNI treats the name without fields as if it were a name with a single field.

RNI treats trailing empty fields as if they were not present. For example "Rosanne|Taylor Smith|" is treated the same as "Rosanne|Taylor Smith".

Alternatively, you have the option of specifying that there is an unknown value in a field. To specify an unknown name field, replace the field with `*?*`.

3.3.3. Names Containing Special Characters

When using JSON objects with RNI, special characters must be properly escaped when used in strings. RNI requires a backslash to escape the special character and then JSON requires another backslash to escape the first backslash. Thus, In RNI, the proper escape character for names containing a special character is a double backslash (\\).

The | used in fielded names [6] is one example of a special character embedded within a name, where | is used to separate the fields. For proper processing of the vertical bar character, RNI needs to be able to distinguish when the user intends to build a fielded name and a name which contains the vertical bar character.

Let's assume we have a name that includes a |; it is not indicating a fielded name: "John|Smith". RNI requires that you escape the vertical bar with a backslash; e.g. "John\|Smith". Then, JSON requires that the backslash character be escaped with a backlash. The correct syntax for the name "John|Smith" is "John\\|Smith". If the entry were representing a fielded name, the correct syntax would be "John|Smith" without any backslashes.

3.4. Verify the RNI SDK Version

To verify the version of the RNI SDK being used by the plugin, send a GET request to {index_name}/rni_plugin/_get_version:

```
curl -XGET 'localhost:9200/rni-test/rni_plugin/_get_version'
```

This call also verifies that the RNI plugin is installed and running successfully.

3.5. Bulk Insert

Bulk insert allows you to add multiple documents to Elasticsearch in a single API call, improving the throughput for uploading documents by orders of magnitude. We recommend you use bulk indexing to create and index your data wherever possible.

1. Create the index.
2. Define the mapping.
3. Run Bulk Insert.

TIP

Do not perform any queries or searches on the cluster while indexing data via the bulk index API. Doing so can cause significant performance issues.

The structure for all Elasticsearch bulk API calls is:

```
{ action_to_be_performed: { metadata_related_to_action}}\n
{ request_body_data_to_index }\n
```

3.5.1. Bulk Insert Example

We're going to continue the example that we started. The index is `rni-test`. The mapping [5] defines a `primary_name`, `aka`, and `occupation`.

1. Create the index.

   ```
   curl -X PUT http://localhost:9200/rni-test
   ```

2. Define the mapping.
   The previously defined mapping:

   ```
   {
       "properties" : {
          "primary_name" : { "type" : "rni_name" },
          "aka" : { "type" : "rni_name" },
          "occupation" : { "type" : "text" }
       }
   }
   ```

   You can put the mapping in a JSON file and create it from the command line. The following curl command creates the mapping using a file (`es_mapping.json` in this example):

   ```
   curl -X PUT -H"Content-Type:application/json" -d @es_mapping.json http://localhost:9200/rni-test/_mapping
   ```

3. Create a data file in newline delimited JSON (NDJSON) format. Save the file as `bulknames.json`. The file MUST end with a newline after the final record.

   ```
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "Joaquín Guzmán","entityType":"PERSON"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "René Lindström Jones","entityType":"PERSON"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "Guadalupe Hernandez","entityType":"PERSON"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "Chris Joseph Arsenault","entityType":"PERSON"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "ABC","entityType":"ORGANIZATION"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "Basis Technology","entityType":"ORGANIZATION"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "Australian Boradcasting Corporation","entityType":"ORGANIZATION"}}
   {"index":{"_index":"rni-test","_id":null}}
   {"primary_name":{"data": "Amazon","entityType":"ORGANIZATION"}}
   ```

4. Use the `_bulk` method to load the data file with curl using the following command:

   ```
   curl -X POST -H"Content-Type:application/json" --data-binary @bulknames.json http://localhost:9200/rni-test/_bulk
   ```

NOTE
If you're providing text file input to curl, use the `--data-binary` flag instead of plain `-d` to preserve the newlines.

4. Searching with Queries

At this point you've created an index and loaded data. Now you can start using RNI to search for matches.

A query searches the index and returns a match score. In RNI, the query for a name consists of two parts, a base query [9] and a rescorer [10].

The base query is a standard Elasticsearch query against a name field. The rescorer takes the results of the base query, and uses Elasticsearch rescoring to select the top candidates and perform pairwise matching on the top candidates.

The query returns an RNI match score (`max_score`), the score of the top scoring document.

IMPORTANT
The `entityType` (PERSON, LOCATION, ORGANIZATION) should always be added to a name query to utilize all RNI features. If you don't specify an `entityType`, the type `NONE` will be used and RNI may return less accurate results.

4.1. Base Query

The base query is a standard query against a name field:

```
   "query" : {
       "match" : {
           "primary_name" : "Jo Shmoe"
       }
   }
```

Querying supports the same name properties [5] that you may use when indexing documents. Unlike during document creation, you must pass the JSON object containing the name fields as a string. You should always include the `entityType` property in your query.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "query" : {
       "match" : {
           "primary_name" : "{\"data\" : \"Jo Shmoe\", \"entityType\" : \"PERSON\"}"
       }
   }
}'
```

Much like during indexing, RNI creates a set of keys based on the name and then generates a more complex internal query to match against the indexed keys.

4.2. Rescore with the RNI Pairwise Name Match

The base query returns a ranked list of matching documents. The rescorer takes the top documents from the list and performs pairwise matching algorithms on those documents, and returns a re-ranked list. RNI has a custom rescorer which allows you to further tune the candidates passing to RNI pairwise matcher. Since the pairwise matcher is a computationally intensive process, you want to rescore just enough documents to find the best matches.

Elasticsearch Rescoring includes the following parameters:

- `window_size` (an integer, defaults to 10) specifies how many documents from the base query should be passed to the RNI pairwise matcher.
  Use this parameter to limit the number of compute-intensive name matches that need to be performed. If you set the value too high, the query will take too long, but if you set the value too low, you will increase the number of false negatives.

TIP
A good starting point for `window_size` is to make it the square root of the size of the index. For example, an index of 10,000 entries would use a `window_size` of 100.

- `query_weight` (a float, defaults to 1.0) specifies the weighting of the score returned by the base query.
  In the context of RNI pairwise matching, the base query score has little meaning, so we suggest you set it to 0.0.

- `rescore_query_weight` (a float, defaults to 1.0) specifies the weighting of the maximum RNI pairwise match score.
  If `query_weight` 0.0 and `rescore_query_weight` is 1.0, the score that is returned by rescoring is the RNI pairwise match score [1].

- `score_mode` controls how the query and rescore query scores are combined. The default value is `total` meaning that both scores are added together after being multiplied by their respective weights.

In the following example, pairwise matching is performed on the top 200 names returned by the base query.

Example with RNI Rescorer:

```
"rescore" : {
    "window_size" : 200,
    "query" : {
        "rescore_query" : {
            "function_score" : {
                "name_score" : {
                    "field" : "primary_name",
                    "query_name" : {"data" : "Jo Shmoe", "entityType":"PERSON"}
                }
            }
        },
        "query_weight" : 0.0,
        "rescore_query_weight" : 1.0
    }
}
```

The "name_score" function matches every name in the given field against the query name and returns the maximum score to the rescorer.

The "name_score" function score query must be given at least one object that specifies:

- field: the search field being rescored which must be of type rni_name.
- query: the value of the search field.

The object passed to the name_score function can also include any of the name properties [5].

This example illustrates the full query incorporating both match and rescore, using RNI query parameters.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "match" : {
            "primary_name" : "{\"data\" : \"Jo Shmoe\",\"entityType\" : \"PERSON\"}"
        }
    },
    "rescore" : {
        "window_size" : 200,
        "query" : {
            "rescore_query" : {
                "function_score" : {
                    "name_score" : {
                        "field" : "primary_name",
                        "query_name" : {"data" : "Jo Shmoe", "entityType":"PERSON"}
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

This query returns an RNI match score against **"Joe Shmoe"** in the **"_score"** field:

```
{
  "_index": "rni-test",
  "_type": "_doc",
  "_id": "1",
  "_score": 0.80217975,
  "_source": {
    "primary_name": "Joe Shmoe",
    "aka": "Bossman",
    "occupation": "business owner"
  }
}
```

4.2.1. Advanced Rescorer

RNI includes a customized RNI rescorer query parameter, rni_query, which utilizes RNI advanced features. The RNI custom rescorer uses the parameters above as well as the following parameters to determine the number of documents that will be rescored. Rescoring fewer documents increases speed, but can be at the cost of accuracy if the best documents are not passed to the rescorer.

- score_to_rescore_restriction (a float, defaults to 0.4, cannot be negative) dynamically controls the minimum query score a document needs to be passed to the RNI rescorer.
  A value of 0.0 will not cut off any documents from being rescored. Higher values rescore fewer documents, increasing speed at the cost of accuracy.

- `window_size_allowance` (a float, defaults to 0.5, must be in interval (0, 1]) dynamically controls the window size for rescoring. No more than `window_size` names will be scored.
  A value of 1.0 will not cut off any documents from being rescored. Higher values rescore more documents, increasing accuracy at the cost of speed.
- `score_if_null` (a float, defaults to -1.0 indicating the feature is off). Set to a value between 0 and 1 to enable it. When set, that value is returned when the field is missing from the document.

NOTE
When using the advanced rescorer, the default value for score_mode is `max`, not `total`. In this mode, the maximum of the original score and rescore query score is used.

In the following example, pairwise matching is performed on the top 200 names returned by the base query.

Example with the Advanced Rescorer

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "match" : {
            "primary_name" : "{\"data\" : \"Jo Shmoe\", \"entityType\" : \"PERSON\"}"
        }
    },
    "rescore" : {
        "window_size" : 200,
        "rni_query" : {
            "rescore_query" : {
                "rni_function_score" : {
                    "name_score" : {
                        "field" : "primary_name",
                        "query_name" : {"data" : "Jo Shmoe", "entityType":"PERSON"},
                        "score_to_rescore_restriction": 1.0,
                        "window_size_allowance": 0.5
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

4.3. Representing Arrays in Elasticsearch

If the name field in your documents is structured as an array, such as first name and last name fields, wrap the field in a nested object. The `nested` datatype allows arrays of objects to be indexed and queried independently of each other.

Since Elasticsearch flattens object hierarchies into a simple list of field names and values, if you don't use the `nested` type, you can lose the relationship between the fields. For example, the following document:

```
  "names" : [
    {
      "first" : "Joe",
      "last" :  "Smith"
    },
    {
      "first" : "Mike",
      "last" :  "Shmoe"
    }
  ]
```

would be transformed internally into a document that looks more like this:

```
{
  "names.first" : [ "mike", "joe" ],
  "names.last" :  [ "smith", "shmoe" ]
}
```

The `names.first` and `names.last` fields are flattened into multi-value fields, and the association between **Joe** and **Smith** is lost. This document would incorrectly match a query for **mike** and **smith**.

If you wrap an array field in a nested object, you will get more accurate search results.

Include a field of type "`nested`" containing the name field in the mapping:

```
  "nested_names" : {
      "type" : "nested",
      "properties" : {
          "name" : { "type" :"rni_name" }
      }
  }
```

Multiple names can be added to the nested field:

```
{
    "nested_names" : [
        {
            "name" : "Joe Smith"
        },
        {
            "name" : "Mike Shmoe"
        }
    ]
}
```

Update the query to refer to the nested object. Set the "`score_mode`" to "`max`".

```
{
    "query" : {
        "nested" : {
            "path" : "nested_names",
            "query" : {
                "match" : {
                    "nested_names.name" : "Mike Shmoe"
                }
            }
        }
    },
    "rescore" : {
        "query" : {
            "rescore_query" : {
                "nested" : {
                    "path" : "nested_names",
                    "score_mode" : "max",
                    "query" : {
                        "function_score" : {
                            "name_score" : {
                                "field" : "nested_names.name",
                                "query_name" : "Mike Shmoe"
                            }
                        }
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}
```

Please see the Elasticsearch documentation for more detailed information on nested objects and queries.

4.3.1. Nested Names Example

Let's consider an example of a database that includes alias names along with a primary name.

Nested Mapping:

```
"properties" : {
   "primary_name" : {"type" : "rni_name"},
   "aliases" : {
      "type" : "nested",
      "properties" : {
        "alias_name" : { "type" : "rni_name" }
      }
   }
}
```

The curl command to create the mapping:

```
curl -XPUT "http://localhost:9200/rni-test/_mapping" -H 'Content-Type: application/json' -d '{
    "properties" : {
        "primary_name" : { "type" : "rni_name"},
        "aliases" : {
            "type" : "nested",
            "properties": { "alias_name" : { "type" : "rni_name" }
         }
       }
    }
}'
```

Each record includes a primary name. Each primary name can have multiple aliases.

```
"primary_name" : "John Smith",
"aliases": [
  {"alias_name": "John Shark"},
  {"alias_name": "Smithy"},
  {"alias_name": "Johnny boy"}
]
```

The curl command to add the data:

```
curl -XPUT "http://localhost:9200/rni-test/_doc/null" -H 'Content-Type: application/json' -d '{
    "primary_name" : "John Smith",
    "aliases" : [
        {"alias_name": "John Shark"},
        {"alias_name": "Smithy"},
        {"alias_name": "Johnny boy"}
    ]
}'
```

The query will try to match one of the aliases. Specify `score_mode: max` to return the highest match score of the aliases.

```
curl -XGET "http://localhost:9200/rni-test/_search" -H 'Content-Type: application/json' -d '{
    "query" : {
        "nested" : {
            "path" : "aliases",
            "query" : {
                "match" : { "aliases.alias_name": "Johnny" }
            }
        }
    },
    "rescore" : {
        "query" : {
            "rescore_query" : {
                "nested" : {
                    "path" : "aliases",
                    "score_mode" : "max",
                    "query" : {
                        "function_score" : {
                            "name_score" : {
                                "field" : "aliases.alias_name",
                                "query_name" : "Johnny"
                            }
                        }
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

4.4. Sorting Results by rni_name

Elasticsearch supports the ability to sort search results by the values of their document fields. In the case of RNI, one may want to sort on an rni_name field. Because these fields are internally composed of many subfields, it is necessary to specify the subfield to sort on. Below are a couple of subfields that you may be interested in:

IndexFields enum value	Raw subfield name	Example for "John' Smith"	Explanation
ORIGINAL_NAME_FIELD	bt_rni_name_original	John Smith	original input data for the name
NORMALIZED_DATA_FIELD	bt_rni_name_normalized	john smith	normalized name

As an example, if your field's name is **primaryName**, you can sort on the original name data by referring to `primaryName.bt_rni_name_original` in your sort specification.

In the Java API, these fields can be referenced through the `IndexFields` enum. Regarding the previous example, one could refer to the same subfield in Java:

```
"primaryName." + IndexFields.ORIGINAL_NAME_FIELD.fieldName()
```

5. Understanding Name Match Scores

To fully understand the name match scores, you need to understand how the match scores are determined. The match score is a value between 0.0 and 1.0; the higher the score, the stronger the match. The score is a relative indication of how similar the two names are; it is not an absolute value. When comparing different name matches, the relative values of the match scores are more relevant than the actual score. Similar name matches in different languages may generate different match scores.

A value of 1.0 is returned if and only if the two names are identical. Character strings, languages, languages of origin, and entity types must all match for the two names to be considered identical.

Calculating the match score is a complex process that involves multiple steps and algorithms.

1. Identify and normalize the tokens [16] in each name. Each name will usually have multiple tokens.
2. Compare each token from name 1 with each token from name 1, calculating [17] the score for every token pair.
3. Once all the token pairs have been scored, the best combination of tokens is selected to maximize the complete score.
4. Score unmatched tokens as deletions or conflicts [17].
5. Compute a weighted average score [18].
6. Adjust [18] the final score. For example, the score is decreased if the gender of the two names does not appear to match.

The Pairwise Match Endpoint [60] REST endpoint performs a pairwise match between two names, and returns detailed information about how the match scores were determined. You can use Parameter Universe [20] and the Parameters [58] REST endpoint to modify parameters and using the information returned by the pairwise match to determine the optimal parameter set for your use case.

5.1. Tokenize, Normalize, and Transliterate

Before any matching algorithms can be run, the names have to be transformed into tokens that can be compared. This step, as with many of the steps in name matching, often has language-specific components.

This step includes:

- Removing stop words, such as Mr. or Senator or General. Stop words are language-dependent.
- Transliterating into English and/or translating if necessary, including:
  - Adding vocalizations, or vowels in the correct location for languages such as Arabic which are often written in an unvocalized form.
  - Adding spaces (segmentation) to languages such as Chinese, Japanese, Korean, and Thai that don't use spaces, separating given and surname tokens.
- Normalization, including removing diacritical marks, to get a canonical representation of the token.

The resulting token output enhances search accuracy and increases relevancy.

5.2. Calculate Scores for Token Pairs

Once the base query is completed, the rescorer selects the names to send to the pairwise matcher. The pairwise matcher takes the query name (Name 1) and performs a pairwise match against each candidate document passed on (Name 2).

Every token in Name 1 is matched and scored against every token in Name 2 to find the matching pairs that will result in the highest total score for the pair. All candidate token pairs are scored to determine the best match alignment.

Token scorers are modular, allowing different scorers to be chosen for each token pair. The choice of scorer applied will depend on the type of tokens, the type of matches, and the languages of the names.

There are multiple types of scorers used, including:

- Fuzzy Matcher
- Nicknames and Cognates
- Initials
- Truncation
- String Similarity
  Names that are very similar by their string edit distance (insertions, deletions, substitutions) but not considered close by the fuzzy matcher will still match.

Each matcher returns a (`ts`) score for the pair.

5.3. Score Deletions and Conflicts

To be considered a match, token pairs must score higher than both the `conflictThreshold` and `estimatedConflictOrDeletion` parameters. Tokens not part of a satisfactory pair in this regard will be considered either conflicts or deletions.

A conflict is a score given to a token pair which context suggests *should* have matched, but whose score was not high enough to be considered a match. For example, when “Johann Sebastian Bach” is matched against “Johann Ambrosius Bach”, **Sebastian** is in conflict with **Ambrosius**. The token pair is considered a conflict.

A deletion is a score given to individual tokens that are not part of successful match pairs or conflict pairs.

There are two types of deletions:

- A *deletion* is an unmatched token where its immediate surrounding tokens are matched.
- An *Out-of-order deletion* is an unmatched token where its immediate surrounding tokens are not matched.

5.4. Calculate the Weighted Score

The token pair scorer returns 2 values, a (`ts`) score and a (`cs`) score. The `ts` is the score of how well the tokens match, while the `cs` score includes the placement of the token in the score calculation. The `cs` will be lower if the tokens match but are out of order.

Each token has a weight. The weightings determine how important the token pair match is in calculating the final score. Full names are rated higher than initials, and unusual tokens get a higher weighting than more common names because it is more significant when they match. For example, Andrew is less common than John, so it gets a higher weighting. These weightings are used in calculating the final score, which is a weighted average of the `cs` scores.

5.5. Adjust the Final Score

At the end, all selected tokens and token-pairs are considered together and some final adjustments are made to the score. Examples of adjustments include:

- A penalty is applied if the two names do not appear to have the same gender. This, of course, is language-dependent.
- Sometimes there is a weighting penalty applied early in the process, which when considered at the end, in the full context of the match, is not as significant as initially determined. These values may be adjusted in the final score.

6. Configuring Name Matching

There are many ways to configure RNI to better fit your use case and data. The two primary mechanisms are by modifying match parameters and editing overrides. You can also train a custom language model.

6.1. Tuning Match Parameters

The default values of the RNI match parameters are tuned to perform well on most queries and datasets. However, every use case uses different data with distinct match requirements. You can modify match parameters to optimize match results for your data and business case.

The typical process for tuning parameters is as follows:

1. Gather a list of names to index and queries to run against them to use as a set of test data. Ideally the test data set should be big enough to reflect the diversity in your real data with at least 100 queries.
2. After indexing the data, run the queries using RNI and determine a match score threshold that appears to provide the best results.
3. Analyze the results to discover cases that RNI failed to score high enough or that RNI incorrectly scored higher than the threshold.
4. Choose a subset of these name pairs that RNI scored too low or too high that will be used as examples to tune your parameters.

5. Tune the match parameters to change the match scores of the test set of undesirable results, so that the score is correctly above or below your threshold. For name or address pairs that have to match in a specific way and are very dissimilar (eg. aliases), we recommend you add them as token or full-name overrides [28].

6. Run the large set of queries through RNI again to test that the new parameter values still return the desired matches, and not new undesired results.

6.1.1. Parameter Configuration Files

Individual name tokens are scored by a number of algorithms or rules. These algorithms can be optimized by modifying configuration parameters, thus changing the final similarity score.

The parameter files are contained in two .yaml files located in `plugins/rni/bt_root/rlpnc/data/etc`. The parameters are *defined* in `parameter_defs.yaml` and *modified* in `parameter_profiles.yaml`.

- `parameter_defs.yaml` lists each match parameter along with the default value and a description. Each parameter may also have a minimum and maximum value, which is the system limit and could cause an error if exceeded. A parameter may also have a recommended minimum (`sane_minimum`) and recommended maximum (`sane_maximum`) value, which we advise you do not exceed.

- `parameter_profiles.yaml` is where you change parameter values based on the language pairs in the match.

IMPORTANT

Do not modify the `parameter_defs.yaml` file. All changes should be made in the `parameter_profiles.yaml` file.

Do refer to the `parameter_defs.yaml` file for definitions and usage of all available parameters.

Parameter Profiles

The parameters in the `parameter_profiles.yaml` file are organized by **parameter profiles**. Each profile contains parameter values for a specific language pair. For example, matching "Susie Johnson" and "Susanne Johnson" will use the `eng_eng` profile. There is also an `any` profile which applies to all language pairs. The

Parameter profiles have the following characteristics:

- Parameter profile names are formed from the language pairs they apply to. The 3 letter language codes are always written in alphabetical order, except for English (`eng`), which always comes last. The two languages can be the same. Examples:

  - `spa_eng`

  - `ara_jpn`

  - `eng_eng`

- They can include the entity type being matched, such as `eng_eng_PERSON`. The parameter values in this profile will only be used when matching English names with English names, where the entity type is specified as PERSON.
- Parameter profiles can inherit mappings from other parameter profiles. The global `any` profile applies to all languages; all profiles inherit its values.
- The `any` profile can include an entity type; `any_PERSON` applies to all PERSON matches regardless of language.
- Specific language profiles inherit values from global profiles. The profile matching person names is named `any_PERSON`. The profile for matching Spanish person against English person names is named `spa_eng_PERSON`. It inherits parameter values from the `spa_eng` profile and the `any_PERSON` profile. The `any_PERSON` profile will not override parameter values from more specific profiles, such as the `spa_eng` profile.

IMPORTANT

Global changes are made with the `any` profile.

Any changes to address parameters should go under the `any` profile, and will affect all fields for all addresses.

Parameter Universe

A parameter universe is a named profile containing a set of RNI parameter profiles with values. Each universe has a name and can contain multiple parameter profiles, including the global `any` profile. A parameter universe profile can also include the entity type being matched, just like regular parameter profiles. Examples:

For example, the **MyParameterUniverse** universe may include the following parameter profiles:

- `"name": "MyParameterUniverse/any"` applies to all language pairs.
- `"name": "MyParameterUniverse/spa_eng"` applies to English - Spanish name pairs.
- `"name": "MyParameterUniverse/spa_eng_PERSON"` applies to all PERSON English - Spanish name pairs.

Each parameter in the profile must match the name of a parameter declared in the `parameters_defs.yaml` file, along with a value.

A parameter universe can be defined dynamically [21] or added to the `parameter_profiles.yaml` file. We recommend that you use dynamic parameter universes for testing and tuning only. For production use, add all parameter universes to the `parameter_profiles.yaml` file.

TIP
You can define multiple named parameter profiles.

Define the parameter universe in the `parameter_profiles.yaml` file. Example:

```
parameterUniverseOne/spa_eng_PERSON:
    reorderPenalty: 0.4
    HMMUsageThreshold: 0.8
    stringDistanceThreshold: 0.1
    useEditDistanceTokenScorer: true
parameterUniverseOne/eng_eng:
    reorderPenalty: 0.6
```

Using a Parameter Universe

To use a parameter universe, add it as part of the `name_score` function when querying the index. All parameter values defined in the parameter universe will be used, where appropriate.

```
curl -XPOST "http://localhost:9200/_search" -H 'Content-Type: application/json' -d'{
  "query": {
    "match": {
      "full_name": "A Ely Taylor"
    }
  },
  "rescore": {
    "window_size": 3,
    "rni_query": {
      "rescore_query": {
        "rni_function_score": {
          "name_score": {
            "field": "full_name",
            "query_name": "A Ely Taylor",
            "score_to_rescore_restriction": 1,
            "window_size_allowance": 0.5,
            "universe": "parameterUniverseOne"
          }
        }
      },
      "query_weight": 0,
      "rescore_query_weight": 1
    }
  }
}'
```

Dynamic Parameter Universes

When tuning RNI, you can use the Parameters [58] REST API endpoint to dynamically create or update a parameter universe, overriding the existing parameter values without having to restart Elasticsearch. Once the optimum values are determined for each parameter, add the parameter universe to the `parameter_profiles.yaml` file for production use.

TIP

Dynamic parameter universes are best suited for testing and tuning the RNI match parameters. Once you determine the best set of parameters, add the parameter universe to the `parameter_profiles.yaml` file for production use. Using dynamic parameter universes can slow your system down considerably.

Use the Parameters [58] endpoint to create a parameter universe, with parameters and values.

```
curl -XPOST "http://localhost:9200/rni_plugin/_parameter_universe" -H 'Content-Type: application/json' -d'{
  "profiles": [
    {
      "name": "parameterUniverseOne/spa_eng_PERSON",
      "parameters": {
        "reorderPenalty": 0.4,
        "HMMUsageThreshold": 0.8,
        "stringDistanceThreshold": 0.1,
        "useEditDistanceTokenScorer": true
      }
    }
  ]
}'
```

The name of the parameter universe is **parameterUniverseOne** and it applies to matching person names between Spanish and English.

6.1.2. Modifying Name Parameters

To start tuning the parameters, run the RNI pairwise match on the test set and look at the match reasons in the response. These match reasons will serve as a guide for which parameters to tune, which are defined in `parameter_defs.yaml`. For additional support on tuning the parameters, contact support@rosette.com.

Once you define a profile and set a parameter value, rerun the RNI pairwise match, scoring the match with the edited `parameter_profiles.yaml` file.

Commonly Modified Name Parameters

Given the large number of configurable name match parameters in RNI, you should start by looking at the impact of modifying a small number of parameters. The complete definition of all available parameters is found in the `parameter_defs.yaml` file.

Parameter	Description	Behavior
`conflictScore`	The score that is assigned to unmatched conflict tokens	Increasing leads to higher final score
`initialsConflictScore`	The score that is assigned to unmatched conflict initials	Increasing leads to higher final score
`initialsScore`	The score that is assigned to an initial matching a token	Increasing leads to higher final score
`initialismScore`	Score assigned to initialism matching a name	The score that is assigned to an initial matching a token
`stuckInitialScore`	Score applied when initial is “stuck” to previous token	Increasing leads to higher final score
`deletionScore`	Score applied to an unmatched token when surrounding tokens are matched	Increasing leads to higher final score

Parameter	Description	Behavior
`outOfOrderDeletionScore`	Score applied to an unmatched token when surrounding tokens are also unmatched	Increasing leads to higher final score
`reorderPenalty`	Penalty applied to matching tokens with different positions	Increasing leads to lower final score
`initialsDeletionPenalty`	Multiplier on token deletion score when deleted token is an initial	Increasing leads to higher final score
`genderConflictPenalty`	Penalty applied when name genders don't match	Increasing leads to lower final score
`crossLanguageGenderConflictPenalty`	Penalty applied when name genders of different languages don't match	Increasing leads to lower final score
`boostWeightAtRightEnd`	Boost applied to tokens at the right end of the name (i.e. surnames in English)	
`boostWeightAtBothEnds`	Boost applied to tokens at either end of the name (i.e. less weight for middle names in English)	
`adjustOneSideDeletionScores`	Multiplier on the token deletion score when all deleted tokens are on one side of the name	Increasing leads to higher score
`reorderCorrection`	Boost to final score if one name's tokens are a reordering of the others	Increasing leads to higher final score
`finalBias`	Helps normalize scores	Increasing leads to a higher score for ALL names

The following examples describe the impact of parameter changes in more detail.

Token Conflict Score `conflictScore`

Let's look at the two names: 'John Mike Smith' and 'John Joe Smith'. 'John' from the first and second name will be matched as well the token 'Smith' from each name. This leaves unmatched tokens 'Mike' and 'Joe'. These two tokens are in direct conflict with each other and users can determine how it is scored. A value closer to 1.0 will treat 'Mike' and 'Joe' as equal. A value closer to 0.0 will have the opposite effect. This parameter is important when you decide names that have tokens that are dissimilar should have lower final scores. Or you may decide that if two of the tokens are the same, the third token (middle name?) is not as important.

Initials Score (`initialsScore`)

Consider the following two names: 'John Mike Smith' and 'John M Smith'. 'Mike' and 'M' trigger an initial match. You can control how this gets scored. A value closer to 1.0 will treat 'Mike' and 'M' as equal and increase the overall match score. A value closer to 0.0 will have the opposite effect. This parameter is important when you know there is a lot of initialism in your data sets.

Token Deletion Score (`deletionScore`)

Consider the following two names: 'John Mike Smith' and 'John Smith'. The name token 'Mike' is left unpaired with a token from the second name. In this example a value closer to 1.0 will not penalize the missing token. A value closer to 0.0 will have the opposite effect. This parameter is important when you have a lot of variation of token length in your name set.

Token Reorder Penalty (`reorderPenalty`)

This parameter is applied when tokens match but are in different positions in the two names. Consider the following two names: 'John Mike Smith', and 'John Smith Mike'. This parameter will control the extent to which the token ordering ('Mike Smith' vs. 'Smith Mike') decreases the final match score. A value closer to 1.0 will penalize the final score, driving it lower. A value closer to 0.0 will not penalize the order. This parameter is important when the order of tokens in the name is known. If you know that all your name data

stores last name in the last token position, you may want to penalize token reordering more by increasing the penalty. If your data is not well-structured, with some last names first but not all, you may want to lower the penalty.

Right End Boost/Both Ends Boost (`boostWeightAtRightEnd`,`boostWeightAtBothEndsboost`)

These parameters boost the weights of tokens in the first and/or last position of a name. These parameters are useful when dealing with English names, and you are confident of the placement of the surname. Consider the following two names: "John Mike Smith' and 'John Jay M Smith'. By boosting both ends you effectively give more weight to the 'John' and 'Smith' tokens. This parameter is important when you have several tokens in a name and are confident that the first and last token are the more important tokens.

6.1.3. Ignore malformed and null value parameters for RNI types

You can update the behavior of RNI to index documents with unsupported languages by updating the `ignoreBadData` parameter and you can also index null values by updating the `allowNullValue` parameter. By default, these parameters are disabled. If `ignoreBadData` parameter is enabled, any document containing a name of an unsupported language will be successfully indexed but search capabilities will be limited to supported languages (the same applies when `allowNullValue` parameter is enabled and we are dealing with documents that contain null values). These features are useful when performing bulk operations in Elasticsearch.

The file name is `parameter_profiles.yaml`, located in `plugins/rni/bt_root/rlpnc/data/etc/`.

To turn any of these features on, set the value of the parameter `ignoreBadData` or `allowNullValue` in the above file to `true`.

6.1.4. Evaluating Parameter Configuration

To evaluate the newly tuned parameter values, query a large dataset of names or addresses that does not include your test set. For an exact evaluation, query an annotated dataset that includes the correct answers for a number of queries. For a general evaluation, measure the number of pair matches that have scores above your threshold, compared to before tuning the parameter values. If there were too many matches before, now there should be fewer matches. If there were too few matches before, there should be more now. If the number of matches increases or decreases dramatically, then there is a higher chance of missing correct matches below the threshold or including incorrect matches above the threshold.

If you find new pair matches that you want to score above or below your threshold, collect them into a test set to retune the parameters. Then evaluate the parameters again using a large dataset to review results. It is important to frequently evaluate new parameter settings on separate test data to ensure the parameters continue to return correct results.

6.2. Configuring Name Overrides

RNI includes override files (UTF-8 encoded) to improve name matching. There are different types of override files:

- Stop patterns and stop word prefixes designate name elements to strip during indexing and queries, and before running any matching algorithms.
- Name pair matches specify scores to be assigned for specified full-name pairs.
- Token pair overrides specify name token pairs that match along with a match score.
- Token normalization files specify the normalized form for tokens and variants to normalize to that form.

- Low weight tokens specify parts of names (such as suffixes) that don't contribute much to name matching accuracy.

The name matching override files are in the `plugins/rni/bt_root/rlpnc/data/rnm/ref/override` directory.

You can modify these files and add additional files in the same subdirectory to extend coverage to additional supported languages. You can also create files that only apply to a specified entity type, such as PERSON.

6.2.1. Stop Patterns and Stop Word Prefixes

Before running any matching algorithms, the names are transformed into tokens that can be compared. RNI uses stop patterns and stop word prefixes to remove patterns, including titles such as Mr., Senator, or General, that you do not want to include in name matching. Both stop patterns and stop word prefixes are used to strip matching name elements during indexing and querying. Stop words are string literals and are processed much more quickly than stop patterns, which are regular expressions. You should use stop words for the most efficient removal of prefixes, such as titles. Stop words are language-dependent.

For each name, RNI performs the following steps in order:

1. Character-level normalization, stripping punctuation (except for periods, commas, and hyphens). White space is reduced to single spaces and all characters are lower-cased. Diacritical marks are removed.
2. Stop patterns are applied.
3. Stop words are applied.

RNI cycles its way through the stop patterns then the stop words, each cycle removing the patterns and words that strip nothing, until the list of stop patterns and stop words is empty.

Stop Pattern

A stop pattern is a regular expression that excludes matching name elements during indexing and queries. You can use any regular expression supported by the Java 1.8 `java.util.regex.Pattern`; see the Javadoc for detailed documentation.

Stop patterns for a given language are specified in a UTF-8 file with the ISO 639-3 three-letter language code in the filename:

```
stopregexes_LANG[_TYPE].txt
```

where *LANG* is a three-letter language code.

Each row in the file, except for rows that begin with `#`[4] is a regular expression. Leading and trailing whitespace is removed from regex lines, so use `\s` at the beginning and end as needed.

TIP

Include *_TYPE*, where *TYPE* designates an entity type, such as PERSON if you want the override to apply only if the name, matching names, or matching tokens have been assigned this entity type. If the filename does not include *_TYPE*, it will be applied to all names, regardless of the entity type.

[4] `#` may also be used after an entry on the same line to begin a comment.

Name elements matching any of these regular expressions are removed. Longer stop patterns are applied before shorter stop patterns, so the presence of a shorter stop pattern does not prevent the stripping of a longer pattern that includes the shorter pattern. For example, the `brigadier[-]general` stop pattern is applied first, but `general` is also a stop pattern and will be applied as well.

RNI includes files with stop patterns for names in English (generic and ORGANIZATION), Japanese (PERSON), Spanish (generic), and Chinese (PERSON). These files are in `plugins/rni/bt_root /rlpnc/data/rnm/ref/override`. The generic (non-entity-specific) English file is `stopregexes_eng.txt`. For example, the entries

```
^fnu\d
^lnu\d
```

indicate that the common indicators for first-name-unknown and last-name-unknown followed by nothing are to be removed.

You can also specify which field the regex is to be applied to when processing a fielded name. Simply add **Tab**`n`, where `n` is the field number. To search multiple fields, include an entry for each field, as illustrated below. When processing a name without fields, the field parameter is ignored. For example,

```
^lnu\d    2
^lnu\d    3
```

indicates that the regex is to be applied to fields 2 and 3 in fielded names.

You can modify the contents of this file. To add stop patterns for a different language, create an additional UTF-8 file in the same subdirectory with the three-letter language code in the filename. For example, `stopregexes_ara.txt` would include regular expressions with Arabic text; `stopregexes_eng_PERSON.txt` would include regular expression to remove elements from PERSON names in English text.

Use of complex patterns may increase processing time. When possible, use stop word prefixes.

Stop Word Prefixes

A stop word prefix is a string literal that strips the matching prefix from name elements during indexing and querying.

Stop word prefixes for a given language are specified in a UTF-8 file with the ISO 639-3 three-letter language code in the filename:

```
stopprefixes_LANG[_TYPE].txt
```

where *LANG* is a three-letter language code. Each row in the file, except for rows that begin with `#`, is a string literal. Prefixes matching any of these string literals are removed.

Like stop patterns, longer stop word prefixes take precedence over shorter prefixes contained within the longer stop word. For example, the `lieutenant colonel` stop word prefix is applied where applicable when `colonel` is also a stop word prefix.

RNI includes files with generic stop word prefixes for names in Arabic, English, Greek, Hungarian, Spanish, and Thai. These files are in `plugins/rni/bt_root /rlpnc/data/rnm/ref/override`: `stopprefixes_eng.txt` and `stopprefixes_spa.txt`. You can modify the contents of these files. To

add stop word prefixes for another language, create a UTF-8 file in the same directory with the three-letter language code in the filename. For example, `stopprefixes_rus.txt` would include stop word prefixes for use with Russian text.

6.2.2. Overriding Name Pair Matches

You can create UTF-8 text files that specify the scores to be assigned for specified full-name pairs. The filename uses the ISO 639-3 three-letter language codes to designate the language of each full name in each of the full-name pairs:

```
fullnames_LANG1_LANG2[_TYPE].txt
```

where *LANG1* is the three-letter language code for the first name and *LANG2* is the three letter language code for the second name.

TIP

Include *_TYPE*, where *TYPE* designates an entity type, such as PERSON if you want the override to apply only if the name (for stop patterns), matching names, or matching tokens have been assigned this entity type. If the filename does not include *_TYPE*, it will be applied to all names, regardless of the entity type.

Each row in the file, except for rows that begin with `#`, is a tab-delimited full-name pair and score:

```
name1 Tab name2 Tab score
```

The scores must be between 0 and 1.0, where 0 indicates no match, and 1.0 indicates a perfect match.

TIP

Since the minimum score for names returned by RNI queries must be greater than 0, an RNI query will not return the name if the override score is 0. Name match operations, on the other hand, will return an override score of 0.

The installation includes a sample file with sample entries commented out: `plugins/rni/bt_root/rlpnc/data/rnm/ref/override/fullnames_eng_eng.txt`. Any non-commented-out entries in this file assign scores to English queries applied to English names in an RNI index. For example,

```
John Doe    Joe Bloggs    1.0
```

indicates that the query name `John Doe` matches the index name `Joe Bloggs` (both used in different regions to indicate 'person unknown') with a score of 1.0.

These match patterns are commutative. The previous entry also specifies a match score of 1.0 if the query name is `Joe Bloggs` and the index includes a document with an `rni_name` field containing `John Doe`.

You can add entries for English to English name matches to `fullnames_eng_eng.txt`, and create additional override files, using the filename to specify the languages. For example the following entries could appear in `fullnames_jpn_eng.txt`:

```
外山恒        Toyama Koichi    1.0
ヒラリークリントン    Hillary Clinton    1.0
```

6.2.3. Overriding Token Pair Matches

You can create text files that specify token (name-element) pairs that match. Token pair overrides are supported[5] for English-English, Japanese-English, Chinese-English, Russian-English, Spanish-English, Japanese-Japanese, Russian-Russian, English-Korean, Korean-Korean, Spanish-Spanish, Greek-English and Hungarian-English token pairs. Such pairs may include proper name and nickname, such as Peter and Pete, and cognate names such as Peter and Pedro. Tokens cannot contain whitespace. When RNI evaluates two names, each of which contains an element from the pair, it enhances the value of the resulting name match score. For example, if `Abigail` and `Abby` constitute a token pair, then the match score for `Abigail Harris` and `Abby Harris` will be higher than it would be if the token pair had not been specified.

The token pairs may be within a language or cross-lingual, as indicated by the file name:

```
tokens_LANG1_LANG2_[TYPE].txt
```

where *LANG1* is the three-letter language code for the first token in each pair and *LANG2* is the three-letter language code for the second token in each pair. Each entry in the file, except for rows that begin with `#`, is a tab-delimited token pair and may include a raw score between 0.0 and 1.0 or an indicator that at least one of the tokens is a nickname or that the tokens are cognates:

```
Token1 Tab Token2 Tab [[0.0-1.0]|NICKNAME|COGNATE|VARIANT]
```

A token pair override score (raw score or indicator) serves as a minimum score, but you can write "/force" after a token score to force it to be exactly that value:

```
Token1 Tab Token2 Tab [([0.0-1.0]|NICKNAME|COGNATE|VARIANT)/force]
```

If you would like to prevent a token pair from matching, you can use the SUPPRESS indicator as an alias for "0.0/force". If you do not include NICKNAME, COGNATE, VARIANT, or SUPPRESS, RNI assumes NICKNAME.

RNI includes `plugins/rni/bt_root/rlpnc/data/rnm/ref/override/tokens_eng_eng.txt`, which contains a list of English/English token pairs. For example:

```
Peter    Pete    NICKNAME
Peter    Pedro   COGNATE
```

This directory also contains Chinese to English token overrides for LOCATION and ORGANIZATION: `tokens_zho_eng_LOCATION.txt`, `tokens_zho_eng_ORGANIZATION.txt`.

When you create an additional file in the same location, use the ISO 639-3 three-letter language name in the filename to identify the language of each name element in the pair. For example `tokens_eng_eng.txt`

[5]Override files are not provided for all supported languages. Specifically, while no files are provided for Russian or Korean, you can create token pair files for these languages.

indicates that the contents match English names to English names; `tokens_eng_eng_ORGANIZATION.txt` indicates that the contents match English ORGANIZATION names to English ORGANIZATION names. The SDK includes a sample file for matching English/English tokens in LOCATION entities: `tokens_eng_eng_LOCATION.txt`.

We recommend that you enter the language names in alphabetical order in the filename and token pairs. Keep in mind that the order has no influence on the resulting score, since the scoring is commutative.

6.2.4. Normalizing Token Variants

You can create text files that specify the normalized form for tokens (name elements) and variants to normalize to that form. The file name indicates the language and optionally the entity type for the tokens to be normalized:

```
equivalenceclasses_LANG_[TYPE].txt
```

For example, `equivalenceclasses_jpn.txt` would contain entries for normalizing Japanese token variants for any entity type to a normalized form.

Each entry in the file contains a normalized form followed by one or more variant forms. The syntax is as follows:

```
[normal_form1]
variant1_1
variant1_2
variant1_3
[normal_form2]
variant2_1
variant2_2
variant2_3
...
```

RNI includes `plugins/rni/bt_root/rlpnc/data/rnm/ref/override/equivalenceclasses_eng_PERSON.txt`, which contains a list of variant renderings to normalize to `muhammad`:

```
[muhammad]
mohammed
mahamed
mohamed
mohamad
mohammad
muhammed
muhamed
muhammet
muhamet
md
mohd
muhd
```

You can add lists of variants to this file, including the normalized form in square brackets to start each list.

6.2.5. Unimportant tokens

You can edit the list of tokens that are given low influence in RNI. These low weight tokens are parts of a name (such as suffixes) that don't contribute much to the name matching accuracy.

The file name is `lowWeightTokens_LANG.txt`.

For example, `plugins/rni/bt_root/rlpnc/data/rnm/ref/lowWeightTokens_eng.txt` contains entries for tokens in English that you may want to put less emphasis on: "jr", "sr", "ii", "iii", "iv", "de".

7. Address Matching

The RNI plugin can match addresses returning a match score reflecting the similarity of two addresses.

As with name and date matching, the process is to create an index containing addresses, then query an address against the index.

NOTE
RNI is optimized for addresses in English. Non-English addresses in Latin script may also be matched; results will vary by language.

7.1. Address Definition

Addresses can be defined either as a set of address fields or as a single string. When defined as a string, the jpostal library[6] is used to parse the address string into address fields.

When entered as a set of fields, the address may include any of the fields below. At least one field must be specified, but no specific fields are required.

RNI optimizes the matching algorithm to the field type. Named entity fields, such as street name, city, and state, are matched using a linguistic, statistically-based algorithm that handles name variations. Numeric and alphanumeric fields such as house number, postal code, and unit, are matched using numeric-based methods.

Supported Address Fields

Field Name	Description	Example(s)
`house`	venue and building names	"Brooklyn Academy of Music", "Empire State Building"
`houseNumber`	usually refers to the external (street-facing) building number	"123"
`road`	street name(s)	"Harrison Avenue"
`unit`	an apartment, unit, office, lot, or other secondary unit designator	"Apt. 123"
`level`	expressions indicating a floor number	"3rd Floor", "Ground Floor"
`staircase`	numbered/lettered staircase	"2"
`entrance`	numbered/lettered entrance	"front gate"
`suburb`	usually an unofficial neighborhood name	"Harlem", "South Bronx", "Crown Heights"
`cityDistrict`	these are usually boroughs or districts within a city that serve some official purpose	"Brooklyn", "Hackney", "Bratislava IV"
`city`	any human settlement including cities, towns, villages, hamlets, localities, etc.	"Boston"

[6]RNI depends on the jpostal binding for the open source libpostal [4] library to parse unfielded addresses as a pre-processing step. Though jpostal is not officially supported on Windows, our tests have shown it to function as expected. Please contact support@basistech.com if you discover any issues.

Field Name	Description	Example(s)
island	named islands	"Maui"
stateDistrict	usually a second-level administrative division or county	"Saratoga"
state	a first-level administrative division	"Massachusetts"
countryRegion	informal subdivision of a country without any political status	"South/Latin America"
country	sovereign nations and their dependent territories, anything with an ISO-3166 code	"United States of America"
worldRegion	currently only used for appending "West Indies" after the country name, a pattern frequently used in the English-speaking Caribbean	"Jamaica, West Indies"
postCode	postal codes used for mail sorting	"02110"
poBox	post office box: typically found in non-physical (mail-only) addresses	"28"

7.2. Using Address Matching

7.2.1. Index Addresses

1. Create an index.

```
curl -XPUT 'http://localhost:9200/rni-test'
```

2. Define a mapping for fields that will contain addresses. The type for each of these fields is `"rni_address"`.

```
curl -XPUT 'http://localhost:9200/rni-test/_mapping' -H'Content-Type: application/json' -d '{
  "properties" : {
    "primary_name" : { "type" : "rni_name" },
    "residence" : { "type" : "rni_address" }
  }
}'
```

3. Index documents containing an address field.

```
curl -XPUT 'http://localhost:9200/rni-test/_doc/1' -H'Content-Type: application/json' -d '{
  "primary_name" : "Joe Schmoe",
  "residence" : {
    "houseNumber" : "123",
    "road" : "Main St",
    "city" : "Boston",
    "state" : "Massachusetts",
    "postCode" : "02110"
  }
}'
```

The address in the document can also be defined as a string.

```
curl -XPUT 'http://localhost:9200/rni-test/_doc/1' -H'Content-Type: application/json' -d '{
    "primary_name" : "Joe Schmoe",
    "residence" : "123 Main St, Boston, Massachusetts, 02110"
}'
```

7.2.2. Query Field Addresses

RNI compares the fields in the query with the fields in the index, matching each non-blank field. Addresses do not have to contain all the same fields to be compared and matched.

As with other objects, the query for an address consists of two parts: the base query and the RNI pairwise address match rescore query.

Base Query. The base query is a standard query against the address field. Refer to Query the Index [9].

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "query" : {
    "match" : {
      "residence" : "{\"road\" : \"Main\", \"state\" : \"MA\"}"
    }
  }
}'
```

RNI Rescore with Addresses. Refer to Rescoring with RNI Pairwise Name Match [10].

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "query" : {
    "match" : { "residence" : "{\"road\" : \"Main\", \"state\" : \"MA\"}" }
  },
  "rescore" : {
    "query" : {
      "rescore_query" : {
        "function_score" : {
          "address_score" : {
            "field" : "residence",
            "query_address" : {
              "road" : "Main",
              "state" : "MA"
            }
          }
        }
      },
      "query_weight" : 0.0,
      "rescore_query_weight" : 1.0
    }
  }
}'
```

The query returns a hit with the RNI address match score.

```
"hits": {
 "total" : 1,
  "max_score" : 0.6057692,
  "hits" : [
    {
      "_index" : "rni-test",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 0.6057692,
      "_source" : {
        "primary_name" : "Joe Schmoe",
        "residence" : {
          "houseNumber" : "123",
          "road" : "Main St",
          "city" : "Boston",
          "state" : "Massachusetts",
          "postCode" : "02110"
        }
      }
    }
  ]
}
```

The address match score is a measure of how similar the addresses are. Similar addresses have a stronger match and their address match score is closer to 1.

7.2.3. Query String Addresses

The address can be structured as a string for queries. The address structure for the query is independent of the format of the address in the original document. A string can be used in the query regardless of whether the indexed address was formatted with fields or as a string.

Base Query. The base query constructed with an address string.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "query" : {
    "match" : {"residence" : "Main, MA"}
  }
}'
```

RNI Rescore with Addresses. The rescore query with an address string.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "query" : {
     "match" : { "residence" : "Main, MA" }},
   "rescore" : {
     "query" : {
       "rescore_query" : {
         "function_score" : {
           "address_score" : {
             "field" : "residence",
             "query_address" : "Main, MA"
          }
         }
     },
     "query_weight" : 0.0,
     "rescore_query_weight" : 1.0
     }
   }
}'
```

The response displayed here returns the address as a string because the indexed document used in this example represented the address as strings. The response will return the address in the same format as the indexed document. The format of the query does not have to match the format of the indexed documents.

```
"hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 0.4552421,
    "hits" : [
      {
        "_index" : "rni-test",
        "_type" : "_doc",
        "_id" : "1",
        "_score" : 0.4552421,
        "_source" : {
          "primary_name" : "Joe Schmoe",
          "residence" : "123 Main St, Boston, Massachusetts, 02110"
        }
      }
    ]
  }
```

The address match score is a measure of how similar the addresses are. Similar addresses have a stronger match and their address match score is closer to 1.

7.2.4. Advanced Rescorer for Address Matching

RNI includes a customized RNI rescorer query parameter, `rni_query`, which utilizes RNI advanced features. The RNI custom rescorer uses the parameters above as well as the following parameters to determine the number of documents that will be rescored. Rescoring fewer documents increases speed, but can be at the cost of accuracy if the best documents are not passed to the rescorer.

- `score_to_rescore_restriction` (a float, defaults to 0.4, cannot be negative) dynamically controls the minimum query score a document needs to be passed to the RNI rescorer.
  A value of 0.0 will not cut off any documents from being rescored. Higher values rescore fewer documents, increasing speed at the cost of accuracy.

- `window_size_allowance` (a float, defaults to 0.5, must be in interval (0, 1]) dynamically controls the window size for rescoring. No more than `window_size` names will be scored.
  A value of 1.0 will not cut off any documents from being rescored. Higher values rescore more documents, increasing accuracy at the cost of speed.

In the following example, pairwise matching is performed on the top 200 names returned by the base query.

Example with the Advanced Rescorer

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "match" : {
            "primary_name" : "{\"data\" : \"Jo Shmoe\", \"entityType\" : \"PERSON\"}"
        }
    },
    "rescore" : {
        "window_size" : 200,
        "rni_query" : {
            "rescore_query" : {
                "rni_function_score" : {
                    "name_score" : {
                        "field" : "primary_name",
                        "query_name" : {"data" : "Jo Shmoe", "entityType":"PERSON"},
                        "score_to_rescore_restriction": 1.0,
                        "window_size_allowance": 0.5
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

8. Configuring Address Matching

Addresses have their own match parameters and override files that you can customize to achieve the best results for your data.

There are two types of override files for addresses:

- Stop patterns and stop word prefixes designate address field elements to strip during indexing and queries.
- Token pair overrides specify address field elements pairs that match.

File Directories

- The parameters [19] are modified in the `plugins/rni/bt_root/rlpnc/data/etc/parameter_profiles.yaml` file.
- The address matching override files are in the `plugins/rni/bt_root/rlpnc/data/addresses/ref/overrides` directory.
- The address stop word files are in the `plugins/rni/bt_root/rlpnc/data/addresses/ref/stopwords` directory.

8.1. Modifying Address Parameters

To start tuning the parameters, run address matching on the test set and look for any unexpected results. Tunable parameters are defined in `parameter_defs.yaml`. The parameter files are described in Parameter Configuration Files [19].

NOTE
Only the `any` profile is supported for address parameters; language-specific profiles are not supported for addresses.

An example parameter to tune is `addressJoinedTokenLimit`, which controls leniency towards joining or separating tokens. For some use cases, you may decide that joining many tokens within a field is acceptable. To adjust this parameter, find an existing parameter profile or define a new one, add the parameter and modify the value. By increasing the parameter value, the `addressJoinedTokenLimit` will be allowed to merge more tokens.

Another example parameter is `houseNumberAddressFieldWeight`, which controls the weight of the `houseNumber` score when calculating the overall score. This type of parameter is available for all address fields, and is weighted evenly at 1 by default. For example, `cityAddressFieldWeight` controls the weight of the city field when matching addresses.

Once you define a profile and set a parameter value, rerun the address pairwise match, scoring the match with the edited `parameter_profiles.yaml` file.

8.1.1. Address Parameters

Parameter	Description	Behavior
`addressFinalBias`	Helps normalize scores	Increasing leads to a higher score for ALL names
`addressReorderPenalty`	Penalty for token reordering within a comparison	Increasing leads to lower final score

Parameter	Description	Behavior
addressDeletionScore	Score for deleted token within a comparison	
addressUnpairedFieldScore	Score for an unpaired field	
addressOverrideDefaultScore	Score for override matches	
addressJoinedTokenLimit	Maximum sum of the number of tokens considered when matching two address fields	
addressCrossFieldScoreThreshold	Minimum value a cross-field score must have to be included in the final score	
addressSameGroupPenalty	Multiplier on field comparisons from the same group	Increasing leads to lower final score
addressDifferentGroupPenalty	Multiplier on field comparisons from different groups.	Increasing leads to lower final score
houseAddressFieldWeight	Weight used during comparison of the house field	
houseNumberAddressFieldWeight	Weight used during comparison of the house number field	
roadAddressFieldWeight	Weight used during comparison of the road field	
unitAddressFieldWeight	Weight used during comparison of the unit field	
levelAddressFieldWeight	Weight used during comparison of the field	
staircaseAddressFieldWeight	Weight used during comparison of the field	
entranceAddressFieldWeight	Weight used during comparison of the entrance field	
suburbAddressFieldWeight	Weight used during comparison of the suburb field	
cityDistrictAddressFieldWeight	Weight used during comparison of the cityDistrict field	
cityAddressFieldWeight	Weight used during comparison of the city field	
islandAddressFieldWeight	Weight used during comparison of the island field	
stateDistrictAddressFieldWeight	Weight used during comparison of the stateDistrict field	
stateAddressFieldWeight	Weight used during comparison of the state field	
countryRegionAddressFieldWeight	Weight used during comparison of the countryRegion field	
countryAddressFieldWeight	Weight used during comparison of the country field	
worldRegionAddressFieldWeight	Weight used during comparison of the worldRegion field	
postCodeAddressFieldWeight	Weight used during comparison of the postCode field	
poBoxAddressFieldWeight	Weight used during comparison of the poBox field	

8.2. Stop Patterns and Stop Word Prefixes

RNI uses stop patterns and stop word prefixes to remove patterns from address fields during indexing and queries before matching algorithms are applied. Using string literals to strip prefixes can be performed more quickly than the application of stop patterns (regular expressions), so you should use stop words for the efficient removal of prefixes, such as *the*, that you do not want to include in address matching.

For each address field, RNI performs the following steps in order:

1. Character-level normalization, stripping punctuation including periods, commas, hyphens, and the number sign. White space is reduced to single spaces and all characters are lower-cased.
2. Stop patterns are applied.
3. Stop words are applied.

8.2.1. Stop Pattern

A stop pattern is a regular expression that excludes matching address field elements during indexing and queries. You can use any regular expression supported by the Java 1.8 `java.util.regex.Pattern` class; see the Javadoc for detailed documentation.

Stop patterns for a given address field are specified in a UTF-8 file with the `AddressField` name:

```
stopregexes_FIELD.txt
```

where *FIELD* is an `AddressField` name. Each row in the file, except for rows that begin with `#`,[7] is a regular expression. Leading and trailing whitespace is removed from regex lines, so use `\s` at beginning and end where needed.

Elements in the address fields matching any of these regular expressions are removed. Longer stop patterns are applied before shorter stop patterns, so the presence of a shorter stop pattern does not prevent the stripping of a longer pattern that includes the shorter pattern.

Stop pattern files are arranged by field in `plugins/rni/bt_root/rlpnc/data/addresses/ref/stopwords`. You can add patterns to existing files, or if the file doesn't exist, create an UTF-8 file in the directory and include the respective `AddressField` name in the filename. For example, `stopregexes_stateDistrict.txt` would include regular expressions to remove elements from the `stateDistrict` address field.

Use of complex patterns may increase processing time. When possible, use stop word prefixes.

8.2.2. Stop Word Prefixes

A stop word prefix is a string literal that strips the matching prefix from address field elements during indexing and queries.

Stop word prefixes for a given address field are specified in a UTF-8 file with the `AddressField` name:

```
stopprefixes_FIELD.txt
```

where *FIELD* is an `AddressField` name. Each row in the file, except for rows that begin with `#`,[8] is a string literal.

Prefixes in the address field matching any of these string literals are removed.

Like stop patterns, longer stop word prefixes take precedence over shorter prefixes that the longer stop word contains.

RNI includes files with stopword prefixes for addresses in English (for `house`, `houseNumber`, `road`, `unit`, `city`, `state`, `country`, `postCode` and `poBox` fields). These files are in `plugins/rni/bt_root/rlpnc/data/addresses/ref/stopwords`. You can modify the contents of these files. To add stop word prefixes for a different address field, create an additional UTF-8 file in the same subdirectory and include the respective `AddressField` name in the filename. For example, `stopprefixes_stateDistrict.txt` would include stopword prefixes for use on `stateDistrict` address field.

[7] `#` may also be used after an entry on the same line to begin a comment.
[8] `#` may also be used after an entry on the same line to begin a comment.

8.3. Overriding Token Pair Matches

NOTE
Token pair overrides are only supported for English-English token pairs.

You can create text files that specify token (address field element) pairs that match. Tokens cannot contain whitespace. When RNI evaluates two address fields, each of which contains an element from the pair, it enhances the value of the resulting address match score. For example, if `road` and `rd` constitute a token pair, then the match score for `Stuart Road` and `Stuart Rd` will be higher than it would be if the token pair had not been specified.

The token pairs file name format is:

```
FIELD.txt
```

where *FIELD* is the `AddressField` name. Each entry in the file, except for rows that begin with `#`, is a tab-delimited token pair and may include a raw score between 0.0 and 1.0. If no score is provided, the `addressOverrideDefaultScore` parameter value will be used.

```
Token1 Tab Token2 Tab [0.0-1.0]
```

A token pair override score serves as a minimum score, but you can write `/force` after a token score to force it to be exactly that value:

```
Token1 Tab Token2 Tab [0.0-1.0]/force
```

If you would like to prevent a token pair from matching, you can use the SUPPRESS indicator as an alias for "0.0/force".

RNI includes `plugins/rni/bt_root/rlpnc/data/addresses/ref/override/state.txt`, which contains a list of U.S. state abbreviations. For example:

```
Massachusetts  MA
California  CA
```

This directory also contains English to English token overrides for `unit`, `level`, `road`, `city`, `state` fields: `unit.txt`, `level.txt`, `road.txt`, `city.txt`, `state.txt`.

When you create an additional file in the same location, use the respective `AddressField` name in the filename to identify the address field each token element in the pair pertains to. For example `cityDistrict.txt` indicates that the contents match cityDistrict address fields.

9. Date Matching

RNI can match dates returning a data match score reflecting the time similarity of the two dates. Dates that are closer together are considered a stronger match and return a match score closer to 1.

For example, 11/05/1993 and 11/07/1993 have a high score, as they are very similar and just two days apart. However, 11/05/1993 and 11/05/1995 yield a low score as they differ by two years.

The process is similar to name matching:

- Index the dates in connection to the related names.
- Query the date and name, receiving back a match score.

The query will return separate match scores for the name and for the associated date of birth. You may decide that the name is more important than the birth date. Within your system, you can weight and combine the name and date match scores to determine the final match score.

9.1. Date Definition

A date contains a year, month, and day, but not all fields are required for matching. All common delimiters for English dates are supported and dates can be expressed with various orderings. RNI will filter out some non-date related words. Formats that include time of day are not supported unless you specify an Elasticsearch date format that includes time information in the mapping. The time component will be ignored.

Omit fields if you do not have the value for one or more fields. For example: 1955-12-30, 1955--03, 12/30, -12-, --30, 1955, 1955-12-.

9.1.1. Supported Date Formats

RNI supports a wide variety of date formats.

- Days can be represented by 1 or 2 digits
- Months can entered as numerics (1 or 2 digits) or English characters (full name or 3 character abbreviation)
- Years can be represented as 1, 2, 3 or 4 digits
- Supported delimiters include `,` `.` `-` `/`, as well as a space
- Partial fields can be entered

Examples: All of the following are acceptable:

- 3/24/1984
- March 1984
- 3-24
- -24-84
- March 24, 1984

- 24-3-84
- March
- 1984

9.2. Using Date Matching

9.2.1. Index Dates

1. Create an index.

```
curl -XPUT 'http://localhost:9200/rni-test'
```

2. Define a mapping for fields that will contain dates. The type for a date field when matching is `"rni_date"`.

```
curl -XPUT 'http://localhost:9200/rni-test/_mapping' -H'Content-Type: application/json' -d '{

    "properties" : {
        "birth_date" : { "type" : "rni_date" },
        "primary_name" : { "type" : "rni_name" }
    }
}'
```

Optionally, in the mapping, you can specify an Elasticsearch date format. All dates must adhere to the specified format. If you specify a format that includes time information, RNI ignores the time component of the date.

WARNING

Specifying an Elasticsearch format disables support for unspecified fields. If, for example, you select a format that does not include a day field ("MM-yyyy"), you will get an error when you use the date format in a query.

```
curl -XPUT 'http://localhost:9200/rni-test/_mapping' -H'Content-Type: application/json' -d '{
    "properties" : {
        "birth_date" : {
            "type" : "rni_date",
            "format" : "MM-yyyy-dd"
        },
        "primary_name" : {
            "type" : "rni_name"
        }
    }
}'
```

3. Index documents containing a date field.

```
curl -XPUT 'http://localhost:9200/rni-test/_doc/1' -H'Content-Type: application/json' -d '{
    "primary_name" : "Joe Schmoe",
    "birth_date" : "07-1955-24"
}'
```

9.2.2. Query Dates

There are many ways to incorporate date matching within your query. Here are two examples, one with date matching by itself, and one with date and name matching.

Basic Date Matching

Base Query. The base query is a standard query against the date field. Refer to Query the Index [9].

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "match" : {
            "birth_date" : "08-1955-25"
        }
    }
}'
```

RNI Rescore with Dates. Refer to Rescoring with RNI Pairwise Name Match [10].

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "match" : { "birth_date" : "08-1955-25" }
    },
    "rescore" : {
        "query" : {
            "rescore_query" : {
                "function_score" : {
                    "date_score" : {
                        "field" : "birth_date",
                        "query_date" : "08-1955-25"
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

The query returns a hit, with the RNI date match score.

```
"hits": {
   "total": 1,
   "max_score": 1.618923,
   "hits": [
     {
       "_index": "test",
       "_type": "_doc",
       "_id": "AVXMepnorGuybmuiQtQr",
       "_score": 0.8120856,
       "_source": {
         "primary_name": "Joe Schmoe",
         "birth_date": "07-1955-24"
       }
     }
   ]
 }
```

Date and Name Match

Base Query. The base query is a standard query against the date and name fields.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "primary_name": "Joe S."
          }
        },
        {
          "match": {
            "birth_date": "08-1955-25"
          }
        }
      ]
    }
  }'
```

RNI Rescore with Dates. Use the `doc_score` function in the rescore when matching a combination of Elasticsearch field types instead of the functions for a single type (`name_score` and `date_score`). The name field is also added to the rescore.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
  "rescore": {
    "query": {
      "rescore_query": {
        "function_score": {
          "doc_score": {
            "fields": {
              "primary_name": {
                "query_value": "Joe S."
              },
              "birth_date": {
                "query_value": "08-1955-25"
              }
            }
          }
        }
      },
      "query_weight" : 0.0,
      "rescore_query_weight" : 1.0
    }
  }
}'
```

9.3. Date Match Parameters

Similarly to the name matching parameters [22], there are a series of date matching parameters. The parameter values can be edited in the `plugins/rni/bt_root/rlpnc/data/etc/parameter_defs.yaml` file.

Parameter Name	Description	Behavior
`tryDayMonthSwap`	Allows for date matching with swapped day and month fields. It is on by default.	This parameter attempts to correct for parsing errors by swapping the day and month. Turn it off if you only want to match the dates exactly as indexed.
`timeDistanceWeight`	Weight for the time distance (i.e. #days) between two dates). The score is based on the number of days between the two dates.	1979-12-31 and 1980-1-1 look different, but their time difference is very close. They will have a high match score.
`yearDistanceWeight`	Weight for the year field comparison of the dates.	Close years will have a high match score.

Parameter Name	Description	Behavior
`monthDistanceWeight`	Weight for the month field comparison of the dates	1 and 12 are far, even if they are close in time. They will have a low match score.
`dayDistanceWeight`	Weight for the day field comparison of the dates	1 and 30 are far, even if they are close in time. They will have a low match score.
`stringDistanceWeight`	The edit difference between the two dates, when converted to a standard string (05021974 for 5/2/1974)	1979-12-31 and 1980-1-1 will be 19791231 and 198000101. They will have a low match score.

The date weighting fields control the relative strength of each aspect of the date-matching algorithm. A separate score is calculated for each match type. The final match score is calculated by performing a weighted arithmetic mean over each of the similarity scores. If a field is missing from a record, that field is ignored and its weight evenly distributed across other fields.

Dates with a high **time** match score may have a very low **string** match score. Time finds dates that are close together; string gives high scores to similarly formatted dates.

Because dates are sometimes written *month day* and other times written *day month*, swap tries matching the date fields as written as well as with the month and date fields switched. The best score is returned as the match score. For example, if the dates in question are 1970-3-5 and 1970-6-4, this feature will match the following four pairs:

1970-3-5 ↔ 1970-6-4
1970-3-5 ↔ 1970-4-6
1970-5-3 ↔ 1970-6-4
1970-5-3 ↔ 1970-4-6

10. Record Matching

A search can include multiple fields and return a single match and match score. The fields can be any combination of type `rni_name`, `rni_date`, `rni_address`, or any other Elasticsearch field type.

Each field can be assigned a weight to reflect its importance in the overall matching logic. When searching for a match, some fields are more important in determining a match than others. For example, the name field is likely more important in determining a match than an address field. If no weights are defined, each field is weighted equally.

When matching records, a similarity score is calculated for each field. Then the final match score is then calculated by performing a weighted arithmetic mean over each of the similarity scores. If a field is missing from a document, that field is removed from the score calculation and its weight is evenly distributed across other fields. You can override this behavior by using the `score_if_null` option to specify a score to be returned if the field is null in the index document.

Use the `doc_score` function in the rescore query when matching records that include multiple field types, instead of the functions for a single type, such as the `name_score` and `date_score` functions. The `doc_score` function has built-in similarity functions for many core types. It does not, however, currently support multiple nested fields.

If your record query contains types which the `doc_score` function doesn't support, you can create a custom similarity function using the Elasticsearch `script_score` function in the rescore query.

10.1. Supported field types

The `doc_score` function has default support for `rni_name`, `rni_date`, `rni_address`, and many of the Elasticsearch core field types. All default similarity scores are between 0.0 and 1.0.

Field Type(s)	Default Similarity Function	Example(s)
rni_name	name_score (refer to RNI pairwise match score [1])	'John David Smith' vs 'Jon D Smith' = 0.88
rni_date, date	date_score (refer to Date Matching [39])	'2010-11-4' vs '2010-5-11' = 0.92
rni_address	address_score (refer to Address Matching [30]	'Red Cedar Ct' vs 'Cedar Ct' = 0.53
keyword, text, string	Normalized edit distance	'37 Congress St.' vs '35 Congres St.' = 0.875
integer, long, short, double, float	Normalized difference (eg. percentage)	'65' vs '59' = 0.908
boolean	Equality	'true' vs 'true' = 1.0, 'true' vs 'false' = 0.0
geo_point	Log function over Haversine distance	'[lat=42.361145, lon=-71.057083]' vs '[lat=42.3736, lon=-71.1097]' = 0.83

10.2. Using Record Matching

10.2.1. Index Records

1. Create an index with a mapping containing fields with different types

```
curl -XPUT 'http://localhost:9200/rni-test' -H'Content-Type: application/json' -d '{
    "mappings" : {
         "properties" : {
            "name" : { "type" : "rni_name" },
            "dob" : { "type" : "rni_date" },
            "address" : { "type" : "rni_address" },
            "height" : { "type" : "integer" },
            "nationality" : { "type" : "keyword" }
        }
    }
}'
```

2. Index documents that contain those fields

```
curl -XPUT 'http://localhost:9200/rni-test/_doc/1' -H'Content-Type: application/json' -d '{
    "name" : "Ryan McDonagh",
    "dob" : "11/19/1987",
    "address" : {
        "houseNumber" : "47",
        "road" : "Park St",
        "city" : "Boston",
        "state" : "MA"
    },
    "nationality" : "USA",
    "height" : 65
}'
```

10.2.2. Basic Multi-Field Query

The query can be a record containing multiple fields. The fields in the query record must be mapped to those of the indexed documents.

Base Query. The base query is a standard Elasticsearch query containing multiple fields that will return candidates for rescoring.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "bool" : {
            "should" : [
                { "match" : { "name" : "Brian McDonough", "entityType": "PERSON" } },
                { "match" : { "dob" : "10/19/87" } },
                { "match" : { "address" : "{ \"houseNumber\" : \"48\",
                                             \"road\" : \"Parker St\",
                                             \"city\" : \"Boston\",
                                             \"state\" : \"MA\" }" } }
            ]
        }
    }
}'
```

RNI Rescore with Records. Use the `doc_score` function to rescore the indexed documents against a query record.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "bool" : {
            "should" : [
                { "match" : { "name" : "Brian McDonough" } },
                { "match" : { "dob" : "10/19/87" } },
                { "match" : { "address" : "{ \"houseNumber\" : \"48\",
                                             \"road\" : \"Parker St\",
                                             \"city\" : \"Boston\",
                                             \"state\" : \"MA\" }" } }
            ]
        }
    },
    "rescore" : {
        "rni_query" : {
            "rescore_query" : {
                "function_score" : {
                    "doc_score" : {
                         "fields" : {
                             "name" : { "query_value": "Brian McDonough" },
                             "dob" : { "query_value": "10/19/87" },
                             "address" : {
                                  "query_value" : {
                                      "houseNumber" : "48",
                                      "road" : "Parker St",
                                      "city" : "Boston",
                                      "state" : "MA"
                                   }
                             },
                             "height" : { "query_value": 67 },
                             "nationality" : { "query_value": "CANADA" }
                         }
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

As with addresses, the `query_value` of names can be an object to match additional name information. The rescore query above can easily be modified to additionally match against a name's `entityType` field:

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "bool" : {
            "should" : [
                { "match" : { "name" : "Brian McDonough" } },
                { "match" : { "dob" : "10/19/87" } },
                { "match" : { "address" : "{ \"houseNumber\" : \"48\",
                                             \"road\" : \"Parker St\",
                                             \"city\" : \"Boston\",
                                             \"state\" : \"MA\" }" } }
            ]
        }
    },
    "rescore" : {
        "rni_query" : {
            "rescore_query" : {
                "function_score" : {
                    "doc_score" : {
                         "fields" : {
                             "name" : {
                                  "query_value": {
                                      "data": "Brian McDonough",
                                      "entityType": "PERSON"
                                  }
                             },
                             "dob" : { "query_value": "10/19/87" },
                             "address" : {
                                  "query_value" : {
                                      "houseNumber" : "48",
                                      "road" : "Parker St",
                                      "city" : "Boston",
                                      "state" : "MA"
                                   }
                             },
                             "height" : { "query_value": 67 },
                             "nationality" : { "query_value": "CANADA" }
                         }
                    }
                }
            },
            "query_weight" : 0.0
            "rescore_query_weight" : 1.0
        }
    }
}'
```

NOTE

The quotes in the `query` above are escaped because you can't pass an object to the basic Elasticsearch query; it requires a string. The rescore queries can handle objects because they are using RNI functions to parse the values.

10.2.3. Weighted Multi-Field Query

Each field can be given a weight to reflect its importance in the overall matching logic.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "bool" : {
            "should" : [
                { "match" : { "name" : "Brian McDonough" } },
                { "match" : { "dob" : "10/19/87" } },
                { "match" : { "address" : "{ \"houseNumber\" : \"48\",
                                             \"road\" : \"Parker St\",
                                             \"city\" : \"Boston\", \"state\" : \"MA\" }" } }
            ]
        }
    },
    "rescore" : {
        "query" : {
            "rescore_query" : {
                "function_score" : {
                    "doc_score": {
                         "fields": {
                             "name": { "query_value": "Brian McDonough", "weight": 4 },
                             "dob": { "query_value": "10/19/87", "weight": 2 },
                             "address" : {
                                  "query_value" : {
                                      "houseNumber" : "48",
                                      "road" : "Parker St",
                                      "city" : "Boston",
                                      "state" : "MA"
                                   },
                                   "weight" : 2
                             },
                             "height" : { "query_value": 67, "weight": 0.5},
                             "nationality" : { "query_value": "CANADA", "weight": 1 }
                         }
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

By default, if a queried-for field is null in the index, the field is removed from the score calculation, and the weights of the other fields are redistributed. However, you can override this behavior by using the `score_if_null` option to specify what score should be returned for this field if it is null in the index document.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "bool" : {
            "should" : [
                { "match" : { "name" : "Brian McDonough" } },
                { "match" : { "dob" : "10/19/87" } },
                { "match" : { "address" : "{ \"houseNumber\" : \"48\",
                                             \"road\" : \"Parker St\",
                                             \"city\" : \"Boston\", \"state\" : \"MA\" }" } }
            ]
        }
    },
    "rescore" : {
        "query" : {
            "rescore_query" : {
                "function_score" : {
                    "doc_score" : {
                         "fields" : {
                             "name" : { "query_value": "Brian McDonough", "weight": 4, "score_if_null" : 0.0  },
                             "dob": { "query_value": "10/19/87", "weight": 2 },
                             "address" : {
                                  "query_value" : "{
                                      "houseNumber" : "48",
                                      "road" : "Parker St",
                                      "city" : "Boston",
                                      "state" : "MA"
                                   },
                                   "weight" : 2
                             },
                             "height" : { "query_value": 67, "weight": 0.5},
                             "nationality" : { "query_value": "CANADA", "weight": 1 ,  "score_if_null" : 1.0  }
                         }
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

NOTE

The quotes in the `query` above are escaped because you can't pass an object to the basic Elasticsearch query; it requires a string. The rescore queries can handle objects because they are using RNI functions to parse the values.

10.2.4. Multi-Field Query with Multiple Nested Fields

The `doc_score` function for rescoring does not currently support search queries containing multiple nested fields. To perform these queries, chain multiple rescorers and adjust the `query_weight` and `rescore_query_weight` parameters to control the relative importance of the original query and of the rescore query, respectively. When chaining multiple RNI advanced rescorers, be sure to add `"score_mode":"total"` to each rni_query object to ensure the final score is properly accumulated.

This example expands the previous examples, adding alias names and modifying the single date of birth (dob field) to contain a list of dates of birth, one for each alias (`dob` field).

1. Create an index with a mapping containing multiple nested fields

```
curl -XPUT "http://localhost:9200/rni-test" -H 'Content-Type: application/json' -d'{
    "mappings": {
    "properties": {
      "name": {
        "type": "rni_name"
      },
      "aliases": {
        "type": "nested",
        "properties": {
          "alias_name": {
            "type": "rni_name"
          }
        }
      },
      "dobs": {
        "type": "nested",
        "properties": {
          "dob": {
            "type": "rni_date"
          }
        }
      },
      "address": {
        "type": "rni_address"
      },
      "height": {
        "type": "integer"
      },
      "nationality": {
        "type": "keyword"
      }
    }
  }
}'
```

2. Index documents that contain the fields

```
curl -XPUT "http://localhost:9200/rni-test/_doc/1" -H 'Content-Type: application/json' -d'{
  "name": "Ryan McDonagh",
  "aliases": [
    {
      "alias_name": "Rayan McDonagh"
    },
    {
      "alias_name": "R. McDonagh"
    },
    {
      "alias_name": "Rayan M."
    }
  ],
  "dobs": [
    {
      "dob": "11/19/1987"
    },
    {
      "dob": "11/20/1987"
    },
    {
      "dob": "10/19/1987"
    }
  ],
  "address": {
    "houseNumber": "47",
    "road": "Park St",
    "city": "Boston",
    "state": "MA"
  },
  "nationality": "USA",
  "height": 65
}'
```

3. Query index with chained multiple rescorers

```
curl -XGET "http://localhost:9200/rni-test/_search" -H 'Content-Type: application/json' -d'{
  "query": {
    "bool": {
     "should": [
      {
        "nested": {
          "path": "dobs",
          "query": {
            "bool": {
              "should": {
                "match": { "dob": "10/19/87"}
              }
            }
          }
        }
      },
      {
         "nested": {
           "path":"aliases",
           "query": {
             "bool": {
               "should": {
                 "match": {"name": "Brian McDonough"}
               }
             }
           }
         }
      },
      {
         "match":{
           "address": "{\"houseNumber\": \"48\", \"road\": \"Parker St\", \"city\": \"Boston\", \"state\": \"MA\" }"
         }
       }
    ]
  }
 },
 "rescore": [
   {
     "rni_query": {
       "rescore_query": {
         "nested": {
           "score_mode": "max",
           "path": "aliases",
           "query": {
             "rni_function_score": {
               "name_score": {
                 "field": "aliases.alias_name",
                 "query_name": "Brian McDonough",
                 "score_to_rescore_restriction": 0.01,
                 "window_size_allowance": 1
               }
             }
           }
         }
       },
       "score_mode": "total",
       "query_weight": 0.0,
       "rescore_query_weight": 1.0❶
       }
     },
     {
      "rni_query": {
        "rescore_query": {
          "nested": {
            "score_mode": "max",
            "path": "dobs",
            "query": {
```

```
            "rni_function_score": {
              "date_score": {
                "field": "dobs.dob",
                "query_date": "10/19/87"
              }
            }
          }
        }
      },
      "score_mode": "total",
      "query_weight": 0.67,
      "rescore_query_weight": 0.33 ❷
      }
    },
    {
     "rni_query": {
       "rescore_query": {
         "rni_function_score": {
           "address_score": {
             "field": "address",
             "query_address": {
               "houseNumber": "48",
                "road": "Parker St",
                "city": "Boston",
                "state": "MA"
              }
            }
          }
        },
        "score_mode": "total",
        "query_weight": 0.75,
        "rescore_query_weight": 0.25 ❸
       }
     },
    {
      "query": {
      "rescore_query": {
        "match": {
          "height": 67
        }
      },
      "query_weight": 0.89,
      "rescore_query_weight": 0.1 ❹
      }
     },
    {
      "query": {
      "rescore_query": {
        "match": {
          "nationality": "CANADA"
        }
      },
      "query_weight": 0.9,
      "rescore_query_weight": 0.1 ❺
    }
   }
  ]
}'
```

To calculate the `rescore_query_weight` for each nested field, you have to work from bottom to top, dividing each field's desired weight by the product of the already-calculated `query_weight` values. The `query_weight` is calculated by subtracting the `rescore_query_weight` from 1.

If there are no previous `query_weight` values, the `rescore_query_weight` is simply the desired field weight.

In this example, the desired field weights are 0.4, 0.2, 0.2, 0.1, and 0.1 for the alias, dob, address, height, and country fields, respectively.

❶ **Rescore based on alias**

Name field weight = 0.4

rescore_query_weight = 0.4 / (0.667 x 0.75 x 0.89 x 0.9) = 1

query_weight = 1 - 1 = 0

❷ **Rescore based on date of birth**

DOB field weight = 0.2

rescore_query_weight = 0.2 / (0.75 x 0.89 x 0.9) = 0.333

query_weight = 1 - 0.33 = 0.667

❸ **Rescore based on address**

Address field weight = 0.2

rescore_query_weight = 0.2 / (0.9 * 0.89) = 0.25

query_weight = 1 - 0.25 = 0.75

❹ **Rescore based on height**

Height field weight = 0.1

rescore_query_weight = 0.1 / 0.9 = 0.11

query_weight = 1 - 0.11 = 0.89

❺ **Rescore based on nationality**

Country field weight = 0.1

rescore_query_weight = 0.1

query_weight = 1 - 0.1 = 0.9

10.2.5. Weighted Multi-Field Query with Custom Similarity Function

While the doc_score function has built-in similarity functions for many core field types, a custom similarity function can be provided at query time. In this manufactured example, we'll use a simple `script_score` function that matches **CANADA** and **USA** with a high score. Refer to the Elasticsearch documentation for more details about Elasticsearch scripting. Any other function can also be used.

```
curl -XGET 'http://localhost:9200/rni-test/_search' -H'Content-Type: application/json' -d '{
    "query" : {
        "bool" : {
            "should" : [
                { "match" : { "name" : "Brian McDonough" } },
                { "match" : { "dob" : "10/19/87" } },
                { "match" : { "address" : "{ \"houseNumber\" : \"48\",
                                             \"road\" : \"Parker St\", \"city\" : \"Boston\",
                                             \"state\" : \"MA\" }" } }
            ]
        }
    },
    "rescore" : {
        "query" : {
            "rescore_query" : {
                "function_score" : {
                    "doc_score": {
                        "fields": {
                            "name": { "query_value": "Brian McDonough", "weight": 4 },
                            "dob": { "query_value": "10/19/87", "weight": 2 },
                            "address" : {
                                  "query_value" : {
                                      "houseNumber" : "48",
                                      "road" : "Parker St",
                                      "city" : "Boston",
                                      "state" : "MA"
                                   },
                                   "weight" : 2
                            },
                            "height": { "query_value": 67, "weight": 0.5 },
                            "nationality": {
                               "function": {
                                   "function_score": {
                                       "script_score": {
                                           "script": {
                                               "lang": "painless",
                                               "params": {
                                                   "query_value": "CANADA"
                                               },
                                               "inline": "if (params.query_value == '\''CANADA'\'' &&
                                                doc['\''nationality'\''].value == '\''USA'\'') {return 0.8}
                                                else {return 0.2}"
                                           }
                                       }
                                   }
                               },
                               "weight": 1
                            }
                        }
                    }
                }
            },
            "query_weight" : 0.0,
            "rescore_query_weight" : 1.0
        }
    }
}'
```

NOTE

The quotes in the `query` above are escaped because you can't pass an object to the basic Elasticsearch query; it requires a string. The rescore queries can handle objects because they are using RNI functions to parse the values.

11. Dynamic Configuration Endpoints

The plugin includes Elasticsearch REST APIs to customize and tune matching through stop words, token overrides, and parameter universes. These endpoints allow you to add and modify these configuration values, without having to restart Elasticsearch.

TIP

To use any of the configuration REST APIs, the parameter `enableDynamicConfigurationEndpoints` must be set to `true` in the `parameter_profiles.yaml` file in the `any:` profile [19]. By default, this parameter is set to `false`. These endpoints should be used for testing and tuning only. When the dynamic configuration endpoints are enabled, they can slow the system down considerably.

TIP

To use dynamic configuration endpoints in an elasticsearch deployment using SSL encryption, the RNI Elasticsearch plugin must be aware of the server's certificate file. To accomplish this, start elasticsearch with:

```
ES_JAVA_OPTS="-Dbt.ssl.certificate=<path_to_certificate>"
```

11.1. Stop words

The `_stopwords` endpoint allows you to ADD, GET and DELETE stop words without restarting the Elasticsearch server. See Stop Patterns and Stop Word Prefixes [25] for more detailed information on stop words.

The following properties are used when creating stop words. The `entity_type` is optional; all other fields are required when adding stop words through the API.

Stop Word Properties

Property	Required	Description
`"lang"`	✓	ISO 639-3 code for the language of the stop word(s).

Property	Required	Description
`stopword_type`	✓	Type of stop word(s), either `regexes` or `prefixes`
`"entity_type"`		Entity type for which to apply the stop word(s), defaults to "ALL".
`"stop words"`	✓	List of stop words to be added.

11.1.1. Create stop word(s)

The `POST_stopword` adds one or more stop words. The `entity_type` field is optional, but the other fields are all required.

```
curl -XPOST "http://localhost:9200/rni_plugin/_stopwords" -H 'Content-Type: application/json' -d '{
    "lang": "eng",
    "stopword_type": "prefixes",
    "entity_type": "PERSON",
    "stopwords": [
        "honorable",
        "senior correspondent"
    ]
}'
```

11.1.2. Get stop word(s)

The `GET _stopwords` method returns all stop words for a given language and stop word type. You can search by just language or by language and type.

Returns all prefix stop words for PERSON types in English:

```
curl -XGET "http://localhost:9200/rni_plugin/_stopwords/prefixes_eng_PERSON"
```

Returns all regex stop words for ORGANIZATION types in Spanish:

```
curl -XGET "http://localhost:9200/rni_plugin/_stopwords/regexes_spa_ORGANIZATION"
```

Returns all prefix stop words in English with no type specified. By default, this list is empty; data will only be returned if you've populated the file with values:

```
curl -XGET "http://localhost:9200/rni_plugin/_stopwords/prefixes_eng"
```

11.1.3. Delete stop word(s)

The `DELETE _stopwords` method deletes a specified stop word.

```
curl -XDELETE "http://localhost:9200/rni_plugin/_stopwords/prefixes_eng_PERSON/doctor"
```

11.2. Token Overrides

The `_overrides` endpoint allows you to ADD, GET and DELETE token pair overrides without restarting the Elasticsearch server. See Overriding Token Pair Matches [28] for more detailed information on token pair overrides.

The following properties are used when creating token overrides.

Token Overrides Properties

Property	Required	Description
`"lang1"`	✓	ISO 639-3 code for the language of the tokens to be overridden.
`"lang2"`	✓	ISO 639-3 code for the language of the token overrides.
`"entity_type"`		Entity type of the list of token override pairs, defaults to "ALL".
`"token_pairs"`	✓	List of token override pairs to be added.
`"token1"`	✓	Token of `lang1` in the pair to be overridden.
`"token2"`	✓	Token of `lang2` in the pair to be the override.
`"score"`	✓	Raw score of the token pair between 0.0 and 1.0.
`"force"`		Indicates whether to force this score to be exactly that value for the given token pair, defaults to `false`.

11.2.1. Create token override(s)

The `POST _overrides` adds one or more token overrides. As shown in the table above, `entity_type` and `force` are optional, but the other fields are required.

```
curl -XPOST "http://localhost:9200/rni_plugin/_overrides" -H 'Content-Type: application/json' -d'{
        "lang1": "eng",
        "lang2": "eng",
        "entity_type": "PERSON",
        "token_pairs":
        [{
        "token1": "Abigail",
        "token2": "Abbey",
        "score": 0.74,
        "force": true},
        {
        "token1": "Aleksander",
        "token2": "Alex",
        "score": 0.74},
        {
        "token1": "Alfonso",
        "token2": "Alphonse",
        "score": 0.74},
        {
        "token1": "Frederica",
        "token2": "Federica",
        "score": 0.74}]}'
```

11.2.2. Update token override

The `POST _overrides/_update` method updates the `score` / `force` values of a given override pair.

```
curl -XPOST "http://localhost:9200/rni_plugin/_overrides/_update" -H 'Content-Type: application/json' -d'
  {
    "lang1": "eng",
    "lang2": "eng",
    "entity_type": "PERSON",
    "token1": "Frederica",
    "token2": "Federica",
    "score": 0.65,
    "force": true
  }'
```

The following syntax is also correct. It requires you specify the language profile (using the respective ISO 639-3 codes) the token override pair belongs to and, optionally, the entity type. If the entity type is not specified, it defaults to `ALL`.

```
curl -XPOST http://localhost:9200/rni_plugin/_overrides/tokens_eng_eng_PERSON/_update?\
token1=Frederica&token2=Federica&score=0.65&force=true
```

11.2.3. Get token override(s)

The `GET _overrides` method returns the overrides of a given language profile.

```
curl -XGET "http://localhost:9200/rni_plugin/_overrides/tokens_hun_eng_PERSON"
```

You can also retrieve the score of a given override pair.

```
curl -XGET "http://localhost:9200/rni_plugin/_overrides/tokens_hun_eng_PERSON?token1=abigel&token2=abigail"
```

11.2.4. Delete token override(s)

The `DELETE _overrides` method deletes a given override pair.

```
curl -XDELETE "http://localhost:9200/rni_plugin/_overrides/tokens_hun_eng_PERSON/abigel+abigail"
```

11.3. Parameters

The `_parameter_universe` endpoint allows you to ADD, GET and DELETE parameters through parameter universes, without restarting the Elasticsearch server. See Parameter Universe [20] for more information on tuning parameters with parameter universes.

11.3.1. Add Parameter(s)

The `POST _parameter_universe` method creates a parameter universe and the parameter profiles within the universe. If you try to add a parameter universe that already exists, it overrides it with the new values. The parameter universe method uses the following syntax:

`SomeParameterUniverseName/xxx_yyy` where `xxx_yyy` is the language profile the parameters belong to expressed in ISO 639-3 codes. The `parameters` field expects a list of parameters for the given profile where the naming of the parameters should match the ones declared in `parameter_defs.yaml`

```
curl -XPOST "http://localhost:9200/rni_plugin/_parameter_universe" -H 'Content-Type: application/json' -d'
  {
    "profiles": [
      {
        "name": "SomeParameterUniverseName/any",
        "parameters": {
          "translatorResultsToKeep": 4,
          "deletionScore": 0.269,
          "doQueryTokenOverrides": true,
          "fieldDeletionScore": 0.27,
          "yearDistanceWeight": 0.2
        }
      },
      {
        "name": "SomeParameterUniverseName/eng_eng",
        "parameters": {
          "HMMUsageThreshold": 0.8,
          "stringDistanceThreshold": 0.1,
          "useEditDistanceTokenScorer": true,
          "finalBias": 2.4,
          "reorderPenalty": 0.2
        }
      }
    ]
  }'
```

11.3.2. Update Parameter(s)

The `POST _parameter_universe/_update` updates the values of existing parameters. You need to provide the parameter universe name along with the parameters and their values. If a parameter doesn't exist it is ignored.

```
curl -XPOST "http://localhost:9200/rni_plugin/_parameter_universe/_update" -H 'Content-Type: application/json' -d'{ "prof
      "name": "SomeParameterUniverseName/any",
      "parameters": {
        "translatorResultsToKeep": 4
      }
    },
    {
      "name": "SomeParameterUniverseName/eng_eng",
      "parameters": {
        "useEditDistanceTokenScorer": false,
        "finalBias": 2.4,
        "reorderPenalty": 0.2
      }
    }
  ]
}'
```

You can also use query parameters to update the value of a parameter. The name of the parameter universe is included as a path parameter. You must then specify two query parameters: `param` and `value`. `param` is made up of the language profile followed by the name of the parameter, `value` contains the value of the parameter.

```
curl -XPOST "http://localhost:9200/rni_plugin/_parameter_universe/SomeParameterUniverseName\
/_update?param=eng_eng.reorderPenalty&value=0.2"
```

11.3.3. Get Parameter(s)

The `GET _parameter_universe` method retrieves a given parameter universe. The name of the parameter universe is provided as a path parameter:

```
curl -XGET "http://localhost:9200/rni_plugin/_parameter_universe/SomeParameterUniverseName"
```

If you add the name of the profile and parameter, it returns the value of the parameter.

```
curl -XGET "http://localhost:9200/rni_plugin/_parameter_universe/SomeParameterUniverseName/eng_eng.reorderPenalty"
```

11.3.4. Delete Parameter(s)

The `DELETE _parameter_universe` method deletes the parameter universe provided as a path parameter.

```
curl -XDELETE "http://localhost:9200/rni_plugin/_parameter_universe/SomeParameterUniverseName"
```

If you add the name of the profile and parameter, it deletes the parameter from the parameter universe. It will then use the value in the `parameter_defs.yaml` file.

```
curl -XDELETE "http://localhost:9200/rni_plugin/_parameter_universe/SomeParameterUniverseName/eng_eng.reorderPenalty"
```

12. Pairwise Match Endpoint

You can perform a pairwise match between two rni_names, rni_dates, rni_addresses, or other datatypes [61] through the `POST _pair_match` method. The results provide insight into how the match scores were calculated, including tokens and token scores. This endpoint can help you understand the impact a specific match parameter has on the final score, and can aid in testing and debugging RNI.

The type of pairwise match being performed is provided to the query, along with the values being compared (`data1` and `data2`). You can also specify one or more parameters and see how they impact the match scores.

Request

```
curl -XPOST "http://localhost:9200/rni_plugin/_pair_match? type=rni_date" -H 'Content-Type: application/json' -d'
{"dataPair": {"data1": "12/25/19","data2": "1/15/20"},
 "parameters": {
     "timeDistanceWeight": ".8",
     "stringDistanceWeight": "0"}}'
```

Response

```
{
  "score" : 0.730683530798762,
  "type" : "ORIGINAL",
  "preSwapPenaltyScore" : 0.730683530798762,
  "swapPenaltyFactor" : 1.0,
  "preFinalBiasScore" : 0.730683530798762,
  "finalBias" : 1.0,
  "debugInfo" : """[{"name":"TIME","weight":0.8,"score":0.6949591099211685},
                   {"name":"YEAR","weight":0.2,"score":0.9330329915368074},
                   {"name":"MONTH","weight":0.2,"score":0.6830201283771977},
                   {"name":"DAY","weight":0.1,"score":0.7071067811865476},
                   {"name":"STRING","weight":0.0,"score":0.375}]"""
}
```

12.1. Supported Types

The following data types are supported by the pairwise match endpoint.

- rni_name
- rni_date
- rni_address
- date
- keyword
- text
- string
- integer
- long
- short
- double
- float
- boolean
- geo_point

Request

```
curl -XPOST "http://localhost:9200/rni_plugin/_pair_match?type=text " -H 'Content-Type: application/json' -d'
{
   "dataPair":
   {
    "data1": "word1",
    "data2": "word2"
  }
}'
```

Response

```
{
  "score" : 0.8333333333333334
}
```

12.2. Name Matching Example

Request

Parameters are specified directly in the request. The source language (`language`) of the name is optional, but recommended if known.

```
curl -XPOST "http://localhost:9200/rni_plugin/_pair_match?type=rni_name" -H 'Content-Type: application/json' -d'
{
  "dataPair":
   {
     "data1":
     {
      "data": "John Robert Edward Smith",
      "language": "eng",
      "entityType": "PERSON"

    },
    "data2":
    {
      "data": "John Smyth",
      "language": "eng",
      "entityType": "PERSON"
    }
  },
  "parameters": {
    "deletionScore": 0.469
  }
}'
```

Response

The response includes detailed information on how the names were matched [16].

```
  "score": 0.8679031451202058,
  "type": "TOKEN_BY_TOKEN",
  "avgMatchedTokenLMBinLeft": 1,
  "avgMatchedTokenLMBinRight": 1,
  "annotations": [
    {
      "type": "One-sided deletion boost",
      "parameter": 0,
      "oldScore": 0.7033713195696144,
      "newScore": 0.7324475249463439
    },
    {
      "type": "Final bias",
      "parameter": 0,
      "oldScore": 0.7324475249463439,
      "newScore": 0.8679031451202058
    }
  ],
  "leftTokens": [
    {
      "data": "john",
      "field": 0,
      "originalData": "john",
      "spanStart": 0,
      "spanEnd": 4,
      "type": "UNKNOWN",
      "weight": 0.2547722342733189
    },
    {
      "data": "robert",
      "field": 0,
      "originalData": "robert",
      "spanStart": 5,
      "spanEnd": 11,
      "type": "UNKNOWN",
      "weight": 0.24522776572668115
    },
    {
      "data": "edward",
      "field": 0,
      "originalData": "edward",
      "spanStart": 12,
      "spanEnd": 18,
      "type": "UNKNOWN",
      "weight": 0.24522776572668115
    },
    {
      "data": "smith",
      "field": 0,
      "originalData": "smith",
      "spanStart": 19,
      "spanEnd": 24,
      "type": "UNKNOWN",
      "weight": 0.2547722342733189
    }
  ],

}
```

```
  "rightTokens": [
    {
      "data": "john",
      "field": 0,
      "originalData": "john",
      "spanStart": 0,
      "spanEnd": 4,
      "type": "UNKNOWN",
      "weight": 0.5
    },
    {
      "data": "smyth",
      "field": 0,
      "originalData": "smyth",
      "spanStart": 5,
      "spanEnd": 10,
      "type": "UNKNOWN",
      "weight": 0.5
    }
  ],"spanMatches": [
    {
      "leftSpan": "Span(0, 4)",
      "rightSpan": "Span(0, 4)",
      "reason": "MATCH"
    },
    {
      "leftSpan": "Span(19, 24)",
      "rightSpan": "Span(5, 10)",
      "reason": "HMM_MATCH"
    },
    {
      "leftSpan": "Span(5, 18)",
      "rightSpan": null,
      "reason": "DELETION"
    }
  ],
  "finalTuples": [
    {
      "originalScore": 1,
      "packingScore": 3.019088937093276,
      "score": 1,
      "reason": "MATCH",
      "leftIndex0": 0,
      "leftIndex1": 0
    },
    {
      "originalScore": 0.7237045473947534,
      "packingScore": 2.730932483146512,
      "score": 0.7237045473947534,
      "reason": "HMM_MATCH",
      "leftIndex0": 3,
      "leftIndex1": 3
    },
    {
      "originalScore": 0.469,
      "packingScore": 0,
      "score": 0.469,
      "reason": "DELETION",
      "leftIndex0": 1,
      "leftIndex1": 2
    }
  ],
  "otherTuples": [],
  "debugInfo": """
Begin [ John Robert Edward Smith Latn eng:eng NONE (john robert edward smith)] [ John Smyth Latn eng:eng NONE (john smyth
```

```
 -- Token data ------------
 john (john) bin=1.0 (w/bias = 1.0000)
 robert (robert) bin=1.0 (w/bias = 1.0000)
 edward (edward) bin=1.0 (w/bias = 1.0000)
 smith (smith) bin=1.0 (w/bias = 1.0000)
 john (john) bin=1.0 (w/bias = 1.0000)
 smyth (smyth) bin=1.0 (w/bias = 1.0000)
 --------------------------
 john/25@0:0 john/50@0:0 -> 1.0000 <+S>
 john/25@0:0 smyth/50@0:1 -> 0.0000 <null>
 robert/25@0:1 john/50@0:0 -> 0.0000 <null>
 robert/25@0:1 smyth/50@0:1 -> 0.0000 <null>
 edward/25@0:2 john/50@0:0 -> 0.0000 <null>
 edward/25@0:2 smyth/50@0:1 -> 0.0000 <null>
 smith/25@0:3 john/50@0:0 -> 0.0000 <null>
 smith/25@0:3 smyth/50@0:1 -> 0.7237 <pi[4]=0.4782 pj[4]=0.4402 +S>
 johnrobertedwardsmith/100@0:0..3 johnsmyth/100@0:0..1 => 0.0000
 -- All Tuples ------------
  john/25@0:0 == john/50@0:0        t=1.0000 MATCH (s=2 q=3.0191 o=1.0000)
  smith/25@0:3 == smyth/50@0:1        t=0.7237 HMM_MATCH (s=2 q=2.7309 o=0.7237)
 --------------------------
john/25@0:0 == john/50@0:0        t=1.0000 MATCH (s=2 q=3.0191 o=1.0000)
smith/25@0:3 == smyth/50@0:1        t=0.7237 HMM_MATCH (s=2 q=2.7309 o=0.7237)
robertedward/49@0:1..2 == <DEL>        t=0.4690 DELETION (s=0 q=0.0000 o=0.4690)
One-sided deletion boost: 0.7034 -> 0.7324
Final bias: 0.7324 -> 0.8679
Score = 0.8679: [ John Robert Edward Smith Latn eng:eng NONE (john robert edward smith)] [ John Smyth Latn eng:eng NONE (
 -- Token data ------------
 john (john) bin=1.0 (w/bias = 1.0000)
 robert (robert) bin=1.0 (w/bias = 1.0000)
 edward (edward) bin=1.0 (w/bias = 1.0000)
 smith (smith) bin=1.0 (w/bias = 1.0000)
 john (john) bin=1.0 (w/bias = 1.0000)
 smyth (smyth) bin=1.0 (w/bias = 1.0000)
 --------------------------
Score[alt] = 0.0000: [ John Robert Edward Smith Latn eng:eng NONE (john robert edward smith)] [ John Smyth Latn eng:eng N
End 0.8679

"""
}
```

12.3. Address Matching Example

Request

The pairwise match endpoint supports both fielded and unfielded addresses. Fielded addresses must be specified as objects, while unfielded addresses must be specified as strings.

```
curl -XPOST "http://localhost:9200/rni_plugin/_pair_match?type=rni_address" -H 'Content-Type: application/json' -d'
{
  "dataPair":
   {
    "data1":
     {
      "houseNumber": "101",
      "road": "Main st",
      "city": "Cambridge",
      "state": "Massachusetts",
      "country": "United States of America"
    },
    "data2": "101 Main St, Cambridge, MA, USA"
  }
}'
```

Response

The response includes a `score` marking the similarity of the two addresses as well as a `type` field describing the type of match observed. The response also includes detailed information on how each of the fields were matched. In the example below, only part the detailed response for `HOUSE_NUMBER` is included.


```
{
   "score":0.9,
   "type":"OTHER",
   "annotations":[
      {
         "type":"Final bias",
         "parameter":0.0,
         "oldScore":0.9,
         "newScore":0.9
      }
   ],
   "addressFieldPairResults":[
      {
         "leftField":"HOUSE_NUMBER",
         "rightField":"HOUSE_NUMBER",
         "score":1.0,
         "spanMatches":[
            {
               "leftSpan":{
                  "start":0,
                  "end":3,
                  "length":3
               },
               "rightSpan":{
                  "start":0,
                  "end":3,
                  "length":3
               },
               "reason":"MATCH"
            }
         ],
         "leftTokens":[
            {
               "data":"101",
               "field":0,
               "originalData":"101",
               "spanStart":0,
               "spanEnd":3,
               "type":"NONE",
               "weight":1.0
            }
         ],
         "rightTokens":[
            {
               "data":"101",
               "field":0,
               "originalData":"101",
               "spanStart":0,
               "spanEnd":3,
               "type":"NONE",
               "weight":1.0
            }
         ],
         ...
        "debugInfo":""
      },
{
  "score": 0.9,
  "type": "ORIGINAL"
}
...
```

13. Supported Text Domains for Name Indexing and Name Matching

This section includes two tables designed to answer the following three questions:

1. Which language and writing script combinations does indexing support for index entries?

NOTE
"Language" in this appendix refers to the language of use, the language of the document in which the name is found, which may not be the language of origin associated with the name. If the language of use is undetermined, use Unknown ["xxx"]

2. A query in a given language and writing script may return results in what set of language and writing script combinations?

3. For name matching, a query name in a given language and writing script may be matched against a reference name in which language and writing script combinations?

13.1. Name Matching Within a Language

The first table identifies the languages, and for each language the writing scripts that **Rosette Name Indexer** supports.

You can use **Rosette Name Indexer** to index names in any of the language-script combinations in this table.

Language (ISO 639-3)	Scripts (ISO 15924)
Arabic (ara)	Arabic (Arab)
Burmese (mya)	Burmese (Mymr)
Chinese (zho)	Han (Hanzi) (Hani), Han (Simplified variant) (Hans), Han (Traditional variant) (Hant)
English (eng)	Latin (Latn)
French (fra)	Latin (Latn)
German (deu)	Latin (Latn)
Greek (ell)	Greek (Grek)
Hebrew (heb)	Hebrew (Hebr)
Hungarian (hun)	Latin (Latn)
Italian (ita)	Latin (Latn)
Japanese (jpn)	Han (Kanji) (Hani), Hiragana (Hira), Japanese syllabaries (alias for Hiragana + Katakana) (Hrkt), Japanese (alias for Han + Hiragana + Katakana) (Jpan), Katakana (Kana)
Korean (kor)	Hangul (Hangŭl, Hangeul) (Hang), Han (Hanja) (Hani), Korean (alias for Hangul + Han) (Kore)
Pashto (pus)	Arabic (Arab)
Persian (fas) [a]	Arabic (Arab)
Persian, Afghan (prs)	Arabic (Arab)
Persian, Iranian (pes)	Arabic (Arab)
Portuguese (por)	Latin (Latn)
Russian (rus)	Cyrillic (Cyrl)
Spanish (spa)	Latin (Latn)
Thai (tha)	Thai (Thai)

Language (ISO 639-3)	Scripts (ISO 15924)
Urdu (urd)	Arabic (Arab)
Vietnamese (vie)	Latin (Latn)

[a]Persian is the macrolanguage that includes Afghan Persian ("prs") and Iranian Persian ("pes")

13.2. Cross-Language Matches

This table identifies the range of cross-language searching and matching that **Rosette Name Indexer** and name matching support. If your query is a name in an Arabic document in Arabic script, the query may return one or more names in English documents in Latin script, in addition to names from Arabic documents in Arabic script. If the query is a name in English and Latin script, it may return documents from any of the supported languages and their native scripts.

Query Domain		Index Domain / Match Domain	
Language (ISO 639-3)	**Script (ISO 15924)**	**Language (ISO 639-3)**	**Scripts (ISO 15924)**
Arabic (ara)	Arabic (Arab)	Arabic (ara)	Arabic (Arab)
		English (eng)	Latin (Latn)
Burmese (mya)	Burmese (Mymr)	Burmese (mya)	Burmese (Mymr)
		English (eng)	Latin (Latn)
Chinese (zho)	Han (Hani),(Hans),(Hant)	Chinese (zho)	(Hani), (Hans), (Hant)
		English (eng)	Latin (Latn)
		Japanese (jpn)	(Hani), (Hira),(Jpan), (Hrkt), (Kana)
		Korean (kor)	(Hani), (Hang), (Kore)
English (eng)	Latin (Latn)	Arabic (ara)	Arabic (Arab)
		Afghan Persian (prs)	Arabic (Arab)
		Burmese (mya)	Burmese (Mymr)
		Chinese (zho)	(Hani), (Hans), (Hant)
		English (eng)	Latin (Latn)
		French (fra)	Latin (Latn)
		German (deu)	Latin (Latn)
		Greek (ell)	Greek (Grek)
		Hebrew (heb)	Hebrew (Hebr)
		Hungarian (hun)	Latin (Latn)
		Iranian Persian (pes)	Arabic (Arab)
		Italian (ita)	Latin (Latn)
		Japanese (jpn)	(Hani), (Hira),(Jpan), (Hrkt), (Kana)
		Korean (kor)	(Hani), (Hang), (Kore)
		Pashto (pus)	Arabic (Arab)
		Persian (fas)	Arabic (Arab)
		Portuguese (por)	Latin (Latn)
		Russian (rus)	Cyrillic (Cyrl)
		Spanish (spa)	Latin (Latn)
		Thai (tha)	Thai (Thai)
		Urdu (urd)	Arabic (Arab)
		Vietnamese (vie)	Latin (Latn)
French (fra)	Latin (Latn)	English (eng)	Latin (Latn)
		French (fra)	Latin (Latn)
German (deu)	Latin (Latn)	English (eng)	Latin (Latn)
		German (deu)	Latin (Latn)

Query Domain		Index Domain / Match Domain	
Language (ISO 639-3)	**Script (ISO 15924)**	**Language (ISO 639-3)**	**Scripts (ISO 15924)**
Greek (ell)	Greek (Grek)	English (eng)	Latin (Latn)
		Greek (ell)	Greek (Grek)
Hebrew (heb)	Hebrew (Hebr)	English (eng)	Latin (Latn)
		Hebrew (heb)	Hebrew (Hebr)
Hungarian (hun)	Latin (Latn)	English (eng)	Latin (Latn)
		Hungarian (hun)	Latin (Latn)
Italian (ita)	Latin (Latn)	English (eng)	Latin (Latn)
		Italian (ita)	Latin (Latn)
Japanese (jpn)	(Hani), (Hira),(Jpan),(Hrkt),(Kana)	Chinese (zho)	(Hani), (Hans), (Hant)
		English (eng)	Latin (Latn)
		Japanese (jpn)	(Hani), (Hira),(Jpan), (Hrkt), (Kana)
		Korean (kor)	(Hani), (Hang), (Kore)
Korean (kor)	(Hani), (Hang), (Kore)	Chinese (zho)	(Hani), (Hans), (Hant)
		English (eng)	Latin (Latn)
		Japanese (jpn)	(Hani), (Hira),(Jpan), (Hrkt), (Kana)
		Korean (kor)	(Hani), (Hang), (Kore)
Pashto (pus)	Arabic (Arab)	English (eng)	Latin (Latn)
		Pashto (pus)	Arabic (Arab)
Persian[a] (fas)	Arabic (Arab)	English (eng)	Latin (Latn)
		Persian (fas)	Arabic (Arab)
Persian, Afghan (prs)	Arabic (Arab)	Afghan Persian (prs)	Arabic (Arab)
		English (eng)	Latin (Latn)
Persian, Iranian (pes)	Arabic (Arab)	English (eng)	Latin (Latn)
		Iranian Persian (pes)	Arabic (Arab)
Portuguese (por)	Latin (Latn)	English (eng)	Latin (Latn)
		Portuguese (por)	Latin (Latn)
Russian (rus)	Cyrillic (Cyrl)	English (eng)	Latin (Latn)
		Russian (rus)	Cyrillic (Cyrl)
Spanish (spa)	Latin (Latn)	English (eng)	Latin (Latn)
		Spanish (spa)	Latin (Latn)
Thai (tha)	Thai (Thai)	English (eng)	Latin (Latn)
		Thai (tha)	Thai (Thai)
Urdu (urd)	Arabic (Arab)	English (eng)	Latin (Latn)
		Urdu (urd)	Arabic (Arab)
Vietnamese (vie)	Latin (Latn)	English (eng)	Latin (Latn)
		Vietnamese (vie)	Latin (Latn)

[a]Persian is the macrolanguage that includes Afghan Persian ("prs") and Iranian Persian ("pes")