BlackLynx Open API Library User Guide

Revision History

Date	Changes	Minimum and/or Release Software Revision(s)
20190309	• In addition to combination LONG/LAT (or LAT/LONG) fields, PIP now supports separated LONG and LAT fields	r1643
20190107	• Added ARP decodes to PCAP. • PIP vertex files now have more liberal number separator rules.	r1513

20181024	• Currency symbols are now optional in the numeric portions of the currency primitive's query clause.
	• CSV, JSON, and XML are now escape-friendly according to RFC 4180, http://json.org (as of 20180822), and http://www.w3.org/TR/xml (as of 20180822), respectively.
	  ○ An UNESCAPE_DATA=false option is added to not perform escape checking, but this is not recommended.
	  ○ Search and replace operations now handle escaping rules when generating structured output as well.
	• Added the new EMPTY_FIELD primitive.
	• Added two new global options primarily targeting forensics use cases: find-input-recursively and input-is-filelist.
	• Various PCAP primitive enhancements and improvements:
	  ○ Match-associated TCP stream dumps, UDP dumps, and payload dumps can now be generated.
	  ○ New meta field family frame.time[_...] is now available.
	  ○ Output index file generation is now supported, to include options enabling match qualifiers and packet number output. This resulted in an API call change, so existing user programs leveraging the PCAP primitive will need to be updated.
	  ○ IP fragmentation now supports the capability to intelligently filter on the individual fragments for those query operations that do not revolve around payload searches. For payload searches, the properly combined fragmented payloads are leveraged for the search. In all cases, in the output data file that is generated upon request, fragmented source packets now properly appear in-order with respect to other source corpus packets.
	  ○ Added ip.proto and ipv6.nxt PCAP filter support.
	• Extended and enhanced several primitive descriptions to be more complete.
	• Updated all examples and added a few new ones throughout this guide.
	• The Java and Python bindings have been revamped. Please note that existing Java and Python user programs may need to be modified to match the updated API bindings.
		r1344
		r1293

Table of Contents

1 Scope

This document explains the usage of the BlackLynx Open API Library. It describes the functions available in the API and explains the details of their usage. Additionally, usage examples are provided as a guide.

2 Overview

This API describes the interface between end user programs and BlackLynx's powerful parallel software framework supporting a variety of COTS x86 hardware, including support for PCIe acceleration boards such as Xilinx SDAccel-enabled FPGAs. The API and BlackLynx software also run on a variety of Amazon Web Services (AWS) instances, including but not limited to the I3, C4, C5, and F1 instances.

The libraries are provided as a standard Linux shared-object dynamic library (e.g., libryftx.so and libryftx_pcap.so), providing a set of accelerated primitive operations that can be performed on a set of data. User programs can then be written to execute a sequence of primitive operations on arbitrary files residing on standard linux file systems.

A C language library is provided, and can be natively accessed using either the C or C++ languages.

Native language bindings for both Python and Java are also provided.

Other languages, such as R and Scala, are easily supported using standard wrapper function mechanisms.

Third party applications can leverage the API to connect to arbitrary standards-based interfaces, such as ODBC, JDBC, RESTful interfaces, and so on. BlackLynx makes several of these connectors openly available to BlackLynx customers. More details about these alternate APIs can be found at https://www.ryft.com/api. Additionally, the latest version of this API Guide can always be found at that link.

3 Available Functions

The API provides a basic set of functions:

- Dataset Manipulation Functions
- Primitives
- Return Error Information
- Return Operation Statistics

These and related topics are discussed in the remainder of this guide.

3.1 Open API Library Dataset

While many parameters used by the API are standard C language types, `rx_data_set_t` is not. The `rx_data_set_t` is an opaque data type that is used to reference the dataset that is being operated on by calls to the API. Typically, this data type consists of a collection of input files along with metadata about them. The same opaque structure is used for results from various library functions. When operating as a results type, the data type contains metadata and statistics relating to operations that are invoked against an input-side dataset.

For example, when using the library's search functions, the data to be searched will be of the `rx_data_set_t` type and the result of that search will also be of the `rx_data_set_t` type. The `rx_data_set_t` type cannot be used outside of the Open API. The following section details the usage and manipulation of datasets.

3.2 Dataset Manipulation Functions

In order to be able to perform operations, a dataset must first be created, which defines the data to be used as part of an operation. Once a dataset is created, the user must call the `rx_ds_add_file()` function to add input files to the dataset. Once all of the input files to be analyzed are added to a dataset, the user can then use API primitives to operate on the dataset.

For every operation on a dataset, a new dataset will be produced. This new dataset is used to reference the results of an operation, which can be examined using the "return error information" and "return operation statistics" functions as detailed in section 3.4 and section 3.5, respectively.

Below are the functions to use to create, delete, or add files to a dataset destined to be operated on by API primitives:

```
rx_data_set_t rx_ds_create();

rx_data_set_t rx_ds_create_with_nodes(
    const uint8_t number_of_processing_nodes);

bool rx_ds_add_file(
    const       rx_data_set_t data_set,
    const char* filename_to_add);

void rx_ds_delete(rx_data_set_t* data_set);
```

- `number_of_processing_nodes` is an integer specifying the number of processing nodes that the algorithm desires to use. The system will attempt to allocate this number of acceleration nodes if they are applicable and available. If the number requested is not available, the system will attempt to use the number that are available. If acceleration nodes are installed but are all currently in use, then if the BlackLynx control logic

detects that the operation would benefit from hardware acceleration, it is placed into a waiting first-in/first-out queue and executed once one or more acceleration nodes become available. If the control logic determines that the operation would not benefit from hardware acceleration, the operation will run in a software-only environment. Terminating a program removes it from its place in the queue and any of its associated resources are released.

- `data_set` is a dataset of type `rx_data_set_t` to be operated on. In this case, a filename will be added to the dataset.
- `filename_to_add` is the filename to add to the dataset, and may contain wildcard characters such as '`*`'.

Note that there are two different functions to create a new dataset – one which requires the user to specify the number of nodes to use during execution and one that does not.

Nodes are an abstract concept managed internally by the system, but in general, when targeting hardware acceleration platforms, the number of nodes refers to the number of hardware acceleration nodes to use for the requested operation. When all-software environments are employed, the requested number of nodes is effectively ignored. For software-only throttling mechanisms, please refer to the `max-spawns` option detailed in section 3.3.3.

If the user creates a new dataset using `rx_ds_create()`, the number of nodes to be used will default to the maximum number of nodes available on the system. With `rx_ds_create_with_nodes()`, the user may request to use a specific number of the available nodes.

Any operations performed on a dataset will be performed on all of the data contained in all of the specified files in the set.

Once a set of data files is processed, the results can be written to a variety of output file types pursuant to the type of operation requested. These output files are detailed throughout this guide.

Once a dataset is no longer needed, it should be deleted by calling the `rx_ds_delete()` function in order to free the appropriate system resources.

3.3 Primitives

API primitives are used to perform a variety of operations related to high performance data analytics on arbitrary input data. The input data can be organized as either unstructured raw data or record-based data. Depending on the type of operation, output can be an index file, a data file, both an index file and a data file, neither, or other types of output.

3.3.1 Primitive Criteria Specification

Various API primitive functions such as the `rx_ds_search` function requires a `match_criteria_query_string` parameter to specify how the operation should be performed. Criteria are made up of one or more expressions, potentially connected using logical operations. The API defines a query language grammar, consisting of an expression which takes the following form:

```
([input_specifier relational_operator ]primitive([expression[, options]]))
```

`input_specifier` specifies how the input data is arranged. The possible values are:

input_specifier Values	Description
RAW_TEXT	The input is a sequence of raw bytes with no implicit formatting or grouping.
RECORD	The input is a series of records. Operate on all records.
RECORD.foo.bar	The input is a series of records. Operate only on the field whose hierarchy is spelled out by (in this example) foo.bar in each record. Note: for JSON and XML input records, field name hierarchies can be specified with '.' separators between them to specify a field hierarchy. For JSON, '[]' separators can be included as appropriate to specify array hierarchy. Whitespace is permitted in JSON fields if the individual field(s) containing whitespace are surrounded in double quotes. For CSV, <field_name> can be either a column name surrounded in double quotes, or a numeric index, where the leftmost column is column 1.

For ease of integration into downstream data analysis ecosystems, note that results from RECORD or RECORD.foo.bar operations include the entire matching records as opposed to partial records.

relational_operator specifies how the input relates to the expression. The possible values are:

relational_operator Values	Description
EQUALS	The input must match expression either exactly for an exact search or within the specified distance for a fuzzy search, with no additional leading or trailing data. Note that this operator has meaning only for record- and field-based searches. If used with raw text input, an error will be generated. When searching raw text data, CONTAINS should be used instead of EQUALS.
NOT_EQUALS	The input must be anything other than expression. Note that this operator has meaning only for record- and field-based searches. If used with raw text input, an error will be generated.
CONTAINS	The input must contain expression. It may also contain additional leading or trailing data.
NOT_CONTAINS	The input must not contain expression. Note that this operator has meaning only for record- and field-based searches. If used with raw text input, an error will be generated. If you desire to determine whether an expression does not exist in raw text input, you should use CONTAINS, and check for zero results.

primitive specifies the primitive associated with the clause. The primitives are discussed in significantly more detail in subsequent sections. The possible values are:

primitive Values	Short Description
EXACT	Search for an exact match.
HAMMING	Perform a fuzzy search using the Hamming distance algorithm.
EDIT_DISTANCE	Perform a fuzzy search using the edit distance (Levenshtein) algorithm.
PCRE2	Search using a PCRE2-compliant regular expression.
DATE	Search for a date or a range of dates.
TIME	Search for a time or a range of times.
NUMBER	Search for a number or a range of numbers.
CURRENCY	Search for a monetary value or a range of monetary values.
IPV4	Search for an IPv4 address or a range of IPv4 addresses.
IPV6	Search for an IPv6 address or a range of IPv6 addresses.

primitive Values	Short Description
EMPTY_FIELD	Search record- or field- based input to detect a field which exists, but whose contents are empty.
PIP	Search geo-location-based data for records inside (or outside) of arbitrary polygons.
PCAP	Search raw PCAP files filtering a variety of information across layers 2-7.
NGRAM	Generate n-gram (unigram, bigram, or trigram) tables for any input corpus.
TERM_FREQUENCY	Generate term frequency (n-gram with n=1) tables for any input corpus.
DOCSIM_TRAINING	Create a trained machine learning model for downstream DOCSIM operations.
DOCSIM	Perform a document similarity machine learning request against a previously trained DOCSIM_TRAINING model.

expression specifies the expression associated with the primitive. The possible values are:

expression Values	Description
Quoted string	Any valid C language string, including backslash-escaped characters. For example, “match this text\n”. This can also include escaped hexadecimal characters, such as “match this text\x0A”, or “\x48\x69\x20\x54\x68\x65\x72\x65\x21\x0A\x00”. If a backslash needs to be placed in the quoted string, use the double backsplash escape sequence “\\”.
Wildcard	A “?” character is used to denote that any **single** character will match. A “?” can be inserted at any point(s) **between** quoted strings. Wildcards are supported only for exact and fuzzy hamming requests. Other uses will result in an error message. For example, “match th”?”s text\n”
Any combination of the above	For example, “match\x20th”?”s text\x0A”, or “match\x20with a wildcard right here”?”and a null at the end\x00”

options specify a comma-separated list of options that can further qualify the request for certain primitives. The possible values are described in section 3.3.2.

logical_operator allows for complex collections of expressions. The possible values are:

logical_operator Values	Description
AND	The logical expression (**a** AND **b**) evaluates to true only if both the expression **a** evaluates to true and the expression **b** evaluates to true.
OR	The logical expression (**a** OR **b**) evaluates to true if either the expression **a** evaluates to true or the expression **b** evaluates to true.
XOR	The logical expression (**a** XOR **b**) evaluates to true if either the expression **a** evaluates to true or the expression **b** evaluates to true, but not both.

Multiple relational_expressions can be combined using the logical operators AND, OR, and XOR, for those primitives where multiple results correlation are appropriate. For example:

```
(RECORD.city CONTAINS EXACT("Rockville")) AND (RECORD.state CONTAINS EXACT("MD"))
```

Parentheses can also be used to control the precedence of operations. Additional whitespace is allowable, which can simplify comprehension. For example:

```
((RECORD.city CONTAINS EXACT("Rockville")) OR (RECORD.city CONTAINS
EXACT("Gaithersburg")) ) AND (RECORD.state CONTAINS EXACT("MD"))
```

Primitives can be mixed-and-matched in the same expression grouping. For example:

```
((RECORD.city CONTAINS EXACT("Rockville")) OR (RECORD.city CONTAINS
EDIT_DISTANCE("Gaithersburg", DISTANCE="1")) ) AND (RECORD.zip CONTAINS NUMBER(NUM >=
"20855", SEPARATOR=",", DECIMAL="."))
```

3.3.2 Parameters and Options List

This subsection details the various possible parameters and options that can be included as part of a configuration expression for various primitives.

Note that it is permissible to include valid but extraneous options, in which case they will be ignored. For example, if a `DISTANCE` option is specified with an `EXACT` primitive, the `DISTANCE` option will be ignored and the operation will still execute.

- `WIDTH="value"`
  - Default: `0`
  - Search Primitive Compatibility: All
  - Other Primitive Compatibility: N/A
  - Description: Specifies a surrounding width as an unsigned 16-bit integer `value`, allowing for surrounding widths of up to 65,535 bytes. Surrounding width means that results will be returned which will contain the specified number of characters (`value`) to the left and right of the match. Note that `WIDTH` and `LINE` are mutually exclusive. Attempting to specify both in the same query will result in an error. When performing `RECORD` or `RECORD.field` operations, width requests are ignored since entire records are always returned.
- `LINE="[true|false]"`
  - Default: `false`
  - Search Primitive Compatibility: All
  - Other Primitive Compatibility: N/A
  - Description: When true, the query will return the (line-feed delimited) line on which the match occurs. Should a match appear multiple times on a single line, only one match line will be generated. If a nearby line feed cannot be found, then the data results are undefined, and become implementation-specific. Note that `WIDTH` and `LINE` are mutually exclusive. Attempting to specify both in the same query will result in an error message. When performing `RECORD` or `RECORD.field` operations, line requests are ignored since entire records are always returned.
- `CASE="[true|false]"`
  - Default: `true`
  - Search Primitive Compatibility: EXACT, HAMMING, and EDIT_DISTANCE
  - Other Primitive Compatibility: NGRAM, TERM_FREQ
  - Description: When false, the query will be run case-insensitive. By default, searches are case sensitive.
- `DICTIONARY="/path/to/dictionary/filename"`
  - Default: N/A
  - Search Primitive Compatibility: EXACT, HAMMING, EDIT_DISTANCE, PCRE2
  - Other Primitive Compatibility: N/A
  - Description: Specifies a query dictionary filename to use, which can be either a relative or absolute filename. When this feature is used, the common primitive query clause must be specified as a null quantity by using back quotes (ie: ""), and the dictionary file referenced should list, one per line, the double-quoted query strings to utilize, exactly how they would appear in the standard expression syntax. For example, a first line of `"Michele"` and a second line `"Sm"?"th"` (`Sm` followed by a wildcard character followed by `th`) would return a match when either (or both) of those two query lines matched. This can be very useful when there is a need to search a common input corpus for a multitude

of values. This could of course be specified as a long sequence of OR clauses, but a query dictionary is often much simpler to maintain over time. Although not required, when using a query dictionary, it is often useful to employ the global `enable-match-qualifiers` option, which appends match qualifier information to the output index file lines, making it much easier for downstream identification of the criteria responsible for each output value. For more details, see section 3.3.5.

- `DISTANCE="value"`
  - Default: `0`
  - Search Primitive Compatibility: HAMMING, EDIT_DISTANCE
  - Other Primitive Compatibility: N/A
  - Description: Specifies the fuzzy search distance for a given Hamming or Edit Distance search primitive. A match is determined if the distance between the match and the search `expression` is less than or equal to this `DISTANCE` option. The resulting distance value for each match encountered is written to the output index file, if specified, which allows for downstream analytics tools to know how close the match was to the request. For `RECORD` based searching (where `input_specifier` is `RECORD` or `RECORD.<field_name>`), the distance is not reported in the output index file, since the purpose of `RECORD` based searching is to return the entire matching `RECORD` which can often have more than one internal match, each of which could have a different distance.
- `REDUCE="[true|false]"`
  - Default: `false`
  - Search Primitive Compatibility: EDIT_DISTANCE
  - Other Primitive Compatibility: N/A
  - Description: When set to true, results will be reduced, thereby eliminating duplicate matches. Duplicate matches are common given how the edit distance (Levenshtein) algorithm works, where it counts insertions, deletions and replacements when calculating fuzziness for the match. For example, if you are searching for “giraffe” with distance less than or equal to one, and the input corpus is “That giraffe is tall,” then three results would be generated: 1) " giraffe", 2) "giraffe", and 3) "iraffe". The first inserts one space before, so that is a match with distance one. The second is an obvious exact match (distance of zero, which is less than one). The third is missing the leading ‘g’, for a distance of one. All three are reported. With this option set to true, the results are reduced such that the “best” match in a cluster of matches will be reported which in this example would have been "giraffe". It qualifies as the best match of the three clustered matches since it represented a match of distance zero.
- `OCTAL="[true|false]"`
  - Default: `false`
  - Search Primitive Compatibility: IPV4
  - Other Primitive Compatibility: N/A
  - Description: By default, the API defines that IPv4 address octets that have leading 0’s will be parsed as standard decimal numbers. Enabling the `OCTAL` option will instead parse IP address octets that have leading 0’s as octal quantities. This option provides a means to select how IPv4 address parsing should occur in situations where a set of input may not strictly adhere to industry guidance for IPv4 addressing.
- `SYMBOL="char"`
  - Default: `$`
  - Search Primitive Compatibility: CURRENCY
  - Other Primitive Compatibility: N/A
  - Description: Specifies the monetary currency symbol that will be used to identify the start of a potential monetary value for the `CURRENCY` primitive. For example, for some countries, this might be specified as `"#"` instead of the default `"$"` value.
- `SEPARATOR="char"`
  - Default: `N/A`
  - Search Primitive Compatibility: NUMBER, CURRENCY
  - Other Primitive Compatibility: N/A

  - Description: Specifies the digits separator that might appear (but does not have to appear) in numbers or monetary values as they are parsed by various primitives. For example, for US numbers, a comma is traditionally used as a digits separator, so this might be specified as `","`.

- `DECIMAL="char"`
  - Default: `N/A`
  - Search Primitive Compatibility: NUMBER, CURRENCY
  - Other Primitive Compatibility: N/A
  - Description: Specifies the decimal place marker that might appear (but does not have to appear) in numbers or monetary values as they are parsed by various primitives. For example, for US numbers, a decimal point (that is, a period) is used as the decimal place marker, so this might be specified as `"."`.
- `SCINOT="[true|false]"`
  - Default: `true` for number search, `false` for currency search
  - Search Primitive Compatibility: NUMBER, CURRENCY
  - Other Primitive Compatibility: N/A
  - Description: By default, scientific notation is enabled for number search and disabled for currency searches. When disabled, the associated state machines will interpret any encountered `"e"` or `"E"` character as a stop character.
- `PREFACE="YY"`
  - Default: `20`
  - Search Primitive Compatibility: DATE
  - Other Primitive Compatibility: N/A
  - Description: Specifies the two implied leading digits for two year date format validation. The default leading digits are `"20"` which means effective years are 2000-2099. Using the preface option allows this value to be changed to any two digits, such as `"19"`, enabling two-year searches in prior (or future) centuries. Note that if date format validation is undesirable or has other specific considerations, a simplified PCRE2 regular expression primitive is recommended instead of the DATE primitive.
- `ALLOWSINGLE="[true|false]"`
  - Default: `true`
  - Search Primitive Compatibility: DATE, TIME
  - Other Primitive Compatibility: N/A
  - Description: By default, single digit values for hours, minutes, seconds, months and days are allowed in the input corpus, and leading zeros are implied for expansion to two-digit values for ranging comparisons during date and time primitive operations. If strict two-digit comparisons are required, set this option to false. Note that this setting affects only the input corpus match detectors. The configuration string values must always be specified as two-digit quantities.
- `FIELD_DELIMITER="char"`
  - Default: ,
  - Search Primitive Compatibility: All, for CSV data
  - Other Primitive Compatibility: N/A
  - Description: This option can be used as a query option when operating against CSV file formats to change the field delimiter from the default comma to a different character, such as a tab. For a tab, use "`\t`", that is, a backslash followed by a little `t`.
- `RECORD_DELIMITER="X[Y]"`
  - Default: `\n`
  - Search Primitive Compatibility: All, for CSV data
  - Other Primitive Compatibility: N/A
  - Description: This option can be used as a query option when operating against CSV file formats to change the record delimiter from the default line feed to a different character (shown as `X`), or to two characters (shown as optional `[Y]`). For example, suppose the common Windows-style carriage return and line feed pair separates a particular file's CSV records. Then `"\r\n"` would be specified, that is, a backslash followed by little `r` followed by a backslash followed by little `n`.

- `OUTPUT_HEADER="true|false"`
  - Default: `N/A`
  - Search Primitive Compatibility: All, for CSV data
  - Other Primitive Compatibility: N/A
  - Description: By default, the system will prepend a header row to a requested output CSV data file if any query clause in a possibly compound clause references a CSV column by name, and if no such column name exists in the query clause, then no header row is generated. The `OUTPUT_HEADER` option overrides this default behavior, such that when set to true, the header row is always prepended to the requested output CSV data file, and if set to false, the header row is never prepended to the requested CSV output data file. Note that when this option is set to true, the first row of an input data file is implied as header, and is therefore not part of the input data operated on by the primitive. When set to false, the first row of data is considered input data and will be operated on by the primitive.
- `VERBOSE_OUTPUT="true|false"`
  - Default: `false`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: NGRAM, TERM_FREQ
  - Description: By default, the n-gram and term frequency primitives will output data in as terse a format as possible, to limit the output data size, which can be quite large for these types of operations. The `VERBOSE_OUTPUT` option overrides this default behavior, such that when set to true, the output data is more verbose. For specific formatting details, refer to section 3.3.7.3.
- `INPUT_IS_FILELIST="true|false"`
  - Default: `false`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: NGRAM, TERM_FREQ, DOCSIM_TRAINING
  - Description: By default, the input files are one or more specific filenames. When this option is set to true, the input file contains the list of files to be processed. The library will therefore read the contents of the input file, expecting to see one filename per line, and the collection of those filenames become the input data set. This is useful when feeding the n-gram, term frequency, or document similarity training primitives with a list of files for processing. Note that the individual files on each line of the list can contain wildcards (such as '*') which will be properly expanded according to standard linux rules. Although this option is specific to the primitives listed, there is a global option `input-is-filelist` available for use by other primitives to achieve a similar capability. See section 3.3.3 for details.
- `TOP_VALUES="integer"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: NGRAM, TERM_FREQ
  - Description: By default, all terms discovered during an n-gram or term frequency primitive are reported in the results, ordered by frequency in descending order. This option allows the number of reported n-grams or terms to be truncated so that only the number of values selected are reported. For example, if this integer value were set to 10, only the top 10 (or fewer if there weren't at least 10 results) results by frequency would appear in the generated output data file. This option affects only the results that are written to the output data file. It does not impact the statistic for the number of unique n-grams or terms, which remains the total number discovered by the primitive. If used, the integer value specified should be >= 1, otherwise a meaningful error will be generated.
- `INCLUSIVE="true|false"`
  - Default: `false`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: The point in polygon primitive defaults to an exclusive construct, such that location points must be fully inside the specified complex polygon in order to be considered a positive match. Therefore, by default, CONTAINS queries will not report points that lie precisely on the polygon

boundary, and NOT_CONTAINS will report them. When set to true, this behavior is flipped and becomes an inclusive operation instead of an exclusive operation, such that CONTAINS queries will report such points, and NOT_CONTAINS queries will not report them.

- `VERTEX_FILE="string"`
  - Default: `N/A`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: This option specifies the vertex file that will be used by the point in polygon primitive. By default, any `VERTEX_FILE` specified to the PIP primitive is expected to consist of a list of polygon vertices, one per line. This behavior can be modified by the `VERTEX_FILE_IS_FILELIST` option as described below.
- `VERTEX_FILE_IS_FILELIST="true|false"`
  - Default: `false`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: By default, any `VERTEX_FILE` specified to the PIP primitive is expected to consist of a list of polygon vertices, one per line. When this option is set to true, the default behavior is overridden, and the `VERTEX_FILE` is instead a file list allowing for a simple mechanism to use an arbitrary set of multiple polygons. Each file in the file list should be fully qualified on its own line in the file specified.
- `FORMAT_DATA="LONG_LAT|LAT_LONG"`
  - Default: `LONG_LAT`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: By default, location points in combination fields that are parsed from the input data corpus associated with a PIP primitive are parsed longitude first, then latitude. When this option is set to `LAT_LONG`, the parsing order is reversed for the input data corpus combination fields, and the first number encountered will be used as the latitude, with the second number encountered as the longitude.
- `FORMAT_POLYGON="LONG_LAT|LAT_LONG"`
  - Default: `LONG_LAT`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: By default, location points parsed from polygon data associated with a PIP primitive are parsed longitude first, then latitude. When this option is set to `LAT_LONG`, the parsing order is reversed for polygon point data, and the first number encountered will be used as the latitude, with the second number encountered as the longitude.
- `LONG_COORD="field"`
  - Default: `N/A`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: To specify an individual record field containing a longitude point only, use this option alongside a `RECORD` query. In addition, option `LAT_COORD` should be used alongside this option to specify the latitude coordinate. When not using these PIP options, a `RECORD.field` query should be used, where the field specified contains both longitude and latitude information in the same field.
- `LAT_COORD="field"`
  - Default: `N/A`
  - Search Primitive Compatibility: PIP
  - Other Primitive Compatibility: N/A
  - Description: To specify an individual record field containing a latitude point only, use this option alongside a `RECORD` query. In addition, option `LONG_COORD` should be used alongside this option to

specify the longitude coordinate. When not using these PIP options, a `RECORD.field` query should be used, where the field specified contains both longitude and latitude information in the same field.

- `IDF_WEIGHTING="max|normal|smooth|prob"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING
  - Description: This is a required option when requesting a DOCSIM_TRAINING operation. The inverse document frequency weighting (IDF_WEIGHTING) provides a method for weighting the relevance of all n-grams used across all documents the corpus. The resulting IDF values are used to form the basis for the document feature vectors used in computing document similarity. Dimensional reduction (configurable by separate options MIN_DF and MAX_DF) is used to reduce the number features used in the document feature matrix. Descriptions of the possible IDF_WEIGHTING values are as follows:
    - `max`: **This is the recommended value for typical human language document similarity training operations**. It uses the same approach as the normal weighting scheme except the IDF values of all n-grams are normalized by the maximum n-gram count in corpus. It is defined as `log(maximum n-gram count in the corpus/(1+n-gram count))`.
    - `normal`: This weighting is defined as `log(number of documents in the corpus / the number of documents containing the n-gram)`.
    - `smooth`: This weighting avoids assigning terms weights of zero when they occur in all documents. It is defined as `log(1 + (number of documents in the corpus / the number of documents containing the n-gram))`.
    - `prob`: This weighting is a probabilistic mode providing a symmetric distribution of values centered around 0 at 50% frequency. It is defined as `log((number of documents in the corpus - the number of documents containing the n-gram) / the number of documents containing the n-gram)`.
- `TF_IDF_WEIGHTING="log|normal|raw|bool|dnormal"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING
  - Description: This is a required option when requesting a DOCSIM_TRAINING operation. The term frequency inverse document frequency weighting (TF_IDF_WEIGHTING) option provides a method for weighting or normalizing the raw n-gram counts for each feature when computing the internal document feature matrix from raw n-gram counts and inverse document frequencies (IDF). The schemes provide a means of weighting features by relevance within a document while IDF provides a relevance weighting for features within the corpus. Descriptions of the possible TF_IDF_WEIGHTING values are as follows:
    - `log`: **This is the recommended value for typical human language document similarity training operations**. Log mode performs a logarithmic scaling of the n-gram counts. It is defined by `1 + log(raw count for the n-gram)`.
    - `normal`: This mode normalizes the n-gram counts as `(the raw count for the n-gram / the total number of n-grams in the document)`.
    - `raw`: Raw mode is the raw n-gram feature count as determined from the n-gram primitive with no weighting or normalization applied.
    - `bool`: Boolean mode is a simple binary weighting scheme. It sets the n-gram count to 1 of the term is contained in the document, or 0 otherwise.
    - `dnormal`: This is a double normalization mode calculated by counting all n-grams occurring in a document and normalizing by the maximum n-gram count in the document. It sets the frequency as `(0.5 + 0.5 * (n-gram count/maximum n-gram count in the document))`. Although this mode has the advantage of preventing term bias due to document size, it is susceptible to outlier terms in a document with very large frequencies. This side effect is less likely (but still possible) when using trigrams (n-grams of size 3).

- `MIN_DF=[percentage value expressed as a decimal between 0 and 1]`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING
  - Description: This is a required option when requesting a DOCSIM_TRAINING operation. This argument performs dimensional reduction of inverse document frequency (IDF) values by discarding features falling below the specified MIN_DF percentage threshold. Therefore, a value of 0.0 will discard no features while a value of 1.0 will discard all features. This can be useful since very small IDF values close to zero indicate that the feature occurs in very few documents and therefore the feature doesn't provide much discriminating information for computing similarity, making them targets for filtering. **A typical, recommended value for MIN_DF is `0.02`.**
- `MAX_DF=[percentage value expressed as a decimal between 0 and 1]`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING
  - Description: This is a required option when requesting a DOCSIM_TRAINING operation. This argument performs dimensional reduction of inverse document frequency (IDF) values by discarding features falling above the specified MAX_DF percentage threshold. Therefore, a value of 1.0 will discard no features while a value of 0.0 will discard all features. This can be useful since very large IDF values close to one indicate that the feature occurs in a majority of the documents and therefore the feature doesn't provide much discriminating information for computing similarity, making them targets for filtering. **A typical, recommended value for MAX_DF is `0.80`.**
- `MAX_TERMS="integer"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING
  - Description: This is a required option when requesting a DOCSIM_TRAINING operation. MAX_TERMS specifies the maximum number of unique n-grams that will be included from each document in the source corpus that feeds the inverse document frequency (IDF) algorithms. Note that this value ensures that no more than the number specified will be used, but it does not guarantee that this number will be used. The value may be reduced automatically by the underlying algorithms in order to ensure that the IDF can be process the requested training corpus. Note that the value utilized by the algorithm is output as part of the standard statistics of the DOCSIM_TRAINING algorithm and in the generated model.
- `N="integer"`
  - Default: `3`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING
  - Description: This is optional when requesting a DOCSIM_TRAINING operation. N specifies whether unigrams (N=1), bigrams (N=2) or trigrams (N=3; default) are used when generating the model file for use by downstream DOCSIM runs. The default value of 3 is recommended for most large input data corpuses with rich data, such as perhaps full-text patent databases, full-text medical journals, and so on. For smaller input corpus sizes with weak signals, or in certain record-based similarity situations, unigrams or bigrams may be preferred.
- `REC_PATH="string"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM_TRAINING, DOCSIM
  - Description: By default, full document similarity is used by the DOCSIM_TRAINING and DOCSIM primitives. This option changes the default behavior to allow for record-based similarity runs by specifying the record or full record.field hierarchy associated with a given training or similarity request. For example, if the input corpus is a collection of CSV records, and the training operation should only

operate on the third field in the CSV records, then a `REC_PATH` string value of `RECORD.3` would be specified.

- `SIMILARITY_TYPE="cosine|euclid|minkowski|jaccard|jaccard_weighted"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM
  - Description: This is a required option when requesting a DOCSIM operation. The similarity type option specifies the type of mathematical measure used for computing similarity between document feature vectors (n-grams). Descriptions of the possible SIMILARITY_TYPE values are as follows:
    - `cosine`: **This is the recommended value for typical human language document similarity operations**. **Cosine similarity** is a commonly used mathematical distance measure across a wide variety of use cases, including text similarity. It is a measure based on the cosine of the angle between two document feature vectors. It is equal to the dot product of the vectors divided by the product of their magnitude. Thus, for an exact vector match, the output of cosine similarity will be 1. This is not to be confused with this API's definition of **cosine distance**, which is defined as `1 - cosine similarity`. The cosine distance value is used in the output of the DOCSIM primitive, since it is significantly more amenable to software visualization environments, where a value of `0` is often, by definition, closest with respect to software visualization frameworks.
    - `euclid`: Euclidean distance is a vector distance based on the Pythagorean Theorem, defining the straight-line distance between two points in Euclidean space. It is the square root of the sum of vector component differences squared.
    - `minkowski`: Minkowski distance is a generalization of Euclidean distance. It is the `p`'th root of the sum of vector components raised to the power `p`.
    - `jaccard`: This distance measure is defined as the cardinality of the intersection of the features in each document vector divided by the cardinality of the union of features for each vector.
    - `jaccard_weighted`: This distance measure is a variant of the Jaccard measure where the weighting for each feature vector is used. It is defined as the sum of the minimum weight for features common to both vectors divided by the sum of the maximum weights for the features in either vector.
- `MIN_DISTANCE="value between 0 and 1 as a float"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM
  - Description: This is a required option when requesting a DOCSIM operation, unless option `MAX_FILE_MATCHES` is used. It is the minimum calculated similarity distance that will produce a similarity match. Any similarity distance less than this value will be ignored. Documents are considered similar only when distance values `d` are within `MIN_DISTANCE <= d <= MAX_DISTANCE`.
- `MAX_DISTANCE="value between 0 and 1 as a float"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM
  - Description: This is a required option when requesting a DOCSIM operation, unless option `MAX_FILE_MATCHES` is used. It is the maximum calculated similarity distance that will produce a similarity match. Any similarity distance greater than this value will be ignored. Documents are considered similar only when distance values `d` are within `MIN_DISTANCE <= d <= MAX_DISTANCE`.
- `MAX_FILE_MATCHES="number"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: DOCSIM

- Description: When specified, this option will override any MIN_DISTANCE and MAX_DISTANCE option set that may exist in the query options list. Instead, the DOCSIM primitive will return the requested number of best-matching results. For example, if 10 results are requested amongst a corpus of 50 files, then the 10 most similar matches will be returned. It is possible to safely request more results than there is available input. For example, if 10 results are requested and there were only 8 input files, then 8 results (representing the entire input corpus) would be generated.

3.3.3 API Global Options List

The previous subsection discussed parameters and options list associated with individual configuration strings. This subsection details additional global optional parameters that, when used, apply to a given control process until they are changed, or until the process exits. Use this syntax to set the global optional parameters list:

```
int rx_set_global_options(
   const char* option_string);
```

- `option_string` is the global option string. The global option string is a comma-separated list of `key=value` pairs.

The possible `key=value` pairs that can be used in the comma-separated global options strings are as follows:

- `max-spawns=integer`
  - Default: `N/A`
  - Search Primitive Compatibility: All
  - Other Primitive Compatibility: All
  - Command Line Equivalent: `--rx-max-spawns X`
    - `X` is the integer value to use.
  - Description: By default, all appropriate computing resources will be used to achieve maximum performance. Note that in general, to control CPU throttling, we recommend using the standard linux `taskset` functionality. However, in some cases, this optional `max-spawns` quantity may be sufficient or even preferred. It instructs the framework to limit the number of parallel software threads used during the heavy-lifting primitive computational steps to the number specified. This can help limit the CPU resource utilization during software-intensive computational algorithms, if desired. Note that this configuration parameter applies only to the computational primitive portions of an accelerated pipeline. Therefore, if CPU throttling is required but a sufficient amount of throttling is not achieved with this optional parameter, linux `taskset` is the recommended approach.
- `max-generated-matches=integer`
  - Default: `N/A`
  - Search Primitive Compatibility: All
  - Other Primitive Compatibility: PCAP
  - Command Line Equivalent: `--rx-max-count X`
    - `X` is the integer value to use.
  - Description: By default, this feature is disabled, a given input corpus is always processed in its entirety. This optional quantity instructs the search primitive engine to halt processing once it encounters the number of matches specified for the currently selected input corpus. This can be used to flag datasets that may need further processing without necessarily processing the entire dataset at the current time. For example, a use case may dictate that it is sufficient to stop processing after finding a single result, rather than to keep processing the entirety of the input corpus. As a point of comparison, this capability is somewhat analogous to the "`--max-count`" option in typical off-the-shelf `grep` implementations.
- `enable-match-qualifiers=bool`
  - Default: `0`

  - Search Primitive Compatibility: EXACT, HAMMING, EDIT_DISTANCE, PCRE2
  - Other Primitive Compatibility: PCAP
  - Command Line Equivalent: `--rx-emq`
  - Description: By default, this feature is disabled with a value of `0`. When set to `1`, any output index file results are extended such that match qualifiers are added to the end of each index file output line, if applicable, showing which portions of the various query strings were matched for Exact, Hamming and Edit Distance searches. The output index file format is extended to append match qualifiers as follows as a name/value(s) pair: `,MQ='"ValueOne" "FirstPart"?"OfValueTwo"'` where the values presented are values from the expression used, not the actual matched data. This distinction is both important and useful, especially for fuzzy searches based on hamming and edit distance. As shown, the match qualifier MQ is the name, and its value is surrounded by single quotes, with internal double-quote entries that are space-separated that reference one or more expression strings that were involved in the match as part of any query dictionary or boolean logic that qualified the query, including any complex internal notations such as wildcards that may have been applicable. Note that given the parallel operation of API primitives and optimizations resulting from complex AND/OR logic, the actual match qualifiers can change from run-to-run for expressions containing OR clauses, as it is non-deterministic which OR clause portion may finish first in a parallel operational environment given typical operating system latencies that can of course vary over time. Note that the match qualifier mechanism will also correctly evaluate AND expressions as well, or any combination of AND/OR logic, pursuant to the notion that the precise OR matches may vary depending on which parallel clause portion finishes first. This means that although the MQ values have value in and of themselves, it may sometimes be valuable to reference them back to the individual query that was used. For information relating to match qualifiers as the apply to the PCAP primitive family, please see sections 3.3.6.6 and 3.3.6.7.

- `find-input-recursively=bool`
  - Default: `0`
  - Search Primitive Compatibility: All
  - Other Primitive Compatibility: PCAP
  - Command Line Equivalent: `--rx-recurse`
  - Description: When set to 1, the input file(s) are treated as a recursive path. For example, the input path `a/b/c/*.txt` will be used as a relative path to recursively look for input files that match the pattern `*.txt` in relative path `a/b/c` and all of its subdirectories. Input paths that do not contain wildcards will attempt to match exact filenames recursively (e.g. `a/b/c/input.json` will attempt to find files named `input.json` in `a/b/c` and all of its subdirectories. Both relative and absolute starting paths are supported. This feature can be quite useful for a variety of forensics use cases.
- `input-is-filelist=bool`
  - Default: `0`
  - Search Primitive Compatibility: All
  - Other Primitive Compatibility: PCAP
  - Command Line Equivalent: `--rx-input-is-filelist`
  - Description: When set to 1, the input file(s) will be treated as a path to a text file containing a list (one per line) of input files or input file wildcard paths. This option can be used in isolation, or, since this option has precedence over the `find-input-recursively` option, both options may be used together to recursively search a specific list of paths for desired files. This feature can be quite useful for a variety of forensics use cases.
- `l4_stream_dump="tx|rx|both:on|off:prefix"`
  - Default: `N/A`
  - Search Primitive Compatibility: N/A
  - Other Primitive Compatibility: PCAP
  - Command Line Equivalent:
    - Stream only: `--rx-follow-stream [tx|rx|both],[prefix]`
    - Stream and payload: `--rx-follow-payload [tx|rx|both],[prefix]`

- Description: Extracts associated TCP stream (and optionally payload) detail for matched packets in a PCAP input corpus. Transmit, receive, or both directions are supported. Extracted stream and payload filenames generated can be arbitrarily prefixed to simplify downstream identification of output. For more details and precise formatting definitions, please refer to section 3.3.6.4.

3.3.4 Search Primitive Family

Use this syntax with the search function:

```
rx_data_set_t rx_ds_search(
    const rx_data_set_t    data_set,
    const char*            results_file,
    const char*            match_criteria_query_string,
    const char*            delimiter_string
    const char*            index_results_file,
    void*                  (*percentage_callback)(uint8_t));
```

- A dataset of type `rx_data_set_t` is the return value, representing the results of the operation. This can be used for subsequent library calls as needed, such as for statistics gathering. The returned dataset should be deleted by the user via a call to `rx_ds_delete()` when it is no longer needed in order to release any associated system resources.
- `data_set` is the input dataset of type `rx_data_set_t` to be searched.
- `results_file` is the output data file to be created. It may be `NULL` if no results file is desired. The data file output for the primitive would include the actual match data, in the same format as the input file.
- `match_criteria_query_string` is the string specifying the search criteria, which varies depending on the type of search primitive(s) that will be employed.
- `delimiter_string` specifies text to append to the results output between search results. For example, it may be useful to place two carriage returns or some other text after each search result to assist with human readability, or with downstream machine parsing of results. Use `NULL` if you do not wish to have delimiter text inserted between search results.
- `index_results_file` is the output index file to be created. It may be `NULL` if no index file is desired. An output index file must be a text file. If the file extension of `index_results_file` is not ".txt", the system will append a ".txt" extension to whatever filename is specified. For example, if you supply "my_index", the result file will be "my_index.txt". If you specify "my_index.txt", no changes will be made. The index file output is a set of comma-separated value lines, one line per match. The columns are, left to right:
  - filename where the match was found,
  - byte offset (where offset 0 is the first byte of the file) where the match begins, which considers surrounding width, LINE, and RECORD modes,
  - the length of the match, which considers surrounding width, LINE, and RECORD modes,
  - the distance, which is usually reported as 0, except for Hamming and Edit Distance invocations,
  - and possibly extra fields for special operations associated with specific API primitives (such as match qualifiers, packet numbers, and so on)
- `percentage_callback` references a user-defined callback function for handling incoming percentage completion messages. For example, a user could provide a callback function for printing the current status of the execution to standard output. It may be `NULL` if no progress reporting is desired.

3.3.4.1 EXACT

The exact search operation will search the input data corpus for exact matches. Exact searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator EXACT(expression[, options]))
```

The available comma-separated `options` for the `EXACT` primitive are documented in section 3.3.2.

When the format of the input data is specified as raw text, the `WIDTH` option specifies how many characters (up to 65,535) on each side of the exact match will be returned as part of the matched data results. This can be useful to assist a human analyst or a downstream machine learning tool to determine the contextual use of a specific matched term when searching data in unstructured fashion.

3.3.4.1.1 Example Program

Suppose you have extracted a list of passengers that had flown through a given airport on a given day, and stored that information in a file called `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

In the following example program, the user requests an exact search for a specific phone number against an input file, generates an output data and an output index file, and displays the total number of matches found to `stdout`.

```
#include <stdio.h>
#include <libryftx.h>
#include <inttypes.h>

int main(void)
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-simple.txt");

    rx_data_set_t search_results = rx_ds_search(
            input_data_set,
            "d.txt",
            "(RAW_TEXT CONTAINS EXACT(\"310-555-6767\", WIDTH=\"16\"))",
            NULL,
            "i.txt",
            NULL);
    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }
    else
    {
        uint64_t search_num_matches = rx_ds_get_total_matches(search_results);
        printf("Total Matches Found: %"PRIu64"\n", search_num_matches);
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);
```

```
    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

When executed, the program generates the following result to `stdout`:

```
Total Matches Found: 1
```

The output data file specified was `d.txt`, and it will contain:

```
tt, 12-14-1949, 310-555-6767
```

We also specified an output index file `i.txt`, and it contains:

```
passengers-simple.txt,224,29,0
```

The third comma-separated value in the output index file represents the total length of the match plus any surrounding detail. The value reported here is `29` which represents 16 bytes to the left of the match, the 12 byte exact match, and the single character trailing line feed that appears to the right of the match before end-of-file is reached, for a total of 29 bytes.

3.3.4.2 HAMMING

The fuzzy Hamming search operation works similarly to exact search except that matches do not have to be exact. Instead, the `DISTANCE` option allows the specification of a "close enough" value to indicate how close the input must be to the match string contained in the `match_criteria_query_string`. The match string can be up to 32 bytes in length.

A “close enough” match is specified as a Hamming distance. The Hamming distance between two strings of equal length is the number of positions at which the corresponding characters are different. As provided to the Hamming search operation, the Hamming distance specifies the maximum number of character substitutions that are allowed in order to declare a match.

Hamming searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator HAMMING(expression, options))
```

The available comma-separated `options` for the `HAMMING` primitive are documented in section 3.3.2. If the `DISTANCE` option is omitted, a default value of `0` will be used, which is effectively the same operation as an exact search.

In addition, similar to exact search, the `WIDTH` and `LINE` options can aid in downstream analysis of contextual use of the match results against unstructured raw text data. When using `LINE` mode with a fuzzy search operation, results generation can be complicated by embedded line feed characters in match strings. The following scenario describes the behavior that can be expected from the API when these situations arise. For a data input file consisting of the following two lines where each line ends in a line feed (0x0a) character:

```
This line ends in a match
And this line has match in the middle and a match a bit later too
```

If a fuzzy operation searches for occurrences of "match A" with case-sensitivity and a distance of 1, three matches would result if `LINE` mode were **not** selected. The first would be "`match<LF>A`" (distance one since the space became a line feed character) and the second would be "`match i`" (distance one since the 'A' became an 'i') and the third would be "`match a`" (distance one because the 'A' became an 'a').

With `LINE` mode specified, a complication arises since in the first match occurrence, a line feed character appears in the actual match result. This can happen quite often when performing fuzzy searches, especially as the distance value is increased. The API behavior in this case will be to output results with an intelligent de-duplication mechanism, using this rule: each match, regardless of its internal byte contents, is expanded by looking for line feed characters on each side of the match. The subsequent offset and length combinations are then de-duplicated if and only if the resulting offsets and lengths are identical. Therefore, when using `LINE` mode in this example, the API will generate exactly two results. The first result will have a starting offset of 0 and a length of 92 (which algorithmically maps to the line feed boundaries occurring completely outside of the first match), and the second consisting of a starting offset of 26 and a length of 66 (which algorithmically maps to the de-duplicated line containing the 2nd and 3rd matches). These two answers capture the totality of the possibilities given the fuzzy nature of the match in conjunction with the `LINE` mode request.

A fully qualified `HAMMING` clause looking for the term `albatross` and return matching lines using a distance of 1 would be:

```
(RAW_TEXT CONTAINS HAMMING("albatross", DISTANCE="1", LINE="true"))
```

Hamming search algorithms are highly parallelizable, so they can run extremely quickly on accelerated hardware. Therefore, if search algorithms can make use of Hamming search, then they will typically perform at very high speed, often at the same speed as exact searches.

Hamming search has a variety of very interesting real-world use cases. It is an excellent tool for looking at single byte column differences for things like serial number comparisons where common human-entry errors can happen given similarities between character sets such as 0/O, i/1/l, 4/A, 5/S, and so on. These same effects often apply to a variety of automated text translation tools, which sometimes make mistakes. There are excellent examples of this in transcribing doctor's notes into medical history records. These effects also hold true for a variety of account number information, street addresses, and even things as simple as time and date formats. All of these result in a variety of "dirty data" problems which can be solved using Hamming distance techniques.

3.3.4.2.1 Example Program

Suppose that you extracted a list of passengers that had flown through a particular airport on a given day, and stored that information in a file called `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

If you were interested in finding all passengers who listed their birthday as "12-14-1949", but weren't sure if the formatting used "-" or "/" to delimit the date fields (or if a human being had entered information without properly checking the syntax requirements), you could create the following program using a search fuzziness parameter of 2 to

allow either "-" or "/" to be recognized, and return 16 characters on each side of the match using a single processing node. In this example, we will use "/".

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-simple.txt");

    rx_data_set_t search_results = rx_ds_search(
            input_data_set,
            "d.txt",
            "(RAW_TEXT CONTAINS HAMMING(\"12/14/1949\", DISTANCE=\"2\", WIDTH=\"16\"))",
            NULL,
            "i.txt",
            NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

This user program will create a data results file called `d.txt`. Data results files will return the matched text including the number of characters requested on each side of the match. The output file contains:

```
Sonny Crockett, 12-14-1949, 310-555-6767
```

The program also requested that an output index file `i.txt` be created. It will contain:

```
passengers-simple.txt,212,41,2
```

Note that the final entry in the lone entry in this index file is "`2`". That means that a Hamming distance of 2 was detected for this entry, since two "-" characters were found in place of the requested "/" characters.

3.3.4.3 EDIT_DISTANCE

Edit distance search performs a search that does not require two strings to be of equal length to obtain a match. Instead of considering individual character differences, edit distance search counts the minimum number of insertions,

deletions and replacements required to transform one string into another. This can make it much more powerful than Hamming search for certain applications.

For example, a Hamming search for a search term of “Michelle” with a distance of 1 would match “Mishelle” since the position ‘c’ is changed to ‘s’. But it would not match against “Mi**schel**le”, since the bolded characters shown don’t match the same positions in the “Michelle” search term, evaluating to a distance of 6, which is greater than the desired distance 1, so no match is declared.

On the other hand, an edit distance search specifying the same “Michelle” match string and a distance of 1 **would** match against the string “Mischelle”, since a single ‘s’ can be inserted into “Michelle” to arrive at “Mischelle”, making it distance 1. The string “Michele” would also be a match because one ‘l’ was deleted to change “Michelle” into “Michele”. “Mishelle” would also be a match, because of the single replacement of an ‘s’ with the ‘c’. However, since the distance specified was only 1, then “Mischele” would not be a match, because that requires two changes: the addition of an ‘s’ and the removal of an ‘l’. If the distance had been specified as 2, then “Mischele” would also have matched, as would the other patterns mentioned, since matches are declared whenever the calculated edit distance between two strings is less than or equal to the desired distance.

Edit distance is an extremely powerful search tool for a variety of data sources, including names, addresses, medical records searching, genomic and disease research data, common misspellings, and more. Unlike Hamming search, edit distance is a more natural search paradigm for many algorithms, since it does not require string matches to be of the same size. The tradeoff is that edit distance is not as amenable to full hardware parallelization, so algorithms reliant on it typically run slower than those that implement Hamming search. When targeting hardware acceleration environments, the match string length added to the distance value must not exceed 32 bytes.

Edit distance searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator EDIT_DISTANCE(expression, options))
```

The available comma-separated `options` for the `EDIT_DISTANCE` primitive are documented in section 3.3.2. Note that the `DISTANCE` option is required, as it sets the edit distance that will be used.

In addition, similar to exact search, the `WIDTH` and `LINE` options can aid in downstream analysis of contextual use of the match results against unstructured raw text data. When using `LINE` mode with a fuzzy search operation, results generation can be complicated by embedded line feed characters in match strings. The following scenario describes the behavior that can be expected from the API when these situations arise. For a data input file consisting of the following two lines where each line ends in a line feed (0x0a) character:

```
This line ends in a match
And this line has match in the middle and a match a bit later too
```

If a fuzzy operation searches for occurrences of "match A" with case-sensitivity and a distance of 1, three matches would result if `LINE` mode were **not** selected. The first would be "`match<LF>A`" (distance one since the space became a line feed character) and the second would be "`match i`" (distance one since the 'A' became an 'i') and the third would be "`match a`" (distance one because the 'A' became an 'a').

With `LINE` mode specified, a complication arises since in the first match occurrence, a line feed character appears in the actual match result. This can happen quite often when performing fuzzy searches, especially as the distance value is increased. The API behavior in this case will be to output results with an intelligent de-duplication mechanism, using this rule: each match, regardless of its internal byte contents, is expanded by looking for line feed characters on each side of the match. The subsequent offset and length combinations are then de-duplicated if and only if the resulting offsets and lengths are identical. Therefore, when using `LINE` mode in this example, the API will generate exactly two results. The

first result will have a starting offset of 0 and a length of 92 (which algorithmically maps to the line feed boundaries occurring completely outside of the first match), and the second consisting of a starting offset of 26 and a length of 66 (which algorithmically maps to the de-duplicated line containing the 2nd and 3rd matches). These two answers capture the totality of the possibilities given the fuzzy nature of the match in conjunction with the `LINE` mode request.

A fully qualified `EDIT_DISTANCE` clause looking for the term “albatross” and return 20 bytes on each side of every match encountered while using a distance of 1 would be:

```
(RAW_TEXT CONTAINS EDIT_DISTANCE("albatross", DISTANCE="1", WIDTH="20"))
```

3.3.4.3.1 Example Program

Suppose that you extracted a list of passengers that had flown through a particular airport on a given day, and stored that information in a file called `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

If you were to search for `Steven` using an edit distance of 1, any single character added to, deleted from, or modified in the string `Steven` would be a match. For example, `Steve` would be a match as well as `Stevens` or `Zteven`. Below is an example user program for searching `passengers-simple.txt` for `Steven` with distance 1, a surrounding width of 16 bytes, with the results reduction (data deduplication) edit distance features enabled:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-simple.txt");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.txt",
        "(RAW_TEXT CONTAINS EDIT_DISTANCE(\"Steven\", WIDTH=\"16\", DISTANCE=\"1\",
REDUCE=\"true\"))",
        "\n",
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
```

```
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Searching for `Steven` using an edit distance search with a distance of 1 and a surrounding width of 16 bytes produces one match to output data file `d.txt`:

```
5, 310-555-2323

Steve McGarett, 12-30-
```

The output index file was requested as `i.txt`. It will contain:

```
passengers-simple.txt,69,38,1
```

The final parameter in the output index file is the resulting edit distance of the match, which in this case was "1," since the edit distance between `Steve` and `Steven` is one.

3.3.4.4 PCRE2

PCRE2 search performs a search that adheres strictly to the PCRE2 regular expression rules. The API supports the PCRE2 specification with respect to query operations as described at http://www.pcre.org/current/doc/html/pcre2syntax.html, as of June 5, 2017.

PCRE2 is a standards-based regular expression format that is heavily used by the search and analytics community for a variety of important search use cases, including cyber use cases.

PCRE2 searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator PCRE2(expression, options))
```

The available comma-separated `options` for the `PCRE2` primitive are documented in section 3.3.2. Note that many query-related options, such as case-insensitive options, are not supported in the options field, but are instead supported through standard PCRE2 syntax internal to the expression itself.

In addition, similar to exact search, the `WIDTH` and `LINE` options can aid in downstream analysis of contextual use of the match results against unstructured raw text data.

A fully qualified `PCRE2` clause looking for various spellings of the often-misspelled word "misspell" in case-insensitive fashion allowing for one or more internal 's' characters would be:

```
(RAW_TEXT CONTAINS PCRE2("(?i)mis+pell"))
```

Similar to a variety of other commercial and open source regular expression frameworks, when the PCRE2 engine finds a match, it starts the next match search at the byte position immediately following the previous match.

3.3.4.4.1 Example Program

Suppose that you extracted a list of passengers that had flown through a particular airport on a given day, and stored that information in a file called `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

Although this is clearly a CSV file by inspection, we can also treat it as an unstructured raw text file, just like we can with any input corpus should we choose. We'll do that here. To quickly search for `Hanibal` or `Hannibal` with a concise clause, a PCRE2-compliant regular expression could nicely do the trick. We will use a regular expression `Hann?ibal`, where the question mark is a greedy quantifier, which means that the previous character '`n`' must appear 0 or 1 times. This effectively means that `Hanibal` or `Hannibal` will result in a match against that PCRE2 regular expression, but nothing else will. In addition, since we want the line(s) containing the match(es) instead of just the match(es) themselves, we will enable the `LINE` mode option to the query.

Below is an example user program to achieve this query.

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-simple.txt");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.txt",
        "(RAW_TEXT CONTAINS PCRE2(\"Hann?ibal\", LINE=\"true\"))",
        "\n",
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc` to compile source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Searching for the PCRE2 compliant regular expression `Hann?ibal` in `passengers-simple.txt` produces one match to output data file `d.txt`, and using the specified surrounding line in the example program:

```
Hannibal Smith, 10-01-1928, 310-555-1212
```

The output index file `i.txt` contains:

```
passengers-simple.txt,0,41,0
```

3.3.4.5 DATE

The Date Search operation allows for validated exact date searches and for searching for dates within a range for both structured and unstructured data. Date validation includes leap year validation and a mechanism is provided to allow for arbitrary two- and four-digit year grammars. The following date formats are supported:

- YYYY/[M]M/[D]D
- YY/[M]M/[D]D
- [D]D/[M]M/YYYY
- [D]D/[M]M/YY
- [M]M/[D]D/YYYY
- [M]M/[D]D/YY

Note: The "/" character in the above list of formats can be replaced by any other single character delimiter, such as "-" or "_", which would enable YYYY-MM-DD and MM_DD_YYYY as valid date formats, respectively.

Date Searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator DATE(expression[, options]))
```

Different date ranges can be searched for by modifying the expression provided. There are two general expression types supported:

```
1.  DATE(DateFormat operator ValueB)
2.  DATE(ValueA operator DateFormat operator ValueB)
```

The box below contains a full list of the supported expressions. DateFormat represents the format of the dates to search for and ValueA and ValueB represent dates to compare input data against.

```
DateFormat  = ValueB
DateFormat != ValueB  (Not equals operator)
DateFormat >= ValueB
DateFormat >  ValueB
DateFormat <= ValueB
DateFormat <  ValueB
ValueA <= DateFormat <= ValueB
ValueA <  DateFormat <  ValueB
ValueA <  DateFormat <= ValueB
ValueA <= DateFormat <  ValueB
```

For example, to find all validated dates after “02/28/12” using a two-digit year format search with the default leading digits of “20” (thus, the year becomes 2012 for date validity comparison) in unstructured raw text, use the following search query criteria:

```
(RAW_TEXT CONTAINS DATE(MM/DD/YY > 02/28/12))
```

To use a two-digit year format to find all validated dates after “02/28/12” with the leading year digits being “19” instead of “20” in unstructured raw text, use the `PREFACE` option added to the search query criteria:

```
(RAW_TEXT CONTAINS DATE(MM/DD/YY > 02/28/12, PREFACE="19"))
```

To find all validated matching dates between “02-28-1998” and “09-19-2017” using a four-digit year format but not including those two dates in a record/field construct where the field tag is `date`, use:

```
(RECORD.date CONTAINS DATE(02-28-1998 < MM-DD-YYYY < 09-19-2017))
```

Note that when specifying months and days in the query string itself, you must use two digit months and day values, which means that leading zeros are required. However, the input corpus being searched does not have to use leading zeros. Matches without leading zeros are also reported, unless the `ALLOWSINGLE` option is set to `false`. For example, the following query string would enforce two-digit months and days in the input corpus (for example, the second day of the month would match only “`02`” and not standalone “`2`”):

```
(RAW_TEXT CONTAINS DATE(MM/DD/YY > 02/02/12, ALLOWSINGLE="false"))
```

All available comma-separated `options` for the `DATE` primitive are documented in section 3.3.2.

3.3.4.5.1 Example Program

Suppose we had the following file, `passengers.csv`, as our input file:

```
name,dob,phone
Hannibal Smith,10-01-1928,011-310-555-1212
DR. Thomas Magnum,01-29-1945,011-310-555-2323
Steve McGarett,12-30-1920,011-310-555-3434
Michael Knight,08-17-1952,011-310-555-4545
Stringfellow Hawke,08-15-1944,011-310-555-5656
Sonny Crockett,12-14-1949,011-310-555-6767
MR. T Magnum,01-29-1946,011-310-555-7878
```

To find passengers who were born after `01-01-1940`, we could use the following user program:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers.csv");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.csv",
```

```
        "(RECORD.\"dob\" EQUALS DATE(MM-DD-YYYY > 01-01-1940))",
        NULL,
        "i.txt",
        NULL);


    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data file results:

```
DR. Thomas Magnum,01-29-1945,011-310-555-2323
Michael Knight,08-17-1952,011-310-555-4545
Stringfellow Hawke,08-15-1944,011-310-555-5656
Sonny Crockett,12-14-1949,011-310-555-6767
MR. T Magnum,01-29-1946,011-310-555-7878
```

The program will also produce the following output index file results:

```
passengers.csv,58,46,0
passengers.csv,147,43,0
passengers.csv,190,47,0
passengers.csv,237,43,0
passengers.csv,280,41,0
```

3.3.4.6 TIME

The Time Search operation can be used to search for validated exact times or for times within a particular range for both structured and unstructured data using the following time formats:

- HH:MM:SS
- HH:MM:SS:ss

Note: 'ss' in the second format indicates hundredths of a second, and if specified should always be two digits.

Time Searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator TIME(expression[, options]))
```

Time ranges can be specified in the expression. There are two general expression types supported:

```
1.  TIME(TimeFormat operator ValueB)
2.  TIME(ValueA operator TimeFormat operator ValueB)
```

The box below contains of list of these supported expressions. TimeFormat represents the format of the times to search for and ValueA and ValueB represent times to compare input data against.

```
TimeFormat  = ValueB
TimeFormat != ValueB (Not equals operator)
TimeFormat >= ValueB
TimeFormat >  ValueB
TimeFormat <= ValueB
TimeFormat <  ValueB
ValueA <= TimeFormat <= ValueB
ValueA <  TimeFormat <  ValueB
ValueA <  TimeFormat <= ValueB
ValueA <= TimeFormat <  ValueB
```

For example, to find all times after “09:15:00”, use the following search query criteria:

```
(RAW_TEXT CONTAINS TIME(HH:MM:SS > 09:15:00))
```

To find all matching times between “11:15:00” and “13:15:00” but not including those times in a record/field construct where the field tag is `time`, use:

```
(RECORD.time CONTAINS TIME(11:15:00 < HH:MM:SS < 13:15:00))
```

Note that when specifying hours, minutes and seconds in the query string itself, you must use two digit values, which means that leading zeros are required. However, the input corpus being searched does not have to use leading zeros. Matches without leading zeros are also reported, unless the `ALLOWSINGLE` option is set to `false`. If the desire is to search for and report two-digit formats only, then the following query string can be used to enforce two-digit hours, minutes and seconds in the input corpus:

```
(RECORD.time CONTAINS TIME(11:15:00 < HH:MM:SS < 13:15:00, ALLOWSINGLE="false"))
```

As an example, if the input corpus had a value “13:1:00,” it would not be considered a valid time since `ALLOWSINGLE` was set to `false`. If it instead appeared as “13:01:00,” it would be considered a valid time. If, on the other hand, `ALLOWSINGLE` was not explicitly set to `false`, the value “13:1:00” would be recognized as a valid time, and a leading zero would be implied for the minutes.

The available comma-separated `options` for the `TIME` primitive are documented in section 3.3.2.

3.3.4.6.1 Example Program

Suppose we have a `passengers-arrivals.txt` input file that includes arrivals of a set of passengers:

```
Hannibal Smith, 10-01-1928, 310-555-1212, 10:45:00
DR. Thomas Magnum, 01-29-1945, 310-555-2323, 10:12:00
Steve McGarett, 12-30-1920, 310-555-3434, 17:42:00
Michael Knight, 08-17-1952, 310-555-4545, 15:30:00
Stringfellow Hawke, 08-15-1944, 310-555-5656, 18:25:22
Sonny Crockett, 12-14-1949, 310-555-6767, 09:13:00
```

Although by inspection this is a CSV input file, it does not contain a proper header line. In this simple case it would be trivial to count to the proper column number and use that number to reference the field of interest for the query. Sometimes, though, there can be hundreds or thousands of columns in CSV files.

An interesting alternative is that there is nothing that stops us from treating any dataset as unstructured raw data. Furthermore, when it is known that CSV files of interest do not leverage escaped new line characters, it is often very trivial to quickly find records (lines) of interest by searching them unstructured and then using the `LINE` mode option to return the entire line associated with a match. This can be a very useful technique when CSV files (like this one) do not include header lines.

Leveraging the unstructured query alongside `LINE` mode methodology, to find all passengers who arrived between `17:00:00` and `19:30:00`, we could use the following user program:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-arrivals.txt");

    rx_data_set_t search_results = rx_ds_search(
       input_data_set,
       "d.txt",
       "(RAW_TEXT CONTAINS TIME(17:00:00 < HH:MM:SS < 19:30:00, LINE=\"true\"))",
       NULL,
       "i.txt",
       NULL);


    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this time search produces the following output data results in `d.txt`:

```
Steve McGarett, 12-30-1920, 310-555-3434, 17:42:00
Stringfellow Hawke, 08-15-1944, 310-555-5656, 18:25:22
```

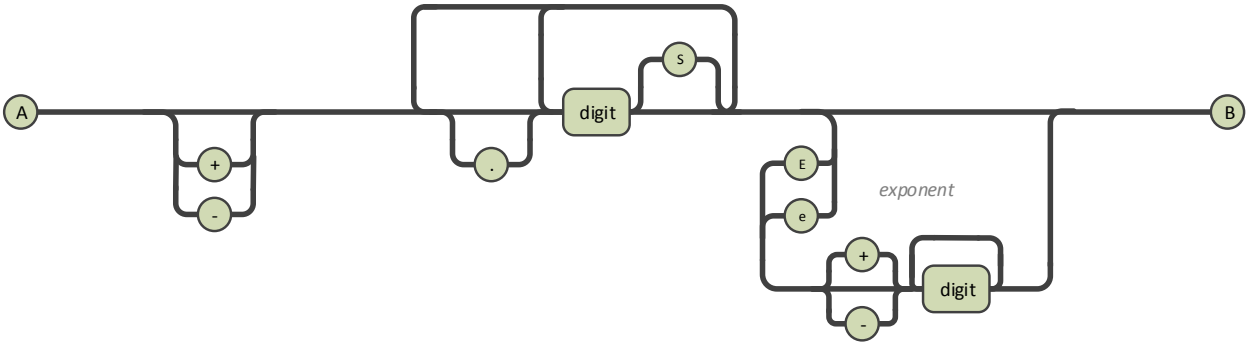
The program also produces the following output index results in `i.txt`:

```
passengers-arrivals.txt,105,51,0
passengers-arrivals.txt,207,55,0
```

3.3.4.7 NUMBER

The Number Search operation can be used to search for exact numbers or numbers in a particular range for both structured and unstructured input data.

At a high level, the number parsing protocol is roughly described via the following diagram:



Please note that the diagram above is a rough guideline and is meant to be informational only. The full details of the protocol rules are:

- Numbers start (the "start character") with a digit (0-9), a minus sign (unless separator is a minus), a plus sign (unless separator is a plus), or a decimal point.
- If a minus sign is the starting character, it may be followed by any amount of spaces or tabs.
- There can be any number of digits (0-9) strung together at any point where digits (0-9) are allowed.
- Any number of digits (0-9) can follow an optional decimal point.
- Unless scientific notation has been disabled via the `SCINOT="false"` option specifier, at any point as digits are recognized, a single 'e' or an 'E' can appear which is interpreted as scientific notation, and a power of ten base is used. Any subsequent 'e' or 'E' that is encountered later in the machine is a stop condition.
- If 'e' or 'E' does appear following a string of digits, then:
  - A plus or minus sign may appear immediately after the 'e' or 'E'
  - Any number of digits may appear immediately after the 'e', 'E', minus sign, or plus sign
- Stop characters are any characters the state machine runs across which are not legal characters for the particular state in question.
- Upon reaching a stop character, the verification engine will run if at least one digit has been encountered since the start character.
- Subsequent searches start with the stop character or the character following it, if optimization is possible.
- In all scenarios, the verification engine uses `strtod` [IEEE-754] to convert the collapsed string to a double precision number for comparison/ranging against the query string value(s) which are also sent through `strtod` to ensure comparison compatibility.
  - The numeric conversion for ranging checks are limited to double precision occupying 64 bits via `strtod` [IEEE-754], which means that the largest possible absolute value number is ~ 1.79769313486231570815... x 10^308, and the smallest absolute value closest to zero is 2.22507385850720138309... x 10^-308.
  - For example, if 123e4567 is detected, it will range to the maximum number, and if the range check passes the criteria, the full match "123d4567" is reported.

  - As another example, if -4E-2020 is the string, since it is smaller than the minimum number closest to zero, it will resolve to zero (and would therefore match an "equals 0" request, and the full match "-4E-2020" would be reported.)
  - Note that these approximations were chosen to be compatible with the industry-accepted and widely used IEEE-754 standards given approximate representations of double precision numbers in binary formats for comparisons.
- When specified, separators are don't-cares internal to a digit (0-9) sequence, except that separators are not allowed to appear back-to-back. If they do, a stop character condition exists. Otherwise, separators are completely stripped from the eventual numeric conversion. This means that if a comma is specified as a separator, representations of "1000.0", "1000", "1,000", "1,000.0", "1,0,0,0.0" and "1,0,0,0,.0" are all equivalent to the value "1000".
- If the configurable number separator option is specified as any 'special' character, such as a plus sign or a minus sign or a decimal point, etc., then those characters are not evaluated for any other purpose, to include negation and scientific notation.
- No whitespace is permitted to appear after a separator character, otherwise it is considered a stop character.

This general solution allows for numbers to be represented in arbitrary forms, including:

- Configurable separators, such as perhaps specifying a comma representing a separator for the US number system (e.g., “7,000”), or specifying a dash separating fields in a phone number or a social security number (e.g., “1-800-555-1212” or “123-45-6789”).
- Configurable decimals, such as perhaps specifying a period to represent the decimal for the US number system (e.g., “7.2”).
- A minus sign to specify a negative value, such as “-7.2”.
- Scientific notation, such as “-2e3”, “-2e-3”, “-2.2E+2”, etc.

Number Searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator NUMBER(expression, options))
```

Different number ranges can be searched for by modifying the expression provided. There are two general expression types supported:

```
1.  NUM operator1 "ValueA"
2.  "ValueA" operator1 NUM operator2 "ValueB"
```

The box below contains a full list of the supported expressions. ValueA and ValueB represent the numbers to compare the input data against.

```
NUM  = "ValueA"
NUM != "ValueA" (Not equals operator)
NUM >= "ValueA"
NUM >  "ValueA"
NUM <= "ValueA"
NUM  < "ValueA"
"ValueA" <= NUM <= "ValueB"
"ValueA" <  NUM <  "ValueB"
"ValueA" <  NUM <= "ValueB"
"ValueA" <= NUM <  "ValueB"
```

The available comma-separated `options` for the `NUMBER` primitive are documented in section 3.3.2. Note that the `SEPARATOR` and `DECIMAL` options are optional.

The `SEPARATOR` is defined as the separating character to use. By default it is disabled. For standard US numbers, a comma might be specified. If other types of numbers are being searched, such as perhaps phone numbers, than a dash would be specified. Note that the `SEPARATOR` does not need to appear in the data stream, but if it does appear, it will be considered part of the value being parsed.

The `DECIMAL` is defined as the decimal specifier to use. By default it is disabled. For standard US numbers, a period (decimal point) would be specified. Note that the `DECIMAL` does not need to appear in the data stream, but if it does appear, it will be considered part of the value being parsed.

Note that if used the `SEPARATOR` and `DECIMAL` must be different. If the same character is specified for both, an error will be generated.

As an example of a fully qualified number search expression, to find all matching numbers using the US number system between but not including “1025” and “1050” in a record/field construct where the field tag is `id`, use:

```
(RECORD.id CONTAINS NUMBER("1025" < NUM < "1050", SEPARATOR=",", DECIMAL="."))
```

The results will contain all numbers that are encountered in the input data that match the specified range. For example, if a scientific notation number "1.026e3" appears in the input data (which expands to "1,026"), then it will be reported as a match since it falls within the specified range. Similarly, the numbers 1049 and 1,049.9 would also match. However, the number -1,234 would not be a match, as it does not fall within the requested range, nor would the number +1,050.00001.

3.3.4.7.1 Example Program

Suppose we have the following input file `pi.txt`:

```
The number pi is a mathematical constant.  Originally defined as the ratio of a circle's
circumference to its diameter, it now has various equivalent definitions and appears in
many formulas in all areas of mathematics and physics.  It is approximately equal to
3.14159.  It is also called Archimedes' constant.

Being an irrational number, pi cannot be expressed as a common fraction (equivalently,
its decimal representation never ends and never settles into a permanently repeating
pattern).  Still, fractions such as 22/7 and other rational numbers are commonly used to
approximate pi.  The digits appear to be randomly distributed.  In particular, the digit
sequence of pi is conjectured to satisfy a specific kind of statistical randomness, but
to date, no proof of this has been discovered.  Also, pi is a transcendental number; that
is, it is not the root of any polynomial having rational coefficients.  This
transcendence of pi implies that it is impossible to solve the ancient challenge of
squaring the circle with a compass and straightedge.

Pi rounded to 14 decimal places is 3.14159265358979.

Pi is often shortened to just 3.14 for many simple calculations.

Pi is not meant to be confused with pie.  Pie is delicious.  I wonder if at any point in
time there have been exactly 3.14E6 uneaten pies in the world.
```

If we want to find all occurrences of the pi between `3.14` and `3.1416` in the above text and adhere to numeric rules (to include scientific notation, so the value `3.14E6` should not appear in the results, but `3.14`, `3.14159`, and `3.14159265358979` would), we might use the following program to do so. We will also arbitrarily specify that a surrounding width of 9 characters should be returned alongside each match. Depending on the use case, some amount

of surrounding width may be useful for downstream processing, such as when feeding mathematical modeling, or natural language processing algorithms although such algorithms would likely fare better with significantly more surrounding width than is being generated in our simple example here. We will also add a separator string consisting of four asterisks bookmarked by a line feed on each end. This will appear between output data results to more easily identify the demarcation points:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "pi.txt");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.txt",
        "(RAW_TEXT CONTAINS NUMBER(\"3.14\" <= NUM <= \"3.1416\", SEPARATOR=\"-\",
DECIMAL=\".\", WIDTH=\"9\"))",
        "\n****\n",
        "i.txt",
        NULL);


    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data results in `d.txt`:

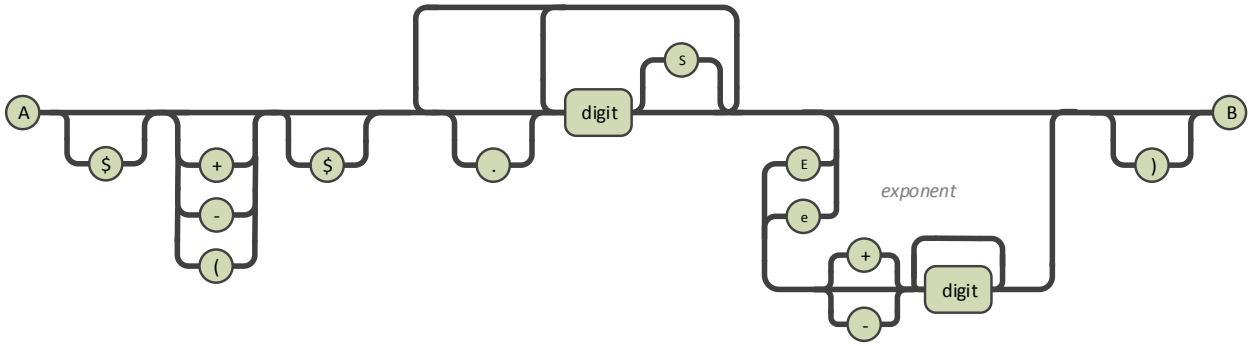
```
equal to 3.14159.  It is
****
laces is 3.14159265358979.

Pi is
****
 to just 3.14 for many
****
```

The program will also produce the following output index results in `i.txt`:

```
pi.txt,253,25,0
pi.txt,1082,34,0
pi.txt,1131,22,0
```

3.3.4.8 CURRENCY

The Currency Search operation follows largely the same rules as the Number Search operation. This includes allowing for leading zeros. Configurable currency identifiers, such as "$" for US currency, are permitted. In addition, negative amounts are parsed using either the minus sign or parenthetical "()" notation. At a high level, the currency parsing protocol is roughly described via the following diagram:



Please note that the diagram above is a rough guideline and is meant to be informational only. The full details of the protocol rules are:

- Currency starts (the "start character") with a currency symbol, a minus sign (unless separator is a minus), an open paren (unless separator is an open paren), or a plus sign (unless separator is a plus).
- If a '+', '-' or '(' starts the currency, then the next non-tab/non-space character must be the currency symbol.
- Mixing-and-matching '-' and '(' is not allowed, and will result in a stop character condition, with the next search starting at the second occurrence (which is an optimization, since no currency symbol appears between the two, or that would have been a stop character).
- If '(' is used, a closing ')', if encountered is the stop character, unless ')' was specified as a separator character. If an alternate stop character is encountered and the opening '(' is not closed, then the value is not considered a legal currency value and the search continues at the stop character.
    - Keep in mind that this means that if ')' is specified as a separator, currencies using parenthetical negation will never match.
- There can be any number of digits (0-9) strung together at any point where digits (0-9) are allowed.
- Any number of digits (0-9) can follow an optional decimal point.
- Unless scientific notation has been disabled via the SCINOT=false option specifier, at any point as digits are recognized, a single 'e' or an 'E' can appear which is interpreted as scientific notation, and a power of ten base is used. Any subsequent 'e' or 'E' that is encountered later in the machine is a stop condition.
- If 'e' or 'E' does appear following a string of digits, then:
    - A plus or minus sign may appear immediately after the 'e' or 'E'
    - Any number of digits may appear immediately after the 'e', 'E', minus sign, or plus sign
- Stop characters are any characters the state machine runs across which are not legal characters for the particular state in question.
- Upon reaching a stop character, the verification engine will run if at least one digit has been encountered since the start character.
- Subsequent searches start with the stop character or the character following it, if optimization is possible.

- In all scenarios, the verification engine uses `strtod` [IEEE-754] to convert the collapsed string to a double precision number for comparison/ranging against the query string value(s) which are also sent through `strtod` to ensure comparison compatibility.
    - The numeric conversion for ranging checks are limited to double precision occupying 64 bits via `strtod` [IEEE-754], which means that the largest possible absolute value number is ~ 1.79769313486231570815... x 10^308, and the smallest absolute value closest to zero is 2.22507385850720138309... x 10^-308.
    - For example, if 123e4567 is detected, it will range to the maximum number, and if the range check passes the criteria, the full match "123d4567" is reported.
    - As another example, if -4E-2020 is the string, since it is smaller than the minimum number closest to zero, it will resolve to zero (and would therefore match an "equals 0" request, and the full match "-4E-2020" would be reported.)
    - Note that these approximations were chosen to be compatible with the industry-accepted and widely used IEEE-754 standards given approximate representations of double precision numbers in binary formats for comparisons.
- When specified, separators are don't-cares internal to a digit (0-9) sequence, except that separators are not allowed to appear back-to-back. If they do, a stop character condition exists. Otherwise, separators are completely stripped from the eventual numeric conversion. This means that if a comma is specified as a separator, representations of "1000.0", "1000", "1,000", "1,000.0", "1,0,0,0.0" and "1,0,0,0,.0" are all equivalent to the value "1000".
- If the configurable number separator option is specified as any 'special' character, such as a plus sign or a minus sign or a decimal point, etc., then those characters are not evaluated for any other purpose, to include negation and scientific notation.
- No whitespace is permitted to appear after a separator character, otherwise it is considered a stop character.

Currency Searches are a type of number searches and extend the general expression defined previously as follows:

```
(input_specifier relational_operator CURRENCY(expression, options))
```

Different currency ranges can be searched for by modifying the expression provided. There are two general expression types supported:

```
1.  CUR operator1 "ValueA"
2.  "ValueA" operator1 CUR operator2 "ValueB"
```

The box below contains a full list of the supported expressions. ValueA and ValueB represent the currency values to compare the input data against.

```
CUR  = "ValueA"
CUR != "ValueA" (Not equals operator)
CUR >= "ValueA"
CUR >  "ValueA"
CUR <= "ValueA"
CUR  < "ValueA"
"ValueA" <= CUR <= "ValueB"
"ValueA" <  CUR <  "ValueB"
"ValueA" <  CUR <= "ValueB"
"ValueA" <= CUR <  "ValueB"
```

The available comma-separated `options` for the `CURRENCY` primitive are documented in section 3.3.2. Note that `SYMBOL`, `SEPARATOR` and `DECIMAL` options are optional.

The `SYMBOL` is defined as the monetary symbol to be used in the state machine which starts a currency match state machine. The default value is the `$` character.

The `SEPARATOR` is defined as the separating character to use in the state machine. By default it is disabled. For standard US currency, a comma might be specified. If other types of currencies are being searched, perhaps a period would need to be specified. For example, US $1,000.00 might be written in certain European markets as "$1.000,00". Note that the `SEPARATOR` does not need to appear in the data stream, but if it does appear, it will be considered part of the value being parsed.

The `DECIMAL` is defined as the decimal specifier to use. By default it is disabled. For standard US currency, a period (decimal point) would be specified. If other types of currencies are being searched, perhaps a comma would need to be specified. For example, US $1,000.00 might be written in certain European markets as "$1.000,00". Note that the `DECIMAL` does not need to appear in the data stream, but if it does appear, it will be considered part of the value being parsed.

Note that the `SEPARATOR` and `DECIMAL` must be different. If the same character is specified for both, an error will be generated.

As an example of a fully qualified currency search expression, to find all values using the US currency system between but not including "$450" and "$10,100.50" in a record/field construct where the field tag is `price`, use:

```
(RECORD.price CONTAINS CURRENCY("$450" < CUR < "$10,100.50", SYMBOL="$", SEPARATOR=",",
DECIMAL="."))
```

Symbol characters used must be consistent throughout the entire expression.

It is not required that the currency be symbol be specified in the numeric portion of the currency expression, however, it must always be specified as the SYMBOL option. As an example, the query shown below is also legal, and would have the same function as the query shown above:

```
(RECORD.price CONTAINS CURRENCY("450" < CUR < "10,100.50", SYMBOL="$", SEPARATOR=",",
DECIMAL="."))
```

The results will contain all prices encountered in the input data that match the specified range. For example, if a price "$692.01" appears in the input data, then it will be reported as a match since it falls within the specified range. But currency values like -$123.00, $449.99 and $100,000 would not match.

3.3.4.8.1 Example Program

Suppose that a file `passengers-fares.csv` lists passengers including the ticket price that each paid:

```
name,dob,phone,fare
Hannibal Smith,10-01-1928,310-555-1212,$1436.00
DR. Thomas Magnum,01-29-1945,310-555-2323,$925.00
Steve McGarett,12-30-1920,310-555-3434,$1207.00
Michael Knight,08-17-1952,310-555-4545,$422.00
Stringfellow Hawke,08-15-1944,310-555-5656,$726.00
Sonny Crockett,12-14-1949,310-555-6767,$214.00
```

To find ticket prices greater than `$450.00`, we could use the following user program. Note that a dollar sign is specified as the currency character, a comma is specified as the separator character, and a period as the decimal. This corresponds to US currency system rules, but anything could be used if other currency systems were desired. Note that

even when specified, the separator (in this example, a comma) does not have to appear in the input data, since the numeric engine is smart enough to recognize that a number `1,000` is the same as a number `1000`.

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-fares.csv");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.csv",
        "(RECORD.\"fare\" CONTAINS CURRENCY(CUR > \"450.00\", SYMBOL=\"$\",
SEPARATOR=\",\", DECIMAL=\".\"))",
        NULL,
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data results in `d.csv`:

```
name,dob,phone,fare
Hannibal Smith,10-01-1928,310-555-1212,$1436.00
DR. Thomas Magnum,01-29-1945,310-555-2323,$925.00
Steve McGarett,12-30-1920,310-555-3434,$1207.00
Stringfellow Hawke,08-15-1944,310-555-5656,$726.00
```

The program will also produce the following output index file results in `i.txt`:

```
passengers-fares.csv,20,48,0
passengers-fares.csv,68,50,0
passengers-fares.csv,118,48,0
passengers-fares.csv,213,51,0
```

3.3.4.9 IPV4

The IPv4 Search operation can be used to search for exact IPv4 addresses or IPv4 addresses in a particular range in both structured and unstructured text using the standard "a.b.c.d" format for IPv4 addresses. Octal parsing support as per various RFC guidelines can also be enabled via an option to the standard IPv4 search query string.

Note that this primitive is meant for searching text, not native binary network captures, such as PCAP files. To natively search binary PCAP files, please refer to the PCAP family of primitives as described in section 3.3.6.

IPv4 searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator IPV4(expression[, options]))
```

Different ranges can be searched for by modifying the expression in the expression above. There are two general expression types supported:

```
1.  IPV4(IP operator "ValueB")
2.  IPV4("ValueA" operator IP operator "ValueB")
```

The box below contains of list of supported expressions. ValueA and ValueB represent the IP addresses to compare the input data against.

```
IP = "ValueB"
IP != "ValueB" (Not equals operator)
IP >= "ValueB"
IP >  "ValueB"
IP <= "ValueB"
IP <  "ValueB"
"ValueA" <= IP <= "ValueB"
"ValueA" <  IP <  "ValueB"
"ValueA" <  IP <= "ValueB"
"ValueA" <= IP <  "ValueB"
```

For example, to find all IP addresses greater than 10.11.12.13, use the following search query criteria:

```
(RAW_TEXT CONTAINS IPV4(IP > "10.11.12.13"))
```

To find all matching IPv4 addresses adhering to 10.10.0.0/16 (that is, all IP addresses from 10.10.0.0 through 10.10.255.255 inclusive) in a record/field construct where the field tag is `ipaddr`, use:

```
(RECORD.ipaddr CONTAINS IPV4("10.10.0.0" <= IP <= "10.10.255.255"))
```

The available comma-separated `options` for the `IPV4` primitive are documented in section 3.3.2.

The default behavior of the IPv4 search primitive is to parse all octets encountered as decimal. When option `OCTAL="true"` is specified, any octets encountered with leading 0's will be parsed as octal quantities instead of decimal quantities for comparison purposes. This can be important in certain cyber analysis scenarios, where obfuscation attempts using octal notation can sometimes fool contemporary systems into decoding incorrect IP addresses if they do not properly parse octal quantities. This is also useful when it is known that certain input datasets may not be strictly adhering to a certain type of leading zero parsing rule.

3.3.4.9.1 Example Program

Suppose example file `passengers-fare-ip.txt` included the IPv4 address that was used by each passenger to purchase the ticket online:

```
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 10.0.0.23
DR. Thomas Magnum, 01-29-1945, 310-555-2323, $925.00, 10.10.0.41
Steve McGarett, 12-30-1920, 310-555-3434, $1207.00, 10.0.2.89
Michael Knight, 08-17-1952, 310-555-4545, $422.00, 192.168.0.22
Stringfellow Hawke, 08-15-1944, 310-555-5656, $726.00, 192.168.1.90
Sonny Crockett, 12-14-1949, 310-555-6767, $214.00, 10.0.0.241
```

Perhaps we need to identify those passengers purchasing tickets from IPv4 addresses with a first octet of "`10`". Looking at the input file, we can see that the file is CSV, but it doesn't contain a header line. Counting the columns, we see that the IPv4 address is in the 5th column. We'll use the following user program to construct a query against the 5th column looking for the IP addresses we care about. Even though the file has a `.txt` extension, the API primitive operation will still auto-detect that the format is CSV, and it will act accordingly. Here's the program we will use:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-fare-ip.txt");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.txt",
        "(RECORD.5 CONTAINS IPV4(\"10.0.0.0\" <= IP < \"11.0.0.0\"))",
        NULL,
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data results:

```
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 10.0.0.23
DR. Thomas Magnum, 01-29-1945, 310-555-2323, $925.00, 10.10.0.41
Steve McGarett, 12-30-1920, 310-555-3434, $1207.00, 10.0.2.89
```

```
Sonny Crockett, 12-14-1949, 310-555-6767, $214.00, 10.0.0.241
```

Usually the full record detail is quite useful for downstream processing, but let's say you only need the names of the individuals as they appear in those records. By inspection, this is trivially obtained by a simple, standard linux `cut` command, feeding it the output data file `d.txt`:

```
$ cut -d ',' -f 1 d.txt
Hannibal Smith
DR. Thomas Magnum
Steve McGarett
Sonny Crockett
```

Note that the program also produced the following output index results, which would allow downstream tools to access the records by offset and length at a later time if needed, without having to perform another search:

```
passengers-fare-ip.txt,0,62,0
passengers-fare-ip.txt,62,65,0
passengers-fare-ip.txt,127,62,0
passengers-fare-ip.txt,321,62,0
```

3.3.4.10 IPV6

The IPv6 Search operation can be used to search for exact IPv6 addresses or IPv6 addresses in a particular range in both structured and unstructured text using the standard "a:b:c:d:e:f:g:h" format for IPv6 addresses. The double colon (::) is also supported, per RFC guidelines.

Note that this primitive is meant for searching text, not native binary network captures, such as PCAP files. To natively search binary PCAP files, please refer to the PCAP family of primitives as described in section 3.3.6.

IPv6 searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator IPV6(expression[, options]))
```

Different ranges can be searched for by modifying the expression in the expression above. There are two general expression types supported:

```
1.  IPV6(IP operator "ValueB")
2.  IPV6("ValueA" operator IP operator "ValueB")
```

The box below contains of list of supported expressions. ValueA and ValueB represent the IP addresses to compare the input data against.

```
IP = "ValueB"
IP != "ValueB" (Not equals operator)
IP >= "ValueB"
IP >  "ValueB"
IP <= "ValueB"
IP <  "ValueB"
"ValueA" <= IP <= "ValueB"
"ValueA" <  IP <  "ValueB"
"ValueA" <  IP <= "ValueB"
"ValueA" <= IP <  "ValueB"
```

For example, to find all IP addresses greater than 1abc:2::8, use the following search query criteria:

```
(RAW_TEXT CONTAINS IPV6(IP > "1abc:2::8"))
```

To find all matching IPv6 addresses between 10::1 and 10::1:1, inclusive, in a record/field construct where the field tag is `ipaddr6`, use:

```
(RECORD.ipaddr6 CONTAINS IPV6("10::1" <= IP <= "10::1:1"))
```

The available comma-separated `options` for the `IPV6` primitive are documented in section 3.3.2.

3.3.4.10.1 Example Program

Suppose a `passengers-fare-ipv6.txt` input file included the printable IPv6 address that was used to purchase the ticket online from an airline carrier by each passenger:

```
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 1abc::11
DR. Thomas Magnum, 01-29-1945, 310-555-2323, $925.00, 1abb:2:3:4:5:6:7:8
Steve McGarett, 12-30-1920, 310-555-3434, $1207.00, 1a0:2b:3c::4d
Michael Knight, 08-17-1952, 310-555-4545, $422.00, ::3ca
Stringfellow Hawke, 08-15-1944, 310-555-5656, $726.00, 1abc:b59:14::21
Sonny Crockett, 12-14-1949, 310-555-6767, $214.00, 1ab0:13:c2::22
```

We are told that we must find passenger information for those passengers who purchased their ticket using an IPv6 address with a first hextet of `1abc`.

Although we can visually determine that the file is indeed a CSV file, let's say for the sake of argument that we didn't know that. Instead, we knew that each record of interest was contained on a single line. We could solve the problem by searching for a valid IPv6 address matching the criteria we need, and then use `LINE` mode to return the lines in question. We will use the following user program to achieve this:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-fare-ipv6.txt");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.txt",
        "(RAW_TEXT CONTAINS IPV6(\"1abc::\" <= IP < \"1abd::\", LINE=\"true\"))",
        NULL,
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }
```

```
    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data results:

```
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 1abc::11
Stringfellow Hawke, 08-15-1944, 310-555-5656, $726.00, 1abc:b59:14::21
```

The program will also produce the following output index results:

```
passengers-fare-ipv6.txt,0,61,0
passengers-fare-ipv6.txt,257,71,0
```

3.3.4.11 EMPTY_FIELD

The Empty Field search operation can be used against record- or field-based input to detect a field which exists, but whose contents are empty. This can be a useful primitive if you need to evaluate the quality of data, or verify the completeness of a particular record-based dataset.

For example, an XML field *<foo></foo>*, since it has no data between the start and end tags, would be considered empty. A JSON field *"foo" = ""* would also be considered empty. A comma-separated value column with no data, such as instantiated by a row containing two consecutive commas, would be considered empty as well. This is extremely useful at scale when validating key fields that are expected to always contain data. These might be things like unique identification numbers, names, addresses, phone numbers, and so on, across millions or even billions of rows of information. Empty fields in some of these cases can often occur as a result of human error (or bugs in software generation tools), making this a valuable primitive operation for ensuring data quality.

Note that determining if a field is empty this is not the same thing as determining if a field is missing. For example, an XML record that does not contain a starting `<foo>` tag is not considered to have an empty value in the `<foo>` tag, since the tag does not exist. Instead, the empty field primitive is meant to perform data verification solely on elements that do exist, but have no contents.

Empty Field searches extend the general expression defined previously as follows:

```
(input_specifier relational_operator EMPTY_FIELD(options))
```

In addition to standard options, for CSV input data alongside this primitive, it is useful to note that the `OUTPUT_HEADER` option is fully supported as described in section 3.3.2.

The value used for `input_specifier` must be `RECORD` for record-based searches or `RECORD.hierarchy` for field-based searches, where hierarchy is of course dependent on the data. For record-based searches, the `relational_operator` must be `CONTAINS` or `NOT_CONTAINS`. For field-based searches, the relational operator must be `EQUALS` or `NOT_EQUALS`.

For example, to find records that contain one or more empty fields, use:

```
(RECORD CONTAINS EMPTY_FIELD())
```

Similarly, to find all records that specify a `foo` field but whose contents are empty, use:

```
(RECORD.foo EQUALS EMPTY_FIELD())
```

3.3.4.11.1 Example Program

Suppose we have an input file `passengers-missing.csv` containing:

```
name,dob,phone
Hannibal Smith,,
DR. Thomas Magnum,01-29-1945,011-310-555-2323
Steve McGarett,12-30-1920,011-310-555-3434
Michael Knight,,
Stringfellow Hawke,08-15-1944,011-310-555-5656
Sonny Crockett,12-14-1949,011-310-555-6767
MR. T Magnum,,
```

Note that in this example, by inspection, three rows have `dob` and `phone` fields that are empty.

To identify the records where the contents of field `dob` and the contents of field `phone` are both empty, we could use the following user program:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-missing.csv");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.csv",
        "(RECORD.\"dob\" EQUALS EMPTY_FIELD()) AND (RECORD.\"phone\" EQUALS
EMPTY_FIELD())",
        NULL,
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
```

```
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data results in `d.csv`:

```
name,dob,phone
Hannibal Smith,,
Michael Knight,,
MR. T Magnum,,
```

The program will also produce the following output index results in `i.txt`:

```
passengers-missing.csv,15,17,0
passengers-missing.csv,121,17,0
passengers-missing.csv,228,15,0
```

3.3.4.12 PIP

The Point In Polygon (PIP) Search operation can be used to isolate data by longitude and latitude, comparing positions against arbitrary complex polygons of arbitrary numbers of vertices. These searches require the input data to be JSON, XML or CSV formatted, since PIP searches make sense only for record/field searches.

PIP searches extend the general expression defined previously as follows:

```
(RECORD[.field] [NOT_]CONTAINS PIP(VERTEX_[LIST | FILE]="data")[,[LONG |
LAT]_COORD="field"])
```

The `field` value is specified if a field contains both a combination longitude/latitude point in the input corpus, for comparison against the specified complex polygon. The specified field should contain a decimal degree coordinate point. The exact format of the point is not important; the first floating point number parsed out of the field will be treated as the longitude and second will be treated as the latitude. If fewer than two floating point numbers are parsed out of the specified field, the field is skipped.

Alternatively, if individual fields contain longitude and separately latitude points, omit the `field` value and instead use options `LONG_COORD` and `LAT_COORD` to specify the appropriate field names. Keep in mind that for CSV input with named fields, an extra set of escaped double quotes will be needed inside the standard double quotes given CSV parsing rules. For example, if a CSV input corpus uses field s_lon for longitude and s_lat for latitude, the options might include: `LONG_COORD="\"s_lon\"", LAT_COORD="\"s_lat\""`. If an extra set of double quotes is used externally to the entire expression, then further escaping may be required, such as: `LONG_COORD=\"\\\"s_lon\\\"\", LAT_COORD=\"\\\"s_lat\\\"\"`.

PIP algorithm calculations use double precision floating point math, meaning coordinates with up to 14 decimal digits are supported, although points beyond 8 decimal digits are extremely uncommon (and often infeasible). Note that the sixth decimal place is sufficient for standard latitude and longitude coordinate resolution down to about 11.1 centimeters (about 4.37 inches). The seventh decimal lace resolves to about 11mm, and the eighth decimal to about 1.1mm.

Since a record is either inside or outside of a given complex polygon, only `CONTAINS` and `NOT_CONTAINS` are supported relational operators, as show above. By default, the primitive uses an exclusive construct, meaning that results contain points that are fully inside the described complex polygon. An option (`INCLUSIVE`) exists if it is desired that points on the polygon boundary itself are also to be considered inside the polygon.

The box below details the data format for the two possible formats, `VERTEX_LIST` and `VERTEX_FILE`.

By default, `VERTEX_LIST` describes a polygon using a format of:

```
long,lat;long,lat;long,lat;...
```

Points that define a polygon cannot be listed in any arbitrary order. The requirement is that adjacent points must define an edge, including the wrap from bottom to top.

The following example would describe a compliant bounding box with four vertices mapping to a portion of the Washington, DC metro area:

```
VERTEX_LIST="-77.305425,38.789232;-76.823540,38.789232;-76.823540,39.037929;-
77.305424,39.037929"
```

`VERTEX_FILE` provides an alternate mechanism for describing a complex polygon, using a specified input text file which contains one point per line, longitude followed by latitude (on the same line), separated and/or surrounded by one or more whitespace characters.

The following example uses the same polygon vertices shown above with VERTEX_LIST, but instead specified by a file "polygon_points.txt":

```
VERTEX_FILE="/path/to/my_polygon_points.txt"
```

The contents of the file might be:

```
$ cat /path/to/my_polygon_points.txt
-77.305425 38.789232
-76.823540 38.789232
-76.823540 39.037929
-77.305424 39.037929
```

Note that `VERTEX_FILE` can be very useful for very large polygons with many hundreds of vertices, such as those that might describe state boundaries, voting districts, international boundaries, oil and gas exploration boundaries, or other arbitrary areas of interest describable by sets of vertices defined by "longitude, latitude" pairs. After the first number is parsed, the parser will skip any non-relevant character it finds (such as the single space separator in the above example) until the next number quantity is found. The start of number character set utilized is a minus sign (-), a dot (.) and 0-9. This means that although that example above used a single space separator, it could have just as easily chosen to use ae comma, for example.

Polygons can be grouped in trivial fashion. This is accomplished by setting the option VERTEX_FILE_IS_FILELIST to true, in which case the VERTEX_FILE specifies a filename that is a list of files, one per line, which each describe individual polygons with the same conventions noted above. An example might resemble:

```
VERTEX_FILE="/path/to/my_list_of_polygon_files.txt", VERTEX_FILE_IS_FILEIST="true"
```

And the contents might contain references to two fully qualified pathnames for polygon descriptions:

```
$ cat /path/to/my_list_of_polygon_files.txt
/path/to/my_polygon_points.txt
/path/to/another_set_of_polygon_points.txt
```

Although convention dictates that the longitude point come first, it is possible to reverse the point positions using a latitude point followed by a separator followed by a longitude point by setting options `FORMAT_DATA` and/or `FORMAT_POLYGON` to `LAT_LONG`. By default, the setting is `LONG_LAT` which means the system will parse points in the input data set and in polygon descriptions as longitude-first. These and other available comma-separated `options` for the `PIP` primitive are documented in detail in section 3.3.2.

3.3.4.12.1 Example Program

In this example we search an input corpus consisting of JSON records to determine if a specific field in the record that combines both longitude and latitude information contains a geospatial coordinate that falls within an arbitrary polygon that we define. In this example, a simple bounding box around the nation of Bermuda defined by the following four coordinates is chosen as the polygon (but any complex polygon with an arbitrary number of vertices and features could be used, even those describing very large geographic regions, such as the state of Alaska and all of its constituent islands):

-64.8960,32.2307
-64.8960,32.3938
-64.6430,32.3938
-64.6430,32.2307

Our input data for this simple example contains the following two JSON records, and is contained in a file `idloc.json`:

```
{"id":"a18929347840059e","attributes":{},"id_str":"121038762064089088","coordinates":{"co
ordinates":[-86.1580423,39.7683765],"type":"Point"},"id":621038762064089088}
{"id":"a18929347840059f","attributes":{},"id_str":"121038762064089089","coordinates":{"co
ordinates":[-64.7855,32.3031],"type":"Point"},"id":121038762064089089}
```

By inspection, it is clear that the first JSON record is not inside the polygon as we've defined it, but the second record is.

The following user program implements the example:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "idloc.json");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.json",
        "(RECORD.coordinates.coordinates CONTAINS PIP(VERTEX_LIST=\"-64.8960,32.2307;-
64.8960,32.3938;-64.6430,32.3938;-64.6430,32.2307\"))",
        "\n",
        "i.txt",
```

```
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc` to compile a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will produce the following output data results in requested output file `d.json`, which is the expected matching record:

```
{"id":"a18929347840059f","attributes":{},"id_str":"121038762064089089","coordinates":{"co
ordinates":[-64.7855,32.3031],"type":"Point"},"id":121038762064089089}
```

The output index file `i.txt` that our program was instructed to create contains:

```
idloc.json,166,159,0
```

This example made use of the `VERTEX_LIST` format for specifying the polygon, instead of the `VERTEX_FILE` approach.

Using `VERTEX_FILE` can be beneficial when describing very complicated polygons with a large number of vertices, especially if that information has already been properly encoded in a separate set of file(s). For more information on the `VERTEX_FILE` format for `PIP`, please refer to section 3.3.4.12.

3.3.4.13 Results: Search Primitives

Index results files contain a single line for each search match, where each line has the following format:

```
Filename,File Offset,Match Length,Actual Matched Distance[,Other Data]
Filename,File Offset,Match Length,Actual Matched Distance[,Other Data]
Filename,File Offset,Match Length,Actual Matched Distance[,Other Data]
…
Filename,File Offset,Match Length,Actual Matched Distance[,Other Data]
```

The `Other Data` is API primitive- and options-dependent, for example, when enabling match qualifiers for some operations, the `Other Data` field will be updated appropriately. For details of those formats, please refer to those topics elsewhere in this guide, such as the match qualifier option information described in section 3.3.2.

The `Actual Matched Distance` field has meaning only when running unstructured searches that are not line-based. For all other searches (including all record-based searches), the `Actual Matched Distance` field will be populated with `n/a` or `0`.

Consider the example line below which indicates that a match was found where the resulting distance for the unstructured match was `1`.

```
passengers.txt,31,3,1
```

Data results files generated by search primitives will have the following format, with a line for each match:

```
[Data before Match Data][Match Data][Data after Match Data]
[Data before Match Data][Match Data][Data after Match Data]
[Data before Match Data][Match Data][Data after Match Data]
…
[Data before Match Data][Match Data][Data after Match Data]
```

The amount of data before and after a raw text match data can be modified by changing the `WIDTH` option in the match query string, as described in section 3.3.2. Note that search result data files, unlike index search result files, will not contain any indication of the matched distance.

If the match query string used the `LINE` option, then the data results file will contain the line delimited by a line feed (0xa) on which the [Match Data] appeared in the input corpus:

```
[Start of line containing Match Data][Match Data][Remainder of line containing Match Data]
[Start of line containing Match Data][Match Data][Remainder of line containing Match Data]
[Start of line containing Match Data][Match Data][Remainder of line containing Match Data]
…
[Start of line containing Match Data][Match Data][Remainder of line containing Match Data]
```

For more information about the `LINE` option, refer to section 3.3.2.

3.3.4.14 SEARCH and REPLACE

Search and replace operations against unstructured data is a well-known problem and has been traditionally solved with tools like `grep`, `sed`, and various related regular expression engines. When it comes to structured data sets, though, search and replace operations are quite a bit more complicated.

This API provides powerful capabilities to perform both simple and complex replacement operations against both unstructured data types as well as XML, JSON and CSV structured data types. This provides a seamless capability for high-performance update operations, akin to similar operations in expensive production databases.

Search and replace functionality is generally a complicated problem that requires extensive knowledge of the data, and the types of search patterns that can be used are often quite limited. Making matters worse, typical search and replace algorithms typically don't perform very well against arbitrary data, especially data that hasn't been indexed.

The search and replace functionality described in this API solves these problems using a powerful parallelization framework library. Use this syntax to invoke the search and replace functionality:

```
rx_data_set_t rx_ds_replace(
    const rx_data_set_t     data_set,
    const char*             filter_query_string,
    const char*             replace_query_string,
    void*                   (*percentage_callback)(uint8_t));
```

- A dataset of type `rx_data_set_t` is the return value, representing the results of the operation. This can be used for subsequent library calls as needed, such as for statistics gathering. The returned dataset should be deleted by the user via a call to `rx_ds_delete()` when it is no longer needed in order to release any associated system resources.
- `data_set` is the input dataset of type `rx_data_set_t` to be searched.
- `filter_query_string` is the string specifying the search criteria to identify records that will be operated on via the eventual `replace_query_string` as described below. If a raw text replacement is desired, set `filter_query_string` to an empty string.
- `replace_query_string` is a constructed string of multiple fields specifying the necessary information needed by the replacement operation, as described below.
- `percentage_callback` references a user-defined callback function for handling incoming percentage completion messages. For example, a user could provide a callback function for printing the current status of the execution to standard output. It may be `NULL` if no progress reporting is desired.

When used to filter specific records, the `filter_query_string` has a similar format to the familiar `match_criteria_query_string` used throughout the search primitive family. The `filter_query_string` can be any valid structured query string incorporating any search primitive, to include complex primitive clauses using boolean logic, using `RECORD` and/or `RECORD.field` constructs targeting structured data in XML, JSON and CSV formats. Any field constructs used in the `filter_query_string` do not need to be the same as those used in the `replace_query_string`; they are treated completely independently.

The `replace_query_string` is a compound string having the following format:

```
(RAW_TEXT|RECORD[.field], "REGEX_FIND", "REGEX_REPLACE", ".file_extension")
```

- Commas separate the fields in the compound replacement string.
- All replacement string fields but the first must be double quoted.
- Either the `RAW_TEXT` or `RECORD` qualifier is required.
- Optional field hierarchies using the appropriate formatting rules for the structured input data can follow a `RECORD` qualifier. For example, if the data corpus is CSV, this might be a numbered column field or a quoted named column header. As another example, if the data corpus is JSON, it might have a data-dependent complex hierarchy, such as something like `.pricing.[].price`.
- Following the `RAW_TEXT` or `RECORD[.field]` qualifier, the first regular expression `REGEX_FIND` searches just the `RECORD[.field]` hierarchy with the specified PCRE2-compliant regular expression. The effective byte positions of the match(es) are stored, and fed to the second regular expression as described below.
- The second regular expression `REGEX_REPLACE` is also a PCRE2-compliant regular expression. It is used to replace the byte offset positions that matched those associated with the match(es) from the previous `REGEX_FIND`.
- Finally, the last portion "`.file_extension`" specifies a file extension that will be appended to each filename in the input corpus file list if a replacement file was generated.

A replacement file is generated only when at least one replacement has been made for a given input file. For example, if `.foo` were specified to be appended to the replacement file extension, and if the input corpus contained a single file `myfile.json`, and at least one replacement occurred, then an output file `myfile.json.foo` would be generated. Note that the input file corpus is always maintained in its original, pristine form.

The combination of the `filter_query_string` and the `replace_query_string` provides a powerful mechanism to arbitrarily filter an entire input corpus to identify a subset of records to operate on for an arbitrary and independent second-level search-and-replace.

3.3.4.14.1 Example Program

In this example, we illustrate a search and replace operation against a structured JSON dataset. Consider the following simple example JSON file called `widgets.json`:

```
{
        "name":"Widget A",
        "description":"This is widget A.  It is really cool!",
        "pricing":[
                {"price":3.09},
                {"price":3.99}
        ]
}
{
        "name":"Widget B",
        "description":"This is widget B.  It isn't nearly as cool as widget A, but some
people like it since it is cheaper.",
        "pricing":[
                {"price":1.09},
                {"price":1.99}
        ]
}
{
        "name":"Widget C",
        "description":"This is widget C.  It is super cool, but it is also the most
expensive widget.  Sometimes it goes on sale for a really nice price.",
        "pricing":[
                {"price":9.99},
                {"saleprice":3.99}
        ]
}
```

In this case, what we'd like to do is execute a search and replace operation such that any record that has the phrase `super cool` in its description, we'd like to change its sale price to `$5.75`.

We first filter the records that contain `super cool` in the description field. Then we replace the `price` field (but not the `saleprice` field) just in those records with a value of `23.25`, appending `.new` to our generated replacement JSON file.

The following user program `test.c` implements the example:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "widgets.json");

    rx_data_set_t results = rx_ds_replace(
        input_data_set,
        "(RECORD.description CONTAINS EXACT(\"super cool\"))",
        "(RECORD.pricing.[].price, \"[0-9]*[.][0-9]*\", \"23.25\", \".new\")",
        NULL);
```

```
    if (rx_ds_has_error_occurred(results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will generate an output file named `widgets.json.new` (which is the same path location where the input corpus file `widgets.json` resides):

```
$ ls widgets*
widgets.json  widgets.json.new
```

Executing a `diff` against the source file `widgets.json` and the replacement results file `widgets.json.new` shows:

```
$ diff widgets.json widgets.json.new
21c21
<               {"price":9.99},
---
>               {"price":23.25},
```

Verifying the results are correct is trivial in this case, since by inspection of the input data corpus it is clear that the third record is the only record that has “super cool” in its description. And the `diff` proves that we did indeed correctly change its price to `23.25`, with the remainder of the file precisely preserved.

Although this example used a single JSON file for the source corpus, the same methodology would apply for multiple file input corpuses to include unstructured, XML, or CSV input data. However, note that it is not permitted to mix-and-match different file types in the same invocation. For example, if search and replace operations are needed across both XML and JSON datasets, then two appropriately configured `rx_ds_replace()` invocations would be required.

3.3.5 Query Dictionaries and Match Qualifiers

The query dictionary functionality can be used alongside the Exact, Hamming, Edit Distance, and PCRE2 Search primitives to search for large numbers of entries very rapidly leveraging internal parallel processing engines. The entries desired are simply listed in a text file, one per line. The primitive operation is informed about the dictionary request via a query clause option DICTIONARY, as described in section 3.3.2.

The match qualifier capability is a powerful feature that can be used in isolation with standard Boolean logic queries or alongside a query dictionary. It is specified as a global option for a particular set of primitive operations, the syntax of which is described in section 3.3.3. It results in appending specific qualifying information to each output index file entry. This allows a user to know which of many possible dictionary entries or lengthy sections of Boolean logic resulted in the match.

We will use the following example to illustrate the capabilities and power of the dictionary and match qualifier functions.

3.3.5.1 Example Program

Suppose we have a list of passenger information stored in `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

In this example, we look for entries for “Hawke” or “Knight” and we only care about those surnames and not about any other data associated with that passenger. We could have used a simple two-clause `OR` expression to achieve this, but sometimes a data dictionary is preferable, especially as the number of comparisons rises. In this example, we create a data dictionary with the two entries we need:

```
$ cat dict.txt
"Hawke"
"Knight"
```

In the user program, we utilize raw text mode and generate a set of index file output results and specify the query dictionary mode using the `DICTIONARY` option, leaving the standard query string nulled via back-to-back double quotes. We also enable the match qualifiers global option, so that we can determine which query specifier from the dictionary resulted in each index file output:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers-simple.txt");

    rx_set_global_options("enable-match-qualifiers=1");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.txt",
        "(RAW_TEXT CONTAINS EXACT(\"\", DICTIONARY=\"dict.txt\"))",
        "\n",
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);
```

```
    return ret;
}
```

An example compilation line using standard `gcc`, making sure to link against the `ryftx` library, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

The output data file `d.txt` will contain:

```
Knight
Hawke
```

The output index file `i.txt` will contain:

```
passengers-simple.txt,134,6,0,MQ='"Knight"'
passengers-simple.txt,180,5,0,MQ='"Hawke"'
```

Each line in the index file represents a match, so we had two matches. Since we specified the global option `enable-match-qualifiers=1`, we include name/value pair match qualifier data appended to each output row which allow us to easily determine which one of the various dictionary entries resulted in the particular result in question. If the user decides they need other information later, they need only leverage the output index file to identify precise information about the location of the specific match qualifiers in the source dataset, since match qualifiers were enabled. If match qualifiers had not been enabled, the output index file would have contained this instead:

```
passengers-simple.txt,134,6,0
passengers-simple.txt,180,5,0
```

3.3.5.2 Match Qualifier Output When LINE Mode Is Used

If the query in the above program had instead been written as follows, then line mode would be enabled alongside the dictionary request:

```
"(RAW_TEXT CONTAINS EXACT(\"\", DICTIONARY=\"dict.txt\", LINE=\"true\"))"
```

Recall that in LINE modes, API primitives will consolidate all matches found on a line and report only a single line in the result, thereby avoiding duplicates.

For example, let's say the following dictionary input file dict.txt contains the following two strings:

```
$ cat dict.txt
"-1952,"
"Knight"
```

Referring back to the `passengers-simple.txt` input corpus, it is clear that these entries will resolve to the same line. With match qualifier output enabled, the index file results would consist of a single line of output, and only one of the two match qualifiers would be shown. In cases where distance values differ, the smallest distance value will be used, and in cases where multiples of those exist, the qualifier with the lowest file offset will be used. In this case, these rules yield:

```
passengers-simple.txt,126,41,0,MQ='"Knight"'
```

Therefore, LINE mode is not recommended for use in situations where it is important to definitively know all possible qualifiers per line result for large dictionaries or long OR clauses. Instead, using surrounding width may be a preferred approach, or, if the data is semi-structured, RECORD modes as described below are the preferred approach.

3.3.5.3 Match Qualifier Output For RECORD Modes

If the query in the above program had instead been written as follows, then RECORD mode would be enabled alongside the dictionary request:

```
"(RECORD.1 CONTAINS EXACT(\"\", DICTIONARY=\"dict.txt\"))"
```

Note that although the input file is still the `passengers-simple.txt` corpus, by invoking `RECORD` mode, the API engines will attempt to auto-detect the file format, independent of its extension. In this case, it will analyze the file and determine that it should be treated as a CSV file, even though its extension is .txt. This is a very useful feature, and we make use of it here as part of this example. In our case, we inform the query engine that we want to search the first column of the CSV file for the various dictionary entries. Like all `RECORD`-based queries, matches will return the entire record(s) of interest.

We revert our dictionary file to the original dictionary data used earlier, but we add a third name string:

```
$ cat dict.txt
"Hawke"
"Knight"
"Michael"
```

Referring back to the `passengers.txt` input corpus, it is clear that we should match two CSV record entries. With match qualifier output enabled, the index file results would consist of two lines describing the records, and all appropriate match qualifiers for each (even when multiple qualifiers result in the same record):

```
passengers-simple.txt,126,41,0,MQ='"Knight" "Michael"'
passengers-simple.txt,167,45,0,MQ='"Hawke"'
```

Thus, unlike LINE modes which de-duplicate and result in single qualifier responses, record modes will show all available match qualifiers. This can be very useful when analyzing semi-structured or structured datasets with large data dictionaries.

3.3.5.4 Match Qualifier Output For Boolean Logic (AND, OR, etc.) Modes

Keeping all else equivalent, if the query in the above program had instead been written as follows, then boolean logic optimizations need to be taken into consideration when analyzing any requested match qualifier results:

```
"(RECORD CONTAINS EXACT(\"Hawke\")) OR (RECORD CONTAINS EXACT(\"Knight\")) OR (RECORD
CONTAINS EXACT(\"Michael\"))"
```

When processing queries containing boolean logic, clauses are optimized in an attempt to minimize the amount of processing work required. In this example, since a standard OR clause is used, the first OR that finishes for a given record will win. For example, referring back to the input corpus, the search for “Knight” and the search for “Michael” result in the same record. Only the qualifier associated with the half of the OR clause that resulted in the winning run

will be included in the match qualifier output for the output record. The associated index file output with match qualifiers enabled in this case might therefore resemble something like:

```
Passengers-simple.txt,126,41,0,MQ='"Michael"'
Passengers-simple.txt,167,45,0,MQ='"Hawke"'
```

This distinction is therefore an important one when compared to the previous data dictionary approach for record searches, where all match qualifiers associated with the record were listed, as all such entries are always run in that mode with no further processing optimization. Therefore, utilizing a data dictionary approach may be better for some use cases (for example, when knowing all possible match qualifiers are of interest), whereas a boolean logic clause approach may be better in other use cases (such as perhaps where performance is the driver).

3.3.5.5 Complex Boolean Logic with Various Primitives Leveraging Query Dictionaries

Since the query dictionary functionality is implemented internal to the boolean logic engine that groups primitives, it is possible to use as many query dictionaries as are needed across any number of boolean logic clauses (such as ANDs and ORs), mixing any number of EXACT, HAMMING, and PCRE2 primitive requests. For example, a complex expression like the one below can be crafted:

```
'((RECORD.Block CONTAINS EXACT("", DICTIONARY="dict1.txt")) AND (RECORD CONTAINS
PCRE2("", DICTIONARY="dict2.txt"))) OR (RECORD CONTAINS HAMMING("",
DICTIONARY="dict3.txt", DISTANCE="1"))'
```

Query dictionary functions can also be mixed-and-matched with non-query dictionary functions:

```
'(RECORD.Equation CONTAINS PCRE2("", DICTIONARY="dict.txt")) AND (RECORD CONTAINS
NUMBER("3.14" < NUM < "3.15", SEPARATOR=",", DECIMAL="."))'
```

3.3.5.6 Specifying a Query Dictionary of PCRE2 Regular Expressions

The query dictionary's value from a performance perspective is its ability to churn through the same resident memory multiple times. This makes it a very attractive alternative when comparing against hundreds or even thousands of PCRE2 regular expressions.

Unfortunately, out of necessity, PCRE2 engines typically deal with buffers in different ways depending on whether or not start-of-line or end-of-line anchors (typically either ^ or $, respectively) are specified in the expression. This is because PCRE2 engines are generally line-based processing tools, which are extended to larger multi-line datasets.

This means that sending a list of expressions where some have anchors and some do not through the PCRE2 engine without changing the data formatting can result in nonsensical output. More concerning is that it isn't possible to know if the possibly bogus output is bogus or not.

To assist with this problem, the dictionary parser provides as much help as it can. Specifically, it will parse the dictionary entries and look at the first and last characters of each dictionary entry, and if at least one entry in the dictionary has at least one anchor used, it will check to ensure that all dictionary entries have at least one anchor type used. The types of anchors don't have to be the same, but they have to be there. If an entry is found that does not use an anchor whereas other entries do, an error message will be generated. This is not a failsafe against all possible written expressions, but it certainly covers the vast majority of real-world use cases.

If you encounter this condition, the solution is straightforward. Since the query dictionary implementation lives alongside a robust boolean logic framework, simply create two (or more, as needed) separate query dictionaries, one

that contains the anchored expressions, and one that contains the remainder. Then construct an OR clause to link them together. This will result in two separate data formatting flows through the PCRE2 engine, thereby eliminating the resident anchor problem.

3.3.5.7 A Replacement Strategy When Using Query Dictionary and Match Qualifiers

There is no built-in all-in-one mechanism to do replacements alongside query dictionary requests, given the effectively infinite number of possibilities surrounding potential end user use cases.

When search-and-replace is needed in these scenarios, it is recommended to run the query dictionary first in its entirety using match qualifiers, and then run appropriate replacements as individual requests based on what was matched. This could be handled via a separately written `bash` script or user program designed for the use case(s) needed.

3.3.6 PCAP Primitive Family

The PCAP primitive family is a set of search primitives specifically geared towards rapid cyber analytics using an arbitrary corpus of one or more binary PCAP (.pcap) files. There are many open source and proprietary tools on the market today which enable PCAP analysis, but many of them perform quite poorly, and routinely run into computational and memory bottlenecks, especially as the input corpus becomes large. In addition, many common tools cannot operate against raw binary PCAP data and require a very time-consuming, up-front transformation on the input data, resulting in a bloated set of XML or JSON data representing the individual packets.

The PCAP primitive family defined in this API does not require such unnecessary transformation steps.

Use this syntax with the PCAP primitive family:

```
rx_data_set_t rx_ds_pcap_search(
    const rx_data_set_t     data_set,
    const char*             results_file,
    const char*             match_criteria_query_string,
    const char*             index_results_file,
    void*                   (*percentage_callback)(uint8_t));
```

- A dataset of type `rx_data_set_t` is the return value, representing the results of the operation. This can be used for subsequent library calls as needed, such as for statistics gathering. The returned dataset should be deleted by the user via a call to `rx_ds_delete()` when it is no longer needed in order to release any associated system resources.
- `data_set` is the input dataset of type `rx_data_set_t` to be searched. Note that all files in the input data set must have the same frame time resolution, or an error will be generated. To handle mixed sets, simply operate on them as two distinct operations, instead of attempting to glob them together in one operation.
- `results_file` is the output data file to be created. It may be `NULL` if no results file is desired. The data results file generated by the PCAP primitive operation include the collection of source packets that match the various complex criteria of the operation requested. The typical result file is therefore a greatly thinned version of a much larger set of PCAP input. The resulting file is itself a valid PCAP file.
- `match_criteria_query_string` is the string specifying the search criteria, which varies depending on the type of PCAP primitive(s) that will be employed. It is described in more detail in subsequent sections.
- `index_results_file` is the output index file to be created. It may be `NULL` if no index file is desired. An output index file must be a text file. If the file extension of `index_results_file` is not ".txt", the system will append a ".txt" extension to whatever filename is specified. For example, if you supply "my_index", the result

file will be “my_index.txt”. If you specify “my_index.txt”, no changes will be made. The index file output is a set of comma-separated value lines, one line per match. The columns are, left to right:
    - filename where the match was found,
    - byte offset (where offset 0 is the first byte of the file) of the start of the packet where the match was found,
    - the length of the packet where the match was found,
    - the distance, which is usually reported as 0, except for Hamming and Edit Distance payload invocations,
    - and the layer 4 payload match qualifier(s) and packet number, if match qualifiers are selected in the global options
- `percentage_callback` references a user-defined callback function for handling incoming percentage completion messages. For example, a user could provide a callback function for printing the current status of the execution to standard output. It may be `NULL` if no progress reporting is desired.
- **Note that unlike all other primitives which link against the standard API’s `libryftx.so` library and use headers from `libryftx.h`, the PCAP primitive must link against the `libryftx_pcap.so` library and make use of headers from `libryftx_pcap.h`.**
    - It is allowable to use both headers in the same program, and link against both shared libraries in the same program.

3.3.6.1 Supported Layers

The PCAP family primitives support a wide variety of operations at protocol layers 2, 3 and 4, plus a mechanism by which any API primitive (such as edit distance, PCRE2, numeric, currency, etc.) can be used against arbitrary layer 4 payloads, thereby enabling complex querying all the way up the protocol stack, through layer 7.

For layers 2, 3 and 4, the PCAP family extend the general expression defined previously as follows:

```
[()[!]layer.field[.subfield] operator value[)]
```

Parenthetical groupings are optional. The `!` is used to represent logical inversion of the expression as needed.

Complex expressions follow the syntax show below, created using one or more logical AND (`&&`) or logical OR (`||`) groupings, where each side of the grouping is an expression as previously defined, and depicted here as the quantity `...`:

```
... [ && | || ...]
```

The box below shows valid `layer` and associated `field` and `value` information along with brief descriptions. Many of these are modeled after similar `Wireshark` nomenclature and semantics:

```
frame – a meta-layer associated with the frame headers attached to each packet in a PCAP
file. The associated field value can be one of:
  time – the frame time in UTC, with format:
    Mmm DD, YYYY HH:MM:SS[.s[s[s[s[s[s[s[s[s]]]]]]]]]
    where Mmm is the three-letter capitalized month, e.g., Jan
  time_delta – the delta time from the previous packet in the input pcap file
    associated with the packet, as a time offset in seconds with up to nine positions
    after the decimal point
  time_delta_displayed – the delta time from the previous packet matched in the query
    filter associated with the operation, as a time offset in seconds with up to nine
    positions after the decimal point
  time_epoch – the frame time as the number of seconds elapsed since midnight
    of January 1, 1970
  time_invalid – 1-bit: 1 if the frame time is invalid (out of range), 0 if valid
```

```
  time_relative – the delta time from the first packet in the input pcap file associated
    with the packet, as a time offset in seconds with up to nine positions after the
    decimal point

eth - Ethernet layer 2 protocol.  The associated field value can be one of:
  src - source address (value: 6 2-hex-digit octets separated by colons)
  dest [synonym: dst] - destination address
  addr - either the source or the destination address
  type - 16 bit Ethertype (integer or hex if leading 0x used)

vlan - Virtual LAN, as defined by IEEE 802.1Q, which resides at or alongside layer 2.
The associated field value can be one of:
  etype - VLAN Ethernet subtype (integer or hex if leading 0x used)
  tci - 16 bit value representing all tag control information (TCI)
  priority | pcp - priority code point 3-bit (0-7) field
  dei | cfi - drop eligible indicator (DEI) one bit (0-1) field
  id | vid - VLAN identifier, lower 12 bits (0-4095) of the TCI.  Note: if double tagging
    is used, both the inner and outer tags will be searched

mpls - Multi-protocol label switching, which is a hybrid layer 2/3 protocol.  The
associated field value can be one of:
  label - 20-bit value (0-1,048575) for the MPLS label
  exp | tc - 3-bit value (0-7) for the traffic class (and QoS and ECN)
  bottom | s - 1-bit: 1 when the current label is the last in the stack
  ttl - 8-bit (0-255) time-to-live indicator

arp – Addres resolution protocol, which resides at layer 2.
  isgratuitous – 1-bit: 1 when the ARP packet is a gratutitous ARP
  src.hw_mac - source MAC of the ARP (value: 6 2-hex-digit colon-separated octets)
  dst.hw_mac – destination MAC of the ARP (value: 6 2-hex-digit colon-separated octets)
  src.proto_ipv4 – source IPv4 of the ARP (value: [0-255].[0-255].[0-255].[0-255])
  dst.proto_ipv4 – destination IPv4 of the ARP (value: [0-255].[0-255].[0-255].[0-255])
  hw.type – 2 bytes: the network link protocol type (Ex: Ethernet is 0x0001)
  hw.size – 1 byte: the size in octets of the hardware address. (Ex: Ethernet is 6)
  proto.type – 2 bytes specifying an EtherType compliant protocol for which the ARP is
    intended. (Ex: IPv4 is 0x0800)
  proto.size – 1 byte: length in octets of layer 3 protocol. (Ex: IPv4 address size is 4)
  opcode – 2 bytes: operation the sender is performing: 1 for request, 2 for reply

ip - IPv4 layer 3 protocol.  The associated field value can be one of:
  src - source address (value: [0-255].[0-255].[0-255].[0-255])
  dest [synonym: dst] - destination address
  addr - either the source or the destination address
  proto – protocol used in the data portion of the IPv4 datagram (value: [0-255])

ipv6 - IPv6 layer 3 protocol.  The associated field value can be one of:
  src - source address (value: 8 2-hex-digit octets separated by colons)
  dest [synonym: dst] - destination address
  addr - either the source or the destination address
  nxt – protocol used in the data portion of the IPv6 datagram (value: [0-255])

icmp - ICMP protocol, for IPv4.  The associated field value can be one of:
  type - the ICMP type (value: [0-255])
  code - the ICMP subtype (value: [0-255])

icmpv6 - ICMP protocol, for IPv6.  The associated field value can be one of:
  type - the ICMP type
  code - the ICMP subtype

udp - UDP layer 4 protocol.  The associated field value can be one of:
```

```
  srcport - the source port (value: [0-65535])
  dstport - the destination port
  port - either the source or the destination port

tcp - TCP layer 4 protocol.  The associated field value can be one of:
  srcport - the source port (value: [0-65535])
  dstport - the destination port
  port - either the source or the destination port
  flags - boolean quantities (0 or 1) representing the TCP flags
    construct, with the following supported subfields:
      fin - the FIN bit
      syn - the SYN bit
      reset - the RST bit
```

The supported operators vary depending on the field and value, but in general adhere to:

```
Equality:                           ==, eq
Inequality:                         !=, ne
Greater than:                       >, gt
Greater than or equal to:           >=, ge
Less than:                          <, lt
Less than or equal to:              <=, le

Not:                                !, not
```

Note that the "Not" operations are currently supported for use directly alongside boolean types only, such as `tcp.flags.[fin|syn|reset]`, and do not propagate through complex expressions.

3.3.6.2 IPv4 Fragmentation Support

PCAP IPv4 (ip) support includes support for native IPv4 packet fragmentation. The library intelligently handles IP fragmentation packets as native, individualized packets, or as combined, de-fragmented payload data, depending on the types of operations requested in the associated query clause. For example, if a message has been fragmented into four packets and results in a payload match, then only one result is reported in the tabulated results, yet all packets that made up the reassembled quantity remain available for detailed analysis.

The library ensures that output PCAP data files maintain the precise order and timing of the fragments with respect to other packets in the source corpus. This means that downstream tools (such as `Wireshark` or `tshark`) can continue to operate on the output PCAP data natively if needed.

3.3.6.3 TCP and UDP Payloads, with Full Layer 7 Primitive Support

Arbitrary API primitives can be used against any layer 4 payload, which by definition includes data up through layer 7, using the following slight variant of the standard search primitive form:

```
(RECORD.payload relational_operator primitive(expression[, options]))
```

The `RECORD.payload` portion shown above in bold is literal. The remainder of the expression is identical to what is described in section 3.3.1. This ensures that the power of any API primitive can be brought to bear on arbitrary payload data of each and every individual packet. This includes even complex primitives such as the edit distance primitive which tools like `Wireshark` do not natively support.

However, with the exception of IPv4 fragmentation as previously described, it is important to understand that the library's PCAP implementation currently does not concatenate cross-packet payloads together across a given stream conversation. The `RECORD.payload` operation works on an individual packet by individual packet basis.

3.3.6.4 TCP Stream Dumps, UDP Stream Dumps and Payload Dumps

Often times it is necessary to look at a conversation in its entirety, as opposed to filtering on individual packet criteria afforded by the standard API PCAP filter operations such as `RECORD.payload`.

One common workflow leveraging contemporary tools that is widely used to extract a full stream for further analysis is as follows:

1. Thin the original PCAP using appropriate API primitive constructs previously described, such as by using source/destination IP address, port, etc.
2. Run the thinned results through `Wireshark` or `tshark` using their "Follow Stream" features to extract and concatenate the stream and payload of interest as part of the conversation.
3. Feed the resulting raw depacketized output through a second set of API payload search primitives.

The workflow noted above can be quite time consuming as tools like `Wireshark` and `tshark` are not full parallel architectures. Furthermore, they often require significant CPU and RAM resources, and can therefore be prohibitive to use on very large input PCAP corpus data.

Thankfully, the BlackLynx APIs provide a mechanism by which the TCP streams, UDP streams, and payloads can be generated in parallel alongside any PCAP operation. This is achieved by using an appropriately crafted global option string entry as defined in section 3.3.3.

The global option string entry to use is `l4_stream_dump`, which takes sub-parameters that specify the direction (transmit as `tx`, receive as `rx`, or `both`), whether (`on`) or not (`off`) to output payload in addition to the isolated packet stream, and finally an output filename `prefix` for the file(s) that will be generated. The precise format is:

```
l4_stream_dump="tx|rx|both:on|off:prefix"
```

The output filenames used will start with the prefix specified, which can include full or relative path detail as desired. For example, if the files should be dumped to `/tmp` and start with `foo`, then the prefix used might be specified as `/tmp/foo`. The full output filename is defined as:

```
prefix_packet_nnn_tx|rx|both.pcap|payload
```

where:

- `prefix` is the `prefix` value from the associated `l4_stream_dump` global option string
- `_packet_` is literal and always appears in the filename
- `nnn` is the integer packet number from the associated input corpus file of the **first match** that resulted in the stream being followed
- `tx|rx|both` is the `tx|rx|both` selection from the associated `l4_stream_dump` global option string
- `.` is the literal extension separator and always appears in the filename
- `pcap|payload` is the extension. One output file with extension `pcap` is always generated and represents the extracted stream. A separate file with extension `payload` is generated as the concatenated L4 payloads, but only if the value on was specified as previously defined in the second sub-parameter of the associated `l4_stream_dump` global option string

Note that use of the `l4_stream_dump` global option string does not preclude the normal generation of output data and output index files. All such features can be used on combination, during the same operation, as desired.

3.3.6.5 Translating Between `Wireshark` / `tshark` and PCAP Query Filters

The query languages used by `Wireshark` and `tshark` are somewhat different than those used by our API engines. The differences are kept to a minimum, but are necessary to allow our engines to support features that `Wireshark` and `tshark` do not allow or are non-intuitive. The user should therefore take care when cut-and-pasting queries between the two. The following table is meant to provide guidance for commonly encountered inconsistencies:

`Wireshark` / `tshark` vs. Our API	**Description**
`not / !`	`Wireshark` / `tshark` and our API have slightly different definitions of the boolean operator `not` (and it's synonym, `!`). We adhere to the strict mathematical definition, and propagate it through a query using standard boolean algebra rules. So, in our query terminology, the expressions `ip.src != 1.2.3.4` and `!(ip.src == 1.2.3.4)` will provide exactly the same results, as standard mathematical boolean algebra rules suggest they should. This is not true in `Wireshark` and `tshark`. Their definition of the not operator, as taken from their display filters at the bottom of the URL at https://wiki.wireshark.org/DisplayFilters is as follows, and does not adhere to strict boolean algebra rules: *If you have a filter expression of the form name op value, where name is the name of a field, op is a comparison operator such as == or != or < or..., and value is a value against which you're comparing, it should be thought of as meaning "match a packet if there is at least one instance of the field named name whose value is (equal to, not equal to, less than, ...) value". The negation of that is "match a packet if there are no instances of the field named name whose value is (equal to, not equal to, less than, ...) value"; simply negating op, e.g. replacing == with != or < with >=, give you another "if there is at least one" check, which is not the negation of the original check.* This means that a `Wireshark` / `tshark` filter written as `!(ip.src == 1.2.3.4)` will return not only packets that would be equivalent to `ip.src != 1.2.3.4`, but it will also include *any and all packets that do not include an IP layer at all*! This can be extremely misleading (and frustrating) when doing packet analysis as filtered output may include 'extra' packets, such as spanning tree packets, and so on, which have nothing to do with IP addresses that the user was intending to tabulate in the first place. As such, the `Wireshark` / `tshark` user is cautioned to closely investigate the use of boolean negation operations in standard `Wireshark` / `tshark` queries. In general, it is better to not use them to avoid mathematical confusion, and to construct your query in a way that agrees with standard mathematical rules for boolean logic, both for analytics sanity and to ensure simplified conversions between `Wireshark` / `tshark` and our API filters.
`frame.time_delta_displayed`	The `Wireshark` / `tshark` filter for `frame.time_delta_displayed` has several known anomalies that make it non-intuitive in many scenarios, since it is defined to depend on the previous frame that is displayed, which depends on the display filter, and therefore scenarios can arise where it changes its own

Wireshark / tshark vs. Our API	Description
	data. These inconsistencies are well-documented on a variety of common Wireshark forums online. Our implementation solves these anomalies by intelligently leveraging the frame.time_delta_displayed only after all other relevant filter criteria have been met, making it a much more useful statistic. However, this means that it is possible (and even likely) that when comparing and contrasting this filter parameter between our API and standard Wireshark, different results will be seen. In general, when leveraging Wireshark to achieve a similarly intuitive capability, you are better off leveraging two separate Wireshark queries: the first as a content filter generating a new PCAP dataset, and then further filter just that data using the more specific frame.time_delta. With our API, there is no need to go through such hoops. Instead, you can simply leverage frame.time_delta_displayed in a standard complex query and achieve intuitive results.

3.3.6.6 Results: PCAP Family

One way to make meaningful use of the PCAP family of primitives is within the confines of a larger cyber analytics ecosystem, to quickly filter very large input data corpus that contemporary tools often struggle to process.

The data results file generated by the PCAP primitive operation include the collection of source packets that match the various complex criteria of the operation requested. The typical result file is therefore a greatly thinned version of a much larger set of PCAP input. Since results files are themselves valid PCAP files, they can be natively used by downstream tools such as Wireshark and tshark to perform further analysis.

In addition to output data file generation, the PCAP primitive also supports output index file generation. The index file output is a set of comma-separated value lines, one line per match. The columns are, left to right:

- filename where the match was found,
- byte offset (where offset 0 is the first byte of the file) of the start of the packet where the match was found,
- the length of the packet where the match was found,
- the distance, which is usually reported as 0, except for Hamming and Edit Distance payload invocations,
- the layer 4 payload match qualifier(s) as MQ='"data1"["data2"][...]' where dataN is the relevant portion(s) of the query string that applied, if match qualifiers are selected in the global options, and
- the associated packet number as PN='xx' where xx is the packet number, if match qualifiers are selected in the global options

The example program below discusses an output scenario in detail.

3.3.6.7 Example Program

In this example, to illustrate PCAP family operation, we have captured a set of packets using Wireshark native support for the popular PCAP file format, which happened to include a set of transfers to the popular CNN news organization's website, via the following URL:

```
http://www.cnn.com/
```

Unlike other examples in this guide, given the binary nature of raw network capture files, the input file data is not shown here in-place. For purposes of this example, we have stored the network packet capture data to a local file input-lin.pcap. For this simple example, the actual source file is approximately 5MB in size.

We will filter the conversation looking for just IP traffic to and from a particular known address that is believed to be associated with `cnn.com` (in this case IPv4 address `151.101.201.67`). In addition to filtering by the IP address, we further qualify our query to investigate the payload contents to determine if case-insensitive `cnn` is present.

The following user program `test.c` implements the example:

```
#include <stdio.h>
#include <libryftx_pcap.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "/home/ryftuser/ryft/regression/data/input-lin.pcap");

    rx_set_global_options("enable-match-qualifiers=1");

    rx_data_set_t results = rx_ds_pcap_search(
        input_data_set,
        "d.pcap",
        "ip.addr == 151.101.201.67 && (RECORD.payload CONTAINS EXACT(\"cnn\",
CASE=\"false\"))",
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` linked against the `libryftx_pcap.so` shared library would be:

```
gcc test.c -o test -lryftx_pcap
```

Running this user program will generate an output file adhering to the standard PCAP file format that contains only those packets that met the criteria set forth in our example program above. The size of the results file compared to the input corpus can be seen in the following output box, and we can see that we had output that matched our query, and the result was a significantly filtered input dataset. In this example, the original input data size was already somewhat manageable, but the same principles would apply for any size input data set.

```
$ ls -al *.pcap
-rw-rw-r-- 0 user group 5085683 Dec 18 14:21 input-lin.pcap
-rw-rw-r-- 0 user group  155621 Dec 18 14:21 d.pcap
```

Furthermore, the results file `d.pcap` could be fed through subsequent API operations or other external programs (such as `Wireshark` or `tshark`) that support standard-based PCAP files for further processing.

In addition, we generated an output index file, and since we enabled match qualifiers, in addition to standard source corpus filename, byte offset and packet length, the index file will also show the layer 4 match qualifier data that matched as it appeared in the query string, as well as include a reference to the associated packet number from the source input corpus:

```
input-lin.pcap,3346596,276,0,MQ='"cnn"',PN='9308'
input-lin.pcap,3351190,2818,0,MQ='"cnn"',PN='9323'
input-lin.pcap,3354090,2818,0,MQ='"cnn"',PN='9325'
input-lin.pcap,3357250,2818,0,MQ='"cnn"',PN='9329'
input-lin.pcap,3360606,2818,0,MQ='"cnn"',PN='9334'
input-lin.pcap,3363582,2818,0,MQ='"cnn"',PN='9337'
input-lin.pcap,3368418,2818,0,MQ='"cnn"',PN='9345'
input-lin.pcap,3371318,2818,0,MQ='"cnn"',PN='9347'
input-lin.pcap,3374218,2818,0,MQ='"cnn"',PN='9349'
input-lin.pcap,3378124,2818,0,MQ='"cnn"',PN='9354'
input-lin.pcap,3382080,2818,0,MQ='"cnn"',PN='9358'
input-lin.pcap,3384980,2818,0,MQ='"cnn"',PN='9360'
input-lin.pcap,3387880,2818,0,MQ='"cnn"',PN='9362'
input-lin.pcap,3390780,2818,0,MQ='"cnn"',PN='9364'
input-lin.pcap,3394252,2818,0,MQ='"cnn"',PN='9368'
input-lin.pcap,3397666,2818,0,MQ='"cnn"',PN='9371'
input-lin.pcap,3401416,4186,0,MQ='"cnn"',PN='9376'
input-lin.pcap,3405684,1450,0,MQ='"cnn"',PN='9378'
input-lin.pcap,3407216,2818,0,MQ='"cnn"',PN='9380'
input-lin.pcap,3410116,2818,0,MQ='"cnn"',PN='9382'
input-lin.pcap,3413016,2818,0,MQ='"cnn"',PN='9384'
input-lin.pcap,3415916,2818,0,MQ='"cnn"',PN='9386'
input-lin.pcap,3421716,2818,0,MQ='"cnn"',PN='9390'
input-lin.pcap,3424616,2818,0,MQ='"cnn"',PN='9392'
input-lin.pcap,3427516,2818,0,MQ='"cnn"',PN='9394'
input-lin.pcap,3430416,2818,0,MQ='"cnn"',PN='9396'
input-lin.pcap,3433316,2818,0,MQ='"cnn"',PN='9398'
input-lin.pcap,3441178,2818,0,MQ='"cnn"',PN='9407'
input-lin.pcap,3444078,2818,0,MQ='"cnn"',PN='9409'
input-lin.pcap,3446978,2818,0,MQ='"cnn"',PN='9411'
input-lin.pcap,3449878,2818,0,MQ='"cnn"',PN='9413'
input-lin.pcap,3452778,2818,0,MQ='"cnn"',PN='9415'
input-lin.pcap,3455678,2818,0,MQ='"cnn"',PN='9417'
input-lin.pcap,3459150,2818,0,MQ='"cnn"',PN='9421'
input-lin.pcap,3463168,2818,0,MQ='"cnn"',PN='9426'
input-lin.pcap,3466068,2818,0,MQ='"cnn"',PN='9428'
input-lin.pcap,3468968,2818,0,MQ='"cnn"',PN='9430'
input-lin.pcap,3471868,2818,0,MQ='"cnn"',PN='9432'
input-lin.pcap,3474768,2818,0,MQ='"cnn"',PN='9434'
input-lin.pcap,3477668,2818,0,MQ='"cnn"',PN='9436'
input-lin.pcap,3480568,2818,0,MQ='"cnn"',PN='9438'
input-lin.pcap,3483468,2818,0,MQ='"cnn"',PN='9440'
input-lin.pcap,3486368,2818,0,MQ='"cnn"',PN='9442'
input-lin.pcap,3489514,2818,0,MQ='"cnn"',PN='9445'
input-lin.pcap,3492414,2818,0,MQ='"cnn"',PN='9447'
input-lin.pcap,3495314,2818,0,MQ='"cnn"',PN='9449'
input-lin.pcap,3498214,2818,0,MQ='"cnn"',PN='9451'
input-lin.pcap,3501114,2818,0,MQ='"cnn"',PN='9453'
input-lin.pcap,3504014,2818,0,MQ='"cnn"',PN='9455'
input-lin.pcap,3506914,2818,0,MQ='"cnn"',PN='9457'
input-lin.pcap,3509814,2818,0,MQ='"cnn"',PN='9459'
input-lin.pcap,3512714,2818,0,MQ='"cnn"',PN='9461'
input-lin.pcap,3515614,2818,0,MQ='"cnn"',PN='9463'
input-lin.pcap,3520576,2818,0,MQ='"cnn"',PN='9470'
input-lin.pcap,3523476,2818,0,MQ='"cnn"',PN='9472'
input-lin.pcap,3526376,3149,0,MQ='"cnn"',PN='9474'
```

The packet numbers can be useful information if a user must operate on the original source corpus using downstream tools such as `Wireshark` and `tshark`, since packets of interest can often be referenced directly by their packet number, as opposed to relying on the relatively slow native search operations of those tools.

Simple aggregations are easy to calculate from some of this information as well. For example, if at some later time you find that you need to determine how many packets matched, simply run the index file through `wc`, since each line in the index file corresponds to one packet that was matched:

```
$ wc -l i.txt
56 i.txt
```

Thus, there were 56 packets that matched. You could of course have retrieved this information as a statistic at run-time by adding an appropriate API call to `rx_ds_get_total_matches`. See section 3.5 for more details on available statistics.

3.3.7 N-gram and Term Frequency Primitive Family

These sections detail the available types of n-gram and term frequency operations. Use this syntax with the n-gram primitive family:

```
rx_data_set_t rx_ds_ngram (
    const rx_data_set_t     data_set,
    const char*             results_file,
    const char*             criteria_string,
    void*                   (*percentage_callback)(uint8_t));
```

- A dataset of type `rx_data_set_t` is the return value, representing the results of the operation. This can be used for subsequent library calls as needed, such as for statistics gathering. The returned dataset should be deleted by the user via a call to `rx_ds_delete()` when it is no longer needed in order to release any associated system resources.
- `data_set` is the input dataset of type `rx_data_set_t`.
- `results_file` is the output data file to be created. It may be `NULL` if no results file is desired.
- `criteria_string` is the string specifying the n-gram expression, which varies depending on the type of n-gram operation requested. It is described in more detail in subsequent sections.
- `percentage_callback` references a user-defined callback function for handling incoming percentage completion messages. For example, a user could provide a callback function for printing the current status of the execution to standard output. It may be `NULL` if no progress reporting is desired.

3.3.7.1 N-GRAM

An n-gram is a contiguous sequence of `n` items (such as words) from a given sequence of text. The n-gram primitive supports values of `n` of 1, 2, and 3. Note that a value of 1 is equivalent to a traditional term frequency operation.

The purpose of the n-gram primitive is to tabulate the number of occurrences of each n-gram in a particular input corpus. This information is useful for classifying and correlating disparate documents with one another, such as grouping them into clusters of like-information. For example, many popular “page rank” algorithms have an n-gram / term frequency component to them. As another example, a means of grouping social media data from multiple input sources together via topic, sentiment, or other criteria can be largely driven by n-gram / term frequency analysis. Still another example is feeding a downstream document similarity algorithm, to find all documents in a larger corpus that are similar in concept to a specific selected document. For example, our DOCSIM family of primitives make heavy use of this and related concepts.

N-gram and term frequency analysis is a notoriously complex problem for traditional software to implement with any kind of reasonable performance. This is because of the multi-faceted nature of the problem space. First, the term(s) have to be identified, second, they must be counted/tabulated, and third, they are typically sorted by frequency. Each of these sub-operations are both compute and memory intensive operations, which can stress traditional software architectures, often resulting in operations that simply cannot complete in reasonable time at reasonable cost, even when clustered.

The API’s n-gram and term frequency primitives solve these problems.

The n-gram primitive's expression is defined as follows:

```
(NGRAM(RAW_TEXT | RECORD[.field.hierarchy][, N="1|2|3"][, options]))
```

The value of `N` is optional and will default to 1. Values o f 2 and 3 can be used to specify 2-gram (bigram) or 3-gram (trigram) n-grams, respectively. The value of `N` that you choose will likely depend on the downstream operations (such as the document similarity primitive described in section 3.3.8 in this API) that will leverage the output of the n-gram generator. In general, N-grams of length 3 are recommended for feeding document similarity algorithms as well as certain natural language processing algorithms for human language input corpus data with relatively rich features, especially for large document sets where each document contains at least a couple of pages worth of data.

Unlike many commercial and open source tools that struggle to generate n-grams of length 2 or 3 in reasonable time, our n-gram primitive is extremely fast even for very large corpus sizes.

The available comma-separated `options` are documented in section 3.3.2.

The following expression shows an example which would characterize a set of JSON input records by 2-gram in case-insensitive fashion solely considering the field hierarchy `employee.lastname`:

```
(NGRAM(RECORD.employee.lastname, N="2", CASE="false"))
```

3.3.7.2 TERM FREQUENCY

Term frequency is a special class of the n-gram primitive, where n=1. The overall behavior is identical to the n-gram primitive, but an alias definition is made available, using:

```
(TERM_FREQ(RAW_TEXT | RECORD[.field.hierarchy][, options]))
```

Internally, this maps precisely to an n-gram operation with `N="1"`.

The available comma-separated `options` are documented in 3.3.2.

3.3.7.3 Results: N-gram and Term Frequency

The n-gram and term frequency primitives have slightly different output formats depending on the nature of the input.

By default, when the responsible n-gram or term frequency primitive request was RECORD based, results files for the n-gram and term frequency primitives adhere to the following format, with one line of output corresponding to an individual record in an individual input file:

```
Filename1,Record1 Offset,Record1 Length,N-gramA,CountA,N-gramB,CountB,...
Filename1,Record2 Offset,Record2 Length,N-gramC,CountC,N-gramD,CountD,...
...
Filename?,Record? Offset,Record? Length,N-gramE,CountE,N-gramF,CountF,...
```

For RAW_TEXT invocations, the default output is shortened, with one line of output corresponding to each individual input file:

```
Filename1,,,N-gramA,CountA,N-gramB,CountB,...
Filename2,,,N-gramC,CountC,N-gramD,CountD,...
...
```

```
FilenameX,,,N-gramE,CountE,N-gramF,CountF,...
```

For both the RECORD and RAW_TEXT modes, a more verbose output format is available, which will include only a single n-gram and associated count per line. However, the more verbose format is not recommended, since the output results data size will be significantly larger, due to the repetition of the filename and record offset/length information on each output line.

The verbose n-gram output format follows for RECORD-based input:

```
Filename1,Record1 Offset,Record1 Length,N-gramA,CountA
Filename1,Record1 Offset,Record1 Length, N-gramB,CountB
...
Filename1,Record2 Offset,Record2 Length,N-gramC,CountC
Filename1,Record2 Offset,Record2 Length,N-gramD,CountD
...
FilenameY,RecordZ Offset,RecordZ Length,N-gramE,CountE
FilenameY,RecordZ Offset,RecordZ Length,N-gramF,CountF
```

The verbose n-gram output format follows for RAW_TEXT operations:

```
Filename1,,,N-gramA,CountA
Filename1,,,N-gramB,CountB
...
Filename2,,,N-gramC,CountC
Filename2,,,N-gramD,CountD
...
FilenameY,,,N-gramE,CountE
FilenameY,,,N-gramF,CountF
```

For all output formats per filename (or filename/record), results are output in sorted order by count, and for situations where the count is identical, secondarily by the n-gram.

3.3.7.4 Example Program

To illustrate n-gram primitive functionality, we will use `input.txt`, which contains the following raw text:

```
This is a set of test input data.  That makes this a rather short test, but it will be
sufficient to highlight ngram operations.
```

To generate a set of n-gram output for this input in case-insensitive fashion with `N=1` (equivalent to a standard single-term term frequency operation), we could use the following user program. Note that the default value for `N` is 1, but we have chosen to specify it in the following program for completeness:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "input.txt");

    rx_data_set_t results = rx_ds_ngram(
        input_data_set,
```

```
        "d.txt",
        "(NGRAM(RAW_TEXT, CASE=\"false\", N=\"1\"))",
        NULL);

    if (rx_ds_has_error_occurred(results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will generate the default terse-mode output data file `d.txt`, and its contents will be as follows, with the n-grams appearing in ASCII order:

```
input.txt,,,a,2,test,2,this,2,be,1,but,1,data,1,highlight,1,input,1,is,1,it,1,makes,1,ngr
am,1,of,1,operations,1,rather,1,set,1,short,1,sufficient,1,that,1,to,1,will,1
```

If the configuration string contained in the above program had instead been specified as `(NGRAM(RAW_TEXT, CASE=\"false\", VERBOSE_OUTPUT=\"true\"))`, then the output file `d.txt` would have been created in verbose mode, and it would consist in a single output line for every individual n-gram per input file, sorted first by frequency and secondly by ASCII order of the n-gram:

```
input.txt,,,a,2
input.txt,,,test,2
input.txt,,,this,2
input.txt,,,be,1
input.txt,,,but,1
input.txt,,,data,1
input.txt,,,highlight,1
input.txt,,,input,1
input.txt,,,is,1
input.txt,,,it,1
input.txt,,,makes,1
input.txt,,,ngram,1
input.txt,,,of,1
input.txt,,,operations,1
input.txt,,,rather,1
input.txt,,,set,1
input.txt,,,short,1
input.txt,,,sufficient,1
input.txt,,,that,1
input.txt,,,to,1
input.txt,,,will,1
```

Note that the repetitive nature of the input filename makes the verbose output format will consume significantly more output disk space, so the default terse mode is recommended.

N-grams of length 2 or 3 are also possible by appending option `N="2"` or `N="3"`, respectively, to the query clause. Example output in each of these cases in terse mode would be:

```
input.txt,,,a rather,1,a set,1,be sufficient,1,but it,1,data that,1,highlight
ngram,1,input data,1,is a,1,it will,1,makes this,1,ngram operations,1,of test,1,rather
short,1,set of,1,short test,1,sufficient to,1,test but,1,test input,1,that makes,1,this
a,1,this is,1,to highlight,1,will be,1
```

```
input.txt,,,a rather short,1,a set of,1,be sufficient to,1,but it will,1,data that
makes,1,highlight ngram operations,1,input data that,1,is a set,1,it will be,1,makes this
a,1,of test input,1,rather short test,1,set of test,1,short test but,1,sufficient to
highlight,1,test but it,1,test input data,1,that makes this,1,this a rather,1,this is
a,1,to highlight ngram,1,will be sufficient,1
```

3.3.8 Document Similarity Primitive Family

Leveraging the n-gram primitive capability, the document similarity (DOCSIM) family of operations consists of powerful text-based machine learning training and inference algorithms to rapidly generate a reusable model which allows the user to identify documents (or records) that are similar to a seed document (or record) across arbitrarily large source corpus data with repeatable results which include relative similarity measures that can be used as confidence factors.

Document similarity is most useful when operating against relatively rich data sets (for example, hundreds of thousands or millions of documents that each have at least a couple of pages of interesting content). Examples of these datasets would resemble what you'd likely find in a large data lake.

It is typically the case that existing systems struggle performing similarity matching across such a wide array of input in reasonable timeframes given typical CPU and system memory constraints. In many cases, they impose unacceptable limitations on the input corpus, which effectively renders relative similarity significantly less useful.

Our text-based machine-learning document similarity engines solve those problems, and our API provides full access to the document similarity engine's capability set.

Since the document similarity operation utilizes machine learning principles, the system must be trained before a typical document similarity operation can be requested. Training is accomplished using the DOCSIM_TRAINING primitive. Before discussing the primitive in more detail, it will help to generally understand how the training and similarity algorithms work.

At a high level, the principles for text-based machine learning are similar in many ways to how we train both human beings and machines for things like image recognition. For example, we might show a child several images (perhaps of a cat, a dog, a boat, a tree, and so on) and tell the child what they are, and then show a child many other images and ask them to identify them, or to pick out images that are similar to another one that they have already learned. These are common problems we allow children to solve with a variety of teaching mechanisms, including tile matching games, or even flash cards. Children are typically very good at this, and get better over time, extending upon what is effectively a model that is perpetually stored in the human brain. Extending that to machines, we might run many known images through a training algorithm so that the machine can learn what is in those images, storing that in a model, and then ask the machine those same questions about other images that they weren't trained on, referring back to the model that described how it was trained.

The same general principle applies to text-based machine learning afforded by the document similarity primitives: many documents are fed into the machine and it runs a set of algorithms to learn about those documents, creating a suitable model that describes the training, and allows the machine to refer back to that model whenever needed. Then, new

documents can be pointed at the machine, allowing the machine to recognize similarities in new data with respect to prior data that it was trained upon.

There is one caveat to that analogy, however: the image recognition example given above is an example in supervised learning, whereas our text-based document similarity paradigm is built using an unsupervised process. This important distinction is critical because it means that our document similarity engine can run on arbitrary amounts of data without the system being told anything about the data ahead of time.

Therefore, it will "learn" on its own. Although this might sound like black magic, it is not. The capability is rooted in a set of well-known scientific and mathematical constructs intelligently applied in our API implementation to be highly performant given our novel optimized parallelization frameworks.

Effectively, the text-based training algorithms rely on several complex mathematical functions, but as a foundation, they employ the straightforward concept of term frequency extended to n-grams. Term frequency, in its simplest form, is a collection of unique terms in a document and the number of times each of those terms appear. An n-gram is simply an extension of that concept, where an n-gram is a contiguous sequence of `n` items from a given sequence of text. By default, our document similarity algorithms use a value of n=3, yielding trigrams, which typically yield the best overall accuracy of similarity results within a given performance objective for large input corpuses with strong signals (ie, rich content). Smaller values n=1 (unigrams) and n=2 (bigrams) are often useful for small document corpuses or small records given typically weaker frequency-based signals for larger values of 'n' amongst such datasets.

Once the n-gram tables are built for each document or record in the source corpus, internal document frequency tables are calculated. These feed a very important step called the inverse document frequency step (often referred to in machine learning literature as TF-IDF, representing term frequency inverse document frequency). This step can be highly parameterized to adjust for a variety of real-world use cases. The intent of this step is to determine which terms, in context to the rest of the terms in the document, provide the most meaningful information across the entire input corpus dataset. Generating the TF-IDF tables are typically a very compute-, memory-, and I/O-intensive process, making it amenable to the types of acceleration that the API primitives excel at, which ensures that the model training times are extremely fast.

The simple yet powerful concept of inverse document frequency is precisely the reason why more training data is almost always better. It is analogous in many ways to human learning constructs, in that more data is usually better. For example, it is typically the case that the reader is better educated after reading two books on a subject versus a single book on a subject.

Once the TF-IDF tables are complete and generated, they are combined with n-gram information across the input corpus to generate a concise output model that describes the training in full, and typically only needs to be created once for a given dataset for a given parameterization set. This model file can then feed downstream similarity requests, allowing for extremely rapid similarity assessments. The output of a similarity operation becomes a list of similar documents or records and their respective confidence factors. And that completes the "unsupervised" learning construct, and was arrived at by intelligently correlating sets of n-grams and their frequencies of appearance across arbitrary amounts of source data.

Use this syntax with the document similarity primitive family:

```
rx_data_set_t rx_ds_document_similarity (
    const char*             configuration_string,
    const rx_data_set_t     data_set,
    const char*             results_file,
    void*                   (*percentage_callback)(uint8_t));
```

- A dataset of type `rx_data_set_t` is the return value, representing the results of the operation. This can be used for subsequent library calls as needed, such as for statistics gathering. The returned dataset should be deleted by the user via a call to `rx_ds_delete()` when it is no longer needed in order to release any associated system resources.
- `configuration_string` is the string used to configure the primitive request, which varies depending on the type of operation requested. It is described in more detail in subsequent sections.
- `data_set` is the input dataset of type `rx_data_set_t`.
- `results_file` is the output data file to be created. It may be `NULL` if no results file is desired.
- `percentage_callback` references a user-defined callback function for handling incoming percentage completion messages. For example, a user could provide a callback function for printing the current status of the execution to standard output. It may be `NULL` if no progress reporting is desired.

3.3.8.1 DOCSIM_TRAINING

As described earlier, the document similarity training primitive generates a model for use with a downstream document similarity request. The `configuration_string` passed to the `rx_ds_document_similarity` function uses a variant of the general expression described in section 3.3.1:

```
(DOCSIM_TRAINING([parameters]))
```

The DOCSIM_TRAINING primitive takes the following parameters:

- (required) IDF_WEIGHTING
- (required) TF_IDF_WEIGHTING
- (required) MIN_DF
- (required) MAX_DF
- (required) MAX_TERMS
- (optional) N
- (optional) REC_PATH

More detailed descriptions and full syntax information for the parameters listed above can be found in section 3.3.2.

The output data file of the DOCSIM_TRAINING primitive is a model file. That model file is fed to the DOCSIM primitive when invoking the document similarity (DOCSIM) primitive.

3.3.8.2 DOCSIM

As described earlier, the document similarity training primitive generates a model for use with a downstream document similarity request. The `configuration_string` passed to the `rx_ds_document_similarity` function uses a variant of the general expression described in section 3.3.1:

```
(DOCSIM("/path/to/model.extension", [parameters]))
```

The first parameter to the DOCSIM primitive is the full path to the model file generated from a prior DOCSIM_TRAINING run. In addition, DOCSIM requires that certain options be specified. More detailed descriptions and full syntax information for the parameters and options can be found in section 3.3.2, including:

- (required) SIMILARITY_TYPE
- (optional) MAX_FILE_MATCHES
- (required unless MAX_FILE_MATCHES is used) MIN_DISTANCE
- (required unless MAX_FILE_MATCHES is used) MAX_DISTANCE

- (optional) REC_PATH

3.3.8.2.1 Results: DOCSIM Output File Format

The output data file of the DOCSIM primitive differs from several of the other primitive types.

It remains a comma-separated value file, but each line of the output file contains four fields, each separated by a comma:

- Field 1: The first field is the name of a file that was similar
- Field 2: The second field is empty for full document similarity. In record modes, it becomes the offset in bytes into the file where the similar record begins.
- Field 3: The third field is empty for full document similarity. In record modes, it becomes the length in bytes of the similar record.
- Field 4: The fourth and final field is the calculated similarity distance, which can be used as a relative confidence factor. The output file is sorted most similar to least similar.

A set of example output is shown below in section 3.3.8.3.

3.3.8.3 Example Program

Purely for illustration purposes, we will simplify the large data lake scenario, and instead use a very small and not very rich data corpus via the following manageable set of input data across just eight documents. Note that the principles described here apply to input data corpus of practically any size:

```
./passengers.txt:
--
Name,DoB,Phone,Notes
Hannibal Smith, 10-01-1928,011-310-555-1212,A-team, baby, A-team!
DR. Thomas Magnum, 01-29-1945,310-555-2323,Magnum PI himself.
Steve McGarett, 12-30-1920,310-555-3434,The new Hawaii Five-O.
Michael Knight, 08-17-1952,011-310-555-4545,"Knight Industries Two Thousand.  Kitt.  He's
the driver, sort of."
Stringfellow Hawke, 08-15-1944,310-555-5656,Fictional character who happens to be the
chief test pilot during the development of Airwolf.
Sonny Crockett, 12-14-1949,310-555-6767,Mr. Miami Vice himself.  Rico Tubbs was his
partner.
Michelle Jones,07-12-1959,310-555-1213,Ms. Jones likes to spell her name many different
ways.
Mishelle Jones,07-12-1959,310-555-1213,Ms. Jones proves that she likes to spell her first
name differently.
Michele Jones,07-12-1959,310-555-1213,Ms. Jones once again shows that she doesn't have
command over the spelling of her first name.
T,01-12-1989,310-555-9876,This guy goes by the name 'T'.  No more.  No less.
DJ,04-25-1985,310-555-3425,I wonder if DJ is just this guy's name or his profession?

./passengers-simple.txt:
--
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767

./passengers-line.txt:
```

```
--
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 01:23:45, 10.0.0.23, 1abc::11, A-
team, baby, A-team!
DR. Thomas Magnum, 01-29-1945, 310-555-2323, $925.00, 09:30:01, 10.10.0.41,
1abb:2:3:4:5:6:7:8, Magnum PI himself.
Steve McGarett, 12-30-1920, 310-555-3434, $1207.00, 12:00:00, 10.0.2.89, 1a0:2b:3c::4d,
The new Hawaii Five-O.
Michael Knight, 08-17-1952, 310-555-4545, $422.00, 12:00:01, 192.168.0.22, ::3ca, "Knight
Industries Two Thousand.  Kitt.  He's the driver, sort of."
Stringfellow Hawke, 08-15-1944, 310-555-5656, $726.00, 23:00:00:01, 192.168.1.90,
1abc:b59:14::21, Fictional character who happens to be the chief test pilot during the
development of Airwolf.
Sonny Crockett, 12-14-1949, 310-555-6767, $214.00, 22:59:59, 10.0.0.241, 1ab0:13:c2::22,
Mr. Miami Vice himself.  Rico Tubbs was his partner.

./passengers-simple-time.txt:
--
Hannibal Smith, 10-01-1928, 310-555-1212, 10:45:00
DR. Thomas Magnum, 01-29-1945, 310-555-2323, 10:12:00
Steve McGarett, 12-30-1920, 310-555-3434, 17:42:00
Michael Knight, 08-17-1952, 310-555-4545, 15:30:00
Stringfellow Hawke, 08-15-1944, 310-555-5656, 18:25:22
Sonny Crockett, 12-14-1949, 310-555-6767, 09:13:00

./passengers-ipv4.txt:
--
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 10.0.0.23
DR. Thomas Magnum, 01-29-1945, 310-555-2323, $925.00, 10.10.0.41
Steve McGarett, 12-30-1920, 310-555-3434, $1207.00, 10.0.2.89
Michael Knight, 08-17-1952, 310-555-4545, $422.00, 192.168.0.22
Stringfellow Hawke, 08-15-1944, 310-555-5656, $726.00, 192.168.1.90
Sonny Crockett, 12-14-1949, 310-555-6767, $214.00, 10.0.0.241

./passengers-ipv6.txt:
--
Hannibal Smith, 10-01-1928, 310-555-1212, $1436.00, 1abc::11
DR. Thomas Magnum, 01-29-1945, 310-555-2323, $925.00, 1abb:2:3:4:5:6:7:8
Steve McGarett, 12-30-1920, 310-555-3434, $1207.00, 1a0:2b:3c::4d
Michael Knight, 08-17-1952, 310-555-4545, $422.00, ::3ca
Stringfellow Hawke, 08-15-1944, 310-555-5656, $726.00, 1abc:b59:14::21
Sonny Crockett, 12-14-1949, 310-555-6767, $214.00, 1ab0:13:c2::22

./passengers-line-pound.txt:
--
Hannibal Smith, 10-01-1928, 310-555-1212, #1436.00, 01:23:45, 10.0.0.23, 1abc::11, A-
team, baby, A-team!
DR. Thomas Magnum, 01-29-1945, 310-555-2323, #925.00, 09:30:01, 10.10.0.41,
1abb:2:3:4:5:6:7:8, Magnum PI himself.
Steve McGarett, 12-30-1920, 310-555-3434, #1207.00, 12:00:00, 10.0.2.89, 1a0:2b:3c::4d,
The new Hawaii Five-O.
Michael Knight, 08-17-1952, 310-555-4545, #422.00, 12:00:01, 192.168.0.22, ::3ca, "Knight
Industries Two Thousand.  Kitt.  He's the driver, sort of."
Stringfellow Hawke, 08-15-1944, 310-555-5656, #726.00, 23:00:00:01, 192.168.1.90,
1abc:b59:14::21, Fictional character who happens to be the chief test pilot during the
development of Airwolf.
Sonny Crockett, 12-14-1949, 310-555-6767, #214.00, 22:59:59, 10.0.0.241, 1ab0:13:c2::22,
Mr.  Miami Vice himself.  Rico Tubbs was his partner.

./passnum.txt:
--
```

```
Name,DoB,Phone,Notes
Hannibal Smith, 10-01-1928,310-555-1212,A-team, baby, A-team!, $1436.00
DR. Thomas Magnum, 01-29-1945,310-555-2323,Magnum PI himself., $925.00
Steve McGarett, 12-30-1920,310-555-3434,The new Hawaii Five-O., $1207.00
Michael Knight, 08-17-1952,310-555-4545,"Knight Industries Two Thousand.  Kitt.  He's the
driver, sort of.", $422.00
Stringfellow Hawke, 08-15-1944,310-555-5656,Fictional character who happens to be the
chief test pilot during the development of Airwolf., $726.00
Sonny Crockett, 12-14-1949,310-555-6767,Mr. Miami Vice himself.  Rico Tubbs was his
partner., $214.00
Michelle Jones,07-12-1959,310-555-1213,Ms. Jones likes to spell her name many different
ways., $213.99
Mishelle Jones,07-12-1959,310-555-1213,Ms. Jones proves that she likes to spell her first
name differently., $213.99
Michele Jones,07-12-1959,310-555-1213,Ms. Jones once again shows that she doesn't have
command over the spelling of her first name., $213.99
T,01-12-1989,310-555-9876,This guy goes by the name 'T'.  No more.  No less., $449.99
DJ,04-25-1985,310-555-3425,I wonder if DJ is just this guy's name or his profession?, $0
```

We will train a model based on that input corpus. After generating the model, we can feed that model to any number of downstream subsequent document similarity requests over time. Although we could use any document we desire to look for similarities against the model, instead we will send one of the documents contained in the corpus. This allows us to easily prove the accuracy, as an accurate model will, of course, provide one exact match in the corpus, plus a collection of other similar documents according to the criteria specified.

To generate a model from our document corpus and then run similarity against the model, we will invoke two primitives. In our example, we will do this within the confines of a single user program. But note that since the initial DOCSIM_TRAINING primitive will generate an output model file and store it to the location specified, subsequent DOCSIM requests can occur at any future time needed when operating against that model file.

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t training_data = rx_ds_create_with_nodes(1);
    rx_ds_add_file(training_data, "my_file_list.txt");

    rx_data_set_t training = rx_ds_document_similarity(
        "(DOCSIM_TRAINING(INPUT_IS_FILELIST=\"true\", IDF_WEIGHTING=\"max\",
TF_IDF_WEIGHTING=\"log\", MIN_DF=\"0.02\", MAX_DF=\"0.80\", MAX_TERMS=\"1000\"))",
        training_data,
        "model.bin",
        NULL);

    if (rx_ds_has_error_occurred(training))
    {
        printf("Error during training: %s\n", rx_ds_get_error_string(training));
        ret = 1;
    }
    else
    {
        rx_data_set_t seed_document = rx_ds_create_with_nodes(1);
        rx_ds_add_file(seed_document, "passengers.txt");
        rx_data_set_t results = rx_ds_document_similarity(
```

```
            "(DOCSIM(\"model.bin\", SIMILARITY_TYPE=\"cosine\", MIN_DISTANCE=\"0.0\",
MAX_DISTANCE=\"1.0\"))",
            seed_document,
            "d.txt",
            NULL);

        if (rx_ds_has_error_occurred(results))
        {
            printf("Error during similarity: %s\n", rx_ds_get_error_string(results));
            ret = 1;
        }

        rx_ds_delete(&seed_document);
        rx_ds_delete(&results);
    }

    rx_ds_delete(&training_data);
    rx_ds_delete(&training);

    return ret;
}
```

Note that in the example code above, we used an input file list construct via the `INPUT_IS_FILELIST` option to avoid having to invoke `rx_ds_add_file` multiple times for each individual filename. For this mode, the contents of the input file `my_file_list.txt` was as follows:

```
passengers.txt
passengers-simple.txt
passengers-line.txt
passengers-simple-time.txt
passengers-ipv4.txt
passengers-ipv6.txt
passengers-line-pound.txt
passnum.txt
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Running this user program will result in two outputs. The first is the model binary `model.bin`, as output from `DOCSIM_TRAINING`. For very small corpus sizes like our example case, model sizes of tens of kilobytes are common. For very large datasets, say hundreds of thousands of documents totaling hundreds of gigabytes in size, the resulting model file is typically on the order of just one or two gigabytes. This is quite different than many model-based paradigms and general indexing solutions where such constructs can approach sizes that are ten times *larger* than the input corpus. This nasty artifact does not occur with our similarity modeling.

The second output file from our program is the output from the `DOCSIM` primitive. It is represented in our example as the output data file `d.txt`. We also chose to limit the output to relative similarity values between 0 and 0.9. The resulting output contents will be as follows. Note the output is comma-separated value (CSV), where the first output line will always consist of column headers. This means that this output file can be readily ingested into any downstream tool that handles standards-based CSV input (such as Microsoft Excel, etc.):

```
filename,fileOffset,recordLength,cosine distance
passengers.txt,,,0.000000000000
```

```
passnum.txt,,,0.406487703323
passengers-line-pound.txt,,,0.879519760609
passengers-line.txt,,,0.879519760609
```

Note that for our raw text example, the fileOffset and recordLength fields are empty. These are populated only when a record-based document similarity operation is chosen using the `REC_PATH` options as described in section 3.3.2.

The ordering of the `DOCSIM` output data file is always most-similar to least-similar, with values closer to 0.0 cosine distance being more similar to the seed document specified. As described in more detail in section 3.3.2 with respect to `SIMILARITY_TYPE`, note that cosine distance is the output value that is leveraged here as opposed to the more mathematically pure cosine similarity value. This allows for simpler integration to a variety of downstream ecosystem tools that the output of the DOCSIM primitive is meant to feed, including a variety of software visualization tools that tend to prefer values (distances) of 0 to represent closest values, making them mathematically agnostic.

Given that definition, in our example, the seed document `passengers.txt` that we provided was also part of the input corpus and was identified as the closest match, with a precisely aligned "0.0" cosine distance value. The algorithm found that corpus file passnum.txt was the next most similar document in the corpus, and so on. A visual inspection of the input data shows this to be true. Note that all results are relative, which is one of the powerful features of cosine distance similarity approaches to text-based document similarity machine learning algorithms. That is, even amongst documents that are all somewhat similar, those that are most similar to a seed document gravitate to the top.

If we had not limited the `MAX_DISTANCE` to 0.9, but instead set it to 1.0, we would receive relative scores for all documents in the input corpus. In our example here, the four documents that did not muster values between 0.0 and 0.9 all happened to be relative scores of 1.0, meaning they are much less similar than the top results. This can be seen by re-running the algorithm using `MAX_DISTANCE` of 1.0 would yield an output results data file that looks like:

```
filename,fileOffset,recordLength,cosine distance
passengers.txt,,,0.000000000000
passnum.txt,,,0.406487703323
passengers-line-pound.txt,,,0.879519760609
passengers-line.txt,,,0.879519760609
passengers-ipv4.txt,,,1.000000000000
passengers-ipv6.txt,,,1.000000000000
passengers-simple-time.txt,,,1.000000000000
passengers-simple.txt,,,1.000000000000
```

3.4 Return Error Information

It is possible to determine whether or not an operation was completed successfully by calling the `rx_ds_has_error_occurred` function on the results dataset that was returned by the operation. If this routine returns true, you may use the `rx_ds_get_error_string` function to retrieve an error string that contains details about the issue and, if available, its cause. Use this syntax to request error details:

```
bool rx_ds_has_error_occurred(const rx_data_set_t data_set);

const char* rx_ds_get_error_string(const rx_data_set_t data_set);
```

- `data_set` is the dataset to obtain information about, such as the `rx_data_set_t` return value of `rx_ds_search()`.

Errors will be reported for invalid user inputs as well as for any internal errors that occur. Note that each example program in this guide handles errors in the recommended fashion, by testing for the condition by invoking `rx_ds_has_error_occurred` and then calling `rx_ds_get_error_string` as needed.

3.5 Return Dataset, Operation Statistics and Aggregations

In addition to being able to calculate statistics and aggregations leveraging the output data and output index files alongside a variety of common linux tools (such as `wc`, `cut`, etc.), some built-in statistics and some aggregations can be obtained for primitive operations using the following API functions:

```
uint64_t rx_ds_get_start_time(const rx_data_set_t data_set);
uint64_t rx_ds_get_execution_duration(const rx_data_set_t data_set);
uint64_t rx_ds_get_fabric_execution_duration(const rx_data_set_t data_set);
uint64_t rx_ds_get_total_bytes_processed(const rx_data_set_t data_set);
uint64_t rx_ds_get_total_matches(const rx_data_set_t data_set);

uint64_t rx_ds_get_total_replacements(const rx_data_set_t data_set);
uint64_t rx_ds_get_total_unique_terms(const rx_data_set_t data_set);
uint64_t rx_ds_get_total_documents_processed(const rx_data_set_t data_set);
uint64_t rx_ds_get_total_similar_documents(const rx_data_set_t data_set);
uint32_t rx_ds_get_max_terms_per_document(const rx_data_set_t data_set);
```

- `data_set` is the dataset to obtain information about, such as the `rx_data_set_t` return value of `rx_ds_search()`.

Below is a table that summarizes the different types of operation statistics:

Function	Description	Units
`rx_ds_get_start_time`	Time at which the operation began, represented as Epoch time. Epoch time is defined as the number of milliseconds that have elapsed since 00:00:00 UTC Thursday, January 1, 1970.	milliseconds
`rx_ds_get_execution_duration`	Amount of time taken to complete the specified operation.	milliseconds
`rx_ds_get_fabric_execution_duration`	Amount of time taken to complete the specified operation, measured at the computation layer and not including software overhead.	milliseconds
`rx_ds_get_total_bytes_processed`	Amount of input data processed.	bytes
`rx_ds_get_total_matches`	Number of matches found for search-based primitives.	N/A
`rx_ds_get_total_replacements`	Number of replacements that were made to the input corpus for a search and replace operation.	N/A
`rx_ds_get_total_unique_terms`	Number of unique n-grams found for n-gram and term frequency primitives.	N/A
`rx_ds_get_total_documents_processed`	Number of documents processed in a document similarity primitive family operation.	N/A
`rx_ds_get_total_similar_documents`	Number of similar documents found for a document similarity operation.	N/A
`rx_ds_get_max_terms_per_document`	Maximum number of terms used for a document similarity primitive family operation.	N/A

In addition, the following functions are also available for obtaining information about a dataset:

```
uint8_t rx_ds_get_number_of_processing_nodes(const rx_data_set_t data_set);
uint32_t rx_ds_get_number_of_files(const rx_data_set_t data_set);
char** rx_ds_get_file_list(const rx_data_set_t data_set);
```

Below is a table that summarizes these functions:

Function	Description	Units
`rx_ds_get_number_of_processing_nodes`	Returns the requested number of processing nodes that may have been physically allocated to the dataset	N/A
`rx_ds_get_number_of_files`	Returns the number of files in the dataset	N/A
`rx_ds_get_file_list`	Returns a list of the files in the dataset	N/A

3.6 XML

XML queries must include the full desired hierarchy to include the starting record tag. For an XML file containing a set of records for passengers where we are interested in a phone number attached to a passenger, a schema may exist where each record begins with a `<passenger>` tag and internal to the record may be a `<phone>` tag. A query against that field might then resemble: `'(RECORD.passenger.phone CONTAINS EXACT("310-555-5656"))'`.

XML data is required to be XML standards-compliant. The API supports the extensible markup language (XML) 1.0 (fifth edition) specification, as documented at: https://www.w3.org/TR/2008/REC-xml-20081126/. Note that data that is not standards-compliant will result in an error message, describing the non-compliant event to the extent possible. This can be an effective means of identifying and subsequently cleaning any 'dirty' input data.

API support includes the following five valid XML escape characters as show below:

	Physical Character	Escaped Form
Ampersand	`&`	`&`
Less-than	`<`	`<`
Greater-than	`>`	`>`
Quotes	`"`	`&quot;`
Apostrophe	`'`	`&apos;`

XML escaping is handled natively amongst all pertinent primitives. For example, when a user inputs a query, they are not required to know that they are operating against XML data or its rules, and they should use the standard physical characters. The internal API control frameworks will automatically adjust values appropriately to adhere to XML specifications.

API primitives can auto-detect XML file types when the RECORD keyword is used appropriately during an operation, but the user always has the explicit option to specify XML by using the XRECORD keyword instead.

3.6.1 Example Program

Here's an input file `passengers.xml` that is composed of `passenger` records, each with three fields: `name`, `dob`, and `phone`:

```
<passenger>
      <name>  Hannibal Smith        </name>
      <dob>   10-01-1928            </dob>
      <phone> 011-310-555-1212      </phone>
```

```
</passenger>
<passenger>
      <name>  DR. Thomas Magnum     </name>
      <dob>   01-29-1945            </dob>
      <phone> 011-310-555-2323      </phone>
</passenger>
<passenger>
      <name>  Steve McGarett        </name>
      <dob>   12-30-1920            </dob>
      <phone> 011-310-555-3434      </phone>
</passenger>
<passenger>
      <name>  Michael Knight        </name>
      <dob>   08-17-1952            </dob>
      <phone> 011-310-555-4545      </phone>
</passenger>
<passenger>
      <name>  Stringfellow Hawke    </name>
      <dob>   08-15-1944            </dob>
      <phone> 011-310-555-5656      </phone>
</passenger>
<passenger>
      <name>  Sonny Crockett        </name>
      <dob>   12-14-1949            </dob>
      <phone> 011-310-555-6767      </phone>
</passenger>
<passenger>
      <name>  MR. T Magnum          </name>
      <dob>   01-29-1946            </dob>
      <phone> 011-310-555-7878      </phone>
</passenger>
```

Here is an example user program test.c to search each record in the file for `310-555-5656` appearing in any field:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers.xml");

    rx_data_set_t search_results = rx_ds_search(
            input_data_set,
            "d.xml",
            "(RECORD.passenger CONTAINS EXACT(\"310-555-5656\"))",
            "\n\n",
            "i.txt",
            NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);
```

```
    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Both data results and index results will be produced by this user program. The program is configured to store the output data results in `d.xml`:

```
<passenger>
        <name>  Stringfellow Hawke      </name>
        <dob>   08-15-1944              </dob>
        <phone> 011-310-555-5656        </phone>
</passenger>
```

The program is configured to store the output index results in `i.txt`:

```
passengers.xml,502,127,0
```

3.7 JSON

This API supports the full complement of JSON objects, arrays, values, strings and numbers, according to the definitions as published in standard ECMA-404 2nd Edition, December 2017. The standard can be directly downloaded from the managing body at http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, and is also referenced at the popular website `json.org`.

The API primitives can auto-detect JSON file types when the `RECORD` keyword is used appropriately during an operation, but the user always has the explicit option to specify JSON by using the `JRECORD` keyword instead.

3.7.1 Example Program

Consider the following simple example JSON input file shown below called `widgets.json`:

```
{
        "name":"Widget A",
        "description":"This is widget A.  It is really cool!",
        "pricing":[
                {"price":3.09},
                {"price":3.99}
        ]
}
{
        "name":"Widget B",
        "description":"This is widget B.  It isn't nearly as cool as widget A, but some
people like it since it is cheaper.",
        "pricing":[
                {"price":1.09},
                {"price":1.99}
        ]
}
```

```
{
        "name":"Widget C",
        "description":"This is widget C.  It is super cool, but it is also the most
expensive widget.  Sometimes it goes on sale for a really nice price.",
        "pricing":[
                {"price":9.99},
                {"saleprice":3.99}
        ]
}
```

Searching JSON formatted data is very similar to other record-based searches. The full path to the field must be specified in the query, and this includes any array hierarchies. For example, to search the `saleprice` field in the embedded array in the example `widgets.json` file above for an exact value of `3.99`, the full path to the field (i.e. "`RECORD.pricing.[].saleprice`") must be specified.

Here is an example user program implementing that search using a single processing node, with custom progress reporting enabled:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "widgets.json");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.json",
        "(RECORD.pricing.[].saleprice EQUALS EXACT(\"3.99\"))",
        NULL,
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

This user program will create a data results file called `d.json`. Data results files will return the matched record(s), in this case a single JSON object. The output file therefore contains:

```
{
        "name":"Widget C",
        "description":"This is widget C.  It is super cool, but it is also the most
expensive widget.  Sometimes it goes on sale for a really nice price.",
        "pricing":[
                {"price":9.99},
                {"saleprice":3.99}
        ]
}
```

In addition, the program requested an output index file `i.txt`. Its contents will be:

```
widgets.json,437,283,0
```

3.8 CSV

As applicable for CSV functionality, the API supports the RFC 4180 October 2005 grammar, which is publicly available for download at https://www.ietf.org/rfc/rfc4180.txt.

The API implementation extends RFC 4180-defined character escaping to apply not just to comma-separated constructs, but also for arbitrary character-separated (such as tab) value functionality, which our API collectively refers to as CSV (where the 'C' stands for character).

When a user requests an operation, they are not required to know that they are operating against CSV data or its rules, and they should use the standard physical characters. The API primitives will automatically adjust values on the fly to match CSV requirements for both input and output operations when operating against CSV datasets.

In addition, the API primitives can auto-detect CSV file types when the `RECORD` keyword is used appropriately during an operation, but the user always has the explicit option to specify CSV by using the `CRECORD` keyword instead.

By default, the field delimiter and the record delimiter are set to be a comma ',' and a line feed '`\n`', respectively, which enables comma-separated value support. To adjust these values, use the `FIELD_DELIMITER` and `RECORD_DELIMITER` query options, respectively, as described in section 3.3.2. Changing these values allows you to use the CSV mechanisms against other character-separated filetypes, such as tab-separated value (often referred to as TSV) files, for example.

Query clauses operating against CSV input can utilize named CSV column headers by surrounding the field name in double quotes, or can reference the column by number, where the leftmost column is column 1. These mechanisms are described in more detail in the example below.

When a column name is referenced anywhere in a query clause, such as in our example below, the first line of the input data is considered the column header line, and by default it will be prepended to the requested output data file. This also holds true when column headers and column indexes are mixed as part of a compound query. To override this functionality, the query option `OUTPUT_HEADER` can be used, as describe in section 3.3.2.

On the other hand, if column index numbers had been exclusively used in the query, then the first line of the input data would have been considered data, and not a column header, and no column header line would have been prepended to the output data file.

3.8.1 Example Program

Consider the following simple character-separated value file stored in `passengers.csv`, where the column separator in this example is a comma. In our example, the file has an initial header row that lists the column names:

```
name,dob,phone
Hannibal Smith,10-01-1928,011-310-555-1212
DR. Thomas Magnum,01-29-1945,011-310-555-2323
Steve McGarett,12-30-1920,011-310-555-3434
Michael Knight,08-17-1952,011-310-555-4545
Stringfellow Hawke,08-15-1944,011-310-555-5656
Sonny Crockett,12-14-1949,011-310-555-6767
MR. T Magnum,01-29-1946,011-310-555-7878
```

This input file is composed of `passenger` records, each with three fields: `name`, `dob`, and `phone`, which are listed in the first row, which is common to many real-world CSV datasets. This allows the various fields to be specified by field name if desired instead of using a column number.

Here is an example user program to search for CSV records with a `phone` field that contains a part of a phone number, `310-555-56`. Given the input corpus, one result will be returned by this example program:

```
#include <stdio.h>
#include <libryftx.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "passengers.csv");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.csv",
        "(RECORD.\"phone\" CONTAINS EXACT(\"310-555-56\"))",
        NULL,
        "i.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

Note that the query string is the third parameter to the `rx_ds_search()` call. It used a named column header. Columns can also be specified by their numerical index, where the leftmost column is 1. This means that the query could have used a numeric column index of 3 since we know from the header row of this CSV file that `phone` is the 3rd column in the specified `passengers.csv` dataset. The record could therefore have been specified as "`(RECORD.3 CONTAINS EXACT(\"310-555-56\"))`" and the same results would be generated. When column headers are used, they must be surrounded by double quotes. When column indexes are used, quotes are not used. Using column index numbers allows for flexibility when specifying CSV fields in arbitrary CSV files, whether or not a column header row exists in the source data.

Since no specific field delimiter character was specified in the query clause via the `FIELD_DELIMITER` option, the default value of a comma was used.

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

Both data results and index results will be produced by this user program. The program is configured to store the output data results in `d.csv` and will include the column header line as pulled from the source data since a valid header name was used in the query:

```
name,dob,phone
Stringfellow Hawke,08-15-1944,011-310-555-5656
```

The program is configured to store the output index results in `i.txt`:

```
passengers.csv,190,47,0
```

4 Large Input Example with No ETL and No Indexing Required

With the exception of the PCAP example, most of the examples in this Guide are shown to operate on very small data sets, so that it is easy for you to repeat the operations by cut-and-pasting the data and programs to learn the API at your own pace. In this section, we deviate from that rationale once more, and provide an example leveraging a very large dataset organized as many independent JSON files on standard linux file systems.

We will operate against a set of partial twitter media collected over the span of one month in 2015 and two months in 2017. The input corpus that we will use is approximately 23GB in size, and contains over 5 million tweets. Obviously, given the size of the corpus, it is not possible to show the contents in-line in this Guide. For reference, they are organized in a set of directories, one for each of the 93 days that were tracked, in a pattern that can be globbed as: `data/*/*json`. These are just standard, regular twitter JSON-based data files, and nothing special was done to them.

We want to know how many of the tweets have case-insensitive `texas` in the tweet's text field. Since we just want the count and don't care about generating an output data or output index file, the following program will achieve what we want:

```
#include <stdio.h>
#include <libryftx.h>
#include <inttypes.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "data/*/*json");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d1.json",
        "(RECORD.text CONTAINS EXACT(\"texas\", CASE=\"false\"))",
        NULL,
        "i1.txt",
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }
    else
    {
        uint64_t search_num_matches = rx_ds_get_total_matches(search_results);
        printf("Total matches containing case-insensitive \"texas\": %"PRIu64"\n",
search_num_matches);
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

An example compilation line using standard `gcc`, and assuming a source file name `test.c` and a target executable named `test` would be:

```
gcc test.c -o test -lryftx
```

The program also generates the following output to `stdout` (or `stderr` as the case may be). In addition, a portion of the result of the prefaced `time` operation is shown for clarity:

```
Total matches containing case-insensitive "texas": 7458

real    0m11.649s
```

Thus, it took only a shade under 12 seconds to perform a structured query against this entire 23GB corpus of over 5 million tweets spread across JSON files in 93 directories. The data was not pre-cached. It underwent zero ETL operations. It was completely unindexed. The API implementation ran it using a single server with a single CPU socket leveraging all available cores in true parallel fashion, without the aid of any external hardware acceleration. This is an excellent example of the kind of extraordinary performance that can be achieved using the BlackLynx parallel processing framework utilizing simple API primitives to tackle a variety of complex problems in hardware- and platform-agnostic fashion, using portable code constructs.

But now let's take this example one step further.

What if we need to know the number of tweets mentioning `texas` but only those tweets that were known to be generated in the state of Texas, not as some system or corporation or commercial mapping entity might describe Texas, but instead the official geographic coordinates of Texas as published by the United States Geological Service (USGS), by way of the US Census Bureau (USCB). That is, the official set of longitude/latitude points that represent the set of polygons that define the state of Texas.

The following program will do exactly that, first executing the same primitive that we ran above, but this time creating an output data file that then feeds a second point-in-polygon (PIP) primitive, all within the same program. This highlights the power of chaining primitives together to solve interesting problems.

The PIP primitive will be told to leverage a set of files that were separately pulled from the USCB that represent the longitude/latitude coordinates as arbitrary polygon vertices that officially define Texas according to the United States government. In this example, we simplify this using a file `/tmp/Texas.txt` that itself is a list of those specific vertex files, one per line, of the various polygons described by longitude/latitude sets that together make up the state of Texas, including outlying landmass attributable to Texas in the Gulf of Mexico.

The program becomes:

```
#include <stdio.h>
#include <libryftx.h>
#include <inttypes.h>

int main( void )
{
    int ret = 0;

    rx_data_set_t input_data_set = rx_ds_create_with_nodes(1);
    rx_ds_add_file(input_data_set, "data/*/*json");

    rx_data_set_t search_results = rx_ds_search(
        input_data_set,
        "d.json",
        "(RECORD.text CONTAINS EXACT(\"texas\", CASE=\"false\"))",
```

```
        NULL,
        NULL,
        NULL);

    if (rx_ds_has_error_occurred(search_results))
    {
        printf("Error during search: %s\n", rx_ds_get_error_string(search_results));
        ret = 1;
    }
    else
    {
        uint64_t search_num_matches = rx_ds_get_total_matches(search_results);
        printf("Total matches containing case-insensitive \"texas\": %"PRIu64"\n",
search_num_matches);

        rx_data_set_t pip_dataset = rx_ds_create_with_nodes(1);
        rx_ds_add_file(pip_dataset, "d.json");
        rx_data_set_t pip_results = rx_ds_search(
            pip_dataset,
            "d2.json",
            "(RECORD.coordinates.coordinates CONTAINS PIP(VERTEX_FILE=\"/tmp/Texas.txt\",
VERTEX_FILE_IS_FILELIST=\"true\"))",
            NULL,
            "i2.txt",
            NULL);

        if (rx_ds_has_error_occurred(pip_results))
        {
            printf("Error during PIP: %s\n", rx_ds_get_error_string(pip_results));
            ret = 1;
        }
        else
        {
            search_num_matches = rx_ds_get_total_matches(pip_results);
            printf("Total matches containing case-insensitive \"texas\" in the state of
Texas: %"PRIu64"\n", search_num_matches);
        }

        rx_ds_delete(&pip_dataset);
        rx_ds_delete(&pip_results);
    }

    rx_ds_delete(&input_data_set);
    rx_ds_delete(&search_results);

    return ret;
}
```

How many longitude/latitude points make up the state of Texas given this definition? Using the USGS corpus it is trivial to run a wc operation against them to count the lines:

```
$ cat /tmp/Texas.txt
/tmp/Texas-000.txt
/tmp/Texas-001.txt
/tmp/Texas-002.txt
/tmp/Texas-003.txt
/tmp/Texas-004.txt
/tmp/Texas-005.txt
/tmp/Texas-006.txt
/tmp/Texas-007.txt
/tmp/Texas-008.txt
/tmp/Texas-009.txt
```

```
/tmp/Texas-010.txt
/tmp/Texas-011.txt
/tmp/Texas-012.txt
/tmp/Texas-013.txt
/tmp/Texas-014.txt
/tmp/Texas-015.txt
/tmp/Texas-016.txt
/tmp/Texas-017.txt
/tmp/Texas-018.txt
/tmp/Texas-019.txt
/tmp/Texas-020.txt
/tmp/Texas-021.txt
/tmp/Texas-022.txt
/tmp/Texas-023.txt
/tmp/Texas-024.txt
/tmp/Texas-025.txt
/tmp/Texas-026.txt
/tmp/Texas-027.txt
/tmp/Texas-028.txt
/tmp/Texas-029.txt
/tmp/Texas-030.txt

$ wc -l /tmp/Texas-* | grep total
 12969 total
```

12,969 longitude/latitude pairs make up the definition of the state of Texas that we used. That's a lot of points, and obviously represents a variety of disjoint land mass constructs that make up the state of Texas. You might think that this operation is going to take a very long time when we execute this program, given the massive amount of mathematical operations that would seem to need to occur to bound the results to such a complex set of disjoint polygons with over ten thousand edges.

Let's see how long it takes.

We compile this program the same way as the previous program, except this program was named `test2.c`:

```
gcc test2.c -o test2 -lryftx
```

The program was run prefaced with the linux `time` command like this:

```
$ time ./test2
```

The program generates both an output data file `d2.json` and an output index file `i2.txt`:

```
$ ls -al *2*
-rw-rw-r--  1 user group 1621585 Aug 27 13:28 d2.json
-rw-rw-r--  1 user group   10910 Aug 27 13:28 i2.txt
```

The program also generates the following output to `stdout` (or `stderr` as the case may be). In addition, a portion of the result of the prefaced `time` operation is shown for clarity:

```
Total matches containing case-insensitive "texas": 7458
Total matches containing case-insensitive "texas" in the state of Texas: 498

real    0m11.973s
```

The complete two-stage chained operation set took only a few hundred milliseconds longer than the prior example which used only the first-stage, and we have our desired answer(s). There were 7458 total tweets across the corpus containing "texas" in case-insensitive fashion, which is of course consistent with the earlier example's result. And now we also know that of those results, 498 originated from within the official geographic bounds of the state of Texas.

Once again, just like in the first example, everything was done without performing any ETL operations and without using any pre-cached indexing data.

The entire set of primitives provided by the API can be used and chained together in a nearly infinite number of ways to solve difficult problems against any data source quickly and efficiently without requiring any upfront ETL or indexing operations. The parallel frameworks that implement the API will always make the best use of the compute resources available, including hardware acceleration resources as appropriate, and will automatically and seamlessly schedule them internally to get the best performance, easily operating on the smallest of datasets, to large datasets in the tens to hundreds of gigabytes range, to very large datasets in the hundreds of terabytes range, or to massive datasets in the multiple petabyte range, by way of both scale-up and scale-out architectures. And most importantly, all of these facets are expertly abstracted by the API implementation, so that end users can focus on business results instead of platform management.

5 Java Bindings

5.1 Java Bindings API

The Open API Java bindings are distributed in JAR files.

On Ubuntu 16.04 LTS, Java 8 is the system default. The JAR files will therefore appear at default location:

- `/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/ext/libryftx-java.jar`
- `/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/ext/libryftx_pcap-java.jar`

On older Ubuntu 14.04 LTS systems, Java 7 is the system default. The JAR files will therefore appear at default location:

- `/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/ext/libryftx-java.jar`
- `/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/ext/libryftx_pcap-java.jar`

In both cases, import lines in Java user programs would read:

- `import com.ryft.ryftx.*`
- `import com.ryft.ryftx_pcap.*`

5.1.1 Java Bindings for All Primitives Except PCAP

For all primitives except PCAP, the public interface describing the Java bindings is defined as two Java classes, `RyftDataSet` and `RyftOperationResult`. `RyftDataSet` is defined as follows:

```
public class RyftDataSet detail:

    //
    // Class constructors
    //
    // @param requestedNumberOfNodes the number of nodes to use to
    //        process this dataset
    // @param dataSetPointer the abstracted dataSet 'pointer'
    // @param inputFiles the list of input files
    // @param requestedNumberOfNodes the number of nodes to use to
    //        process this dataset
    //
    public RyftDataSet(int requestedNumberOfNodes)
    public RyftDataSet(long dataSetPointer)

    public RyftDataSet(
        String[]    inputFiles,
        int         requestedNumberOfNodes)

    //
    // Add a file to the dataset.
    //
    // @param filename the file to add
    //
    // @return TRUE on success, FALSE otherwise
    //
    public boolean addFile(String filename)

    //
    // Delete the dataset.  This method must be called after
    // usage of the dataset is complete to avoid memory leaks
```

```
    //
    public void delete()

    //
    // Percent complete callback method that can be called
    // periodically to report the progress of the operation.
    // This method can/should be overridden as necessary by
    // user programs
    //
    // @param percentComplete the current percent complete
    //
    public void percentCompleteCallback(int percentComplete)

    //
    // Perform a search of the dataset using the provided
    // parameters.
    //
    // @param resultsFilename the name of the result file to be
    //        created, or null for none
    // @param queryString string specifying the search criteria
    // @param delimiterString string to be inserted into the
    //        results in between each result
    // @param indexResultsFilename the name of the index file to
    //        be created, or null for none
    //
    // @return a result object containing the operation results
    //
    public RyftDataSet search(
        String  resultsFilename,
        String  queryString,
        String  delimiterString,
        String  indexResultsFilename)

    //
    // Perform a replace operation using the provided
    // parameters.
    //
    // @param filterQuery the filter to apply to narrow the data
    //                    before replacement
    // @param replaceQuery the replacement to perform
    //
    // @return a result object containing the operation results
    //
    public RyftDataSet replace(
        String  filterQuery,
        String  replaceQuery)

    //
    // Perform an n-gram operation.
    //
    // @param resultsFilename the name of the result file to be
    //        created, or null for none
    // @param queryString string specifying the n-gram
    //        configuration
    //
    // @return a result object containing the operation results
    //
    public RyftDataSet ngram(
        String  resultsFilename,
        String  queryString)
```

```
    //
    // Perform a document similarity operation.
    //
    // Perform a document similarity operation.
    //
    // @param resultsFilename the name of the result file to be
    //        created, or null for none
    // @param queryString string specifying similarity criteria
    //
    // @return a result object containing the operation results
    //
    public RyftDataSet documentSimilarity(
        String  resultsFilename,
        String  queryString)

    //
    // Get the results of an operation.  The results will
    // populate one or more of the following:
    //
    //    error status
    //    error string
    //    operation start time, in msec
    //    operation duration, in msec
    //    total number of bytes processed
    //    total matched terms (search operation only)
    //    total unique terms (n-gram operation only)
    //    total documents processed (docsim operation only)
    //    total similar documents (docsim operation only)
    //    max terms per document (docsim operation only)
    //
    // @return the results of the operation
    //
    public RyftOperationResult getResults()
```

`RyftOperationResult` is defined as follows:

```
public class RyftOperationResult:

    //
    // class constructor
    //
    // @param isOperationSuccessful was the operation successful ?
    // @param errorString the error string describing why the operation
    //        was unsuccessful
    // @param startTimeInMsec the operation starting time, in msec
    // @param executionDurationInMsec the operation duration, in msec
    // @param fabricExecutionDurationInMsec the fabric operation duration,
    //        in msec
    // @param totalBytesProcessed the total number of bytes processed
    //        during the operation
    // @param totalNumberOfMatches the total number of matches
    // @param totalNumberOfReplacements the total number of replacements
    // @param totalUniqueTerms the total number of unique terms (n-grams)
    // @param totalDocumentsProcessed the total number of documents processed
    //        for a document similarity operation
    // @param totalSimilarDocuments the total number of similar documents
    //        determined by a document similarity operation
```

```
    // @param maxTermsPerDocument the number of max terms (n-grams) used when
    //        performing a document similarity operation
    //
    public RyftOperationResult(
        boolean isOperationSuccessful,
        String  errorString,
        long    startTimeInMsec,
        long    executionDurationInMsec,
        long    fabricExecutionDurationInMsec,
        long    totalBytesProcessed,
        long    totalNumberOfMatches,
        long    totalNumberOfReplacements,
        long    totalUniqueTerms,
        long    totalDocumentsProcessed,
        long    totalSimilarDocuments,
        long    maxTermsPerDocument)

    //
    // check whether the operation was successful
    //
    // @return TRUE if the operation was successful, FALSE otherwise
    //
    public boolean isOperationSuccessful()

    //
    // return the error string describing why the operation failed
    //
    // @return the error string, or empty string if no error
    //
    public String getErrorString()

    //
    // get the start time of the operation
    //
    // @return the operation start time, in msec
    //
    public long getStartTimeInMsec()

    //
    // get the execution duration of the operation
    //
    // @return the execution duration, in msec
    public long getExecutionDurationInMsec()

    //
    // get the fabric execution duration of the operation
    //
    // @return the fabric execution duration, in msec
    public long getFabricExecutionDurationInMsec()

    //
    // get the total number of bytes processed
    //
    // @return the total number of bytes processed
    //
    public long getTotalBytesProcessed()

    //
    // get the total number of unique terms (n-grams)
    //
```

```
    // @return totalUniqueTerms the total number of unique terms (n-grams)
    //
    public long getTotalUniqueTerms()

    //
    // get the total number of matches
    //
    // @return the total number of matches
    //
    public long getTotalNumberOfMatches()

    //
    // get the total number of replacements
    //
    // @return the total number of replacements
    //
    public long getTotalNumberOfReplacements()

    //
    // get the total number of documents processed for a document similarity
    // operation
    //
    // @return the total number of documents processed for a document
    //         similarity operation
    //
    public long getTotalDocumentsProcessed()

    //
    // get the total number of similar documents for a document similarity
    // operation
    //
    // @return the total number of similar documents for a document
    //         similarity operation
    //
    public long getTotalSimilarDocuments()

    //
    // get the number of max terms (n-grams) used when performing a document
    // similarity operation
    //
    // @return the number of max terms (n-grams) used when performing a
    //         document similarity operation
    //
    public long getMaxTermsPerDocument()
```

5.1.2 Java Bindings for PCAP

For the PCAP primitive family, the public interface describing the Java bindings is defined as two Java classes, `RyftPcapDataSet` and `RyftOperationResult`. `RyftPcapDataSet` is defined as follows:

```
public class RyftPcapDataSet details:

    //
    // Class constructors
    //
    // @param requestedNumberOfNodes the number of nodes to use to
    //        process this dataset
    // @param dataSetPointer the abstracted dataSet 'pointer'
```

```
    // @param inputFiles the list of input files
    //
    public RyftPcapDataSet(int requestedNumberOfNodes)
    public RyftPcapDataSet(long dataSetPointer)
    public RyftPcapDataSet(
        String[]    inputFiles,
        int         requestedNumberOfNodes)

    //
    // Add a file to the dataset.
    //
    // @param filename the file to add
    //
    // @return TRUE on success, FALSE otherwise
    //
    public boolean addFile(String filename)

    //
    // Delete the dataset.  This method must be called after
    // usage of the dataset is complete to avoid memory leaks
    //
    public void delete()

    //
    // Percent complete callback method that can be called
    // periodically to report the progress of the operation.
    // This method can/should be overridden as necessary by
    // user programs
    //
    // @param percentComplete the current percent complete
    //
    public void percentCompleteCallback(int percentComplete)

    //
    // Perform a PCAP search of the dataset using the provided
    // parameters.
    //
    // @param resultsFilename the name of the result file to be
    //        created, or null for none
    // @param queryString string specifying the search
    //        criteria
    // @param indexResultsFilename the name of the index file to
    //        be created, or null for none
    //
    // @return a result object containing the operation results
    //
    public RyftPcapDataSet pcapSearch(
        String  resultsFilename,
        String  queryString,
        String  indexResultsFilename)

    //
    // Get the results of an operation.  The results will
    // populate one or more of the following:
    //
    //    error status
    //    error string
    //    operation start time, in msec
    //    operation duration, in msec
    //    total number of bytes processed
```

```
    //    total matched terms (search operation only)
    //    total unique terms (n-gram operation only)
    //    total documents processed (docsim operation only)
    //    total similar documents (docsim operation only)
    //    max terms per document (docsim operation only)
    //
    // @return the results of the operation
    //
    public RyftOperationResult getResults()
```

`RyftOperationResult` is defined as follows:

```
public class RyftOperationResult details:

    //
    // class constructor
    //
    // @param isOperationSuccessful was the operation successful ?
    // @param errorString the error string describing why the operation
    //        was unsuccessful
    // @param startTimeInMsec the operation starting time, in msec
    // @param executionDurationInMsec the operation duration, in msec
    // @param fabricExecutionDurationInMsec the fabric operation duration,
    //        in msec
    // @param totalBytesProcessed the total number of bytes processed
    //        during the operation
    // @param totalNumberOfMatches the total number of matches
    //
    public RyftOperationResult(
        boolean isOperationSuccessful,
        String  errorString,
        long    startTimeInMsec,
        long    executionDurationInMsec,
        long    fabricExecutionDurationInMsec,
        long    totalBytesProcessed,
        long    totalNumberOfMatches)

    //
    // check whether the operation was successful
    //
    // @return TRUE if the operation was successful, FALSE otherwise
    //
    public boolean isOperationSuccessful()

    //
    // return the error string describing why the operation failed
    //
    // @return the error string, or empty string if no error
    //
    public String getErrorString()

    //
    // get the start time of the operation
    //
    // @return the operation start time, in msec
    //
    public long getStartTimeInMsec()

    //
    // get the execution duration of the operation
```

```
    //
    // @return the execution duration, in msec
    public long getExecutionDurationInMsec()

    //
    // get the execution duration of the operation
    //
    // @return the execution duration, in msec
    public long getFabricExecutionDurationInMsec()

    //
    // get the total number of bytes processed
    //
    // @return the total number of bytes processed
    //
    public long getTotalBytesProcessed()

    //
    // get the total number of matches
    //
    // @return the total number of matches
    //
    public long getTotalNumberOfMatches()
```

5.2 Example Java User Program

Consider the following example input file `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

An example program named `ApiUgJavaSearch.java` that uses the Java bindings to search the input data shown above using an exact search for the characters `555` is shown below:

```
import com.ryft.ryftx.*;

public class ApiUgJavaSearch
{
    public static void main(String args[])
    {
        RyftDataSet dataSet = new RyftDataSet(1);
        dataSet.addFile("passengers-simple.txt");

        RyftDataSet searchDataSet = dataSet.search(
            null,
            "(RAW_TEXT CONTAINS EXACT(\"555\"))",
            "\n",
            null);

        RyftOperationResult searchResult = searchDataSet.getResults();
        boolean boolSuccess = searchResult.isOperationSuccessful();
        if (boolSuccess)
        {
```

```
            System.out.println("   Duration: " +
                searchResult.getExecutionDurationInMsec() + "ms");
            System.out.println("   Bytes: " +
                searchResult.getTotalBytesProcessed());
            System.out.println("   Matches: " +
                searchResult.getTotalNumberOfMatches());
        }
        else
        {
            System.out.println("Error = " +
                searchResult.getErrorString());
        }

        searchDataSet.delete();
        dataSet.delete();
    }
}
```

To compile and execute the example program shown above, perform the following steps:

```
$ javac ApiUgJavaSearch.java
$ java ApiUgJavaSearch
```

Output to `stdout` would resemble:

```
   Duration: 6ms
   Bytes: 253
   Matches: 6
```

6 Python Bindings

6.1 Python Bindings API

The Ryft Open API Python bindings are distributed for use with `python3`.

On most Ubuntu 16.04 LTS systems which standardize on python 3.5, the default install locations for the `.egg` and `.so` files are:

- `/usr/local/lib/python3.5/dist-packages/ryftx-0.3.0-py3.5-linux-x86_64.egg`
- `/usr/local/lib/python3.5/dist-packages/ryftx_pcap-0.1.0-py3.5-linux-x86_64.egg`
- `/usr/local/lib/python3.5/dist-packages/ryft/ryftx.cpython-35m-x86_64-linux-gnu.so`
- `/usr/local/lib/python3.5/dist-packages/ryft/ryftx_pcap.cpython-35m-x86_64-linux-gnu.so`

On most legacy Ubuntu 14.04 LTS systems which standardize on python 3.4, the default install locations are:

- `/usr/local/lib/python3.4/dist-packages/ryftx-0.3.0-py3.4-linux-x86_64.egg`
- `/usr/local/lib/python3.4/dist-packages/ryftx_pcap-0.1.0-py3.4-linux-x86_64.egg`
- `/usr/local/lib/python3.4/dist-packages/ryft/ryftx.cpython-34m.so.`
- `/usr/local/lib/python3.4/dist-packages/ryft/ryftx_pcap.cpython-34m.so.`

Python3 programs leveraging the Ryft Open API would import the packages as needed like this:

- `import ryft.ryftx`
- `import ryft.ryftx_pcap`

6.1.1 Python Bindings for All Primitives Except PCAP

A description of the Python3 bindings for all primitives except the PCAP family can be displayed directly using the interactive `python3` interpreter. As an alternative, the following shell script can be used to show help for the bindings:

```
#!/usr/bin/python3
import ryft.ryftx
help(ryft.ryftx)
```

Typical help output is as follows:

```
Help on module ryft.ryftx in ryft:
NAME
    ryft.ryftx - RyftX Open API Python bindings
CLASSES
    builtins.Exception(builtins.BaseException)
        ryftx.error
    builtins.object
        dataset

    DataSet = class dataset(builtins.object)
     |  A DataSet is an opaque data type that is used to reference the
     |  dataset that is being operated on by calls to the Ryft API
     |  Library.  For example, when using the library's Search
     |  function,
     |  the data to be searched will be of the DataSet type and the
     |  result of that search.
     |
     |  The DataSet can be initialized in several ways where the:
     |  number of nodes is an integer from 1-4:
     |      DataSet(<num_nodes>)
```

```
|      DataSet(<input_file_string>, <num_nodes>)
|      DataSet(<input_file_list>, <num_nodes>)
|      DataSet(<input_file_tuple>, <num_nodes>)
|
|
|  The DataSet can be initialized with just the number of
|  requested processing nodes:
|      my_dataset = DataSet(int) - Where (int) can be 1-4
|
|  Methods defined here:
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate
|      signature.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for
|      accurate signature.
|
|  add_file(...)
|      Add a file to a dataset
|
|      Args:
|          filename_to_add (str): The filename to add to the
|          dataset, and may contain wildcard specifications - required
|
|      Returns:
|        Boolean
|
|      Raises:
|        ryftx.error: An error occurred during file addition
|
|  document_similarity(...)
|      Perform a document similarity operation
|
|      Args:
|          configuration_string (str) - The configuration string
|          - required
|          results_file (str) - Output result file to be created
|          - optional
|          callback function: - References a user-defined
|          callback function
|           for handling incoming percentage completion messages.
|           For example, a user could provide a callback function
|           for
|           printing the current status of the execution to
|           standard output. - optional
|
|      Returns:
|        Dataset object
|
|      Raises:
|        ryftx.error: An error occurred during search
|
|  ngram(...)
|      Perform n-gram/term frequency analysis of an input data
|
|      Args:
|          match_string (str) - Relevant n-gram options to
|          consider - optional
```

```
|          results_file (str) - Output result file to be created
|          - optional
|          callback function: - References a user-defined
|          callback function
|           for handling incoming percentage completion messages.
|           For example, a user could provide a callback function
|           for
|           printing the current status of the execution to
|           standard output. - optional
|
|      Returns:
|        Dataset object
|
|      Raises:
|        ryftx.error: An error occurred during term frequency
|        operation
|
|  replace(...)
|      Perform a replace
|
|      Args:
|          filter_string (str) - The filter to be applied before
|          replacement - required
|          replace_string (str) - The search & replace parameters
|          - required
|          callback function: - References a user-defined
|          callback function
|           for handling incoming percentage completion messages.
|           For example, a user could provide a callback function
|           for
|           printing the current status of the execution to
|           standard output. - optional
|
|      Returns:
|        Dataset object
|
|      Raises:
|        ryftx.error: An error occurred during search
|
|  search(...)
|      Perform a search
|
|      Args:
|          match_string (str) - The search configuration string -
|          required
|          results_file (str) - Output result file to be created
|          - optional
|          index_results_file (str): Output index file to be
|          created - optional
|          del_string (str) - The string to delineate between
|          data search results - optional
|          callback function: - References a user-defined
|          callback function
|           for handling incoming percentage completion messages.
|           For example, a user could provide a callback function
|           for
|           printing the current status of the execution to
|           standard output. - optional
|
|      Returns:
```

```
|         Dataset object
|
|     Raises:
|       ryftx.error: An error occurred during search
|
|  ----------------------------------------------------------------
|  --------
|  Data descriptors defined here:
|
|  error_occurred
|      Boolean indicating whether an error occurred on the last
|      search
|
|  error_string
|      String indicating the error from the last search
|
|  execution_duration
|      Long indicating the total number of milliseconds the last
|      search took
|
|  files
|      Tuple of input files contained by the dataset
|
|  max_terms_per_document
|      Long indicating the max terms used for calculations during
|      a document similarity operation
|
|  num_files
|      Long indicating the number of files operated on by a
|      dataset
|
|  num_processing_nodes
|      Long indicating the number of processing nodes requested
|      for the dataset
|
|  start_time
|      Long indicating when the program started in milliseconds
|      relative to epoch
|
|  total_bytes_processed
|      Long indicating the total number of bytes processed during
|      the last search
|
|  total_documents_processed
|      Long indicating total number of documents processed by a
|      document similarity operation
|
|  total_matches
|      Long indicating the total number of matches from the last
|      search
|
|  total_replacements
|      Long indicating the total number of replacements from the
|      last replace
|
|  total_similar_documents
|      Long indicating the total number of similar documents
|      found for a document similarity operation
|
|  total_unique_terms
```

```
     |      Long indicating the total number of unique terms from the
     |      last n-gram operation

    ...
```

6.1.2 Python Bindings for PCAP

A description of the Python3 bindings for the PCAP family can be displayed directly using the interactive `python3` interpreter. As an alternative, the following shell script can be used to show help for the bindings:

```
#!/usr/bin/python3
import ryft.ryftx_pcap
help(ryft.ryftx_pcap)
```

Typical help output is as follows:

```
Help on module ryft.ryftx_pcap in ryft:
NAME
    ryft.ryftx_pcap - RyftX Pcap Open API Python bindings
CLASSES
    builtins.Exception(builtins.BaseException)
        ryftx.error
    builtins.object
        dataset

    PcapDataSet = class dataset(builtins.object)
     |  A PcapDataSet is an opaque data type that is used to reference
     |  the
     |  dataset that is being operated on by calls to the Ryft API
     |  Library.  For example, when using the library's Search
     |  function,
     |  the data to be searched will be of the DataSet type and the
     |  result of that search.
     |
     |  The PcapDataSet can be initialized in several ways where the:
     |  number of nodes is an integer from 1-4:
     |      PcapDataSet(<num_nodes>)
     |      PcapDataSet(<input_file_string>, <num_nodes>)
     |      PcapDataSet(<input_file_list>, <num_nodes>)
     |      PcapDataSet(<input_file_tuple>, <num_nodes>)
     |
     |
     |  The PcapDataSet can be initialized with just the number of
     |  requested processing nodes:
     |      my_dataset = PcapDataSet(int) - Where (int) can be 1-4
     |
     |  Methods defined here:
     |
     |  __init__(self, /, *args, **kwargs)
     |      Initialize self.  See help(type(self)) for accurate
     |      signature.
     |
     |  __new__(*args, **kwargs) from builtins.type
     |      Create and return a new object.  See help(type) for
     |      accurate signature.
     |
     |  add_file(...)
```

```
|      Add a file to a dataset
|
|      Args:
|          filename_to_add (str): The filename to add to the
|          dataset, and may contain wildcard specifications - required
|
|      Returns:
|        Boolean
|
|      Raises:
|        ryftx.error: An error occurred during file addition
|
|  pcap_search(...)
|      Perform a PCAP search
|
|      Args:
|          match_string (str) - The configuration string for the
|          PCAP operation - required
|          results_file (str) - Output result file to be created
|          - optional
|          index_results_file (str): Output index file to be
|          created - optional
|          callback function: - References a user-defined
|          callback function
|           for handling incoming percentage completion messages.
|           For example, a user could provide a callback function
|           for
|           printing the current status of the execution to
|           standard output. - optional
|
|      Returns:
|        Dataset object
|
|      Raises:
|        ryftx.error: An error occurred during search
|
|  ----------------------------------------------------------------
|  --------
|  Data descriptors defined here:
|
|  error_occurred
|      Boolean indicating whether an error occurred on the last
|      search
|
|  error_string
|      String indicating the error from the last search
|
|  execution_duration
|      Long indicating the total number of milliseconds the last
|      search took
|
|  files
|      Tuple of input files contained by the dataset
|
|  num_files
|      Long indicating the number of files operated on by a
|      dataset
|
|  num_processing_nodes
|      Long indicating the number of processing nodes requested
```

```
     |      for the dataset
     |
     |  start_time
     |      Long indicating when the program started in milliseconds
     |      relative to epoch
     |
     |  total_bytes_processed
     |      Long indicating the total number of bytes processed during
     |      the last search
     |
     |  total_matches
     |      Long indicating the total number of matches from the last
     |      search

    ...
```

6.2 Example Python User Program

Consider the following example input file `passengers-simple.txt`:

```
Hannibal Smith, 10-01-1928, 310-555-1212
DR. Thomas Magnum, 01-29-1945, 310-555-2323
Steve McGarett, 12-30-1920, 310-555-3434
Michael Knight, 08-17-1952, 310-555-4545
Stringfellow Hawke, 08-15-1944, 310-555-5656
Sonny Crockett, 12-14-1949, 310-555-6767
```

An example Python3 program that uses the Python bindings is shown below as a shell script which can be executed from the command line. It performs a Hamming search against the input file shown above, returning all lines that contain "`Dr.`" with distance 1, outputting the results to `d.txt`:

```
#!/usr/bin/python3

import ryft.ryftx

input_files = ("passengers-simple.txt")
input_data_set = ryft.ryftx.DataSet(input_files, 1)

search_results = input_data_set.search(match_string="(RAW_TEXT CONTAINS HAMMING(\"Dr.\",
DISTANCE=\"1\", LINE=\"true\"))", results_file="d.txt")

print("   Duration: " + str(search_results.execution_duration) + "ms")
print("   Bytes: " + str(search_results.total_bytes_processed))
print("   Matches: " + str(search_results.total_matches))
```

Output to `stdout` would resemble:

```
   Duration: 9ms
   Bytes: 253
   Matches: 1
```

And the output file `d.txt` contains:

```
DR. Thomas Magnum, 01-29-1945, 310-555-2323
```