Using Search

Contents

You must first enable Riak Search in your environment to use it.

Introduction

Riak Search is a distributed, full-text search engine that is built on Riak Core and included as part of Riak open source. Search provides the most advanced query capability next to MapReduce, but is far more concise; easier to use, and in most cases puts far less load on the cluster.

Search indexes Riak KV objects as they're written using a precommit hook. Based on the object's mime type and the Search schema you've set for its bucket, the hook will automatically extract and analyze your data and build indexes from it. The Riak Client API is used to invoke Search queries that return a list of bucket/key pairs matching the query. Currently the PHP, Python, Ruby, and Erlang client libraries support integration with Riak Search.

Features

- Support for various mime types (JSON, XML, plain text, Erlang, Erlang binaries) for automatic data extraction
- Support for various analyzers (to break text into tokens) including a white space analyzer, an integer analyzer, and a no-op analyzer
- Robust, easy-to-use query language

- Exact match queries
    - Wildcards
    - Inclusive/exclusive range queries o AND/OR/NOT support
    - Grouping
    - Prefix matching
    - Proximity searches
    - Term boosting
- Solr-like interface via HTTP (not Solr compatible)
- Protocol buffers interface
- Scoring and ranking for most relevant results
- Search queries as input for MapReduce jobs

When to Use Search

- When collecting, parsing, and storing data like user bios, blog posts, journal articles, etc. to facilitate fast and accurate information retrieval in addition to some reasonable form of ranking.
- When indexing JSON data. Extractors pull the data apart depending on mime type, analyzers tokenize the fields, and the schema describes what fields to index, how to analyze them and how to store them.
- When fast information retrieval with a powerful query language is needed.

When Not to Use Search

- When only simple tagging of data is needed with exact match and range queries. In this case Secondary Indexes would be easier.
- When the data is not easily analyzed by Search. For example mp3, video, or some other binary format. In this case a Secondary Index interface is recommended.
- When built-in anti-entropy/consistency is required. At this time Search has no read-repair mechanism. If Search index data is lost, the entire data set must be re-indexed.

Indexing Data

Before you can search for values, you first must index them. In its standard form, Riak Search requires you to index a value manually. You can find a detailed list of indexing commands in the Search Indexing Reference.

If you want a simpler, but less explicit form of indexing, check out theSearch, KV and MapReduce section of Advanced Search.

Query Interfaces

Querying via Command Line

Riak Search comes equiped with a command line tool `search-cmd` for testing your query syntax.

This example will display a list of document ID values matching the title "See spot run".

Shell

```
bin/search-cmd search books "title:\"See spot run\""
```

Solr

Riak Search supports a Solr-compatible interface for searching documents via HTTP. By default, the select endpoint is located at `http://hostname:8098/solr/select`.

Alternatively, the index can be included in the URL, for example `http://hostname:8098/solr/INDEX/select`.

The following parameters are supported:

- `index=INDEX`: Specifies the default index name.
- `q=QUERY`: Run the provided query.

- `df=FIELDNAME`: Use the provided field as the default. Overrides the "default_field" setting in the schema file.
- `q.op=OPERATION`: Allowed settings are either "and" or "or". Overrides the "default_op" setting in the schema file. Default is "or".
- `start=N`: Specify the starting result of the query. Useful for paging. Default is 0.
- `rows=N`: Specify the maximum number of results to return. Default is 10.
- `sort=FIELDNAME`: Sort on the specified field name after the given rows are found. Default is "none", which causes the results to be sorted in descending order by score.
- `wt=FORMAT`: Choose the format of the output. Options are "xml" and "json". The default is "xml".
- `filter=FILTERQUERY`: Filters the search by an additional query scoped to inline fields.

- `presort=key|score`: Sorts all of the results by bucket key, or the search score, before the given rows are chosen. This is useful when paginating to ensure the results are returned in a consistent order.

> ### Limitations on Presort
>
> When paginating results with **presort**, note that the results may only be sorted by the search **score** or sorted by the **key order**. There is currently no way to pre-sort on an arbitrary field. This means that if you with to paginate on some field, build your keys to include that field value then use `presort=key`.

To query data in the system with Curl:

HTTP

```
$ curl "http://localhost:8098/solr/books/select?start=0&rows
```

Riak Client API

The Riak Client APIs have been updated to support querying of Riak Search. See the client documentation for more information. Currently, the Ruby, Python, PHP, and Erlang clients are supported.

The API takes a default search index as well as as search query, and returns a list of bucket/key pairs. Some clients transform this list into objects specific to that client.

MapReduce

The Riak Client APIs that integrate with Riak Search also support using a search query to generate inputs for a MapReduce operation. This allows you to perform powerful analysis and computation across your data based on a search query. See the client documentation for more information. Currently, the Java, Ruby, Python, PHP, and Erlang clients are supported.

Kicking off a MapReduce query with the same result set over HTTP would use a POST body like this:

Json

```json
{
  "inputs": {
             "bucket":"mybucket",
             "query":"foo OR bar"
            },
  "query":...
 }
```

or

Json

```json
{
  "inputs": {
             "bucket":"mybucket",
             "query":"foo OR bar",
             "filter":"field2:baz"
            },
  "query":...
 }
```

The phases in the "query" field should be exactly the same as usual. An initial map phase will be given each object matching the search for processing, but an initial link phase or reduce phase will also work.

The query field specifies the search query. All syntax available in other Search interfaces is available in this query field. The optional filter field specifies the query filter.

The old but still functioning syntax is:

Json

```json
{
  "inputs": {
               "module":"riak_search",
               "function":"mapred_search",
               "arg":["customers","first_name:john"]
             },
  "query":...
 }
```

The "arg" field of the inputs specification is always a two-element list. The first element is the name of the bucket you wish to search, and the second element is the query to search for.

Query Syntax

Search queries use the same syntax as Lucene and supports most Lucene operators including term searches, field searches, boolean operators, grouping, lexicographical range queries, and wildcards (at the end of a word only).

Terms and Phrases

A query can be as simple as a single term (ie: "red") or a series of terms surrounded by quotes called a phrase ("See spot run"). The term (or phrase) is analyzed using the default analyzer for the index.

The index schema contains a setting that determines whether a phrase is treated as an AND operation or an OR operation. By default, a phrase is treated as an OR operation. In other words, a document is returned if it matches any one of the terms in the phrase.

Fields

You can specify a field to search by putting it in front of the term or phrase to search. For example:

Shell

```
color:red
```

Or:

Shell

```
title:"See spot run"
```

You can further specify an index by prefixing the field with the index name. For example:

Shell

```
products.color:red
```

Or:

Shell

```
books.title:"See spot run"
```

If your field contains special characters, such as ('+','-','/','[',']','(',')',':' or space), then either surround the phrase in single quotes, or escape each special character with a backslash.

Shell

```
books.url:'http://mycompany.com/url/to/my-book#foo'
```

-or-

Shell

```
books.url:http\:\/\/mycompany.com\/url\/to\/my\-book\#foo
```

Wildcard Searches

Terms can include wildcards in the form of an asterisk (*) to allow prefix matching, or a question mark (?) to match a single character.

Currently, the wildcard must come at the end of the term in both cases, and must be preceded by a minimum of two characters.

For example:

- "bus*" will match "busy", "business", "busted", etc.
- "bus?" will match "busy", "bust", "busk", etc.

Proximity Searches

Proximity searching allows you to find terms that are within a certain number of words from each other. To specify a proximity search, use the tilde argument on a phrase.

For example:

Shell

```
"See spot run"~20
```

Will find documents that have the words "see", "spot", and "run" all within the same block of 20 words.

Range Searches

Range searches allow you to find documents with terms in between a specific range. Ranges are calculated lexicographically. Use square brackets to specify an inclusive range, and curly braces to specify an exclusive range.

The following example will return documents with words containing “red” and “rum”, plus any words in between.

Shell

```
"field:[red TO rum]"
```

The following example will return documents with words in between “red” and “rum”:

Shell

```
"field:{red TO rum}"
```

Boosting a Term

A term (or phrase) can have its score boosted using the caret operator along with an integer boost factor.

In the following example, documents with the term “red” will have their score boosted:

Shell

```
red^5 OR blue
```

Boolean Operators - AND, OR, NOT

Queries can use the boolean operators AND, OR, and NOT. The boolean operators must be capitalized.

The following example return documents containing the words “red” and “blue” but not “yellow”.

Shell

```
red AND blue AND NOT yellow
```

The required (+) operator can be used in place of “AND”, and the prohibited (-) operator can be used in place of “AND NOT”. For example, the query above can be rewritten as:

Shell

```
+red +blue -yellow
```

Grouping

Clauses in a query can be grouped using parentheses. The following query returns documents that contain the terms “red” or “blue”, but not “yellow”:

Shell

```
(red OR blue) AND NOT yellow
```

These May Also Interest You

- Five-Minute Install
- Advanced Secondary Indexes
- Replication Properties
- Advanced Commit Hooks
- Advanced MapReduce
- Advanced Search