Using Secondary Indexes

Contents

When using the LevelDB or Memory backend, you have the ability to retrieve objects by using external indexes.

Introduction

Secondary Indexing (2i) in Riak gives developers the ability, at write time, to tag an object stored in Riak with one or more queryable values.

Since the KV data is completely opaque to 2i, the user must tell 2i exactly what attribute to index on and what its index value should be, via key/value metadata. This is different from Search, which parses the data and builds indexes based on a schema. Riak 2i currently requires the LevelDB or Memory backend.

Features

- Allows two types of secondary attributes: integers and strings (aka binary).
- Allows querying by exact match or range on one index.
- Allows pagination of results.
- Allows streaming of results.
- Query results can be used as input to a MapReduce query.

When to Use Secondary Indexes

- When you want to find data based on terms other than an objects' bucket/key pair. With 2i, indexes can be created by adding metadata to Riak objects.
- When the value being stored is an opaque blob, like a binary file, and you want to index it via added attributes.
- When you want or need an easy-to-use search mechanism. Secondary indexing does not require a schema (as in Search) and comes with a basic query interface.
- When you want or need anti-entropy. Since 2i is just metadata on the KV object and the indexes reside on the same node, 2i piggybacks off of read-repair.

When Not to Use Secondary Indexes

- If your ring size exceeds 512 partitions: 2i can cause performance issues in large clusters. (See "How it Works").
- When you need more than the exact match and range searches that 2i supports.
- When you want to use composite queries. A query like "last_name=zezeski AND state=MD" would have to be split into two queries and the results merged (or involve MapReduce).

Query Interfaces and Examples

Typically, the result set from a 2i query is a list of object keys from the specified bucket that include the index values in question. As we'll see below, when executing range queries in Riak 1.4 or higher it is possible to retrieve the index values along with the object keys.

In this example, a bucket/key pair of "users/john_smith" is used to store user data. The user would like to add a twitter handle and email address as secondary indexes:

Inserting the object with Secondary Indexes:

HTTP

```
$ curl -X POST \
  -H 'x-riak-index-twitter_bin: jsmith123' \
  -H 'x-riak-index-email_bin: jsmith@basho.com' \
  -d '...user data...' \
  http://localhost:8098/buckets/users/keys/john_smith
```

- The object has been stored with a primary bucket/key of: users/john_smith
- The object now has a secondary index called “twitter_bin” with a value of: jsmith123
- The object now has a secondary index called “email_bin” with a value of: jsmith@basho.com

Querying the object with Secondary Indexes:

Query the twitter handle…

HTTP

```
$ curl localhost:8098/buckets/users/index/twitter_bin/jsmith
```

Response…

Text

```
{"keys":["john_smith"]}
```

Examples

To run the following examples, ensure that Riak is running on localhost with the HTTP endpoint listing on port 8098, and configured to use an index-capable storage backend. `curl` is required.

Indexing Objects

The following example indexes four different objects. Notice that we're storing both integer and string (aka binary) fields, field names are automatically lowercased, some fields have multiple values, and duplicate fields are automatically de-duplicated:

HTTP

```
$ curl -v -XPUT \
  -d 'data1' \
  -H "x-riak-index-field1_bin: val1" \
  -H "x-riak-index-field2_int: 1001" \
  http://127.0.0.1:8098/riak/mybucket/mykey1

$ curl -v -XPUT \
  -d 'data2' \
  -H "x-riak-index-Field1_bin: val2" \
  -H "x-riak-index-Field2_int: 1002" \
 http://127.0.0.1:8098/riak/mybucket/mykey2

$ curl -v -XPUT \
  -d 'data3' \
  -H "X-RIAK-INDEX-FIELD1_BIN: val3" \
```

```
  -H "X-RIAK-INDEX-FIELD2_INT: 1003" \
  http://127.0.0.1:8098/riak/mybucket/mykey3

$ curl -v -XPUT \
  -d 'data4' \
  -H "x-riak-index-field1_bin: val4, val4, val4a, val4b" \
  -H "x-riak-index-field2_int: 1004, 1004, 1005, 1006" \
  -H "x-riak-index-field2_int: 1004" \
  -H "x-riak-index-field2_int: 1004" \
  -H "x-riak-index-field2_int: 1004" \
  -H "x-riak-index-field2_int: 1007" \
  http://127.0.0.1:8098/riak/mybucket/mykey4
```

The following examples demonstrate what happens when an index field is specified with an invalid field name or type. The system responds with `400 Bad Request` and a description of the error.

Invalid field name:

```
curl -XPUT \
-d 'data1' \
-H "x-riak-index-field2_foo: 1001" \
http://127.0.0.1:8098/riak/mybucket/mykey

# Response
Unknown field type for field: 'field2_foo'.
```

Incorrect data type:

```
curl -XPUT \
-d 'data1' \
-H "x-riak-index-field2_int: bar" \
http://127.0.0.1:8098/riak/mybucket/mykey

# Response
Could not parse field 'field2_int', value 'bar'.
```

Querying

Exact Match

The following examples use the HTTP interface to perform an exact match index query:

Query a binary index

HTTP

```
$ curl http://localhost:8098/buckets/mybucket/index/field1_b
```

Query an integer index

HTTP

```
$ curl http://localhost:8098/buckets/mybucket/index/field2_i
```

The following example performs an exact match query and pipes the results into a MapReduce job:

HTTP

```
$ curl -X POST \
  -H "content-type: application/json" \
  -d @- \
  http://localhost:8098/mapred \
<<EOF
{
   "inputs":{
       "bucket":"mybucket",
       "index":"field1_bin",
       "key":"val3"
   },
   "query":[
      {
         "reduce":{
            "language":"erlang",
```

```
            "module":"riak_kv_mapreduce",
            "function":"reduce_identity",
            "keep":true
        }
    }
  ]
}
EOF
```

Range

The following examples use the HTTP interface to perform a range query:

Query a binary index…

HTTP

```
$ curl http://localhost:8098/buckets/mybucket/index/field1_b
```

Query an integer index…

HTTP

```
$ curl http://localhost:8098/buckets/mybucket/index/field2_i
```

The following example performs a range query and pipes the results into a MapReduce job:

HTTP

```
$ curl -X POST \
  -H "content-type: application/json" \
  -d @- \
  http://localhost:8098/mapred \
<<EOF
{
   "inputs":{
       "bucket":"mybucket",
       "index":"field1_bin",
       "start":"val2",
       "end":"val4"
```

```
    },
    "query":[
        {
            "reduce":{
                "language":"erlang",
                "module":"riak_kv_mapreduce",
                "function":"reduce_identity",
                "keep":true
            }
        }
    ]
}
EOF
```

Range with terms

When performing a range query, it is possible to retrieve the matched index values alongside the Riak keys using `return_terms=true`. An example from a small sampling of Twitter data with indexed hash tags:

Shell

```
curl 'http://localhost:10018/buckets/tweets/index/hashtags_b
{"results":[{"rock":"34922410122478796 8"},{"rocks":"3492236 3
```

Range query term filtering with regular expressions

The term_regex parameter filters secondary index range query results such that only terms that match the regular expression are returned.

For example, the following unfiltered query returns the below term/key pairs:

Shell

```
http://localhost:10018/buckets/b/index/f1_bin/a/z?return_ter

[("baa", "key1"),("aab", "key2"),("bba", "key3")]
```

The same query, executed with the regular expression `^.a` passed in using the term_regex parameter, returns only the items that have an 'a' as the second character in the term:

Shell

```
http://localhost:10018/buckets/b/index/f1_bin/a/z?return_ter

[("baa", "key1"),("aab", "key2")]
```

Pagination

When asking for large result sets, it is often desirable to ask the servers to return chunks of results instead of a firehose. As of Riak 1.4, you can do so using `max_results=<n>`, where `n` is the number of results you'd like to receive.

Assuming more keys are available, a `continuation` value will be included in the results to allow the client to request the next page.

Here is an example of a range query with both `return_terms` and pagination against the same Twitter data set. Results are fed into python for easier reading.

Shell

```
curl 'http://localhost:10018/buckets/tweets/index/hashtags_b
{
    "continuation": "g2gCbQAAAAdyaXBqYWtlbQAAABIzNDkyMjA2ODc
    "results": [
        {
            "rice": "349222574510710785"
        },
        {
            "rickross": "3492228680952176 64"
        },
        {
            "ridelife": "349221819552763905"
        },
        {
            "ripjake": "349220649341952001"
```

```
        },
        {
            "ripjake": "3492206870571294 73"
        }
    ]
}

# Take the continuation value from the previous result set a
curl 'http://localhost:10018/buckets/tweets/index/hashtags_b
{
    "continuation": "g2gCbQAAAAlyb2Jhc2VyaWFtAAAAEjM0OTIyMzc
    "results": [
        {
            "ripjake": "3492211987748085 79"
        },
        {
            "ripped": "3492240173471006 72"
        },
        {
            "roadtrip": "3492212071550320 66"
        },
        {
            "roastietime": "3492213707244912 65"
        },
        {
            "robaseria": "3492237027659120 65"
        }
    ]
}
```

Streaming

It is possible to stream results using `stream=true`. This can be combined with pagination and `return_terms`.

Sorting

Non-paginated 2i queries are unsorted by default. Users have the option to sort per request, or to sort every non-paginated query by default. Per request sorting can be done using the `pagination_sort` parameter. Sorting of all non-paginated secondary index queries by default can be accomplished by adding `{secondary_index_sort_default, true}` to the riak_kv section of the app.config.

An example of this sorting can be seen in the pagination example above. Hash tags (2i keys) are returned in ascending order, and the object keys (Twitter IDs) for the messages which contain the “ripjake” hash tag are also returned in ascending order.

Retrieve all Bucket Keys via $bucket Index

The following example uses the HTTP interface to retrieve the keys for all objects stored in the bucket 'mybucket' using an exact match on the special $bucket index.

HTTP

```
$ curl http://localhost:8098/buckets/mybucket/index/\$bucket
```

Count Bucket Objects via $bucket Index

The following example performs a secondary index lookup on the $bucket index like in the previous example and pipes this into a MapReduce that counts the number of records in the 'mybucket' bucket. In order to improve efficiency, the batch size has been increased from the default size of 20.

HTTP

```
$ curl -XPOST http://localhost:8098/mapred
  -H 'Content-Type: application/json'
  -d '{"inputs":{
          "bucket":"mybucket",
```

```
            "index":"$bucket",
            "key":"mybucket"
        },
        "query":[{"reduce":{"language":"erlang",
                            "module":"riak_kv_mapreduce",
                            "function":"reduce_count_inputs",
                            "arg":{"reduce_phase_batch_size":
                           }
                }]
        }'
EOF
```

These May Also Interest You

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