Using Key Filters

Contents

Key filters are a way to pre-process MapReduce inputs from a full bucket query simply by examining the key without loading the object first. This is especially useful if your keys are composed of domain-specific information that can be analyzed at query-time.

Understanding key filters

Key filters can be thought of as a series or pipeline of transformationsand predicates that attempt to match keys produced by the list-keys operation. Keys that match the predicates are fed into the MapReduce query as if they had been specified manually.

To illustrate this, let's contrive an example. Let's say we're storing customer invoices with a key constructed from the customer name and the date, in a bucket called “invoices”. Here are some sample keys:

```
basho-20101215
google-20110103
yahoo-20090613
```

Given that key scheme, here are some queries we will be able to do simply with key filters:

- Find all invoices from a given customer.
- Find all invoices from a range of dates.
- Find invoices from customers who have names containing the word "solutions".
- Find invoices that were sent on the 3rd of June.

Solutions to these queries are shown in the examples below.

Once the keys are filtered to only the items we care about, the normal MapReduce pipeline can further filter, transform, extract, and aggregate all the data we are interested in.

Constructing key filters

Key filters change the structure of the "inputs" portion of the MapReduce query.

When submitting a query in JSON format, this makes the inputs a JSON object containing two entries, "bucket" and "key_filters". All filters are specified as arrays, even if the filter takes no arguments. Example:

Json

```
{
  "inputs":{
     "bucket":"invoices",
     "key_filters":[["ends_with", "0603"]]
  }
  /* ...rest of mapreduce job */
}
```

When submitting a query from the Erlang local or Protocol Buffers client, the inputs become a two-tuple where the first element is the bucket as a binary, and the second element is a list of filters. Like the JSON format, the filters are specified as lists, even for filters with no arguments, and the filter names are binaries.

Erlang

```
riakc_pb_socket:mapred(Pid, {<<"invoices">>, [[<<"ends_wit
```

Key Filter Functions

Riak Key Filter provides two kinds of function manipulators: transform and predicate.

Transform key-filter functions manipulate the key so that it can be turned into a format suitable for testing by the predicate functions. Each function description is followed by a sample usage in JSON notation.

Predicate key-filter functions perform a test on their inputs and return true or false. As such, they should be specified last in a sequence of key-filters and are often preceded by transform functions.

A full list of keyfilter functions can be found in the Key Filters Reference.

Example query solutions

Find all invoices for a given customer

Json

```
{
  "inputs":{
     "bucket":"invoices"
     "key_filters":[["tokenize", "-", 1],["eq", "basho"]]
   },
   /* ... */
}
```

Find all invoices from a range of dates

Json

```
{
  "inputs":{
     "bucket":"invoices"
     "key_filters":[["tokenize", "-", 2],
                    ["between", "20100101", "20101231"]]
   },
   /* ... */
}
```

Find invoices from customers who have names containing the word “solutions”

Json

```
{
  "inputs":{
     "bucket":"invoices"
     "key_filters":[["tokenize", "-", 1],
                    ["to_lower"],
                    ["matches", "solutions"]]
   },
   /* ... */
}
```

Find invoices that were sent on the 3rd of June

Json

```
{
  "inputs":{
     "bucket":"invoices"
     "key_filters":[["ends_with", "0603"]]
   },
   /* ... */
}
```

These May Also Interest You

- Five-Minute Install
- Advanced Secondary Indexes

- Advanced Secondary Indexes
- Replication Properties
- Advanced Commit Hooks
- Advanced MapReduce
- Advanced Search