Configuring Secondary Indexes

Contents

Configuration

Secondary indexes (2i) are enabled by configuring Riak to use theLevelDB or Memory backend (or one of those in conjunction with theMulti backend.

All nodes in a cluster must be configured to use an indexing-capable backend for secondary indexes to work properly.

Migrating an Existing Cluster

Warning: this should be done with caution to avoid overburdening your cluster. This will involve migrating all data on any given node to other nodes in the cluster, resulting in higher than usual network and disk I/O.

These steps will remove a node from the cluster, enable a backend that supports 2i, and re-add it to the cluster.

1. Choose one node in the cluster. Run `riak-admin leave` on the node.
2. Wait for transfers to complete. Then, run `riak stop`.
3. Turn on 2i by configuring Riak to use the appropriate backend.

4. Run `riak-admin join`
5. Repeat with remaining nodes.

Indexing an Object

To index an object, an application tags the object with one or more field/value pairs, also called index entries. At write time, the object is indexed under these index entries. An application can modify the indexes for an object by reading an object, adding or removing index entries, and then writing the object. Finally, an object is automatically removed from all indexes when it is deleted.

The object's value and its indexes should be thought of as a single unit. There is no way to alter the indexes of an object independently from the value of an object, and vice versa. To think of it another way, you can imagine that every time you write an object, it completely de-indexes the object, and then re-indexes the object based on the new set of provided index entries.

In the case where an object has siblings, the object is indexed under the index entries for *ALL* of the siblings. When the conflict is resolved, it acts just like an index update. Riak de-indexes the object, and then re-indexes the newly resolved object based on the provided index entries.

Indexing is atomic, and is updated in real time when writing an object. This means that an object will be present in future index queries as soon as the write operation completes.

Index Data Types

As of version 1.0, Riak supports two types of indexes, binaries and integers. Because there is no index schema, the data type of an index is encoded within the suffix of an index field's name. Available suffixes are `_int` and `_bin`, indicating an integer or a string (aka binary), respectively. For example, `timestamp_int` is an integer field and `email_bin` is a string field.

More complicated data types (such as dates and timestamps) can be stored by

encoding the date as a string of characters and choosing a format that sorts correctly. For example, a date could be encoded as the string “YYYYMMDD”. Floats can be encoded by deciding on a required precision level, multiplying the value accordingly, and then truncating the float to an int. For example, to store a float with precision down to the thousandths, you would multiply the float by 1000.

Index field names are automatically converted to lowercase. Index fields and values should only contain ASCII characters. For that reason, it's best to normalize or encode any user-supplied data before using it as an index. While Secondary Indexes may appear to work with multibyte encodings, they should be avoided because Secondary Index range queries assume single byte characters, so range queries across multibyte strings will return incorrect results.

When using the HTTP interface, multi-valued indexes are specified by separating the values with a comma (,). For that reason, your application should avoid using a comma as part of an index value.

Index Sizes

The indexes on an object contribute to the overall size of an object. The number of indexes on an object is limited only by the maximum recommended Riak object size (not larger than 1-2MB). Basho has stress tested objects with 1000 index entries, but expect that most applications will use significantly fewer.

The size of an individual index is also limited only by resources, but note that some HTTP proxies impose size limits on HTTP headers. Since indexes are encoded as HTTP headers when using the HTTP interface, this may affect the maximum index value size.

On disk, indexes contribute to the overall size of the object, and are also stored in a separate structure that takes additional disk space. The overhead for each additional index is minimal. LevelDB employs prefix compression, which can also drastically reduce the amount of disk space an index requires.

An Object's Indexes and Value Are

Orthogonal

An object's indexes and an object's value are completely orthogonal. The indexes may or may not reflect information that already exists in the object's value.

The intended usage is that an application will store some information, such as a blob of JSON data, in the value of a Riak object, and then annotate the object with indexes for easier retrieval later. The indexes may or may not repeat data that is already found in the JSON object. For example, an object may have an `email_bin` index field while also having an `email` field in the JSON blob.

This may seem inefficient in the cases where data is duplicated, but it leads to some interesting scenarios. For example, an application can store images in Riak (or MP3s, or other opaque, non-extensible data structures), and can index the images by owner, file size, creation date, etc.

Index Lookups

Index queries are only supported on one index field at a time. The query can specify either an exact match or a range of values. Range queries are inclusive, so results may match the start or end value. The query operation returns a list of matching keys. The application may then decide to loop through each key, looking up the value from Riak.

Currently, the result order is undefined, and there is no way to directly pull back a list of objects using secondary indexes. This may change in the future.

An index query can be piped into a MapReduce job, allowing applications to sort, filter, or process query results in parallel across a cluster.

Lookup Performance

2i uses document-based partitioning, which means that the indexes for an object are stored on the same partition as the object itself. This has implications on query-time performance. When issuing a query, the system must read from what we call a

“covering” set of partitions. The system looks at how many replicas of our data are stored and determines the minimum number of partitions that it must examine to retrieve a full set of results, accounting for any offline nodes.

By default, Riak is configured to store 3 replicas of all objects, so the system can generate a full result set if it reads from one-third of the system's partitions, as long as it chooses the right set of partitions. The query is then broadcast to the selected partitions, which read the index data, generate a list of keys, and send them back to the requesting node.

Special Fields

The `$key` index field is a special field that is implicitly indexed on all objects when 2i is enabled. The value of this field is the object's key, so this field allows an application to perform range queries across the keys in a bucket.

The `$bucket` index field is another special field that is implicitly indexed on all objects. The value of this field is the object's bucket, so this field allows an application retrieve all objects within a bucket based on secondary indexes.

These May Also Interest You

- Downloads
- Log Messages FAQs
- Operating Riak FAQs
- Configuration Files
- Load Balancing and Proxy Configuration
- Basic Cluster Setup