Cockatrice Documentation

Release 0.2.0

Minoru Osuka

Oct 18, 2018

Contents

Cockatrice is the open source search and indexing server written in Python that provides scalable indexing and search, faceting, hit highlighting and advanced analysis/tokenization capabilities.

CHAPTER 1

Features

Indexing and search are implemented by Whoosh. Cockatrice provides it via the RESTful API using Flask.
In cluster mode, uses Raft Consensus Algorithm by PySyncObj to achieve consensus across all the instances of the nodes, ensuring that every change made to the system is made to a quorum of nodes.

- Full-text search and indexing
- Faceting
- Result highlighting
- Easy deployment
- Bringing up cluster
- Index replication
- An easy-to-use RESTful API

CHAPTER 2

Source Codes

https://github.com/mosuka/cockatrice

CHAPTER 3

Requirements

Python 3.x interpreter

CHAPTER 4

Contents

4.1 Getting Started

Installation of Cockatrice on Unix-compatible or Windows servers generally requires Python interpreter and pip command.

4.1.1 Installing Cockatrice

Cockatrice is registered to PyPi now, so you can just run following command:

```
$ pip install cockatrice
```

4.1.2 Starting Cockatrice

Cockatrice includes a command line interface tool called bin/cockatrice. This tool allows you to start Cockatrice in your system.

To use it to start Cockatrice you can simply enter:

```
$ cockatrice server
```

This will start Cockatrice, listening on default port (8080).

```
$ curl -s -X GET http://localhost:8080/
```

You can see the result in plain text format. The result of the above command is:

```
cockatrice <VERSION> is running.
```

4.2 Schema management

First of all, you need to create a schema definition. Cockatrice fully supports the field types, analyzers, tokenizers and filters provided by Whoosh. This section explains how to describe schema definition.

4.2.1 Schema Design

Cockatrice defines the schema in YAML format. YAML is a human friendly data serialization standard for all programming languages.

The following items are defined in YAML:

- schema
- default_search_field
- field_types
- analyzers
- tokenizers
- filters

4.2.2 Schema

The schema is the place where you tell Cockatrice how it should build indexes from input documents.

```
schema:
  <FIELD_NAME>:
    field_type: <FIELD_TYPE>
    args:
      <ARG_NAME>: <ARG_VALUE>
      ...
```

- `<FIELD_NAME>`: The field name in the document.
- `<FIELD_TYPE>`: The field type used in this field.
- `<ARG_NAME>`: The argument name to use constructing the field.
- `<ARG_VALUE>`: The argument value to use constructing the field.

For example, `id` field used as a unique key is defined as following:

```
schema:
  id:
    field_type: id
    args:
      unique: true
      stored: true
```

4.2.3 Default Search Field

The query parser uses this as the field for any terms without an explicit field.

```
default_search_field: <FIELD_NAME>
```

- <FIELD_NAME>: Uses this as the field name for any terms without an explicit field name.

For example, uses text field as default search field as following:

```
default_search_field: text
```

4.2.4 Field Types

The field type defines how Cockatrice should interpret data in a field and how the field can be queried. There are many field types included with Whoosh by default, and they can also be defined directly in YAML.

```
field_types:
  <FIELD_TYPE>:
    class: <FIELD_TYPE_CLASS>
    args:
      <ARG_NAME>: <ARG_VALUE>
```

- <FIELD_TYPE>: The field type name.
- <FIELD_TYPE_CLASS>: The field type class.
- <ARG_NAME>: The argument name to use constructing the field type.
- <ARG_VALUE>: The argument value to use constructing the field type.

For example, defines text field type as following:

```
field_types:
  text:
    class: whoosh.fields.TEXT
    args:
      analyzer:
      phrase: true
      chars: false
      stored: false
      field_boost: 1.0
      multitoken_query: default
      spelling: false
      sortable: false
      lang: null
      vector: null
      spelling_prefix: spell_
```

4.2.5 Analyzers

An analyzer examines the text of fields and generates a token stream. The simplest way to configure an analyzer is with a single class element whose class attribute is a fully qualified Python class name.
Even the most complex analysis requirements can usually be decomposed into a series of discrete, relatively simple processing steps. Cockatrice comes with a large selection of tokenizers and filters. Setting up an analyzer chain is very straightforward; you specify a tokenizer and filters to use, in the order you want them to run.

```
analyzers:
  <ANALYZER_NAME>:
    class: <ANALYZER_CLASS>
    args:
      <ARG_NAME>: <ARG_VALUE>
  <ANALYZER_NAME>:
    tokenizer: <TOKENIZER_NAME>
    filters:
      - <FILTER_NAME>
```

- <ANALYZER_NAME>: The analyzer name.
- <ANALYZER_CLASS>: The analyzer class.
- <ARG_NAME>: The argument name to use constructing the analyzer.
- <ARG_VALUE>: The argument value to use constructing the analyzer.
- <TOKENIZER_NAME>: The tokenizer name to use in the analyzer chain.
- <FILTER_NAME>: The filter name to use in the analyzer chain.

For example, defines analyzers using `class`, `tokenizer` and `filters` as follows:

```
analyzers:
  simple:
    class: whoosh.analysis.SimpleAnalyzer
    args:
      expression: "\\w+(\\.?\\w+)*"
      gaps: false
  ngram:
    tokenizer: ngram
    filters:
      - lowercase
```

4.2.6 Tokenizers

The job of a tokenizer is to break up a stream of text into tokens, where each token is (usually) a sub-sequence of the characters in the text.

```
tokenizers:
  <TOKENIZER_NAME>:
    class: <TOKENIZER_CLASS>
    args:
      <ARG_NAME>: <ARG_VALUE>
```

- <TOKENIZER_NAME>: The tokenizer name.
- <TOKENIZER_CLASS>: The tokenizer class.
- <ARG_NAME>: The argument name to use constructing the tokenizer.
- <ARG_VALUE>: The argument value to use constructing the tokenizer.

For example, defines tokenizer as follows:

```
tokenizers:
  ngram:
    class: whoosh.analysis.NgramTokenizer
```

(continues on next page)

(continued from previous page)

```
    args:
      minsize: 2
      maxsize: null
```

4.2.7 Filters

The job of a filter is usually easier than that of a tokenizer since in most cases a filter looks at each token in the stream sequentially and decides whether to pass it along, replace it or discard it.

```
filters:
  <FILTER_NAME>:
    class: <FILTER_CLASS>
    args:
      <ARG_NAME>: <ARG_VALUE>
```

- `<FILTER_NAME>`: The filter name.
- `<FILTER_CLASS>`: The filter class.
- `<ARG_NAME>`: The argument name to use constructing the filter.
- `<ARG_VALUE>`: The argument value to use constructing the filter.

For example, defines filter as follows:

```
filters:
  stem:
    class: whoosh.analysis.StemFilter
    args:
      lang: en
      ignore: null
      cachesize: 50000
```

4.2.8 Example

Refer to the example for how to define schema.

https://github.com/mosuka/cockatrice/blob/master/example/schema.yaml

4.2.9 More information

See documents for more information.

- https://whoosh.readthedocs.io/en/latest/schema.html
- https://whoosh.readthedocs.io/en/latest/api/fields.html
- https://whoosh.readthedocs.io/en/latest/api/analysis.html

4.3 Index management

You need to create an index after starting Cockatrice. Also you can delete indexes that are no longer needed.

4.3.1 Create an index

A schema is required to create an index, you need to put the schema in the request. Create an index by the following command:

```
$ curl -s -X PUT -H "Content-type: text/x-yaml" --data-binary @./conf/schema.yaml␣
↪http://localhost:8080/rest/myindex | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "status": {
    "code": 202,
    "description": "Request accepted, processing continues off-line",
    "phrase": "Accepted"
  },
  "time": 0.08018112182617188
}
```


4.3.2 Get an index

If you created an index, you can retrieve index information by the following command:

```
$ curl -s -X GET http://localhost:8080/rest/myindex | jq .
```

The result of the above command is:


```
{
  "index": {
    "doc_count": 0,
    "doc_count_all": 0,
    "last_modified": 1539674088.2373083,
    "latest_generation": 0,
    "name": "myindex",
    "storage": {
      "files": [
        "_myindex_0.toc"
      ],
      "folder": "/tmp/cockatrice/index",
      "readonly": false,
      "supports_mmap": true
    },
    "version": -111
  },
  "status": {
    "code": 200,
    "description": "Request fulfilled, document follows",
    "phrase": "OK"
  },
  "time": 0.0008370876312255859
}
```


4.3.3 Delete an index

You can delete indexes that are no longer needed. Delete an index by the following command:

```
$ curl -s -X DELETE http://localhost:8080/rest/myindex | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "status": {
    "code": 202,
    "description": "Request accepted, processing continues off-line",
    "phrase": "Accepted"
  },
  "time": 0.0006439685821533203
}
```


4.4 Document management

Once indices are created, you can update indices.

4.4.1 Index a document

If you already created an index named `myindex`, you can indexing a document by the following command:

```
$ curl -s -X PUT -H "Content-Type:application/json" http://localhost:8080/rest/
↪myindex/_doc/1 -d @./example/doc1.json | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "status": {
    "code": 202,
    "description": "Request accepted, processing continues off-line",
    "phrase": "Accepted"
  },
  "time": 0.00015020370483398438
}
```


4.4.2 Get a document

Getting a document from `myindex` by the following command:

```
$ curl -s -X GET http://localhost:8080/rest/myindex/_doc/1 | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "doc": {
    "fields": {
      "contributor": "43.225.167.166",
      "id": "1",
      "text": "A search engine is an information retrieval system designed to help␣
↪find information stored on a computer system. The search results are usually␣
↪presented in a list and are commonly called hits. Search engines help to minimize␣
↪the time required to find information and the amount of information which must be␣
↪consulted, akin to other techniques for managing information overload. The most␣
↪public, visible form of a search engine is a Web search engine which searches for␣
↪information on the World Wide Web.",
```


(continues on next page)

(continued from previous page)

```
      "timestamp": "20180704054100",
      "title": "Search engine (computing)"
    }
  },
  "status": {
    "code": 200,
    "description": "Request fulfilled, document follows",
    "phrase": "OK"
  },
  "time": 0.011947870254516602
}
```

4.4.3 Delete a document

Deleting a document from `myindex` by the following command:

```
$ curl -s -X DELETE http://localhost:8080/rest/myindex/_doc/1 | jq .
```

You can see the result in JSON format. The result of the above command is:

```
{
  "status": {
    "code": 202,
    "description": "Request accepted, processing continues off-line",
    "phrase": "Accepted"
  },
  "time": 6.699562072753906e-05
}
```

4.4.4 Index documents in bulk

Indexing documents in bulk by the following command:

```
$ curl -s -X PUT -H "Content-Type:application/json" http://localhost:8080/rest/
↪myindex/_docs -d @./example/bulk_index.json | jq .
```

You can see the result in JSON format. The result of the above command is:

```
{
  "status": {
    "code": 202,
    "description": "Request accepted, processing continues off-line",
    "phrase": "Accepted"
  },
  "time": 0.00018596649169921875
}
```

4.4.5 Delete documents in bulk

Deleting documents in bulk by the following command:

```
$ curl -s -X DELETE -H "Content-Type:application/json" http://localhost:8080/rest/
↪myindex/_docs -d @./example/bulk_delete.json | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "status": {
    "code": 202,
    "description": "Request accepted, processing continues off-line",
    "phrase": "Accepted"
  },
  "time": 0.00232696533203125
}
```


4.5 Search documents

Once created an index and added documents to it, you can search for those documents.

4.5.1 Searching documents

Searching documents by the following command:

```
$ curl -s -X GET http://localhost:8080/rest/myindex/_search?query=search | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "results": {
    "hits": [
      {
        "doc": {
          "fields": {
            "contributor": "KolbertBot",
            "id": "3",
            "text": "Enterprise search is the practice of making content from␣
↪multiple enterprise-type sources, such as databases and intranets, searchable to a␣
↪defined audience. \"Enterprise search\" is used to describe the software of search␣
↪information within an enterprise (though the search function and its results may␣
↪still be public). Enterprise search can be contrasted with web search, which␣
↪applies search technology to documents on the open web, and desktop search, which␣
↪applies search technology to the content on a single computer. Enterprise search␣
↪systems index data and documents from a variety of sources such as: file systems,␣
↪intranets, document management systems, e-mail, and databases. Many enterprise␣
↪search systems integrate structured and unstructured data in their collections.[3]␣
↪Enterprise search systems also use access controls to enforce a security policy on␣
↪their users. Enterprise search can be seen as a type of vertical search of an␣
↪enterprise.",
            "timestamp": "20180129125400",
            "title": "Enterprise search"
          }
        },
        "pos": 0,
        "rank": 0,
```


(continues on next page)

(continued from previous page)


```
        "score": 1.7234593504967473
      },
      {
        "doc": {
          "fields": {
            "contributor": "Nurg",
            "id": "5",
            "text": "Federated search is an information retrieval technology that␣
↪allows the simultaneous search of multiple searchable resources. A user makes a␣
↪single query request which is distributed to the search engines, databases or other␣
↪query engines participating in the federation. The federated search then aggregates␣
↪the results that are received from the search engines for presentation to the user.␣
↪Federated search can be used to integrate disparate information resources within a␣
↪single large organization (\"enterprise\") or for the entire web. Federated search,␣
↪unlike distributed search, requires centralized coordination of the searchable␣
↪resources. This involves both coordination of the queries transmitted to the␣
↪individual search engines and fusion of the search results returned by each of them.
↪",
            "timestamp": "20180716000600",
            "title": "Federated search"
          }
        },
        "pos": 1,
        "rank": 1,
        "score": 1.7042117821338238
      },
      {
        "doc": {
          "fields": {
            "contributor": "Aistoff",
            "id": "2",
            "text": "A web search engine is a software system that is designed to␣
↪search for information on the World Wide Web. The search results are generally␣
↪presented in a line of results often referred to as search engine results pages␣
↪(SERPs). The information may be a mix of web pages, images, and other types of␣
↪files. Some search engines also mine data available in databases or open␣
↪directories. Unlike web directories, which are maintained only by human editors,␣
↪search engines also maintain real-time information by running an algorithm on a web␣
↪crawler. Internet content that is not capable of being searched by a web search␣
↪engine is generally described as the deep web.",
            "timestamp": "20181005132100",
            "title": "Web search engine"
          }
        },
        "pos": 2,
        "rank": 2,
        "score": 1.619574615564863
      },
      {
        "doc": {
          "fields": {
            "contributor": "43.225.167.166",
            "id": "1",
            "text": "A search engine is an information retrieval system designed to␣
↪help find information stored on a computer system. The search results are usually␣
↪presented in a list and are commonly called hits. Search engines help to minimize␣
↪the time required to find information and the amount of information which must be␣
↪consulted, akin to other techniques for managing information overload. The most␣
↪public, visible form of a search engine is a Web search engine which searches for␣
↪information on the World Wide Web.",
```


(continues on next page)

(continued from previous page)


```
            "timestamp": "20180704054100",
            "title": "Search engine (computing)"
          }
        },
        "pos": 3,
        "rank": 3,
        "score": 1.5951006619362313
      },
      {
        "doc": {
          "fields": {
            "contributor": "Citation bot",
            "id": "4",
            "text": "A distributed search engine is a search engine where there is no␣
↪central server. Unlike traditional centralized search engines, work such as␣
↪crawling, data mining, indexing, and query processing is distributed among several␣
↪peers in a decentralized manner where there is no single point of control.",
            "timestamp": "20180930171400",
            "title": "Distributed search engine"
          }
        },
        "pos": 4,
        "rank": 4,
        "score": 1.5232201764110038
      }
    ],
    "is_last_page": true,
    "page_count": 1,
    "page_len": 5,
    "page_num": 1,
    "total": 5
  },
  "status": {
    "code": 200,
    "description": "Request fulfilled, document follows",
    "phrase": "OK"
  },
  "time": 0.010915756225585938
}
```


4.6 Cluster management

You already know to start Cockatrice in standalone mode, but that is not fault tolerant. If you need to increase the fault tolerance, bring up Cockatrice cluster.

4.6.1 Create a cluster

Cockatrice is easy to bring up the cluster. You can bring up 3-node cluster with static membership by following commands:

```
$ cockatrice server --http-port=8080 --bind-addr=127.0.0.1:7070 --peer-addr=127.0.0.
↪1:7071 --peer-addr=127.0.0.1:7072 --index-dir=/tmp/cockatrice/node1/index --dump-
↪file=/tmp/cockatrice/node1/raft/data.dump
$ cockatrice server --http-port=8081 --bind-addr=127.0.0.1:7071 --peer-addr=127.0.0.
↪1:7070 --peer-addr=127.0.0.1:7072 --index-dir=/tmp/cockatrice/node2/index --dump-
↪file=/tmp/cockatrice/node2/raft/data.dump
$ cockatrice server --http-port=8082 --bind-addr=127.0.0.1:7072 --peer-addr=127.0.0.
↪1:7070 --peer-addr=127.0.0.1:7071 --index-dir=/tmp/cockatrice/node3/index --dump-
↪file=/tmp/cockatrice/node3/raft/data.dump
```

Above example shows each Cockatrice node running on the same host, so each node must listen on different ports. This would not be necessary if each node ran on a different host.

So you have a 3-node cluster. That way you can tolerate the failure of 1 node.

You can check the cluster with the following command:

```
$ curl -s -X GET http://localhost:8080/rest/_cluster | jq .
```

You can see the result in JSON format. The result of the above command is:


```
{
  "cluster_status": {
    "commit_idx": 4,
    "enabled_code_version": 0,
    "last_applied": 4,
    "leader": "127.0.0.1:7070",
    "leader_commit_idx": 4,
    "log_len": 3,
    "match_idx_count": 2,
    "match_idx_server_127.0.0.1:7071": 4,
    "match_idx_server_127.0.0.1:7072": 4,
    "next_node_idx_count": 2,
    "next_node_idx_server_127.0.0.1:7071": 5,
    "next_node_idx_server_127.0.0.1:7072": 5,
    "partner_node_status_server_127.0.0.1:7071": 2,
    "partner_node_status_server_127.0.0.1:7072": 2,
    "partner_nodes_count": 2,
    "raft_term": 1,
    "readonly_nodes_count": 0,
    "revision": "2c8a3263d0dbe3f8d7b8a03e93e86d385c1de558",
    "self": "127.0.0.1:7070",
    "self_code_version": 0,
    "state": 2,
    "unknown_connections_count": 0,
    "uptime": 159,
    "version": "0.3.4"
  },
  "status": {
    "code": 200,
    "description": "Request fulfilled, document follows",
    "phrase": "OK"
  },
  "time": 0.0001499652862548828
}
```


Recommend 3 or more odd number of nodes in the cluster. In failure scenarios, data loss is inevitable, so avoid deploying single nodes.

Once cluster is created, you can create indices. let's create an index to 127.0.0.1:8080 by the following command:

```
$ curl -s -X PUT -H "Content-type: text/x-yaml" --data-binary @./conf/schema.yaml␣
↪http://localhost:8080/rest/myindex | jq .
```

If the above command succeeds, same index will be created on all the nodes in the cluster. Check your index on each nodes.

```
$ curl -s -X GET http://localhost:8080/rest/myindex | jq .
$ curl -s -X GET http://localhost:8081/rest/myindex | jq .
$ curl -s -X GET http://localhost:8082/rest/myindex | jq .
```

Let's index a document to 127.0.0.1:8080 by the following command:

```
$ curl -s -X PUT -H "Content-Type:application/json" http://localhost:8080/rest/
↪myindex/_doc/1 -d @./example/doc1.json | jq .
```

If the above command succeeds, same document will be indexed on all the nodes in the cluster. Check your document on each nodes.

```
$ curl -s -X GET http://localhost:8080/rest/myindex/_doc/1 | jq .
$ curl -s -X GET http://localhost:8081/rest/myindex/_doc/1 | jq .
$ curl -s -X GET http://localhost:8082/rest/myindex/_doc/1 | jq .
```

4.7 RESTful API Reference

4.7.1 Index APIs

The Index API is used to manage individual indices.

Get Index API

The Get Index API allows to retrieve information about the index. The most basic usage is the following:

```
GET /rest/<INDEX_NAME>
```

- <INDEX_NAME>: The index name.

Create Index API

The Create Index API is used to manually create an index in Cockatrice. The most basic usage is the following:

```
PUT /rest/<INDEX_NAME>?sync=<SYNC>
---
schema:
  id:
    field_type: id
    args:
      unique: true
      stored: true
...
```

- <INDEX_NAME>: The index name.

- <SYNC>: Specifies whether to execute the command synchronously or asynchronously. If `True` is specified, command will execute synchronously. Default is `False`, command will execute asynchronously.
- Request Body: YAML formatted schema definition.

Delete Index API

The Delete Index API allows to delete an existing index. The most basic usage is the following:

```
DELETE /rest/<INDEX_NAME>?sync=<SYNC>
```

- <INDEX_NAME>: The index name.
- <SYNC>: Specifies whether to execute the command synchronously or asynchronously. If `True` is specified, command will execute synchronously. Default is `False`, command will execute asynchronously.

4.7.2 Document APIs

Get Document API

```
GET /rest/<INDEX_NAME>/_doc/<DOC_ID>
```

- <INDEX_NAME>: The index name.
- <DOC_ID>: The document ID to retrieve.

Index Document API

```
PUT /rest/<INDEX_NAME>/_doc/<DOC_ID>?sync=<SYNC>
{
  "name": "Cockatrice",
  ...
}
```

- <INDEX_NAME>: The index name.
- <DOC_ID>: The document ID to index.
- <SYNC>: Specifies whether to execute the command synchronously or asynchronously. If `True` is specified, command will execute synchronously. Default is `False`, command will execute asynchronously.
- Request Body: JSON formatted fields definition.

Delete Document API

```
DELETE /rest/<INDEX_NAME>/_doc/<DOC_ID>?sync=<SYNC>
```

- <INDEX_NAME>: The index name.
- <DOC_ID>: The document ID to delete.
- <SYNC>: Specifies whether to execute the command synchronously or asynchronously. If `True` is specified, command will execute synchronously. Default is `False`, command will execute asynchronously.

Index Documents API

```
PUT /rest/<INDEX_NAME>/_docs?sync=<SYNC>
[
  {
    "id": "1",
    "name": "Cockatrice"
  },
  {
    "id": "2",
  ...
]
```

- <INDEX_NAME>: The index name.
- <SYNC>: Specifies whether to execute the command synchronously or asynchronously. If `True` is specified, command will execute synchronously. Default is `False`, command will execute asynchronously.
- Request Body: JSON formatted documents definition.

Delete Documents API

```
DELETE /rest/<INDEX_NAME>/_docs?sync=<SYNC>
[
  "1",
  "2",
  ...
]
```

- <INDEX_NAME>: The index name.
- <SYNC>: Specifies whether to execute the command synchronously or asynchronously. If `True` is specified, command will execute synchronously. Default is `False`, command will execute asynchronously.
- Request Body: JSON formatted document ids definition.

4.7.3 Search APIs

Search API

```
GET /rest/<INDEX_NAME>?query=<QUERY>&search_field=<SEARCH_FIELD>&page_num=<PAGE_NUM>&
↪page_len=<PAGE_LEN>
```

- <INDEX_NAME>: The index name to search.
- <QUERY>: The unicode string to search index.
- <SEARCH_FIELD>: Uses this as the field for any terms without an explicit field.
- <PAGE_NUM>: The page number to retrieve, starting at `1` for the first page.
- <PAGE_LEN>: The number of results per page.

4.7.4 Cluster APIs

Cluster API

```
GET /rest/_cluster
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search