Simple Search Engine

Implementation documentation

04/22/18
Naveen Mysore

Problem Statement

Build a simple search engine which supports two kind of operations index and query.
* index : insert a document into the database `in: index <document_id> [tokens]`
* query : retrieve documents based on a query `in: query <expression>`

Index command example:

```
in: index 1 salt pepper fish salt pepper milk
in: index 12 fish salt pepper garlic
in: index 12 salt pepper milk
```

*note : duplicating index will over write existing document

Query command example:

```
in: query ((salt|pepper)&(salt&butter))
in: query (butter&pepper)
in: query salt
in: query salt|pepper|milk
```

Response:

Upon document entry response status will be returned with either success or failure.

```
out: index ok 1
out: index error <error message>
```

Similarly for query command if upon successfully retrieval all documents matching the query will be returned else error with message will be returned.

```
out: results 3 1 2 4
out: query error <error message>
```

*note: *response for query results are ranked based on tf-idf ranking. This was not asked on the problem statement but I have implemented it as additional feature.*

Implementation

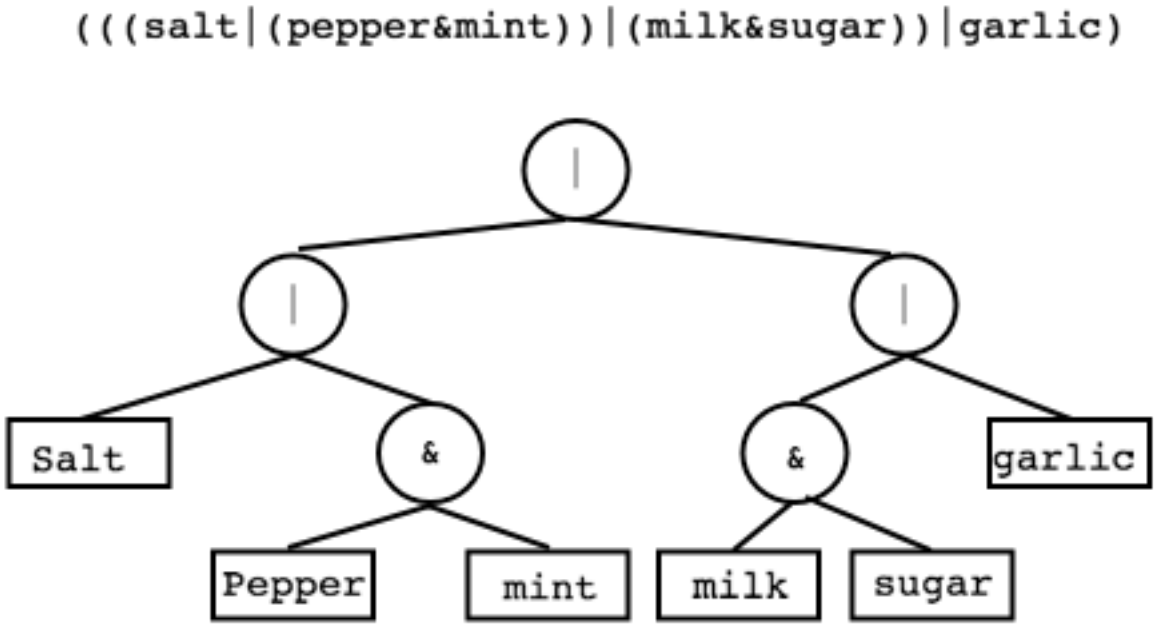
DATABASE

Database is implemented as a simple python dictionary with stored key value pairs. document id's are treated as keys and tokens as raw strings are treated as values.

QUERY

Since query can be nested expressions like `(((salt|(pepper&mint))|(milk&sugar))|garlic)`

In order to retain the hierarchy of query exceptions and results a binary tree structure is used to create execution order.



And this is implemented by `build_query_tree` function and results are reduced by `reduce_tree` function. Documents are retrieved from the database for each tokens and functional programming techniques with Map, Reduce, filter functions are used to logically reduced results at different stages.

In order to parse the query expression two stacks are used (term_stack, operation_stack). Each character is checked sequentially if “|” or “&” character is encountered then its pushed into operation_stack. If valid tokens are encountered like salt then its pushed into term_stack. When valid operation is reached ex: pepper&mint then this operation is performed by retrieving all documents having pepper and mint together and this result is pushed back into term_stack. Then upon encountering next valid operation ie. salt | (*) results from previous execution is popped from the term_stack and new execution salt | ([doc_list]) is performed which yields us all documents having either only salt or (pepper and mind) and this new resulting document list is pushed back to term_stack. This steps are recursively performed until all nodes are visited.

RANKING

Once we get results with a list of all documents satisfying the query, ranking is done based on how well these documents match our query. For example if a document has many occurrences of token "*milk*" and user queries for milk then this document should appear as top result. On the other hand a token occurs multiple time across all documents like "*water*" then this token has less values and when used into nested queries like `(salt|pepper)|water` we need to rank accordingly hence term frequency inverse document frequency technique is used for ranking these documents.

Once we have retrieved all the documents using functional programming techniques number of times each token appears in a document is counted

$$tf=1+\log(f_{t,d}) \;\; V\ terms, documents$$

If a token appears across all or many documents then it carries less weightage on the hand if a token appears only in few documents then it means is a rear token and hence it carried more weightage and this can be computed by inverse document frequency.

$$idf=\log\left(\frac{D}{f_{d,t}}\right) V\ terms, documents$$

Where D is total number of documents and f d,t is number of documents matching a token.

Then for all tokens we computer tf-idf

$$tf\text{-}idf = tf \cdot id$$

Next we need to check how well our query matches these documents and this can be done by comparing net tf-idf values for query terms.

$$\mathrm{S}co\mathrm{re}(q, d) = \sum_{t \in q}^{\|} tf\text{-}idf_{t,d}$$

And finally we sort results based on score values and return them.