Homework 1 Report

103062512 徐丞裕

How to compile and execute your program?

I've written several scripts to run the program automatically.

```
# compile the source files and jar them
# (requires maven build tool)
# pre-built jar is located at /prebuild
$ script/build

# copy input files to HDFS
$ script/setup

# build inverted indexes
$ script/index

# perform full text search using the indexes
$ script/query
=> {enter your query here}
```

Design: explain your algorithm

Building index:

1. Setup
   - Build a list of file names so we can get the document ID based on the list (use `indexOf`).
2. Map
   - Key: a tuple consist of (`docID` , `term`).
   - Value: a tuple consist of (`docID` , `termFreq` , array of `offsets`).

   `term` s are generated using the `Analyzer` or Apache Lucene.

   Special characters are filtered out using a regular expression: `[^a-z0-9]`

   `term` s are also down cased to support case-insensitive queries.

The reason of duplicating `DocID` into values is to make the collection process easier in the reduce phase, since we'll need the `DocID` for each `term`.

3. Combine
   In this phase values are grouped by `term` and values of `offsets` are joint together to create an array of `offsets`.

4. Reduce

   In this phase, the values are grouped by `term` and we join the `(docID, termFreq, offsets)` of each document to create our final inverted indexes.

Retrieval: Calculating Score

1. Setup

   - Query processing:

     - downcase
     - remove special characters
     - check if there's an `or` keyword
     - split query into tokens using whitespace

   - Get #documents (for calculating inverted document frequency)

2. Map

   - Key: `docID`
   - Value: an array consists of multiple tuples (`term` , `idf` , an array of `offsets`)

3. Reduce

   - Output key: (`docID` , `score`)
   - Output value: an array of tuples (`term` , `offsets`)

   In this phase, values are grouped by `docID` in order to calculate the similarity score of query and relevant documents.

   Note here I use the cosine similarity as the score of query and each document.

   The basic idea of this is treating the query as a document and compare it with existing documents.

Retrieval: Sort by Score

Since Hadoop API only support one reduce phase for each job, we'll need to dispatch another job if we'd like to sort the results by their scores.

The calculation here is quite straightforward. We only need to sort the records by scores.

Answers to Questions

How many #phases you used to run `mapreduce` in Inverted Index Is there any other way to do it?

One. There're other ways to do it, one possible solution is creating the `(term, docID, offsets)` in one task and combine them into the final result we want in another task.

What's the pros and cons?

One phase:
- pros: simple, less I/O
- cons: error prone

Multiple phases:
- pros: fault tolerance (since we can start from the intermediate steps without start all over again).
- cons: more I/O

What's your extension?

1. case insensitive query
2. `and` query

What's the most difficult part in your implementation?

The most difficult part of my implementation is supporting `and` query, since we need to ensure that the document contains all of the terms and the distance of terms should not be too large (or it makes no sense).

How do you filter those useless notation?

Using regular expression: `[^a-z0-9]`.

If we need to search these special notations, how to modify your filter?

One possible solution is that building indexes for both filtered and non-filtered terms.

For example, the token `wouldn't` will be converted to both `wouldnt` and `wouldn't`.

Possible Enhancements

1. Consider the position of the terms when calculating scores.
2. Consider the distances of the terms when calculating scores for `and` query.