Cloud Programming Homework 3: Search Engine

103062512 徐丞裕

Instructions

Indexing

1. Build the indexer program:

```Bash
$ pwd
# => ~/hw3/HW3_103062512_index
$ script/build
```

2. Perform preprocessing, PageRank calculation, inverted index building:

```Bash
$ pwd
=> ~/hw3/HW3_103062512_index
$ script/index /shared/HW2/sample-in/input-1G hw3-1G
```

3. Migrate data to HBase:

```Bash
$ pwd
=> ~/hw3/HW3_103062512_index
$ script/migrate hw3-1G
```

Querying

1. Build the query program:

```Bash
$ pwd
# => ~/hw3/HW3_103062512_query
$ script/build
```

2. Perform a query:

```Bash
$ pwd
=> ~/hw3/HW3_103062512_query
$ script/query "cloud programming"
```

Implementation Details

1. Preprocessing

   Implemented in `HW3_103062512_index/src/main/scala/preprocess/Worker.scala`.

   It

   - Converts `<title>` to a numerical ID
   - Creates mapping from `title` to ID and vice versa
   - Removes XML syntax elements in `<text>`

2. Index building

   Instead of using code of the first assignment, I rewrite a more suitable version for this application.

   Since it's implemented on Spark, it's much more faster than the MapReduce version.

   The `idf` is pre-calculated and saved to the output file, so we don't have to re-calculate it in the query stage.

   Also, term frequency is normalized by document, which avoids bias for larger documents.

3. Migration

   I use this command to import data from HDFS to HBase:

```Bash
$ hbase \
  org.apache.hadoop.hbase.mapreduce.ImportTsv \
  -Dimporttsv.columns=HBASE_ROW_KEY,column_family \
  "table_name" /path/to/hdfs/file
```

   I four tables:

ID to title mapping

DocID	Title
9413	Scala

Inverted Index

Word	IDF	Occurrences
Cool	0.2	41;0.1;10,20/13;0.01;30,513

   (each occurrences is in format `docID;term-freq;offsets`)

PageRank

DocID	Score
9413	0.02

Document

DocID	Score
9413	Scala is a ...

4. Querying

   Using HBase Java API to load necessary data to the program and filtering out data that we're not interested in.

   The process is as follows:

   1. For each word we want to query, load the inverted index to memory.
   2. Calculate TF-IDF vectors for each document.
   3. Compute cosine similarities between each document and the query.
   4. Load PageRank for each document.
   5. Sort documents by `cosSim * 0.7 + rank * 0.3` descending.
   6. For fragments in each document, the one with larger `idf` is shown before the one with smaller `idf`s.

   The reason why I weigh the cosine similarity more is that we care about the content.

   And the reason why sorting fragments using `idf` is we want to show the rarest words first, which are usually our point of interests (for searching some infrequent terms).

Experience & Conclusion

Dealing with formats is the most painful part in this assignment.

It's hard to build indexes that are not fragile for any format of the document (weird titles, special characters, etc).