Performance Improvement of Search for Large Data

Bhumi D. Shah[1], Mehul C. Parikh[2], Rahevar Mrugendrasinh L.[3]

[1] ME student, Department of Computer Science and Engineering, GEC Modasa, Gujarat, India
[2] Professor, Department of Computer Science and Engineering, GEC Modasa, Gujarat, India
[3] Professor, Department of Computer Science and Engineering, Charusat Changa, Gujarat, India

ABSTRACT

Indexes are essential to provision of search over massive collections of text information. This paper, present an optimized index construction strategy for distributed file system search engine when files are store on heterogeneous network of computer. Index construction can be worn out 2 phase: apply analyzer rule and index compression severally. B5 approach is used for fast index construction. For efficient query evolution Inverted Index Data structure is employed. In this paper focus is given to the Distributed File system search. As one of the most result of this paper demonstrate the time complexity and space complexity of indexed file.

Keyword : - *Information Retrieval, File System, Distributed File system, Index data structure, index compression.*

---

1. INTRODUCTION

The size and rate of growth of today's text Collection bring new challenges for index construction. Building associate index for an outsized text Collection that is distributed on different node would possibly involve the management of a plenty of distinct words and occurrences of words in text.

Desktop file System search engine has got to handle large heterogeneous collections of documents like word file (.doc), pdf file (.pdf) and so on that demand specialized indexing techniques to allow fast retrieval to documents searched by the users. [1].

for using Blocked Sort-Based indexing with this methodology it creates optimized index for Desktop File System [2].

When data is too large and store on different node indexing metadata with file content is massive so the space and time cost is increases.

Literature survey illustrate, there has been no optimization during Index construction to reduce the space cost, time cost and I/O frequency for distributed file on different node.

2. RELATED WORK

For querying of large collections, many different sorts of index data structure have been described from that 2 most liked data structures are: Inverted Index and Signature file [3]

Inverted Index: The indexing method starts by tokenizing the input documents and forming a list of <term, doc> pair and inserts these tokens into an inverted list. Token provide the link between queries and documents. The list is sorted lexicographically across terms.

Signature file: Signature files are a probabilistic indexing method. For each term, s bits are set in a signature of w bits. The term descriptors for the terms that appear in every document are unit superimposed (i.e., OR'ed together) to obtain a document descriptor, and the document descriptors are then stored in a signature file of size wN bits for N Documents.

Signature File have significant disadvantages [4]
Signature files are not easier to construct and maintain, they require additional disk access for short queries. There is no sensible approach of using signature files for handling ranked queries or for identifying phrases.

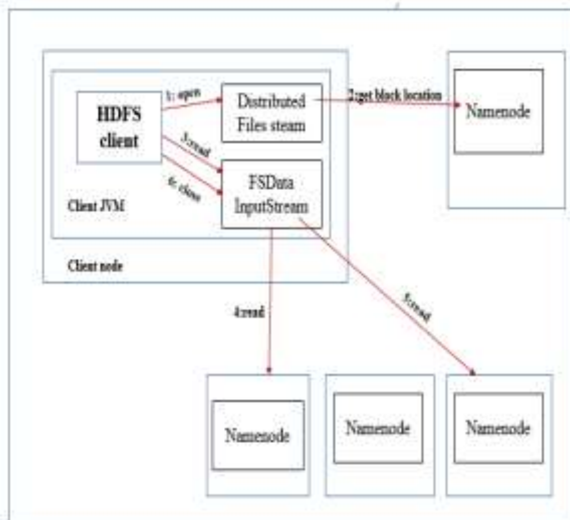
Comparison of Inverted Index and Signature File [5]

TYPES OF QUERY	INVERTED INDEX	SIGNATURE FILE
Term query	Yes	Yes
Range query	Yes	No
Prefix query	Yes	Yes
Boolean query	Yes	Yes
Phrase query	Yes	No
Wild card query	Yes	No
Fuzzy query	Yes	No
Facet query	Yes	No

2.1 Hadoop Distributed File System

Hadoop File System was developed using distributed filing system style. It is run on commodity hardware. Unlike alternative distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware.
HDFS holds very massive quantity of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are hold on redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications obtainable to parallel process.

2.2 HDFS Read and Write Operations:



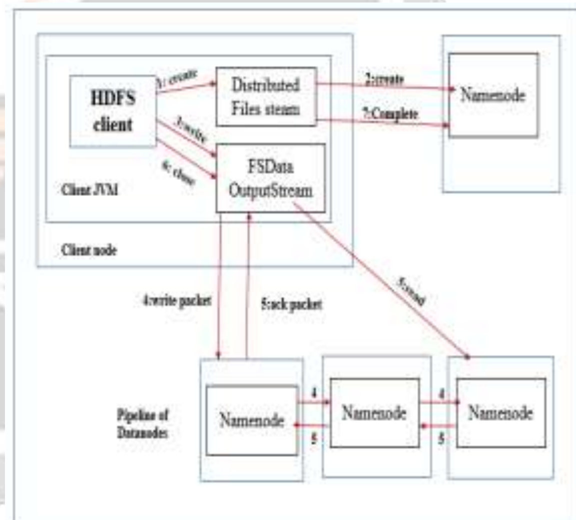


Figure 1: HDFS Read Operation **Figure 2: HDFS Write Operation**

2.2.1 HDFS Read Operation

Step 1: First the Client can open the file by giving a call to open () methodology on File System object, which for HDFS is an instance of DistributedFileSystem class.
Step 2: DistributedFileSystem calls the Namenode, using RPC (Remote Procedure Call), that used to determine the locations of the blocks for the primary few blocks of the file. For each block, the namenode returns the addresses of

all the data nodes that have a duplicate of that block. Client will interact with respective data nodes to read the file. Namenode also offer a token to the client that it shows to data node for authentication.
The DistributedFileSystem returns an object of FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from FSDataInputStream in flip wraps a DFSInputStream, which manages the data node and namenode I/O
Step 3: The client then calls read () on the stream. DFSInputStream, which has keep the data node addresses for the primary few blocks within the file, then connects to the first closest datanode for the 1st the primary block within the file.
Step 4: Data is streamed from the datanode back to the client, which calls read () repeatedly on the stream.
Step 5: When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the subsequent block. This happens transparently to the client, which from its purpose of read is simply reading never-ending stream
Step 6: Blocks are read in order, with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the subsequent batch of blocks as needed. When the client has finished reading, it calls close () on the FSDataInputStream

2.2.2 HDFS Write Operation

Step 1: The client produces the file by calling create () method on DistributedFileSystem.
Step 2: DistributedFileSystem makes an RPC call to the namenode to create a new file in the file system's namespace, with no blocks associated with it.
The namenode performs various checks to create positive the file doesn't exist already which the client has the correct permissions to make the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. TheDistributedFileSystem returns an FSDataOutputStream for the client to bigine writing data to.
Step 3: As the client writes data, DFSOutputStream splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by choosing an inventory of appropriate datanodes to store the replicas. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode within the pipeline, which stores the packet and forwards it to the second datanode in the pipeline.
Step 4: Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.
Step 5: DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the acknowledgment queue. A packet is removed from the acknowledgment queue only it's has been acknowledged by all the datanodes within the pipeline.
Step 6: When the client has finished writing data, it calls close () on the stream.
Step 7: This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete The namenode already knows that blocks the file is created from, so it only has to wait for blocks to be minimally replicated before returning with success.

2.3 Solar Search engine:

Apache Solr is a scalable and ready-to-deploy open source full-text search engine powered by Lucene. It offers key features like trilingual keyword searching, faceted search, intelligent matching, content clustering, and relevancy weighting right out of the box.

3. PROPOSED SCHEME FOR INDEXING

The retrieval system described in this paper is the B5 file system search engine. B5 is similar to other file system search engine, such as Microsoft share point server, Apple Spotlight, or Recall.
1. Before Applying Analyzer Rule and Build Index, Extract Plain Text from Document Collection (. pdf, docs, ppt.. Etc)
2. Analyze Rules are applying
Rule1: Whitespace Analyzer merely splits text into tokens on whitespace characters and makes no alternative effort to normalize the tokens.

Rule2: Character Normalization, Character Normalization improves recall. Means that additional documents are retrieved though the documents don't precisely match the query. i. e. Case Normalization: finding documents with ORACLE when looking out for oracle.
Accent removal: finding documents that contain e` when looking out for e.
Rule3: Eliminating stop words (i.e. a, an, these, aren't)
Rule4: Stemming each token to its root (i.e. gone, going, goes -go) [6]
3. Build Inverted List B5 uses inverted file [7] [8] as its main index data structure. All posting lists contains the exact position of all occurrences of a term
6.Apply Build Index index construction algorithm
Posting list are accumulated in an in-memory index, using a hash table with move Each posting list to-front heuristics [8]. When the entire assortment has been indexed, all terms are sorted in lexicographical order and in-memory data are written to disk, forming an inverted file.

```
Build Index (input Tokenizer) ==
Position ← 0
  while inputTokenizer.hasNext () do
      T ← inputTokenizer.getNext () Do
      Obtain dictionary entry for T;
      create new entry if necessary.
      Append new posting to T's posting list.
      position ← position + 1
sort all dictionary entries in lexicographical order
while (all documents have not been processed)
n←n+l
      block←ParseNextBlockO
      WriteBlock to Disk (block, fn)
Return
```

7. Index Compression Technique
consists of a number of individual postings.
Each posting holds a document identifier (docno.) and the frequency (i. e., count) of the term in that document [9] [10]. Before indexed data written to disk, Variable Byte Integer compression techniques applied over posting list to reduce the disk space and disk traffic.
Index Compression Steps are given below:
1) convert the number to binary (e.g. 5 becomes -> 101)
2) break it into blocks of 7 bits, starting from the smallest amount Vital bit.
3) each 7 bits block gets a 0 in front of it (so it will form a byte), except for the last block which gets a 1 prep ended (this announces we're at the end of the blocks set for that int)
To be more precise, here's an example :-
The first bit is set to 1 for the last byte of the encoded gap, 0 otherwise a gap of size 5 is encoded as 10000101

docIDs	824	829	215406
gaps		5	214577
VBcode	00000110 10111000	10000101	00001101 00001100 10110001

The posting lists for the REUTERS collection are compressed to 116 MB with this technique (original size: 250 MB)
The idea of representing gaps with variable integral number of bytes can be applied with units that differ from 8 bits Larger units can be processed (decompression) faster than little ones, but are less effective in terms of compression rate
8. Index Boosting
The score is computed for every document (d) matching each term (t) in a query (q).

$$\sum_{t \ in \ q} \big(tf(t \ in \ d) \times idf \times boost(t.fieldin \ d) \times lengthNorm(t.fieldin \ d)\big) \times coord(q,d) \times queryNorm(q)$$

tf (t in d): Term frequency factor for the term (t) in the document (d), i.e. how several times the term t happens within the document.

boost (t. field in d): Field & Document boost, as set during indexing. You may use this to statically boost certain fields and certain documents over others.
length Norm (t. field in d): Normalization value of a field, given the number of terms within the field. This value is computed throughout indexing and stored in the index norms. Shorter fields (fewer tokens) get a bigger boost from this factor.
coord (q, d): Coordination factor, based on the number of query terms the document contains. The coordination factor gives an AND-like boost to documents that contain more of the search terms than other document.
query Norm(q): Normalization value for a query, given the sum of the squared weights of each of the query terms.
Boost factors are built into the equation to let you affect a query or field's influence on score

4. PERFORMANCE STUDY

Data Set: For our experiments, we used text collection, having different size, which contains about 1500 pdf documents.

System setup: The experiments were conducted on PC based on an Intel core i-5 CPU@2.40GHZ processor with 4GB main memory and 500GB, 7200-rpm SATA hard drive. The operating system as windows 8.

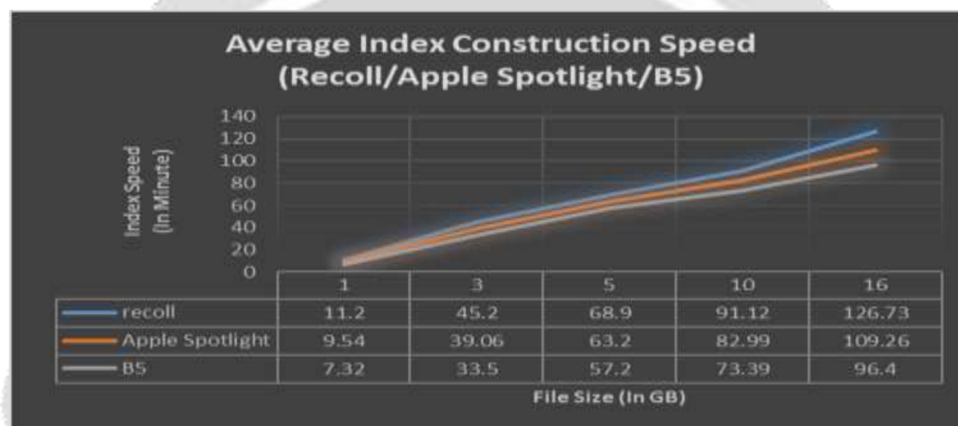


Figure 3: Compare the average index construction speed with Apple spotlight, Recoll and B5

Figure 3 shows the result of average index construction time; no major changes monitor in Index size for small document collection but when document collection growing then index size is much wider. For the 16GB file size, average construction time taken by Recoll and Apple Spotlight is 126.73 min and 109.26 min respectively. While B5 average construction time, 96.4 min, which is lower than recoll and Spot light search system

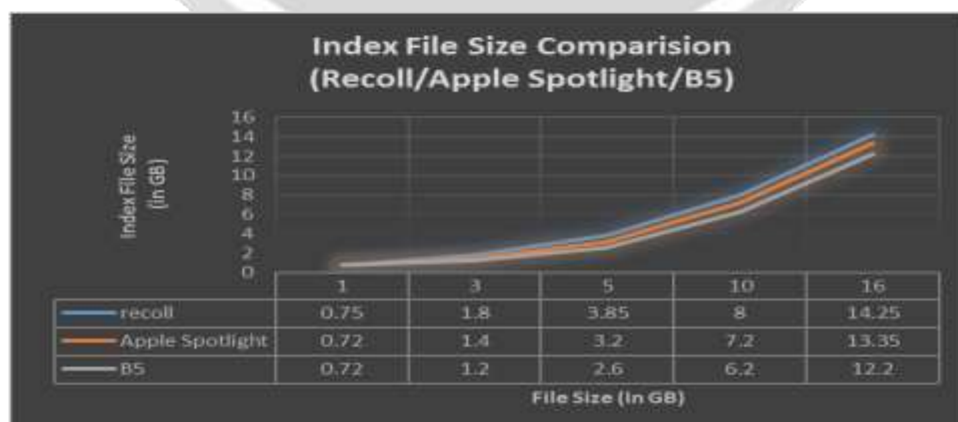


Figure 4: Compare index File size with Apple spotlight, Recoll and B5

Figure 4 show the average index File Size taken by proposed indexing algorithm B5 with compression over posting list and B5 without compression. With using compression technique no major changes for small document collection but when document collection growing then performance gap is Little increase.

Figure 5: Index File Size for B5 with compression and B5 without compression

figure 5 show the average index File Size taken by proposed indexing algorithm B5 with compression over posting list and B5 without compression. With using compression technique no major changes for small document collection but when document collection growing then performance gap is Little increase.

CONCLUSION

Performance Improvement of search for Large data approach discussed in this work. In this experiment different distributed indexing techniques like recoll, apple spotlight is compared with proposed B5 techniques in terms of index construction speed and size of index. This experiment show that b5 approach give better result for index construction speed and index size when documents are large.

6. REFERENCES

[1] S. B"uttcher and C. L. A. Clarke. Indexing time vs. query time: tradeoffs in dynamic information retrieval systems. In CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management, pages 317-318, New York, NY, USA,2005. ACM.

[2] Rehvar Mrugendrasinh L and Mehul C. Parikh. Optimized Index Construction For Large Text Collections Using Blocked sort-based Indexing, IEEE International Conference on Advanced Communication Control and Computing Technologies (lCACCCT) 2014.

[3] William B. Frakes, Ricardo Baeza-Yates Information Retrieval Data Structures & Algorithms (lst ed.), Prentice Hall.

[4] Justin Zobel and Alistair Moffat. Inverted Files for Text Search Engines, ACM Transaction on Database System 23(4), pages 453-490

[5] Justin Zobel, Alistair Moffat and Kotagiri Ramamohanrao. Inverted Files Vs Signature File for Text Indexing. Journal ACM Transactions on Database Systems, Pages 453-490, New York NY, USAACM

[6] Lovins, J. 1968. Development of a stemming algorithm. Mechanical Translation and Computation II, 1-2, 22.

[7] J. Zobel, S. Heinz, and H.E. Williams. In-Memory Hash Tables for accumulating Text Vocabularies. Information Processing Letters, 80(6),2001.

[8] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. Jour. of the American Society for Information Science and Technology, pages 713-729, 2003.

[9] j.wan and s.pan. Performance Evaluation of Compressed Inverted Index in Lucene, IEEE International Conference on Research Challenges in Computer Science (ICRCCS) 2009.

[10] Marcin Zukowski, S'andor H'eman, Niels Nes, Peter Boncz "SuperScalar RAM-CPU Cache Compression", ICDE '06 in Proceeding of the 22nd International Conference, Atlanta, April 2006.