ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 2, February 2015

Building Inverted Index and Search Engine using Apache Lucene

Gita Veer[1], Poonam Rathod[2], Poonam Sinare[3], Prof. R. B. Singh[4]

B.E., Computer Engg, SIT, Lonavala, Maharashtra[1,2,3]

Computer Department, SIT, Lonavala, Maharashtra[4]

Abstract: We describe data structures and an update strategy for e practical implementation of inverted indexes. The context of our discussion is the construction of a dedicated index engine for a distributed full-text information retrieval system but the results have wider applications. Retrieval operations require a single disk access per query term. The online update strategy guarantees the consistency of on-disk data structures. Index compression integrates smoothly.

Keywords: NNquery, Inverted Index, Lucene, Elastic Search.

I. INTRODUCTION

Our general concern is the construction of a distributed full-text information retrieval system.The basic architecture consists of a group of LAN-connected processors each managing its own separate disk and memory. Individual processors act as either text servers, storing documents and servicing requests for portions of these documents or as index engines identifying the portions of documents that match client –generated search criteria. To external clients the group of machines appears to be a single large information retrieval system.A front end processor the Marshaller /Dispatcher coordinates the activities of the group of processors interacting with client applications dispatching queries to the index engines and text servers marshalling query results and returning the results to clients.

II. INVERTED INDEXES

Our specific concern is the data structure design and update strategy used by the index engines. The basic data abstraction implemented by an index engine is an inverted index. File structures based on inverted indexes are standard for implementing information retrieval systems. An inverted index is a function that maps index terms into positions in documents where the terms occur. Index terms are typically words, , but may include document markup tags and other structural information of importance to database clients.

III. PRACTICAL ISSUES

In an operational environment there are a number of practical issues to be considered when implementing inverted lists.

Retrieval Response: Retrieval operations far-out number update operations. Querying the retrieval system is the primary operation used by external clients and response time is of utmost importance. The mapping of an index term into its postings list must require as few disk accesses as possible. Ideally a single disk access would be sufficient to translate any term independent of the size of the dictionary and postings file and independent of the size of the postings list for the particular word.

Update Throughput: Updates are usually additions of new documents. Occasionally deletion of documents and addition and modification of indexing may be required. Since update is primarily a maintenance function rather than an external client service, update throughput, not response time is of importance. Index Compression will increase the amount of dictionary and postings data that can be stored on available disk. Since compression and decompression techniques operate by linearly processing a range of data, this property creates a potential decrease in retrieval response time if random access into the data is limited.

Consistency: The database must be maintained in a consistent state at all times. For example, if a failure occurs during update of the postings file, the dictionary must not be left pointing to an incorrect range of postings.

IV.EXISTING APPROACHES

Most discussions of inverted indexes for information retrieval assume that the file structures are static created by an initial database load operation and not modified thereafter. These file structures generally require multiple disk accesses for term translation. Discussions of updatable inverted indexes generally adopt an append only model of update. Tries, hashing and the ubiquitous B-Tree can all be used to implement an updatable dictionary. An updatable postings file can be implemented using a variety of free space management techniques. The postings for a particular term may be maintained in chained buckets. A new bucket is added to the chain each time a document append causes a bucket to overflow. Alternately, postings may be stored in contiguous extents with free space left after each extent. If the extent overflows , the postings list is copied to a larger extent.

V.OUR APPROACH

In the remainder of the paper we present data structures that efficiently realize the inverted index data abstraction and permit the continuous online application of updates without significantly disrupting retrieval performance. Accessing the inverted index for a term requires a single

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

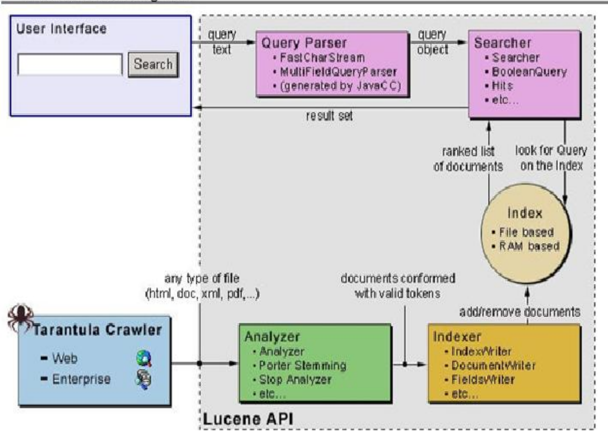
IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 2, February 2015

disk access. Update is an ongoing background process, rewriting the database on an ongoing basis. Updates are maintained in main memory data structures until they can be applied to disk. The update process is occasionally check pointed. If a processor failure occurs the update process may be restarted at the last checkpoint. The integration of caching and index compression is straightforward.

The primary goal of this project is the creation of a prototype distributed full text information retrieval system. Where exposition is simplified and no generality is lost. we use the concrete data structures of the Multi Text Project in our discourse.

System Architecture:



Our general concern is the construction of a distributed fulltext information retrieval system. The basic architecture consists of a group of LAN connected processors, each managing its own separate disk and memory. Individual processors act as either text servers, storing documents and servicing requests for portions of these documents or as index engines, identifying the portions of documents that match client generated search criteria. To external clients, the group of machines appears to be a single large information retrieval system. A front end processor, the Marshaller/Dispatcher, coordinates the activities of the group of processors, interacting with client applications, dispatching queries to the index engines and text servers, marshalling query results and returning the results to clients.

VI. PAGE RANK ALGORITHM

So what is Page Rank?

PageRank is a "vote", by all the other pages on the Web, about how important a page is. A link to a page counts as a vote of support. If there's no link there's no support.

Page Rank is defined like this:

We assume page A has pages T1...Tn which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also C(A) is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

$$PR(A) = (1-d) + d\,(PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one.

PageRank or PR(A) can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web.

but that's not too helpful so let's break it down into sections.

PR(Tn) - Each page has a notion of its own self-importance. That's "PR(T1)" for the first page in the web all the way up to "PR(Tn)" for the last page

C(Tn) - Each page spreads its vote out evenly amongst all of it's outgoing links. The count, or number, of outgoing links for page 1 is "C(T1)", "C(Tn)" for page n, and so on for all pages.

PR(Tn)/C(Tn) - so if our page (page A) has a backlink from page "n" the share of the vote page A will get is "PR(Tn)/C(Tn)"

d(... - All these fractions of votes are added together but, to stop the other pages having too much influence, this total vote is "damped down" by multiplying it by 0.85 (the factor "d")

(1 - d) - The (1 – d) bit at the beginning is a bit of probability math magic so the "sum of all web pages' PageRanks will be one": it adds in the bit lost by the d(.... It also means that if a page has no links to it (no backlinks) even then it will still get a small PR of 0.15 (i.e. 1 – 0.85). (Aside: the Google paper says "the sum of all pages" but they mean the "the normalised sum" – otherwise known as "the average" to you and me.

How is Page Rank Calculated?

This is where it gets tricky. The PR of each page depends on the PR of the pages pointing to it. But we won't know what PR those pages have until the pages pointing to them have their PR calculated and so on… And when you consider that page links can form circles it seems impossible to do this calculation!

PageRank or PR(A) can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web.

What that means to us is that we can just go ahead and calculate a page's PR without knowing the final value of the PR of the other pages. That seems strange but, basically, each time we run the calculation we're getting a closer estimate of the final value. So all we need to do is remember the each value we calculate and repeat the calculations lots of times until the numbers stop changing much.

VII. FURTHER ISSUES

Index Compression

Index compression integrates smoothly into the scheme. Each index block is individually compressed to a variablelength segment. The index map references compressed blocks rather than fixed size blocks. We add a

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 2, February 2015

word to each entry in the index map that indicates the offset of the compressed block in the index. Blocks are decompressed as they are brought into the cache or read by the update process. Since all blocks are cyclically re-written, compression does not hamper update.

System Considerations

This paper has concentrated on the design of the index engine.We look briefly at a few relevant aspects of the remainder of the system. The text server translates a range in the database into the associated text. While the details differ, we organize the text server using data structuring principles similar to those used in the index engine. While our data structures efficiently implement the inverted index data abstraction, they do not efficiently implement queries that are based only on the dictionary. Generally these queries consist of identifying terms that match specified patterns. In systems that separate the dictionary from the index, this type of query can be satisfied by a dictionary search. Besides its other duties, the Marshaller/Dispatcher is responsible for handling these dictionary based queries by maintaining a separate dictionary database of all words in the system. In addition, the Marshaller/Dispatcher is responsible for implementing a term thesaurus.

VIII. WIDER APPICATION

While our exposition has been in the context of a distributed information retrieval system the data structures and update strategy have wider applicability. Inverted indexes are used in applications other than information retrieval. Even if file structures are static and update is not a requirement in the case of aCD ROM, for example) our data structures provide an efficient realization of inverted lists. The update strategy has applicability to other databases with similar update characteristics with the text server being a ready example.

XI. CONCLUSION

The data structures presented in this paper efficiently realize the inverted index data abstraction. A retrieval operation requires a single disk access in all but rare cases. The update strategy provides high throughput with little impact on retrieval performance. The file structures may be compressed to increase the size of index that can be stored on available disk. Although discussed in the context of a distributed fulltext information retrieval system, the results of this paper have applicability to any use of inverted indexes and any database with similar update characteristics.

ACKNOWLEDGEMENT

The authors would like to thank faculty of Computer Department, University of Pune for the opportunity given
In conducting this research.

REFERENCES

[1] Qi H, Li K, Shen Y, Q u W .Object-based image reDerieval with kernel on adjacency matrix and local combined features .ACM Trans Multimed Comput Commun Appl (TOMCCAP) 2012;8(4):54.
[2] Ji C, Dong T, Li Y, Shen Y ,Li K , Qi u W.etal.2012.Inverted grid-based knn query processing with map reduce.In:2012Seven th china Grid annual conference.p.25–32
[3] Hasan M ,Cheema M , Qu W , Lin X .Efficient algorithm stom on it or continuous constrained k nearest neighbour queries .In :Database systems for advanced applications, SpringerBerlinHeidelberg,2010.p.233–49.
[4] Forbes J. Burkowski Surrogate subsets: A free space management strategy for theindex of a text retrieval system. In Proc.13th ACMSIGIR Conference, pages 211-226, Brussels ,1990.

 DOI 10.17148/IJARCCE.2015.4296