International Journal of Computer Science and Engineering Research and Development (IJCSERD), ISSN 2248-9363 (Print), ISSN 2248-9371 (Online), Volume 1, Number 2, May-October (2011) pp. 08-27 © PRJ Publication http://www.prjpublication.com/IJCSERD.asp

IJCSERD

© PRJ PUBLICATION

EXTENDING THE PERFORMANCE AND QUALITY OF SEARCH ENGINE VIA SOFTWARE TEST CASE GENERATION

[1]Hemraj Saini
Associate Professor & Head, Department of Computer Science & Engineering,
Orissa Engineering College, Bhubaneswar (India)-752050

H.N.Pratihari
Professor & Head, Department of Electronics &Communication Engineering,
Orissa Engineering College, Bhubaneswar (India)-752050

ABSTRACT

Nowadays, millions of millions of information has been uploaded through the web in different specialization. Such huge amount of information makes difficult for the normal search methods to be effective. This paper discusses search engine in general perception, refinement of search, software components of search engine and finally the detailed way of working of a search engine by an extensive modular architecture of the Search engine and how it can be improved through software testing.

The paper suggests new technique for the search engine based on the object oriented architecture. Procedural oriented, structural oriented and object oriented are the general techniques, which are adopted to solve the given statements of the problem. Out of these three techniques, object oriented techniques are broadly used, as they better describe the objects and flexibility to maintain. This paper attempts to demonstrate the power of object oriented software testing at class level, a disciplined approach to improve the product search engine, which is very powerful tool of knowledge engineering and management.

In this paper the modular architecture of a search engine is analyzed in an extensive manner with its working. The object oriented technique is analyzed to implement the SearchQueryProcessing class of the search engine. A technique based on class specification is also developed that can be used to generate the test cases at class level testing for object oriented programs. The paper explained how to get the best out of the Internet, working of a search engine in general perception, refinement of search, software components of search engine and finally the detailed way of working of a search engine by an extensive modular architecture of the Search engine. Based on this technique a test model is also developed for generation of test cases. The actual intended behavior is represented by the test model because it is based on the class specification.

[1] Corresponding Author: Hemraj Saini
E-mail: hemraj1977@yahoo.co.in

Although this paper includes discussion of all the test cases of the most crucial class of the search engine, the rest of the classes need to be tested and improvised by the methodology discussed in the paper. Through this paper an attempt is made to implement Object Oriented Approach to improve the performance of the search engine class.

Keywords- Search Engine, Class Level Testing, Specification Based Class Specification, state-space partition, Test Model, Test Case Generation

1. INTRODUCTION

Web and the connection to the information through the Internet have become the most powerful tools around the world for providing information. The provided or uploaded information covers variety range of areas from our life. Therefore finding the right information is vital to get all the benefit from the Web. Thus, the arrangements of information in a specific manner such as indexed one with searching methods are required. The two things, arrangements of information and search methods are incorporated in one single entity named search engines (McCown, F., Nelson,M. L., 2007; Enge, E., 2009; Cameron, R.D., 2001).

Different types of search engines are available for searching the information over the Web. There are essentially five types of search engines (B¨uttcher, S., Clarke, C. L. A.,2006; Tsegay, Y., Turpin, A., Zobel, J., 2007; Brin, S., Lawrence, 2000) which are available- Free text search engines, Index search engines, Multi-search engines, Natural Language engines, Site/subject specific. Each of these works very differently, and one should need to be aware of when it's best to use one, and when it's best to use another. These search engines have a specific method to match the query with its database and returns the matched patterns. This searching method can be efficiently implemented by using object oriented techniques, means by creating the class and its methods as described in further text. These methods of the class need to be tested by various test cases for better and effective performance.

Testing is conducting at two levels, method and system levels. Many researchers have addressed the class-level testing (Doong & Frankl, 1991, 1994; Harrold, McGregor, & Fitzpatrick, 1992; McGregor & Korson, 1994; Murphy, Townsend, & Wong, 1994; Nirmal Gupta, Dinesh Saini, Hemraj Sain, 2008). Formal methods can be used in software testing process to improve the quality and effectiveness of the process (Bernot, Gaudel, & Marre, 1991; Carrington & Stocks, 1994; Crnkovic, Filipe, Larsson, & Lau, 2000).

This paper discusses a modular architecture of search engine first and then a software testing technique in order to improve the processing of a search engine. The testing technique is an object oriented software testing process which can be used to generate test cases systematically and effectively. To generate the test cases for a class, which may have mutable objects (which can be modified after it is created), we will use specification-based testing technique. The purpose of specification-based testing is to derive testing information from a specification of the software under test, rather than from the implementation. Various methods of object-oriented software specifications may be graphical techniques, decomposition specification techniques, communication specification techniques, functional specification techniques and behavior specification

techniques (Wieringa, 1998). The paper deals with finite state diagram which is a behavior specification technique for the generation of test cases of the classes, the mutable objects.

2. WORKING OF SEARCH ENGINE

This section describes the operation of search engines. Initially, all search engines share of what are called 'robots' or 'spiders', which is responsible for surfing the Web from link to another link. The reason for that is to find the new or updated sites that are uploaded to the Web. When they find a new site, or an updated site, they will take snap of information from the discovered sites and store them in their home database. . People can register their web pages with search engines, which means that they usually get listed much more quickly than waiting for the spiders to come across them. Search engines match queries against an index that they create in the home database. The index consists of the words in each document, plus pointers to their locations within the documents. This is called an inverted file (Zhang, J., Long, X., Suel, T., 2008). A search engine comprises four essential modules, a document processor, a query processor, a search and matching function, and a ranking capability (Brin, S., Lawrence, 2000). While users focus on "search", the search and matching function is only one of the four modules. Each of these four modules can have impact on the searching results. The following sections describe the functions.

2.1. Document Processor

The document processor prepares, processes, and inputs the documents, pages, or sites that users search against. The document processor performs some or all of the following steps:

- Normalizes the document stream to a predefined format.
- Breaks the document stream into desired retrievable units.
- Isolates and Meta tags subdocument pieces.
- Identifies potential index able elements in documents.
- Deletes stop words.
- Stems terms.
- Extracts index entries.
- Computes weights.
- Creates the main inverted file against the searched items in order to match queries to documents.

Steps 1-3: **Preprocessing**. While essential and potentially important in affecting the outcome of a search, these first three steps simply standardize the multiple formats encountered when deriving documents from various providers or handling various Web sites. The steps serve to merge all the data into a single consistent data structure that all the downstream processes can handle. The need for a well-formed, consistent format is of relative importance in direct proportion to the sophistication of later steps of document processing. Step two is important because the pointers stored in the inverted file will enable a system to retrieve various sized units — either site, page, document, section, paragraph, or sentence.

Step 4: **Identify elements to index.** Identifying potential elements for indexing in the documents dramatically affects the nature and quality of the document representation that the engine will search against. In designing the system, It is necessary to define the word

"term." Is it the alphanumeric characters between blank spaces or punctuation? If so, what about non-compositional phrases (phrases in which the separate words do not convey the meaning of the phrase, like "skunk works" or "hot dog"), multi-word proper names, or inter-word symbols such as hyphens or apostrophes that can denote the difference between "small business men" versus small-business men." Each search engine depends on a set of rules that its document processor must execute to determine what action is to be taken by the "tokenizer," i.e. the software used to define a term suitable for indexing.

Step 5: **Deleting stop words**. This step helps to save system resources by eliminating from further processing, as well as potential matching, those terms that have little value in finding useful documents in response to a customer's query. This step used to matter much more than it does now when memory has become so much cheaper and systems so much faster, but since stop words may comprise up to 40 percent of text words in a document, it still has some significance. A stop word list typically consists of those word classes known to convey little substantive meaning, such as articles (*a, the*), conjunctions (*and, but*), interjections (*oh, but*), prepositions (*in, over*), pronouns (*he, it*), and forms of the "to be" verb (*is, are*). To delete stop words, an algorithm compares index term candidates in the documents against a stop word list and eliminates certain terms from inclusion in the index for searching.

Step 6: Term Stemming. Stemming removes word suffixes, perhaps recursively in layer after layer of processing. The process has two goals. First, in terms of efficiency, stemming reduces the number of unique words in the index, which in turn reduces the storage space required for the index and speeds up the search process. Second, in terms of effectiveness, stemming improves recall by reducing all forms of the word to a base or stemmed form. For example, if a user asks for *analyze*, they may also want documents which contain *analysis*, *analyzing*, *analyzer*, *analyzes*, and *analyzed*. Therefore, the document processor stems document terms to *analy-* so that documents which include various forms of *analy-* will have equal likelihood of being retrieved; this would not occur if the engine only indexed variant forms separately and required the user to enter all. Of course, stemming does have a downside. It may negatively affect precision in that all forms of a stem will match, when, in fact, a successful query for the user would have come from matching only the word form actually used in the query.

Step 7: Extract index entries. Having completed steps 1 through 6, the document processor extracts the remaining entries from the original document.

The output of step 7 is then inserted and stored in an inverted file that lists the index entries and an indication of their position and frequency of occurrence. The specific nature of the index entries, however, will vary based on the decision in Step 4 concerning to constitutes the terms which can be indexed. More sophisticated document processors will have phrase recognizers, as well as Named Entity Recognizers and Categorizers, to insure index entries such as Milosevic are tagged as a person and entries such as Yugoslavia and Serbia as countries.

Step 8: Term weight assignment. It is used to assign weights to different terms in the index file. The simplest of search engines just assign a binary weight: 1 for presence and 0 for absence. The more sophisticated the search engine, the more complex the weighting scheme. Measuring the frequency of occurrence of a term in the document creates more sophisticated weighting, with length-normalization of frequencies still more

sophisticated. Extensive experience in information retrieval research over many years has clearly demonstrated that the optimal weighting comes from use of "tf/idf." This algorithm measures the frequency of occurrence of each term within a document. Then it compares that frequency against the frequency of occurrence in the entire database.

Step 9: Create index. The index or inverted file is the internal data structure that stores the index information and that will be searched for each query. Inverted files range from a simple listing of every alpha-numeric sequence in a set of documents/pages being indexed along with the overall identifying numbers of the documents in which the sequence occurs, to a more linguistically complex list of entries, the tf/idf weights, and pointers to where inside each document the term occurs. The more complete the information in the index, the better the search results.

2.2. Query Processor

Query processing has seven possible steps, though a system can cut these steps short and proceed to match the query to the inverted file at any of a number of places during the processing. Document processing shares many steps with query processing. More steps and more documents make the process more expensive for processing in terms of computational resources and responsiveness. However, longer the wait for results means higher the quality of results. The steps for a query processing are as follows (with the option to stop processing and start matching indicated as "Matcher"):

- Tokenize query terms.
  Recognize query terms vs. special operators.
  ——————————> Matcher
- Delete stop words.
- Stem words.
- Create query representation.
  ——————————> Matcher
- Expand query terms.
- Compute weights.
  ——————————> Matcher

Step 1: **Tokenizing.** As soon as a user inputs a query, the search engine must tokenize the query stream, i.e., break it down into understandable segments. Usually a token is defined as an alphanumeric string that occurs between white space AND/OR punctuation.

Step 2: Parsing. Since users may employ special operators in their query, including Boolean, adjacency, or proximity operators, the system needs to parse the query first into query terms and operators. These operators may occur in the form of reserved punctuation (e.g., quotation marks) or reserved terms in specialized format (e.g., AND, OR). In the case of an NLP system, the query processor will recognize the operators implicitly in the language used no matter how the operators might be expressed (e.g., prepositions, conjunctions, ordering). At this point, a search engine may take the list of query terms and search them against the inverted file.

Steps 3 and **4: Stop list and stemming.** Some search engines will go further and stop-list and stem the query, similar to the processes described above in the Document Processor section. The stop list might also contain words from commonly occurring querying phrases, such as, "I'd like information about." However, since most publicly available search engines encourage very short queries, as evidenced in the size of query window provided, the engines may drop these two steps.

Step 5: Creating the query. How each particular search engine creates a query representation depends on how the system does its matching. If a statistically based matcher is used, then the query must match the statistical representations of the documents in the system. Good statistical queries should contain many synonyms and other terms in order to create a full representation. If a Boolean matcher is utilized, then the system must create logical sets of the terms connected by AND, OR, or NOT.

An NLP system will recognize single terms, phrases, and Named Entities. If it uses any Boolean logic, it will also recognize the logical operators from Step 2 and create a representation containing logical sets of the terms to be AND'd, OR'd, or NOT'd.

At this point, a search engine may take the query representation and perform the search against the inverted file. More advanced search engines may take two further steps.

Step 6: Query expansion. Since users of search engines usually include only a single statement of their information needs in a query, it becomes highly probable that the information they need may be expressed using synonyms, rather than the exact query terms, in the documents which the search engine searches against. Therefore, more sophisticated systems may expand the query into all possible synonymous terms and perhaps even broader and narrower terms.

Step 7: Query term weighting (assuming more than one query term). The final step in query processing involves computing weights for the terms existed in the query according to their relevance.

Leaving the weighting up to the user is not common, because research has shown that users are not particularly good at determining the relative importance of terms in their queries. They can't make this determination for several reasons. First, they don't know what else exists in the database, and document terms are weighted by being compared to the database as a whole. Second, most users seek information about an unfamiliar subject, so they may not know the correct terminology.

Few search engines implement system-based query weighting, but some do an implicit weighting by treating the first term(s) in a query as having higher significance. The engines use this information to provide a list of documents/pages to the user.

After this final step, the expanded, weighted query is searched against the inverted file of documents.

2.3. Search and Matching Function

This step is based on different theoretical models of information retrieval available in literature (these are not discussed in the paper as its scope becomes too larger). The paper will only make some broad generalizations in the following description of the search and matching function.

Searching the inverted file for documents meeting the query requirements, referred to simply as "matching," is typically a standard binary search, no matter whether the search ends after the first two, five, or all seven steps of query processing (Tomkins, A., Xu, Y., 2006; Long, X., Suel, T., 2005). While the computational processing required for simple, un-weighted, non-Boolean query matching is far simpler than when the model is an NLP-based query within a weighted, Boolean model, it also follows that the simpler the document representation, the query representation, and the matching algorithm, the less relevant the results, except for very simple queries, such as one-word, non-ambiguous queries seeking the most generally known information.

Having determined which subset of documents or pages matches the query requirements to some degree, a similarity score is computed between the query and each document/page based on the scoring algorithm used by the system. Scoring algorithms rankings are based on the presence/absence of query term(s), term frequency, tf/idf, Boolean logic fulfillment, or query term weights. Some search engines use scoring algorithms not based on document contents, but rather, on relations among documents or past retrieval history of documents/pages.

After computing the similarity of each document in the subset of documents, the system presents an ordered list to the user. The sophistication of the ordering of the documents again depends on the model the system uses, as well as the richness of the document and query weighting mechanisms. For example, search engines that only require the presence of any alpha-numeric string from the query occurring anywhere, in any order, in a document would produce a very different ranking than one by a search engine that performed linguistically correct phrasing for both document and query representation and that utilized the proven tf/idf weighting scheme.

However the search engine determines rank, the ranked results list goes to the user, who can then simply clicks and follow the system's internal pointers to the selected document/page. More sophisticated systems will go even further at this stage and allow the user to provide some relevance feedback or to modify their query based on the results they have seen. If either of these are available, the system will then adjust its query representation to reflect this value-added feedback and re-run the search with the improved query to produce either a new set of documents or a simple re-ranking of documents from the initial search.

2.3.1. Ranking

Ranking of the terms can be decided by considering the following different factors associated with a particular term-

Term frequency: How frequently a query term appears in a document is one of the most obvious ways of determining a document's relevance to a query.

Location of terms: Many search engines give preference to words found in the title or lead paragraph or in the metadata of a document. Terms occurring in the title of a document or page that match a query term are therefore frequently weighted more heavily than terms occurring in the body of the document.

Popularity: Search engine and several other search engines add popularity to link analysis to help determine the relevance or value of pages. Popularity utilizes data on the frequency with which all users choose a page as a means of predicting relevance. While popularity is a good indicator at times, it assumes that the underlying information need remains the same.

Date of Publication: Some search engines assume that the more recent the information is, the more likely that it will be useful or relevant to the user. The engines therefore present results beginning with the most recent to the less current.

Length: While length per se does not necessarily predict relevance, it is a factor when used to compute the relative merit of similar pages. So, in a choice between two documents both containing the same query terms, the document that contains a proportionately higher occurrence of the term relative to the length of the document is assumed more likely to be relevant.

Proximity of query terms: When the terms in a query occur near to each other within a document; it is more likely that the document is relevant to the query than if the terms occur at greater distance. While some search engines do not recognize phrases in queries, some search engines clearly rank documents in results higher if the query terms occur adjacent to one another or in closer proximity, as compared to documents in which the terms occur at a distance.

Proper nouns sometimes have higher weights, since so many searches are performed on people, places, or things. While this may be useful, if the search engine assumes that you are searching for a name instead of the same word as a normal everyday term, then the search results may be peculiarly skewed.

3. GENERAL MODULAR ARCHITECTURE OF A SEARCH ENGINE

First, we will provide a high level discussion of the architecture. Then, there are some in-depth descriptions of important data structures. Finally, the major applications: crawling, indexing, and searching will be examined in depth.

3.1. Architecture Overview

Most of Search engines are implemented in C or C++ for efficiency and can run in either Solaris or Linux.

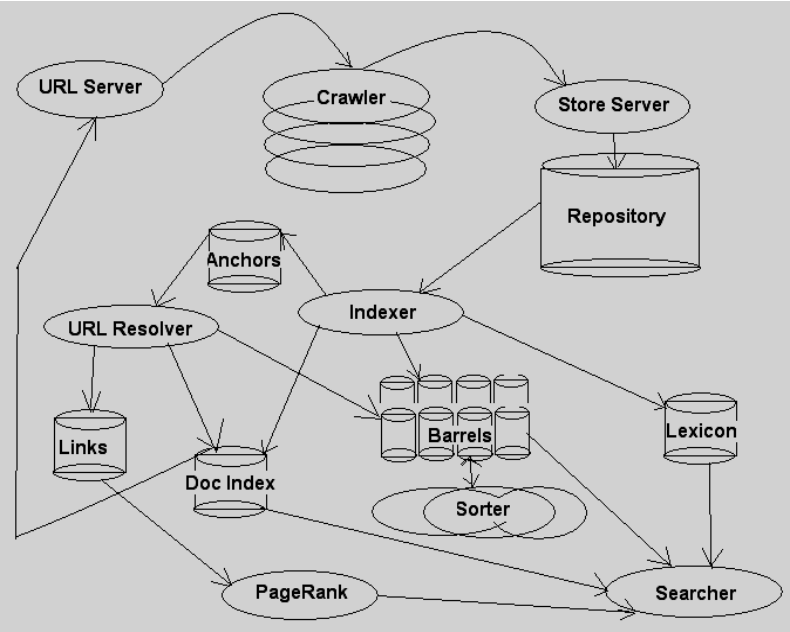

Figure-1: High Level Search engine Architecture

Graphical representation of general modular architecture of a search engine is given by figure-1. In Search engine, the web crawling (downloading of web pages) is done by several distributed crawlers. There is a URLserver that sends lists of URLs to be fetched to the crawlers. The web pages that are fetched are then sent to the storeserver. The storeserver then compresses and stores the web pages into a repository. Every web page has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a web page. The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, uncompressed the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, position in document, an approximation

of font size, and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parses out all the links in every web page and stores important information about them in an anchors file. This file contains enough information to determine where each link points from and to, and the text of the link.

The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents.

The sorter takes the barrels, which are sorted by docID and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

3.2. Major Data Structures

Search engine's data structures are optimized so that a large document collection can be crawled, indexed, and searched with little cost. Although, CPUs and bulk input output rates have improved dramatically over the years, a disk seek still requires about 10 ms to complete. Search engine is designed to avoid disk seeks whenever possible, and this has had a considerable influence on the design of the data structures.

3.2.1. BigFiles

BigFiles are virtual files spanning multiple file systems and are addressable by 64 bit integers. The allocation among multiple file systems is handled automatically. The BigFiles package also handles allocation and deallocation of file descriptors, since the operating systems do not provide enough for our needs. BigFiles also support rudimentary compression options.

3.2.2. Repository

The repository contains the full HTML of every web page. Each page is compressed using zlib. The choice of compression technique is a tradeoff between speed and compression ratio. In the repository, the documents are stored one after the other and are prefixed by docID, length, and URL as can be seen in Figure-2.

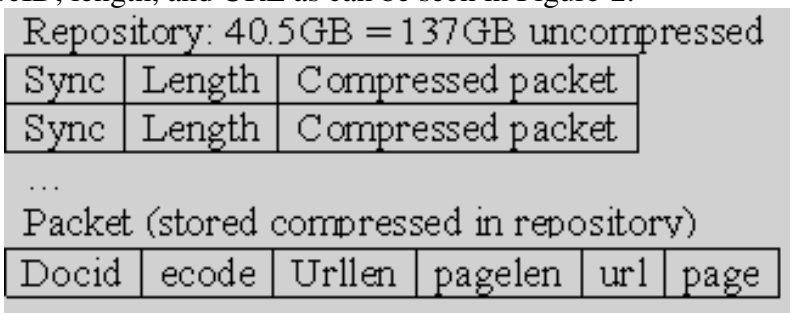

Figure-2: Repository data structure

The repository requires no other data structures to be used in order to access it. This helps with data consistency and makes development much easier; we can rebuild all the other data structures from only the repository and a file, which lists crawler errors.

3.2.3. Document Index

The document index keeps information about each document. It is a fixed width ISAM (Index sequential access mode) index, ordered by docID. The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. If the document has been crawled, it also contains a pointer into a variable width file called docinfo, which contains its URL and title. Otherwise the pointer points into the URLlist, which contains just the URL. This design decision was driven by the desire to have a reasonably compact data structure, and the ability to fetch a record in one disk seek during a search

Additionally, there is a file which is used to convert URLs into docIDs. It is a list of URL checksums with their corresponding docIDs and is sorted by checksum. In order to find the docID of a particular URL, the URL's checksum is computed and a binary search is performed on the checksums file to find its docID. URLs may be converted into docIDs in batch by doing a merge with this file. This is the technique the URLresolver uses to turn URLs into docIDs. This batch mode of update is crucial because otherwise we must perform one seek for every link which assuming one disk would take more than a month for our 322 million link dataset.

3.2.4. Lexicon

The lexicon has several different forms. One important change from earlier systems is that the lexicon can fit in memory for a reasonable price. In the current implementation we can keep the lexicon in memory on a machine with 256 MB of main memory. The current lexicon contains 14 million words (though some rare words were not added to the lexicon). It is implemented in two parts, a list of the words (concatenated together but separated by nulls) and a hash table of pointers.

3.2.5. Hit Lists

A hit list corresponds to a list of occurrences of a particular word in a particular document including position, font, and capitalization information. Hit lists account for most of the space used in both the forward and the inverted indices. Because of this, it is important to represent them as efficiently as possible. Several alternatives are considered for encoding position, font, and capitalization such as simple encoding (a triple of integers), a compact encoding (a hand optimized allocation of bits), and Huffman coding. In the end the hand optimized compact encoding since it required far less space than the simple encoding and far less bit manipulation than Huffman coding. The details of the hits are shown in Figure-3.

The compact encoding uses two bytes for every hit. There are two types of hits: fancy hits and plain hits. Fancy hits include hits occurring in a URL, title, anchor text, or meta tag. Plain hits include everything else. A plain hit consists of a capitalization bit, font size, and 12 bits of word position in a document (all positions higher than 4095 are labeled 4096). Font size is represented relative to the rest of the document using three bits (only 7 values are actually used because 111 is the flag that signals a fancy hit). A fancy hit consists of a capitalization bit, the font size set to 7 to indicate it is a fancy hit, 5 bits to encode the type of fancy hit, and 8 bits of position. For anchor hits, the 8 bits of position are split into 4 bits for position in anchor and 4 bits for a hash of the docID the anchor occurs in. This gives us some limited phrase searching as long as there are not that many anchors for a particular word. Updating of anchor hits are stored to allow for greater resolution in the position and docIDhash fields. Relative font size is used to the

rest of the document because when searching, due to the need of ranking otherwise identical documents differently just because one of the documents is in a larger font.

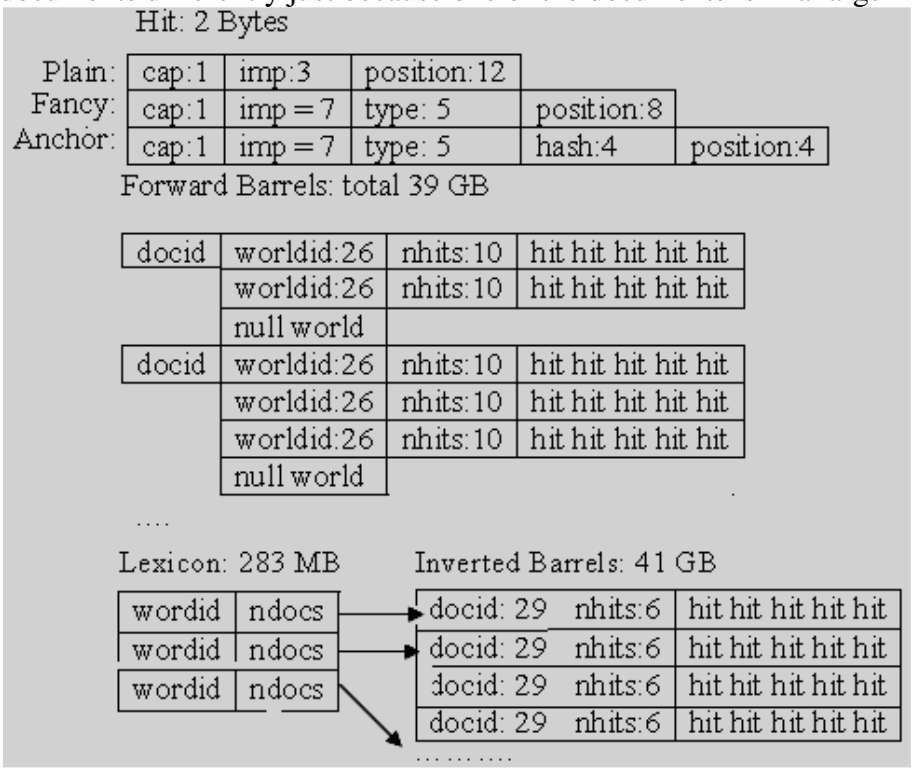

Figure 3 Forward and Reverse Indexes and the Lexicon

The length of a hit list is stored before the hits themselves. To save space, the length of the hit list is combined with the wordID in the forward index and the docID in the inverted index. This limits it to 8 and 5 bits respectively (there are some tricks which allow 8 bits to be borrowed from the wordID). If the length is longer than would fit in that many bits, an escape code is used in those bits, and the next two bytes contain the actual length.

3.2.6. Forward Index

The forward index is already partially sorted, stored in a number of barrels (we used 64). Each barrel holds a range of wordID's. If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordID's with hitlists, which correspond, to those words. This scheme requires slightly more storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter. Furthermore, instead of storing actual wordID's, we store each wordID as a relative difference from the minimum wordID that falls into the barrel the wordID is in. This way, we can use just 24 bits for the wordID's in the unsorted barrels, leaving 8 bits for the hit list length.

3.2.7. Inverted Index

The inverted index consists of the same barrels as the forward index, except that they have been processed by the sorter. For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into. It points to a doclist of docID's together with their

corresponding hit lists. This doclist represents all the occurrences of that word in all documents.

An important issue is in what order the docID's should appear in the doclist. One simple solution is to store them sorted by docID. This allows for quick merging of different doclists for multiple word queries. Another option is to store them sorted by a ranking of the occurrence of the word in each document. This makes answering one word queries trivial and makes it likely that the answers to multiple word queries are near the start. However, merging is much more difficult. Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index. A compromise between these options has be chosen for keeping two sets of inverted barrels such as one set for hit lists which include title or anchor hits and another set for all hit lists. This is the way to check the first set of barrels first and if there are not enough matches within those barrels to check the larger ones.

3.2.8. Indexing the Web

Parsing: Any parser which is designed to run on the entire Web must handle a huge array of possible errors. These range from typos in HTML tags to kilobytes of zeros in the middle of a tag, non-ASCII characters, HTML tags nested hundreds deep, and a great variety of other errors that challenge anyone's imagination to come up with equally creative ones. For maximum speed, instead of using YACC to generate a CFG parser, we use flex to generate a lexical analyzer which we outfit with its own stack. Developing this parser which runs at a reasonable speed and is very robust involved a fair amount of work.

Indexing Documents into Barrels: After each document is parsed, it is encoded into a number of barrels. Every word is converted into a wordID by using an in-memory hash table -- the lexicon. New additions to the lexicon hash table are logged to a file. Once the words are converted into wordID's, their occurrences in the current document are translated into hit lists and are written into the forward barrels. The main difficulty with parallelization of the indexing phase is that the lexicon needs to be shared. Instead of sharing the lexicon, we took the approach of writing a log of all the extra words that were not in a base lexicon, which we fixed at 14 million words. That way multiple indexers can run in parallel and then the small log file of extra words can be processed by one final indexer.

Sorting: In order to generate the inverted index, the sorter takes each of the forward barrels and sorts it by wordID to produce an inverted barrel for title and anchor hits and a full text inverted barrel. This process happens one barrel at a time, thus requiring little temporary storage. Also, we parallelize the sorting phase to use as many machines as we have simply by running multiple sorters, which can process different buckets at the same time. Since the barrels don't fit into main memory, the sorter further subdivides them into baskets which do fit into memory based on wordID and docID. Then the sorter, loads each basket into memory, sorts it and writes its contents into the short inverted barrel and the full inverted barrel.

3.2.9. Searching

The goal of searching is to provide quality search results efficiently. Many of the large commercial search engines seemed to have made great progress in terms of efficiency. Therefore, we have focused more on quality of search in our research, although we

believe our solutions are scalable to commercial volumes with a bit more effort. The search engine query evaluation process is show in figure-4.

To put a limit on response time, once a certain number (currently 40,000) of matching documents are found, the searcher automatically goes to step 7 in figure-4. This means that it is possible that sub-optimal results would be returned.

1. Parse the query.
2. Convert words into wordIDs.
3. Seek to the start of the doclist in the short barrel for every word.
4. Scan through the doclists until there is a document that matches all the search terms.
5. Compute the rank of that document for the query.
6. If we are in the short barrels and at the end of any doclist, seek to the start of the doclist in the full barrel for every word and go to step 4.
7. If we are not at the end of any doclist go to step 4. Sort the documents that have matched by rank and return the top k.

Figure 4: Search Engine Query Evaluation

4. THE CORE CLASS OF A SEARCH ENGINE: *SEARCHQUERYPROCESSING* CLASS

This is the core part of any search engine used to provide the quality search results efficiently. To design it we can use mostly used programming technique named object oriented techniques. The purpose of using the object oriented technique is due to its syntax by which one can represent the scenario in a real world like system. The representation of the scenario is in the form of classes, each having methods and attributes.

4.1. Class Specification

Larch (Guttag, Horning, Garland et al.,1993; Wing, 1983) may be thought of as an approach to formal specification of program modules. This approach is an extension of Hoare's ideas for program specification (Hoare, 1969, 1972). Its distinguishing feature is that it uses two "tiers" (or layers). A class specification will consist of two layers: a functional tier and a conditional tier. The functional tier is an algebraic specification, which is used to define the abstract values of objects as shown in figure-5. In the conditional tier, the class name, invariant, pre and post conditions of methods of each class are specified as shown in figure-6.

```
SearchQueryString  // Query string to be searched by search engine
TokenThreshHolds  //Maximum characters allowed to be tokenized in one cycle
Class SearchQueryProcessing has Tokens
        constructor SearchQueryProcessing()
                TokenThreshHolds=0
        end constructor
        constructor SearchQueryProcessing(Int X)
                TokenThreshHolds=X
                SearchQueryString = SearchQueryString - First X  characters
        end constructor
        operation GetTokens(SearchQueryString) returns tokens
                return Tokens
        end operation
        operation GetParser(Tokens) returns reservedTerms
                return reservedTerms
        end operation
        operation GetQuery(Tokens) returns matchingPatterns
                return matchingPatterns
        end operation
        operation GetGenericQuery(matchingPatterns, reservedTerms) returns      matchingPatterns with
        reservedWords
                return (matchingPatterns with reservedTerms)
        end operation
        operation GetTermWeight(matchingPatterns) returns termWeights
                return termWeights
        end operation
        operation GetFirstKMatchedTerms() returns first K matched terms
                return FirstKMatchedTerms
        end operation
end class
```

Figure-5: Functional tier of the class SearchQueryProcessing

A number of modules may be existed in the functional tier of SearchQueryProcessing class as shown in the figure-5. It is having a default constructor, an overloaded constructor to handle the maximum number of tokens to be processed in one cycle, GetTokens method to separate different tokens, GetParser for parsing the query string having 'AND', 'OR' etc. connectors, GetQuery method to identify the matching patterns, GetGenericQuery method is an overloading method to handle the reservewords and returns the matching patterns, GetTermWeight method is used to generate the inverted file with term weights and their associated term values and GetFirstKMatchedTerms is. a method to return the first K results from the overall results of the query.

```
Class SearchQueryProcessing
        Invariant {Tokens > = 0}.
        constructor SearchQueryProcessing()
                ensure { Tokens = = 0}
        constructor SearchQueryProcessing(Int X)
                ensure { Tokens ! = 0 && TokenThresHold = = X)
        method GetTokens(SearchQueryString SQT)
                requires {SQT !=NULL && Tokens>=0}
                ensures {Token(post-operation) == Tokens(pre-operation)+Tokens}
        method GetParser(Tokens LST)
                requires {LST !=NULL}
                ensures {Token(post-operation) == Tokens(pre-
                operation)+ReservedTerms}
        method GetQuery(Tokens LST)
                requires {LST !=NULL}
                ensures {Token(post-operation) == Tokens(pre-
                operation)+ matchingPatterns }
        method GetGenericQuery(mathchingPatterns MP, reservedWords RW)
                requires {MP !=NULL && RW!=NULL}
                ensures {Token(post-operation) == Tokens(pre-
                operation)+ matchingPatterns }
        method GetTermWeight(matchingPatterns MP)
                requires {MP !=NULL}
                ensures {Length of inverted file!=0} // Inverted file is a way to store
                                                     //the termweights and its
                                                     //associated terms
        method GetFirstKMatchedTerms()
```

Figure-6: The conditional tier of the class SearchQueryProcessing

4.2. The state-space partition of class SearchQueryProcessing

Depending upon the values of attributes a class object may acquire various states. All such states form state space (VanderBrug & Minker, 1975) of a class. In our approach, the state space of a class will be partitioned into sub states. For each sub state, the input space of each method will be partitioned into sub-domains. Partition testing or sub domain testing comprises a broad class of software testing methods that call for dividing a program's input domain into sub domains and then selecting a small number of tests (usually one) from each of them (Weyuker & Jeng, 1991).

Self.Token<100, self.Token>100

Taking into account the clauses in Figure-5, the state space is further partitioned into six substates, In the Figure-6 six substates are shown.

Unborn or initial, self.Token<=0 (invalid), 0< self.Token<100, self.Token=100, self.Token>100, & Final or Decidable

Figure-7: State-space partition of class SearchQueryProcessing

Let us consider the class SearchQueryProcessing. The state space of this class is partitioned into following substates: the token is less than 100 and the token is greater than 100.No further partitioning is necessary in this simple example. It is assumed that an object behaves uniformly in each substate. The test model in this example class is a finite-state machine, which describes the state-dependent behaviors of individual objects of the class as shown in figure-7. The test model will have a set of states and a set of

transitions among the states. Each state is obtained through the state-space partition of the class. Each transition between the states consists of a method, which changes the value of an object from the source state to the target state of the transition, and a guard predicate derived from the pre-condition of the method. Also there are two special states: initial and final states. The former represents the state before an object is created and the latter represents the state after an object is destroyed. The test model of class SearchQueryProcessing is having six states:

S0 = {unborn or initial},
S1 = 0,
S2 = {0 < b < 100},
S3= {b = 100},
S4={b>100},
S5={Final}.

Where S0 is initial state and b denotes the attribute Balance. The transitions are:

t0= SearchQueryProcessing(),
t1=GetTokens() {X=0},
t2=GetTokens() {0<X<100},
t3=GetTokens(X) {X=100},
t4=GetTokens(X) {X>100},
t5= GetParser(LST) {LST.length=0},
t6= GetParser(LST) {0<LST.length <100},
t7={ GetParser(LST) {LST.length=100},
t8=GetParser(LST.length>100},
t9= GetQuery(LST) { LST.length=0},
t10= GetQuery(LST) {0<LST.length <100},
t11= GetQuery(LST) { LST.length=100},
t12= GetQuery(LST) { LST.length>100},
t13= GetGenericQuery(LST, RW) {LST.length<=0, RW.length <=0},
t14=GetGenericQuery(LST,RW){LST.length>=0, RW<=0},
t15 = GetGenericQuery(LST,RW){LST.length>=0, RW.length<=0},
t16 = GetGenericQuery(LST,RW) {LST.length<=0, RW.length>=0},
t17= GetGenericQuery(LST,RW){LST.length>0,RW.length>0},
t18=GetGenericQuery(LST,RW) {LST.length==0, RW.length<0)
t19 = GetGenericQuery(LST,RW) {LST.length<0, RW.length==0},
t20= GetGenericQuery(LST,RW) {0<LST.length<100, 0<RW.length<100},
t21=GetTermWeight(MP) {MP=0},
t22=GetTermWeight(MP) {MP<0},
t23=GetTermWeight(MP) {MP>0},
t24=GetTermWeight(MP) {0<MP<100},
t25=GetTermWeight(MP) {MP=100},
t26=GetTermWeight(MP) {MP>100}.

4.3. The test model for the class SearchQueryProcessing

The test model can be turned into a complete finite-state machine by adding error states, error transitions, undefined states, and undefined transitions as represented by figure-8. The transition from one state to another state is the cause by a specific method, in our case it is mentioned in the table-1.

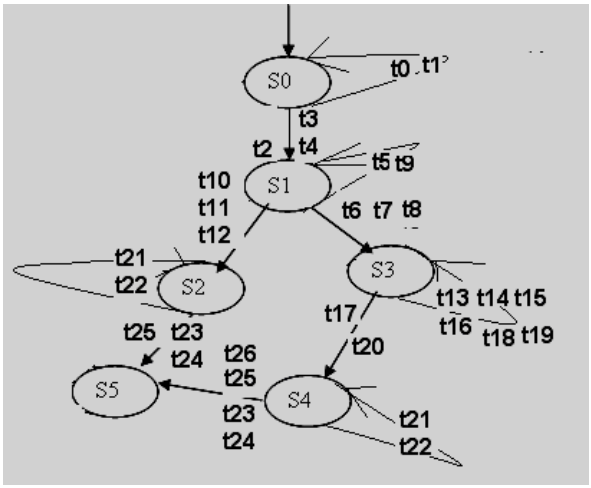

Figure-8: The test model for the class SearchQueryProcessing

Table-1: Mapping in between states and Methods

Current State	Method	Destination State
S0 (unborn state)	GetToken()	S1
S1	GetQuery	S2
S1	GetParser	S3
S2	GetTermWeight	S5
S3	GetGenericQuery	S4
S4	GetTermWeight	S5

4.4. Generation of Test Cases

A test case is generated by traversing the test model from the initial state. Method sequences are derived from the traversed transitions. A set of test cases are required to be generated so that it can cover the test model in the form of state coverage, transition coverage, and path coverage. The input space may also be partitioned. The input space is the sets of values for the input parameters of the method. A valid input space for a method is the subset of the input space satisfying the pre-condition of the method. The input space of a method can be partitioned at least into two sub-domains, whether valid or invalid values. Test data can then be drawn from each sub-domain (Chen & Yu, 2002). The partiotion of input space of the class SearchQueryProcessing is as follows-

Input-space partition of GetToken (SearchQueryString SQT)

In State S1:

p1 = {0<X<100}, p2 = {X=100}, p3 = {0<X<100}

In State S2:

P4 = {0<LST.length<100}, p5 = {LST.length=100}, p6 = { LST.length>100}

In State S3:

P7 = {0<LST.length<100}, p8 = {LST.length=100}, p9 = { LST.length>100}

In State S4:

p10 = {LST.length>0 and RW.length>0}, p11 = { 0<LST.length<100 and 0<RW.length<100}

In State S5:

p12 = {0<MP<100}, p13={MP=100}, p14={MP>100}

Input-space partition of GetParser(Tokens LST)

In State S1:

p13 = {0<X<100}, p14 = {X=100}, p15 = {0<X<100}
In state S3:
P16 = {0<LST.length<100}, p17 = {LST.length=100}, p18 = { LST.length>100}
In State S4:
p19 = {LST.length>0 and RW.length>0}, p20 = { 0<LST.length<100 and 0<RW.length<100}
In state S5:
p21 = {0<MP<100}, p22={MP=100}, p23={MP>100}

Input-space partition of GetQuery(Tokens LST)

In State S1:
p24 = {0<X<100}, p25 = {X=100}, p26 = {0<X<100}
In state S2:
p27 = {0<LST.length<100}, p28 = {LST.length=100}, p29 = { LST.length>100}
In state S5:
p30 = {0<MP<100}, p31={MP=100}, p32={MP>100}

Input-space partition of GetGenericQuery(mathchingPatterns MP, reservedWords RW)

In state S3:
p44 = {0<LST.length<100}, p34 = {LST.length=100}, p35 = { LST.length>100}
In State S4:
p36 = {LST.length>0 and RW.length>0}, p37 = { 0<LST.length<100 and 0<RW.length<100}
In state S5:
p38 = {0<MP<100}, p39={MP=100}, p40={MP>100}

Input-space partition of GetTermWeight(matchingPatterns MP)

In state S2:
P41 = {0<LST.length<100}, p42 = {LST.length=100}, p43 = { LST.length>100}
In State S4:
P44 = {LST.length>0 and RW.length>0}, p45 = { 0<LST.length<100 and 0<RW.length<100}
In state S5:
P46 = {0<MP<100}, p47={MP=100}, p48={MP>100}

For example which calls GetGenericQuery method, Let SearchQueryString SQT = “aaa bbb”, the test case SearchQueryProcessing(),GetToken(SQT), GetParser ({LST = {aaa}, {bbb}, {aaa, bbb}, { }}), GetGenericQuery GetTermWeight ({MP = {aaa}, {bbb}, {aaa, bbb}}, { }) covers the five states in the following sequence:
S0, S1, S2, S4, S5

Another example which calls GetQuery method , Let SearchQueryString SQT = “aaa”, the test case SearchQueryProcessing(),GetToken(SQT), GetParser ({LST = {aaa}), GetQuery GetTermWeight ({MP = {aaa}, {aa}, {a}}) covers the four states in the following sequence:
S0, S1, S2, S5

5. CONCLUSION

In this paper the modular architecture of a search engine is discussed in an extensive manner with its working. The object oriented technique is discussed to implement the SearchQueryProcessing class of the search engine. A technique based on class specification is also developed that can be used to generate the test cases at class level testing for object oriented programs. The paper explained how to get the best out of the Internet, working of a search engine in general perception, refinement of search, software components of search engine and finally the detailed way of working of a search engine by an extensive modular architecture of the Search engine. Based on this technique a test model is also developed for generation of test cases. The actual intended behavior is represented by the test model because it is based on the class specification.

REFERENCES

1. B¨uttcher, S., Clarke, C. L. A. (2006), "A document-centric approach to static index pruning in text retrieval systems", In Proceedings of the 15th ACM International Conference on Information and Knowledge Management (CIKM'06), (pp. 182–189), ACM, New York, NY
2. Bernot, G., Gaudel, M.C., & Marre, B. (1991), "Software testing based on formal specifications: a theory and a tool", Software Engineering Journal, 6(6), 387-405
3. Brin, S., Lawrence (2000), "The Anatomy of a Large-Scale Hyper textual Web Search Engine" Sergey, page@cs.stanford.edu, Computer Science Department, Stanford University, Stanford, CA 94305}
4. Cameron, R.D. (2001), "Information Retrieval and Search" Retrieved on 01st April,2009, Available at: http://www.cs.sfu.ca/~cameron/Teaching/D-Lib/IR.html
5. Carrington, D., & Stocks, P. (1994), "A tale of two paradigms: formal methods and software testing", Proceedings of 8th Annual Z User Meeting (ZUM '94), J.P. Bowen and J.A. Hall (eds.), Workshops in Computing (pp. 51-68). Springer-Verlag, Berlin
6. Chen, T.Y., & Yu, Y.T. (2002), "A decision-theoretic approach to the test allocation problem in partition testing", IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans, 32(6), 733-745
7. Crnkovic, F., J. K., Larsson, M., & Lau, K. K. (2000), "Object-oriented design frameworks: Formal specification and some implementation issues", Proceedings of 4th IEEE International Baltic Workshop, Vol. 2, (pp. 63—77)
8. Doong, R.K., & Frankl, P. (1991), "Case Studies in Testing Object-Oriented Software", Testing, Analysis, and Verification Symposium, Association for Computing Machinery, (pp. 165 – 177). New York
9. Doong, R.K., & Frankl, P. (1994), "The ASTOOT approach to testing object-oriented programs", ACM Transactions on Software Engineering and Methodology,3(2), 101-130
10. Enge, E. (2009), "Search Engine Visibility and Site Crawlability, Part 1", on 1st April, 2009, Available at: http://searchenginewatch.com/3627930

11. Guttag, J. V., & Horning, J. J., Garland, S.J., Jones, K.D., Modet, A., & Wing, J.M. (1993), "Larch: languages and tools for formal specification", Texts and Monographs, Computer Science series NY: Springer-Verlag
12. Harrold, M.J., McGregor, J.D., & Fitzpatrick, K.J. (1992), "Incremental testing of object-oriented class structures", Proceedings of the 14th International Conference on Software Engineering (pp. 68 – 80
13. Hoare, C. A. R. (1969), "An axiomatic basis for computer programming", Comm. ACM, 12(10), 576-583
14. Long, X., Suel, T. (2005), "Three-level caching for efficient query processing in large web search engines", In Proceedings of the 14th International World Wide Web Conference (WWW'05). ACM, New York, NY, 257–266
15. McCown, F., Nelson,M. L. (2007), "Agreeing to disagree: Search engines and their public interfaces", In JCDL '07: Proceedings of the 2007 Conference on Digital Libraries. 309–318
16. McGregor, J.D., & Korson, T.D. (1994), "Testing the polymorphic interactions of classes" Technical Report No- TR-94-103, Clemson University
17. Murphy, G.C., Townsend, P., & Wong, P.S. (1994), "Experiences with cluster and class testing", Communications of the ACM, 37(9), 39 – 47
18. Nirmal Gupta, Dinesh Saini, Hemraj Sain (2008), " Class Level Test Case Generation in Object Oriented Software Testing", Int. J. of Information Technology and Web Engineering, 3(2), 18-26, April-June 2008
19. Tomkins, A., Xu, Y. (2006), "Estimating corpus size via queries", In Proceedings of the 15th ACM International Conference on Information and Knowledge Management (Arlington, VA), (pp. 594–603), ACM. New York
20. Tsegay, Y., Turpin, A., Zobel, J. (2007), "Dynamic index pruning for effective caching", In Proceedings of the 16th ACM conference on Conference on Information and Knowledge Management (CIKM'07). ACM, New York, NY, 987–990
21. VanderBrug, G. J., & Minker, J. (1975), "State-space problem-reduction, and theorem proving—some relationships", Commun. ACM, 18(2), 107-119
22. Weyuker, E. J., & Jeng, B. (1991), "Analyzing partition testing strategies", IEEE Trans. Softw. Eng., 17(7), 703-711
23. Wieringa, R. (1998), "A survey of structured and object-oriented software specification methods and techniques", ACM Comput. Surv., 30(4), 459-527
24. Wing, J.M. (1983), "A two-tiered approach to specifying programs", Technical Report TR-299, Mass. Institute of Technology, Laboratory for Computer Science. N.
25. Zhang, J., Long, X., Suel, T. (2008), "Performance of compressed inverted list caching in search engines", In Proceedings of the 17th International World Wide Web Conference (WWW'08). ACM, New York, NY, 387–396