ISSN:2231-0711
www.ijcset.net

Keyword based Search Engine for Web Applications Using Lucene and JavaCC

Priyesh Wani, Nikita Shah, Kapil Thombare, Chaitalee Zade

Information Technology, Computer Science
University of Pune
priyeshwani@gmail.com
nikitishahu@gmail.com
kapil.thombare@gmail.com
chaitaleezade@gmail.com

***Abstract*—Past Few years IT industry has taken leap towards developing Web based Applications. The need aroused due to increasing globalization. Web applications has revolutionized the way business processes. It provides scalability and extensibility in managing different processes in respective domains. With evolving standard of these web applications some functionalities has become part of that standard, Search Engine being one of them. Organization dealing with large number of clients has to face an overhead of maintaining huge databases. Retrieval of data in that case becomes difficult from developer's point of view. In this paper we have presented an efficient way of implementing keyword based search engine for an application involving such kind of huge database and demonstrated how indexing using Lucene and parsing using JavaCC combined together can make search faster there by reducing the data retrieval time.**

***Keywords*— Lucene, JavaCC**

I. Introduction

Searching is one of the primary needs of any application used by any organization. Clients often seek for faster retrieval of data. Moreover certain time constraints are provided by the organization for searching. There are certain search engines like Google Adsense that can be used directly in any web application. But however in some cases the organization prefers to develop their own search engine due to proprietary issues.

In this paper we have presented a technique of developing keyword based searching using Lucene and JavaCC. This was implemented as a part of project at SunGard Technology Services. The main objective was to provide an enhanced keyword based search engine supporting Google like keywords to an application Fame Energy Services. It is a web based application offering quality-checked, fully-managed data feeds, decision support applications and an enterprise data management system to help you to perform trading and risk management activities, such as market to market valuations, deal pricing, forward curve creation, and financial reporting of P&L

Apache Lucene(TM) is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Some of its features are Scalability, Highly accurate search algorithms, High performance indexing. It is an open-source Java full-text search library which makes it easy to add search functionality to an application or website. Lucene is able to achieve fast search responses because, instead of searching the text directly, it searches an index instead. Which is equivalent to retrieving pages in a book related to a keyword by searching the index at the back of a book, as opposed to searching the words in each page of the book.

JavacCC: Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc. JavaCC works with any Java VM version 1.2 or greater. It has been certified to be 100% Pure Java. JavaCC has been tested on countless different platforms without any special porting requirements. JavaCC is based on LL parsing, but it allows you to use grammars that are not LL. As long as you can use JavaCC's look-ahead specification to guide the parsing where the LLrules are not sufficient, JavaCC can handle any grammar that is not left-recursive. JavaCC thus by providing java API provides java developers an advantage to work around with parsing

The implemented system will accept the keywords to be searched from the user in the form of an expression.The user will be able to enter expressions like "Keyword1" or "Keyword2", "Keyword1" and "Keyword2", "Keyword1" not "Keyword2" as well as expressions consisting parenthesis.It will then perform parenthesis validation and build a search query, based on the tokens generated and the business logic, enabling the accurate retrieval of data and formulate the result set as per the entered expression.

Thus, by using these two technologies viz. Lucene and Javacc, the time efficiency is increased approximately by 50%.

II. History

JavaCC:

In 1996, Sun Microsystems released a parser generator called *Jack*. The developers responsible for *Jack* created their own company called Metamata and changed the *Jack* name to JavaCC. Metamata eventually became part of WebGain. After WebGain shut down its operations, JavaCC was moved to its current home.

Lucene:

Lucene was originally written by Doug Cutting. It was initially available for download from its home at the SourceForge web site. It joined the Apache Software Foundation's Jakarta family of open source Java products in September 2001 and became its own top-level Apache project in February 2005. Until recently, it included a number of sub-projects, such as Lucene Java, Droids, Lucene.Net, Lucy, Mahout, Solr, Nutch, Open Relevance Project, PyLucene, and Tika. Solr has been merged into the Lucene project itself and Mahout, Nutch and Tika have been moved to be independent top-level projects.

Main versions introduced (selected versions):

- 1.01b (July 2001): last SourceForge release
- 1.2 (June 2002): first Apache Jakarta release
- 1.4 (July 2004): enhanced query parser, token positions, span queries, sorting
- 1.9 (February 2006): binary stored fields, date tools, range filters, regexp query
- 2.0 (May 2006): clean up of code, removed deprecated methods
- 2.4 (October 2008): various performance improvements, delete/update
- 2.9 (September 2009): near-realtime search, numeric ranges, cleanup
- 2.9.4 is recommended release for production
- 3.0 (November 2009): cleanup and migration to Java 1.5 (generics, var args)
- 3.1 is latest build released on March 31, 2011

Lucene implementations:

Java (original), C++ (CLucene), .NET (Lucene.NET), C (Lucene4c), Objective-C (LuceneKit), Python (Lupy), PHP 5 (Zend), Perl (Plucene), Delphi (MUTIS), Jruby (Ferret), Common Lisp (Montezuma)

III. WORKING

Overall algorithm :

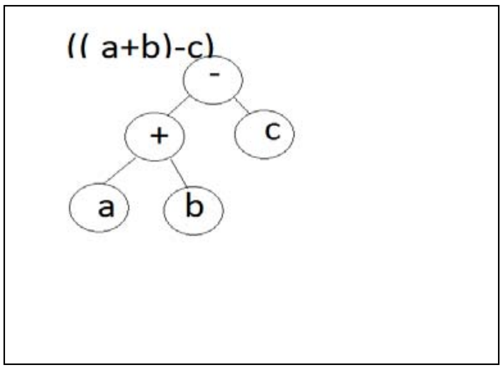
1. Accept parenthesized expression from user.
2. Parse the expression and convert it into an expression tree.
3. Traverse the expression tree in postorder format in order to build lucene queries.
4. Fire the formed lucene query and retrieve the results.

Step 1 and 2 :

We use Javacc here as our parser generator tool and generate a parser which parses the parenthesized expressions . We also make use of the additional functionality available in Javacc called Jjtree to bulid our expression tree. The code for tree building has to be written manually as an automated procedure is not available . The parser generated will parse the input string character by character , we have not changed the value of default LOOKAHEAD (which is 1).

Sample input :

((a+b)-c) , ((a+b)-(c+d+e))



Datastructures used :

A stack called nodes : to push operand

A stack called operator : to push operators

An array oparr[] : to store operators which are popped

An array nodesarr[] : to store operands which are popped

Tokens used in parsing :

IDENTIFIER : any combination of letters (a-z|A-Z) and numbers (0-9) without space

OPERATOR: + , - , |

+ (meaning and), - (meaning not), | (meaning or)

It could easily be extended to parse the strings "and" , "or" , "not"

1. On encountering an opening brace "(" , push it in the operator stack .
2. Go to step 1 untill the parsed character remains to be "("
3. According to the sample input shown , the program expects an IDENTIFIER at this point .So the parser goes ahead only if the next parsed character is an identifier. (a,b,1,2,1a,a1,abc123 etc).
4. When this identifier is parsed, it is pushed into the nodes stack.
5. Now the parser , matches the string of input characters which are to be parsed to regular expression : (operator . Expression) * (closepar)* , where Expression() is defined by : (openpar)* IDENTIFIER (OPERATOR . Expression)* (closepar)* . Here, "openpar" matches the open parenthesis and "closepar" matches the close parenthesis . IDENTIFIER and OPERATOR are as defined above .
6. So following the above regex , whenever an OPERATOR is encountered it is pushed into the operator stack and whenever an IDENTIFIER is encountered it is stored in the nodes stack .On encountering the close parenthesis ,
   6.1. Do
      - Pop from operator stack and store in array oparr[]
      - Pop from nodes stack and store in the array nodesarr[]

      While top of operator stack ! = " ("

6.2 Since the identifiers will always be one more than the number of operator (for eg : a+b , no of operator =1 , number of operand =2) , an additional pop will be required . Pop from the nodes stack and store it in the array nodesarr[] . Pop off the opening bracket.

6.3 Now , we will scan the arrays from right to left:
Do
- The rightmost element in the nodes array is stored as our leftchild , and deleted.
- The element rightmost element now is stored as our rightchild, and deleted.
- We attach the above right and left child to the rightmost operator in the operator array.
- Store this operator node (which has a right and a left child) in the nodesarr in the rightmost.
- Delete the operator from oparr[].

While oparr[] is not empty

6.4 The node in the nodesarr[0] is then pushed into the nodes stack and is deleted from array.

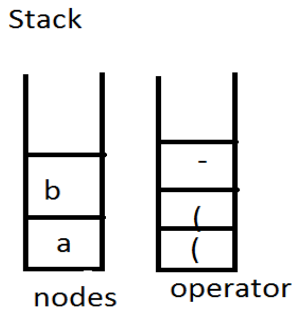
7. Stop

Justification for use of arrays along with stack : The existing algorithms for converting expression into expression trees , use 2 stacks. Consider an expression like (a+(c-z+f) , i.e when more than one operator is inside the bracket. We follow left to right reading of the string. So when we evaluate stack on the parsing of ")" , first we want to evaluate a "+" , and then evaluate "-". To store "-"and respective operands , till we evaluate "+ ", we need to make use of array.

Show working :

For the input: ((a+b)-c)

When the first closing brace in encountered , the contents in stack are.



Following the algorithm,

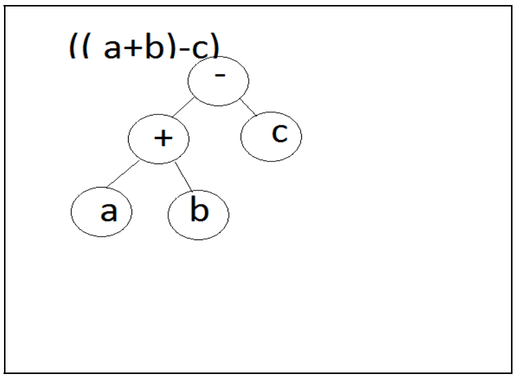
Nodesarr[]

Oparr[] :

Step 3 and 4:

Build Lucene Query: Expression tree formed out of user-entered expression is taken as input to form the Lucene queries. The tree is traversed in postorder format and query is build for keywords as per criterion.

Sample Input: Considering the same example, the tree generated



Data structures used :

An BooleanQuery leftqueru/rightquery : to keep a create a left and right subtree query saperately .

An array keywordcondition to keep a track of the parent operator of the child nodes.

Justification for use of BooleanQuery and array: The existing algorithms for converting expression into expression trees , use a BooleanQuery and a array stacks. Consider an expression like ((a+b)-c) , the tree is as follows:

Here, Boolean query leftquery and rightquery maintain the left and right subtree. . i.e. for operator + the leftquery is a and rightquery is b. However for the operator – the leftquery is ab+ and the rightquery is c. Keywordcondition is used to keep a track of the operators during the reccursive call. If keywordcondition is 1 the operator is + (and), 2 the operator is – (or) else if 3 then the operator is – (not).

Working :

For the input: ((a+b)-c)

As shown in the figure above, first we traverse -, here keywordcondition[0]=2, then the index of the array is incremented. Next when we come across + the valu e of keyworcondition[1]=1. So thateach time the right subtree is popped the index of keywordcondition is decremented and we get the track of the operator of the current subtree.

The postorder traversal of the tree is : ab+c- . i.e. first the roots of the tree are visited and then the operation to be performed between them is decided i.e. +,-,|.

The query is generated for each leafnode initially and then depending upon the keyword criterion the queries are nested and once it reaches the root node, the final query is passed for further processing.

Tokens used for deciding the keyword Criterion:

OPERATOR: + , - , |

+ (meaning and), - (meaning not), | (meaning or)

It could easily be extended to parse the strings "and" , "or" , "not"

Steps involved in generating the query for given expression:

1. Accept the expression tree from the parser.
2. Start postorder traversal
3. For node- has child and node is not visited before, initialize the keywordcondition for corresponding

node (+, - or |). Keywordcondition[]=1(for +), 2(for |), 3(for -). Go to 2

4. If node is equal to leaf node, form the query for leaf node keyword (Lucene Condition: *OCCUR.SHOULD)*
5. If node is not a leaf node and not a root node, get the keyword criterion and form the query applying the criterion on the queries of its child nodes. Go to 2.
6. Node is a root node. Form the final query and return to the main calling function.
7. Pass the query formed out of the expression tree for retrieval of the data from server.
8. Stop.

OBSERVATION

Time complexity :

1. Building the query step 3(Postorder traverdsal): O(n)

Complexity function T(n)

T(n) = T(k) + T(n – k – 1) + c

Where k is the number of nodes on one side of root and n-k-1 on the other side.

Let’s do analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.
T(n) = T(0) + T(n-1) + c
T(n) = 2T(0) + T(n-2) + 2c
T(n) = 3T(0) + T(n-3) + 3c
T(n) = 4T(0) + T(n-4) + 4c
………………………………………….
………………………………………….
T(n) = (n-1)T(0) + T(1) + (n-1)c
T(n) = nT(0) + (n)c

Value of T(0) will be some constant say d. (traversing a empty tree will take some constants time)

T(n) = n(c+d)
T(n) = (-)(n) (Theta of n)

Case 2: Both left and right subtrees have equal number of nodes.

T(n) = 2T(|_n/2_|) + c

This recursive function is in the standard form (T(n) = aT(n/b) + (-)(n)) for master method .

When we solve it, O(n) .

CONCLUSIONS

Thus using Lucene and JavaCC we have successfully generated parser which parses the paranthesized input string into an expression tree notation. The postorder traversal of expression tree helped us to build the lucene queries accurately and efficiently (Since postorder time complexity is O(n) as proved above) there by enabling efficient and optimized searching capability for the application.

ACKNOWLEDGMENT

Success is the manifestation of diligence, perseverance, inspiration, motivation and innovation. Every nice work begins with a systematic approach reaching successsful completion. I am deeply grateful to Prof. M.P. Gandhi and Prathyush Nair(Sungard Technology Services) for guiding us and immensely honoured by her connection.

REFERENCES

[1] *lucenetutorial.com*

[2] JavaCC Documentation.

[3] Introduction to JavaCC. http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf

[4] Information Retrieval Seminar December 05, 2005IBM Research LabHaifa, Israel on Lucene algorithm

[5] http://generatingparserswithjavacc.com/