Cross Lingual Information Retrieval and Error Tracking in search engine

by

Saurabh Garg

Roll No: 140070003

under the guidance

of

Prof. Pushpak Bhattacharyya

RnD Project

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

Abstract

Information retrieval (IR) is the activity of obtaining information resources relevant to an information need from a collection of information resources. Searches can be based on full-text or other content-based indexing. Cross-language information retrieval (CLIR) is a subfield of information retrieval dealing with retrieving information written in a language different from the language of the user's query. For example, a user may pose their query in English but retrieve relevant documents written in Hindi. To do so, most of CLIR systems use translation techniques.

Sandhan - Indian language search engine that is being developed by IIT Bombay along with many other colleges uses cross lingual information retrieval. I will explain offline and online processing in search engine along with a tool that can allow one to track output of every module of a search engine. The tool provides the ability to perform pseudo error-correction by allowing the user to modify these outputs or tune parameters of the modules to check for improvement of results. This is crucial since it saves the immediate need to make changes in the system in terms of resource updation or development efforts as only some surface level changes can help in the improvement of the result quality.

In this report, I will explain query processing pipeline along with the efficacy of the tracker tool in Sandhan search with some java implementation detail.

Declaration

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 19th November, 2016 Saurabh Garg

Place: IIT Bombay, Mumbai Roll No: 140070003

Acknowledgements

I am thankful to the people who have been instrumental in helping me out throughout this project. First and foremost, I express my sincere gratitude towards my supervisor Prof. Pushpak Bhattacharyya for his guidance. My research in Cross Lingual Information Retrieval is being driven by his vision and support. I thank Arjun Atreya and Maulik Vacchani for there guidance throughout this project. For the work and experience that we have shared, I heartedly thank Sandhan team members at IITB. I am also thankful to my friends, family and teachers who have been always there for me whenever I needed them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Information Retrieval

1.1 Boolean Retrieval

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers). The **Boolean retrieval model**[7] is a model for information retrieval in which we can pose any query which is in the form of a Boolean expression of terms,i.e in which terms are combined with the operators AND, OR, and NOT. In **ad-hoc retrieval task**, a system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query. A document is relevant if it is one that the user perceives as containing information of value with respect to their personal information need.

Key terms :

Precision: Fraction of the returned results are relevant to the information need.

Recall: Fraction of the relevant documents in the collection were returned by the system.

Term-Document Incidence Matrix: Matrix with rows as words and columns as documents with each element (r, c) denotes presence of word r in document d.

Inverted index: An index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a set of documents.

Dictionary	**Posting List**
abc	$1 \rightarrow 2 \rightarrow 5 \rightarrow 6$
xyz	$4 \rightarrow 5 \rightarrow 8$
pqrt	$2 \rightarrow 5 \rightarrow 8$
...	...

Table 1.1: Inverted Index example

Posting List: Each document id is called a posting and a set of document ids is a postings list. Basic inverted index is a dictionary of terms each of which is associated with a postings list. Document frequency is length of each posting list. For example see 1.1

1.1.1 Processing Boolean Queries

Consider, a conjunctive query $a\ AND\ b$. Approach is to locate a and b in the Dictionary and then merge their retrieved posting lists.
Merging operation: Maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are x and y, the intersection takes O(x + y) operations. Formally, the complexity of querying is O(N), where N is the number of documents in the collection. Here an implicit assumption is to keep posting lists in the sorted order of docID.
To handle more complex queries like $a\ AND\ b\ AND\ c$, locate a, b and c in the Dictionary and retrieve their posting list. Merge smallest two posting lists initially and then this obtained posting list with third one. For arbitrary Boolean Queries, we have to evaluate and temporarily store the answers for intermediate expressions in a complex expression. However, generally queries that users submit are conjunctive in nature.

1.2 Term vocabulary and posting lists

To construct inverted index :

1. Collect the documents to be indexed
2. Tokenize the text.
3. Do linguistic pre-processing of tokens.
4. Index the documents that each term occurs in.

1.2.1 Document delineation

The first task is to choose a *document unit*. In simple cases each file is regarded as a document unit. But in many cases like in traditional Unix email file stored a sequence of emails which should be regarded as different files and in current days email may contain many attached documents which should be treated as separate files. Going in the opposite direction, various pieces of web software (such as latex2html) take things that you might regard as a single document (e.g., a Power-point file or a LaTeX document) and split them into separate HTML pages for each slide or subsection, stored as separate files. In these cases, you might want to combine multiple files into a single document.
More generally, for very long documents, the issue of *indexing granularity* arises. For a collection of books, it would usually be a bad idea to index an entire book as a document. A search for Indian languages might bring up a book that mentions India in the first chapter and languages in the last chapter, but this does not make it relevant to the query. Instead we might say each chapter or each paragraph or each line should be treated as separate document. Thus clearly there is a precision/recall trade off here. If the units get too small, we are likely to miss important passages because terms were distributed over several mini-documents, while if units are too large we tend to get spurious matches and the relevant information is hard for the user to find.

1.2.2 Determining the vocabulary of terms

Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called *tokens*, perhaps at the same time throwing away certain characters, such as punctuation. **Example :**

Input : Hey! How are you?

Output : Hey How are you

A **token**[11] is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. A **type** is the class of all tokens containing the same character sequence. A **term** is a type that is included in the IR systems dictionary.

The major problem of the tokenization phase is what are the correct tokens to use? In this example, it looks fairly trivial: you chop on whitespaces and throw away punctuation characters. This is a starting point, but even for English there are a number of tricky cases. For example, what do you do about the various uses of the apostrophe for possession and contractions? In term *aren't* we can not chop off at apostrophe and take *aren* and *t*. These issues of tokenization are language-specific. It thus requires the language of the document to be known. *Language identification* based on classifiers that use short character subsequences as features is highly effective; most languages have distinctive signature patterns.

One more problem of *hyphenation and spacing*. Hyphenation is used for various purposes ranging from splitting up vowels in words (co-education) to joining nouns as names (Hewlett-Packard). Handling hyphens automatically can thus be complex: it can either be done as a classification problem, or more commonly by some heuristic rules, such as allowing short hyphenated prefixes on words, but not longer hyphenated forms. Splitting on white space can also split what should be regarded as a single token. This occurs most commonly with names (San Francisco, Los Angeles) but also with borrowed foreign phrases (au fait) and compounds that are sometimes written as a single word and sometimes space separated (such as white space vs. whitespace).For all queries like lowercase, lower-case and lower case should return the same results. One effective strategy in practice, which is

used by some Boolean retrieval systems such as Westlaw and Lexis-Nexis, is to encourage users to enter hyphens wherever they may be possible, and whenever there is a hyphenated form, the system will generalize the query to cover all three of the one word, hyphenated, and two word forms, so that a query for lower-case will search for ”lower-case” OR lower case OR ”lowercase”.
Also different languages presents different problems like *French* (uses of apostrophe) and *Chinese* (no space delimiter).

1.2.3 Stop Words

Stop Words are words which do not contain important significance to be used in Search Queries. Usually these words are filtered out from search queries because they return vast amount of unnecessary information. The general strategy for determining a stop list is to sort the terms by collection frequency(the total number of times each term appears in the document collection) and take most frequent terms, often hand filtered and removed when creating index lists. But removing stop words may give a little harm like ”Prime minister of India” will be then ”Prime Minister” and ”India” if we remove ”of”.

1.2.4 Normalization

Token normalization is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. The most standard way to normalize is to implicitly create *equivalence classes*, which are normally named after one member of the set. An alternative to creating equivalence classes is to maintain relations between un-normalized tokens.
Some normalization forms :

1. *Accents and diacritics*

2. *Capitalization/case-folding*

Query Term	**Terms in documents that should be matched**
Windows	Windows
windows	Windows, windows, window
window	window, windows

Table 1.2: Asymmetric expansion of query

1.2.5 Stemming and lemmatization

Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. *Lemmatization* usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the *lemma*. The most common algorithm for stemming english is *Porter's Algorithm*. It consist of five phases of word reduction. **Example :**
Sample Text : Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation
Porter Stemmer Output : such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

1.2.6 Phrase Queries

Consider query like ”Prime Minister India”. It doesn't make sense to spit it by space and search for ”Prime” and ”Minister”. To be able to support such phrase queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms.
Biword Indexes : Thus in above we generate two vocabulary terms ”Prime Minister” and ”Minister India”.

This can work fairly well but their can be false positives. Thus we often first do Part of speech of Tagging. Also, this concept of biword can be extended to a sequence of words and if index includes variable length word sequences it is called *Phrase Index*.

1.3 Dictionary and tolerant retrieval

1.3.1 Search structures for dictionaries

Given an inverted index and a query, our first task is to determine whether each query term exists in the vocabulary and if so, identify the pointer to the corresponding postings.[8] The vocabulary look up requires one of the two data structure for efficiency: *hashing* and *Search Trees*.

Hashing has been used for dictionary lookup in some search engines. Each vocabulary term (key) is hashed into an integer over a large enough space that hash collisions are unlikely; collisions if any are resolved by auxiliary structures like bins. At query time, we hash each query term separately and following a pointer to the corresponding postings, taking into account any logic for resolving hash collisions.

The best-known search tree is the **binary tree**, in which each internal node has two children. The search for a term begins at the root of the tree. Each internal node (including the root) represents a binary test, based on whose outcome the search proceeds to one of the two sub-trees below that node.

1.3.2 Wildcard Queries

Wildcard query is useful when user if unsure of letters in between or is unaware of some letters at few positions in phrase. A query such as nam* is known as a trailing wildcard query, because the * symbol occurs only once, at the end of the search string. A search tree on the dictionary is a convenient way of handling trailing wildcard queries: we walk down the tree following the symbols n, a and m in turn, at which point we can enumerate

the set W of terms in the dictionary with the prefix nam. Finally, we use $|W|$ lookups on the standard inverted index to retrieve all documents containing any term in W. We also consider a reverse B-tree on the dictionary to handle queries like *nam which then can be handled in similar way.
Thus using a normal B-Tree and a reverse B-tree we can handle more general case like nam*fl by querying in B-tree and reverse B-tree and then taking intersection of results obtained. To handle general wildcard query we consider permutation index. First, we introduce a special symbol $ into our character set, to mark the end of a term. Next, we construct a permuterm index, in which the various rotations of each term (augmented with $) all link to the original vocabulary term. Consider the wildcard query m*n. The key is to rotate such a wildcard query so that the * symbol appears at the end of the string thus the rotated wildcard query becomes n$m*. Next, we look up this string in the permuterm index.
Whereas the permuterm index is simple, it can lead to a considerable blowup from the number of rotations per term. A k-gram is a sequence of k characters. In a k-gram index, the dictionary contains all k-grams that occur in any term in the vocabulary. Each postings list points from a k-gram to all vocabulary terms containing that k-gram.

1.3.3 Spelling correction

Two specific forms of spelling correction that we refer to as isolated-term correction and context-sensitive correction. In isolated-term correction, we attempt to correct a single query term at a time even when we have a multiple-term query.
Techniques for isolated term correction :

1. **Edit Distance :** Given two character strings s1 and s2, the edit distance between them is the minimum number of edit operations required to transform s1 into s2. This is done using dynamic programming paradigm.

2. **k-gram indexes :** k-gram index to retrieve vocabulary terms that have many k-grams

in common with the query.

1.3.4 Phonetic Correction

Algorithms for such phonetic hashing are commonly collectively known as *soundex* algorithms. Idea :

1. Turn every term to be indexed into a 4-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index.
2. Do the same with query terms.
3. When the query calls for a soundex match, search this soundex index.

Algorithm :

1. Retain the first letter of the term.
2. Change all occurrences of the following letters to 0 (zero): A, E, I, O, U, H, W, Y.
3. Change letters to digits as follows: B, F, P, V to 1.
   C, G, J, K, Q, S, X, Z to 2.
   D,T to 3.
   L to 4.
   M, N to 5. R to 6.
4. Repeatedly remove one out of each pair of consecutive identical digits.
5. Remove all zeros from the resulting string. Pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits.

1.4 Index Construction

1.4.1 Blocked sort-based indexing

The algorithm parses documents into termIDdocID pairs and accumu- lates the pairs in memory until a block of a fixed size is full. We choose the block size to fit comfortably into memory to permit a fast in-memory sort. The block is then inverted and written to disk. Inversion involves two steps. First, we sort the termIDdocID pairs. Next, we collect all termIDdocID pairs with the same termID into a postings list, where a posting is simply a docID. The result, an inverted index for the block we have just read, is then written to disk. In the final step, the algorithm simultaneously merges the ten blocks into one large merged index.[10]

1.4.2 Single-pass in-memory indexing

Blocked sort-based indexing has excellent scaling properties, but it needs a data structure for mapping terms to termIDs. For very large collections, this data structure does not fit into memory. A difference between BSBI and SPIMI is that SPIMI adds a posting directly to its postings list. Instead of first collecting all termIDdocID pairs and then sorting them (as we did in BSBI), each postings list is dynamic (i.e., its size is adjusted as it grows) and it is immediately available to collect postings. This has two advantages: It is faster because there is no sorting required, and it saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.

1.4.3 Distributed Indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines either according to term or according to document. MapReduce, a general architecture for distributed computing, is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines. A master node directs the process of assigning and reassigning tasks to individual worker nodes. The map and reduce

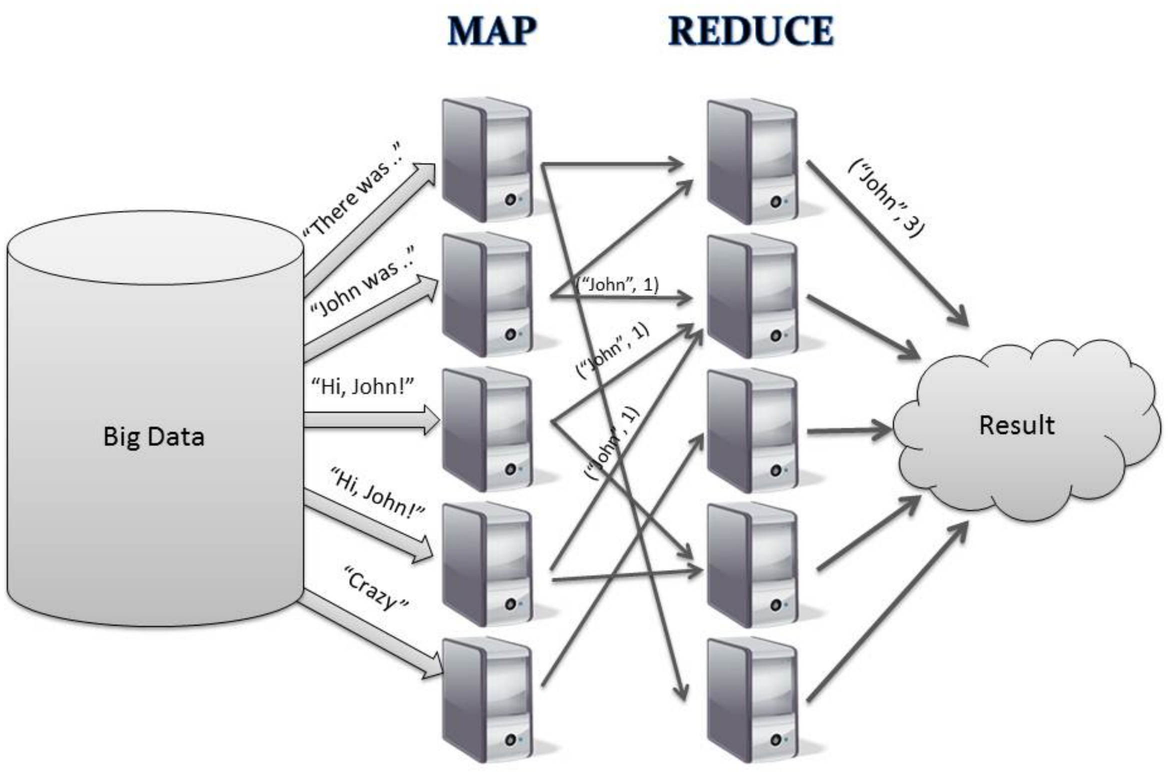


Figure 1.1: Map Reduce (Source: Google Images)

phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently. Splits are not preassigned to ma-

chines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine. In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs.
The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. For the reduce phase, we want all values for a given key to be stored close together, so that they can be read and processed quickly. Collecting all values for a given key into one list is the task of the inverters in the reduce phase. The master assigns each term partition to a different inverter and, as in the case of parsers, reassigns term partitions in case of failing or slow inverters.

1.5 Index Compression

Immediate benefit of compression is need less disk space. But there are two more subtle benefits. The first is increased use of caching. Search systems use some parts of the dictionary and the index much more than others. The second more subtle advantage of compression is faster transfer of data from disk to memory. Efficient decompression algorithms run so fast on modern hardware that the total time of transferring a compressed chunk of data from disk and then decompressing it is usually less than transferring the same chunk of data in uncompressed form.
Two kinds of compression techniques[9]: *lossless* and *lossy compression*. Better compression ratios can be achieved with lossy compression, which discards some information. Case folding, stemming, and stop word elimination are forms of lossy compression.

1.5.1 Heaps' Law: Estimation number of terms

Estimating number of distinct terms M is Heaps law, which estimates vocabulary size as a function of collection size:

$$M = kT^b$$

where T is number of tokens in the collection and b is less than 1.

1.5.2 Dictionary Compression

Using fixed-width entries for terms is wasteful. One can overcome these shortcomings by storing the dictionary terms as one long string of characters. The pointer to the next term is also used to demarcate the end of the current term.
One can further compress the dictionary by grouping terms in the string into blocks of size k and keeping a term pointer only for the first term of each block. We store the length of the term in the string as an additional byte at the beginning of the term. Their is one source of redundancy in the dictionary that consecutive entries in an alphabetically sorted list share common prefixes. This observation leads to front coding.

Chapter 2

Sandhan – Search Engine

A web search engine is a software system that is designed to search for information on the World Wide Web. The search results are generally presented in a line of results often referred to as search engine results pages (SERPs). The information may be a mix of web pages, images, and other types of files. A search system consists of two parts viz. offline processing and online processing.

2.1 Offline Processing

Offline processing mainly consists of two parts crawling and indexing. In crawling, documents from the web are fetched and stored. These documents have to be parsed and stored in optimal way in terms of both storage space and search time. For efficient retrieval of documents, an inverted index of terms in the documents is created.

2.1.1 Crawling

Injection step involves the injection of the seed URLs into the crawling pipeline. This step occurs only once. **Generation** step, the crawler generates the list of URLs to be fetched. This is done by scoring the URLs by some metric and then the top N URLs are selected. **Fetching** step, the web pages of the URLs generated in the Generate phase are downloaded.

Parse step, the downloaded URLs are parsed and the hyperlinks are extracted to be given to the Generate phase again. **UpdateDB**, step the crawlDB is updated with the newer URLs.

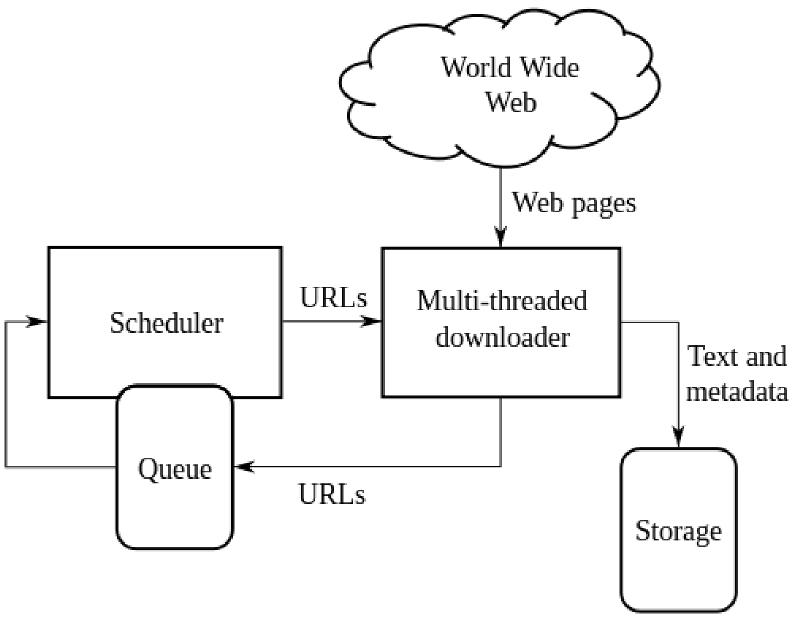


Figure 2.1: Web Crawling cycle (Source: Wikipedia)

2.1.2 Indexing

In this phase, the downloaded pages are sent to the Solr indexer to be indexed.

2.2 Online Processing

Online processing consists of converting the users information need in a format which facilitates the matching of query terms with terms in documents. Query expansion using feedback or thesaurus, semantic search, etc. are different ways of capturing user information need. A naive way to capture information need is to consider query terms as representative of user information need. These terms in the query have to be processed before they can be used to search documents. Processing of terms involves stop word removal, stemming and query formulation. Some of the search engines also do named entity recognition, multi-word recognition or word sense disambiguation to enhance query processing. This processing is done in the form of a pipeline where output of one stage is fed as input to the next stage.

2.2.1 Query Processing

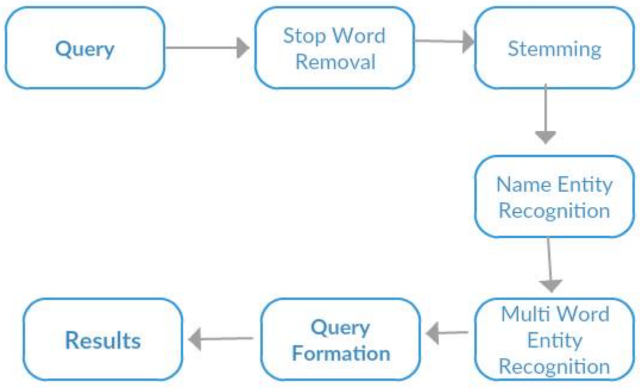


Figure 2.2: Query Processing Pipeline

Consider the diagram 2.2, a user's query is entered in to the pipeline and first stage is stop word removal where stop words like the, is etc. are removed as explained earlier1. Second stage is stemming where each word is stemmed[5] and then passed on to next stages for Name Entity and Multi word Entity recognition from the dictionary which is pre-computed. From this, a new Boolean Query is created which is used to fetch results. The quality of these set of results is a function of the accuracy of individual modules.

Chapter 3

Fora for IR Evaluation

List of fora for IR Evaluation

- **FIRE** (**F**orum for **I**nformation **RE**trieval)
- **TREC** (**T**ext **RE**trieval **C**onference)
- **NTCIR** (**N**II **T**estbeds and **C**ommunity for **I**nformation access **R**esearch)
- **CLEF** (**C**onference and **L**abs of the **E**valuation **F**orum)

provide platform to evaluate the system performance in terms of precision, recall, MAP value, etc. However, these measures indicate end to end performance of the system and do not evaluate performance of individual module in the system. Since the architecture is pipelined, the error propagates and multiplies. In such architecture, tracking the root cause of error is important.

3.1 Fire

Aims of FIRE[4]:

1. Large scale test collections for Indian Language Information Retrieval experiments

2. Provided a common evaluation infrastructure for comparing different information retrieval systems.

3. Encouraged research in Indian language access technologies

3.2 CLEF

CLEF provides[3]:

1. multilingual and multimodal system testing, tuning and evaluation

2. investigation of the use of unstructured, semi-structured, highly-structured, and semantically enriched data in information access

3. creation of reusable test collections for benchmarking

4. exploration of new evaluation methodologies and innovative ways of using experimental data

5. discussion of results, comparison of approaches, exchange of ideas, and transfer of knowledge.

3.3 TREC

Goals of TREC[1]:

1. to encourage research in information retrieval based on large test collections

2. to increase communication among industry, academia, and government by creating an open forum for the exchange of research ideas

3. to speed the transfer of technology from research labs into commercial products by demonstrating substantial improvements in retrieval methodologies on real-world problems

4. to increase the availability of appropriate evaluation techniques for use by industry and academia, including development of new evaluation techniques more applicable to current systems.

3.4 NTCIR

Aims of NTCIR[2]:

1. to encourage research in Information Access technologies by providing large-scale test collections reusable for experiments and a common evaluation infrastructure allowing cross-system comparisons

2. to provide a forum for research groups interested in cross-system comparison and exchanging research ideas in an informal atmosphere

3. to investigate evaluation methods of Information Access techniques and methods for constructing a large-scale data set reusable for experiments.

Tracker is a tool which captures the input and output information of each stage of the pipeline and displays it to the assessor/developer.

Chapter 4

Error Tracker and Pseudo Error Correction

4.1 Errors and impact on performance

Each module can contribute to the error and hence affect the overall performance of the system. Different kinds of error that can occur:

- In Stop word removal, consider the case that stop word not being detected by the module. This can boost non-relevant search results in the top because a high frequency of stop word content in it. On the other hand, consider the case in which an important word is removed as stop word and hence results corresponding to that are removed.
- In Stem detection two possible errors viz. no stem detection(Under Stemming) or wrong stem detection(Over Stemming). And this error will be propagated to further stages.
- Name Entity or Multi Word Entity might not be detected due to wrong stemming which will cause loss of important or close proximity information.

Thus there is need to track and there by correct error of each module. Changing the module for each error and retesting the system after change is quite time consuming. A simpler and elegant solution is to have error detection and pseudo error correction built with the search engine. This involves correcting the output of a particular module temporarily if needed without making a change in the module for detecting errors in further modules.

4.2 Tracker

Tracker[6] comes with capability of error detection and pseudo error correction after each module if needed in the User Interface of search engine itself. It is developed in order to assist developers and assessors to analyze each module for error and accordingly tune the required parameters which can improve the performance of search engine. Tracker captures input and output of each module for a query and displays them to the assessor/developer. The assessor/developer can then manually judge the outputs as correct or incorrect. This helps in detecting errors in modules.

Tracker also allows assessor/developer to replace the output of a particular module with the correct output without actually modifying the module. Once the assessor submits the query again after modifying the output of particular stage, the new corrected output will override the existing output of that stage and will be fed as input to next stage. This can be done incrementally to detect errors in all the modules. And finally necessary changes after analyzing the complete situation can be added to system by developer.

Chapter 5

Implementation details and Results

Tracker is developed in Java. Tracker has a web interface which is linked to the results page of the search engine. When a query is fired, the input and output of each module is stored in an object. This object persists till expiry of session. The values captured in the object are displayed to the assessors on the tracker web interface. The assessor is allowed to modify the output of any module. The modified output is stored back in the object and used for overriding the output of the stages for which modification is done. The object stores change information till either session expires or user fires a different query. In the latter case, the object is used for storing information about new query. Implementing Tracker on Sandhan involved two stages :
(i) Changes in User Interface
(ii) Changes in solr Code

5.1 User Interface Changes

On the homepage of Sandhan a separate search tab added that will search along with tracking error.5.1

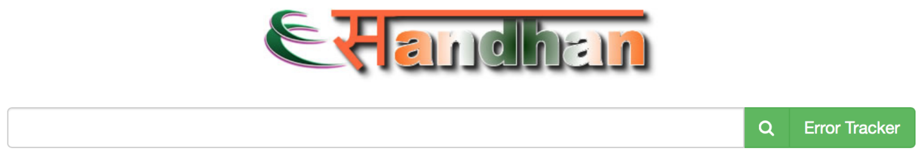


Figure 5.1: Sandhan homepage with added ”Error Tracker” option

Overview of User Interface implementation details in figure5.2. When a user input is sub-

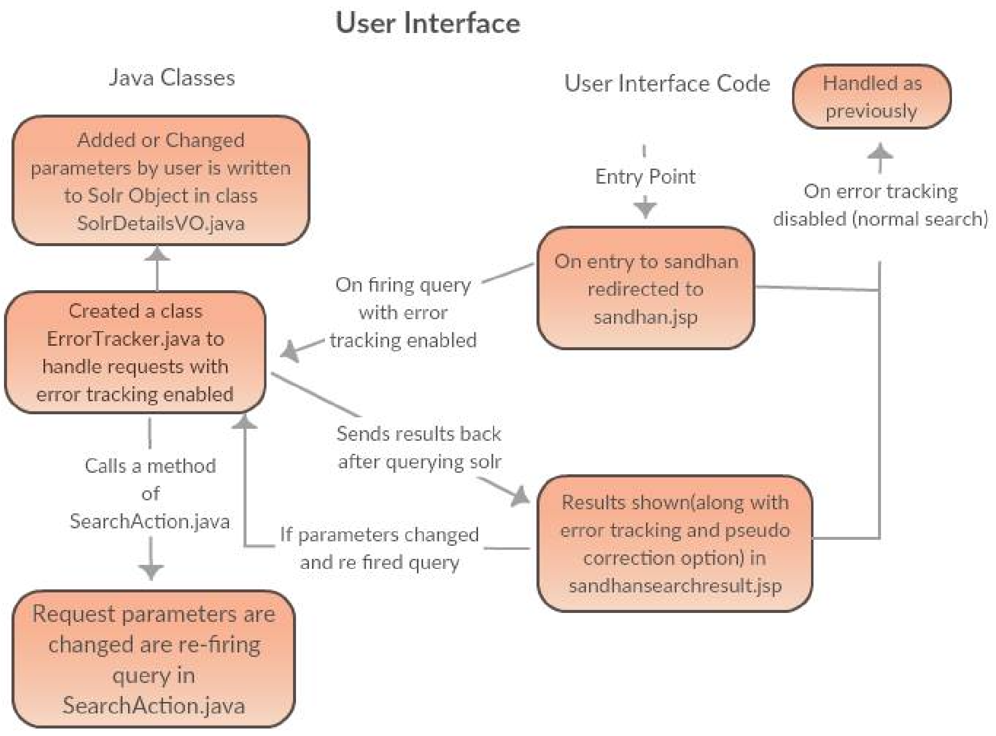

User Interface
Java Classes
User Interface Code
Handled as previously
Added or Changed parameters by user is written to Solr Object in class SolrDetailsVO.java
Entry Point
On error tracking disabled (normal search)
On entry to sandhan redirected to sandhan.jsp
On firing query with error tracking enabled
Created a class ErrorTracker.java to handle requests with error tracking enabled
Sends results back after querying solr
Calls a method of SearchAction.java
Results shown(along with error tracking and pseudo correction option) in sandhansearchresult.jsp
If parameters changed and re fired query
Request parameters are changed are re-firing query in SearchAction.java

Figure 5.2: User Interface Details

mitted and clicked on search, normal search will happen and user will be shown results, as it was done originally. But if user clicks on ”Error Tracker” normal results are shown along with output of each stage of pipeline which can be changed (for pseudo error correction) and query can be re-fired to see changes in results due to changed parameters.

5.2 SOLR Code Changes

Currently, version of Sandhan search engine uses *Apache-solr-3.4*. Overview of Implementation Details in figure5.3. When a query is received SolrParser creates name-value pair for each entry in user query and is added to solrParams class object. These name-value pairs are used everywhere else in the code for query processing particularly in DisMaxQParser.java and when any field is changed by user, corresponding changes are made to XMLWriter object in XMLWriter.java to pass changes back to user Interface.

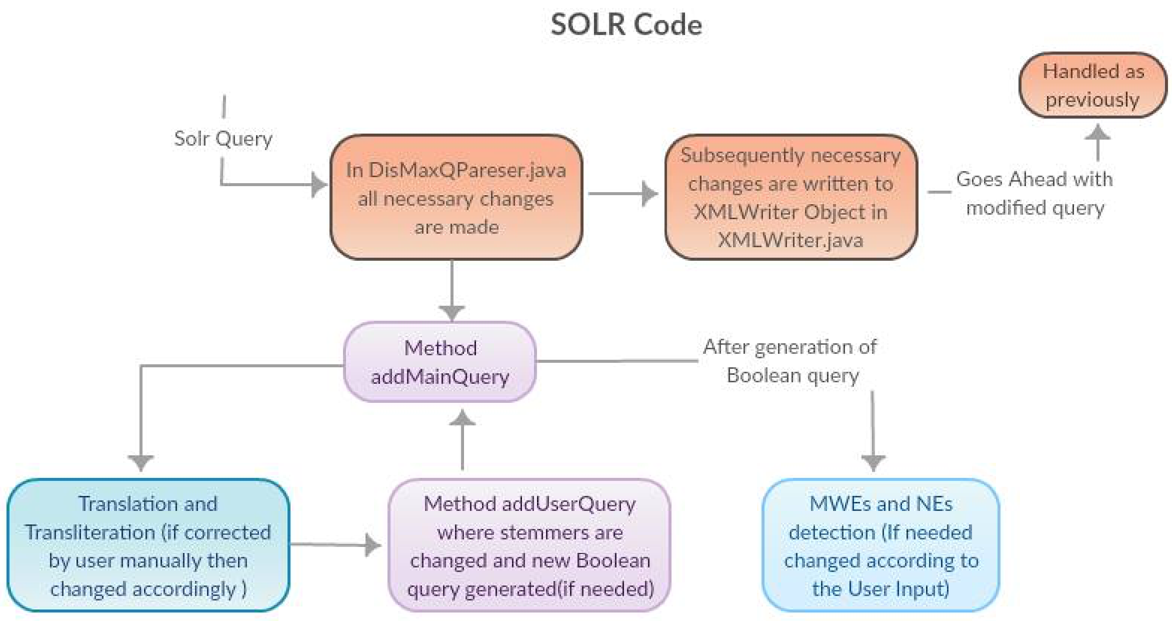

SOLR Code
Solr Query
In DisMaxQPareser.java all necessary changes are made
Subsequently necessary changes are written to XMLWriter Object in XMLWriter.java
Goes Ahead with modified query
Handled as previously
Method addMainQuery
After generation of Boolean query
Translation and Transliteration (if corrected by user manually then changed accordingly)
Method addUserQuery where stemmers are changed and new Boolean query generated(if needed)
MWEs and NEs detection (If needed changed according to the User Input)

Figure 5.3: Solr Code Details

5.3 Results

Consider query in figure 5.4

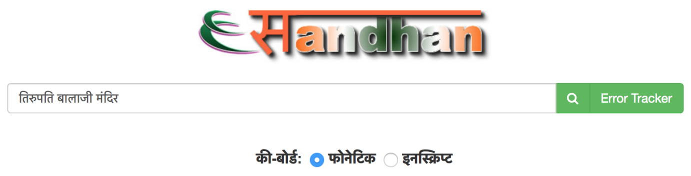
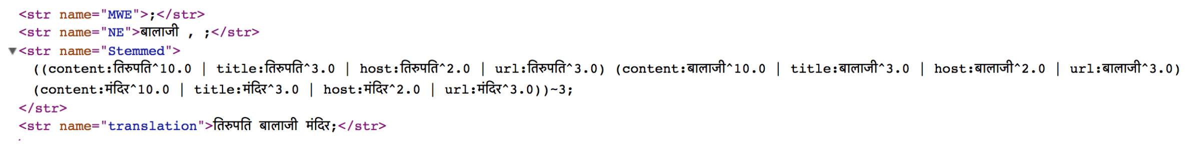


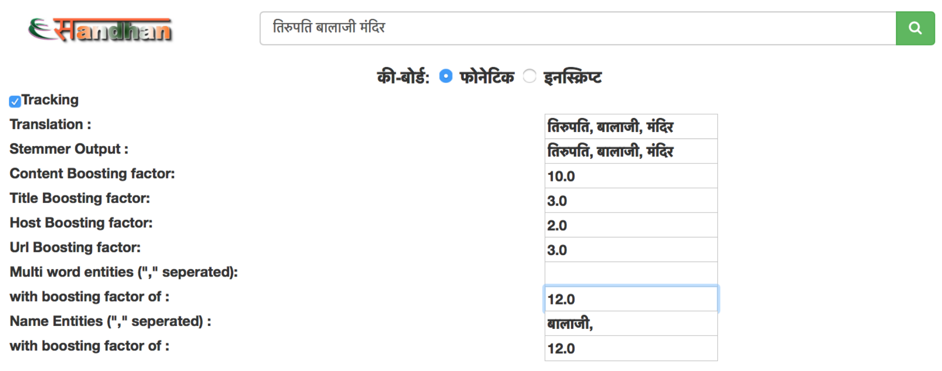
Figure 5.4: Marathi Query as User Input on Sandhan

Output of the solr code that is passed to user interface contains some extra fields along with original retrieved documents and meta data shown in 5.5 (Only extra added fields shown). Search Results shown on the user interface of the Sandhan looks like 5.6. The first column



```
  <str name="MWE">;</str>
  <str name="NE">बालाजी , ;</str>
▼<str name="Stemmed">
    ((content:तिरुपति^10.0 | title:तिरुपति^3.0 | host:तिरुपति^2.0 | url:तिरुपति^3.0) (content:बालाजी^10.0 | title:बालाजी^3.0 | host:बालाजी^2.0 | url:बालाजी^3.0)
    (content:मंदिर^10.0 | title:मंदिर^3.0 | host:मंदिर^2.0 | url:मंदिर^3.0))~3;
  </str>
  <str name="translation">तिरुपति बालाजी मंदिर;</str>
```

Figure 5.5: Solr Code Output



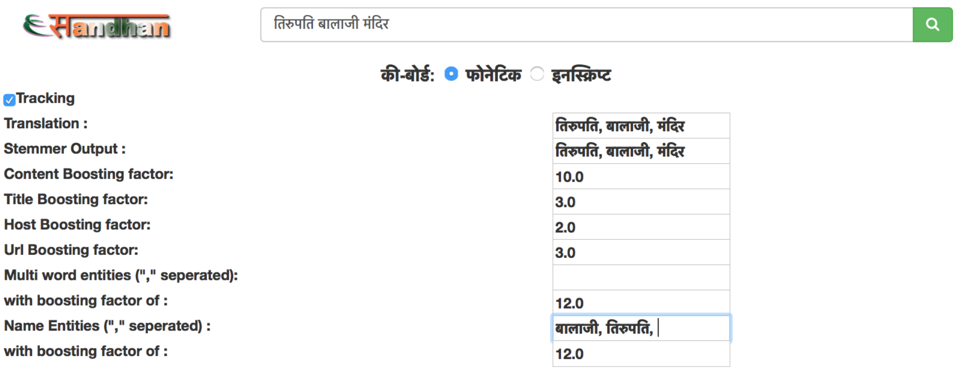
[✓ Tracking]

Translation :	तिरुपति, बालाजी, मंदिर
Stemmer Output :	तिरुपति, बालाजी, मंदिर
Content Boosting factor:	10.0
Title Boosting factor:	3.0
Host Boosting factor:	2.0
Url Boosting factor:	3.0
Multi word entities ("," seperated):	
with boosting factor of :	12.0
Name Entities ("," seperated) :	बालाजी,
with boosting factor of :	12.0

Figure 5.6: Results of Error Tracker on Sandhan on firing query with error Tracking on

specifies the level in module hierarchy, second denotes module name and the last column shows the output of the corresponding output. Since it is pipeline architecture, the output of one module directly forms input of other module and hence inputs are not explicitly shown. The assessors are allowed to edit one level at a time which enables tracking incremental changes.

Thus, on interface output contains necessary fields for error tracking and correction[1]. It also contains one check box for tracking on/off, if it is selected and query is re-fired after editing some fields, then these changed fields will overwrite the corresponding outputs in the query processing pipeline. Using this, a new Boolean query will be generated and search results will now correspond to the updated user query as shown in 5.7. If tracking check box is not checked, then search is directed to the normal search without error tracking outputs.



तिरुपति बालाजी मंदिर

की-बोर्ड: [✓ फोनेटिक] इनस्क्रिप्ट

[✓ Tracking]

Translation :	तिरुपति, बालाजी, मंदिर
Stemmer Output :	तिरुपति, बालाजी, मंदिर
Content Boosting factor:	10.0
Title Boosting factor:	3.0
Host Boosting factor:	2.0
Url Boosting factor:	3.0
Multi word entities ("," seperated):	
with boosting factor of :	12.0
Name Entities ("," seperated) :	बालाजी, तिरुपति,
with boosting factor of :	12.0

Figure 5.7: Results of Error Tracker on Sandhan after editing NEs and re-firing query

[1]Note: Search results also include relavent web pages (not shown here).

Chapter 6

Conclusion

In this report, I started with Boolean Information Retrieval theory and offline & online processing in a search engine, followed up by list of Fora for IR evaluation, followed by importance and need of error tracker in search engine by highlighting benefits it can provide to assessors. Tracker facilitates detection of errors in different modules of the search engine. Pseudo error correction of outputs helps discovering further errors in the system without making a change in the module. Also, along with this various stage of Query processing was briefly explained. This error tracker and corrector can be extended for large scale applications.

Bibliography

[1] Text REtrieval Conference (TREC) home page http://trec.nist.gov/. 1992.

[2] NTCIR home http://research.nii.ac.jp/ntcir/index-en.html. 1999.

[3] The CLEF initiative (Conference and Labs of the Evaluation Forum) homepage http://www.clef-initiative.eu/. 2000.

[4] FIRE - Forum for Information Retrieval Evaluation http://www.isical.ac.in/~clia/. 2008.

[5] M. Bapat, H. Gune, and P. Bhattacharyya. A paradigm-based finite state morphological analyzer for marathi. *Proceedings of the 1st Workshop on South and Southeast Asian Natural Language Processing (WSSANLP)*, pages 26–34, 2010.

[6] S. Chaudhury, A. Atreya, P. Bhattacharyya, and G. Ramakrishnan. Error tracking in search engine development,. *3rd Workshop on South East and South Asian NLP, part of COLING*, 2012.

[7] C. Manning, P. Raghavan, and H. Schtze. Boolean retrieval, introduction to information retrieval, cambridge university press. 2008.

[8] C. Manning, P. Raghavan, and H. Schtze. Dictionaries and tolerant retrieval, introduction to information retrieval, cambridge university press. 2008.

[9] C. Manning, P. Raghavan, and H. Schtze. Index compression, introduction to information retrieval, cambridge university press. 2008.

[10] C. Manning, P. Raghavan, and H. Schtze. Index construction, introduction to information retrieval, cambridge university press. 2008.

[11] C. Manning, P. Raghavan, and H. Schtze. The term vocabulary & postings lists, introduction to information retrieval, cambridge university press. 2008.