AIW & Information Retrieval (UE18CS322)
Unit 2

Aronya Baksy

February 2021

1 Hardware Basics and BSBI

1.1 Hardware Basics

- The memory hierarchy of a system comprises of the registers, cache, RAM, ROM (magnetic drives, tape drives or SSDs) and peripheral storage (USB flash drives etc).
- Registers are the fastest but also the most expensive per byte of memory. Hence registers are also the smallest in capacity.
- Registers are a subset of the cache, cache is a subset of the RAM and so on.
- For a magnetic disk, the total memory access time comprises of seek time (time needed to move the disk head to the right cylinder), the transfer time (time needed to transfer bytes from disk to I/O controller) and the rotational latency (time needed for the right sector to fall under the disk head to be read).
- Contiguous memory access is faster than accessing non-contiguous chunks of data.
- The DMA controller controls the system bus that transfers data, hence the processor is free to process data during this operation.
- An efficient set of compression and decompression algorithms can make the time needed to read compressed data much less than the time for uncompressed data.

1.2 Blocked Sort Based Indexing (BSBI)

- Sorting is a key step of inverted index construction (see Unit 1). Hence efficient sorting algorithm that takes memory structure into consideration is useful.
- An *external sort algorithm* (one that uses data stored on secondary storage) is needed as large corpora will simply not work with internal sorting algorithms (that load all data into the main memory)
- Terms are represented as integer term IDs (instead of strings). This puts an upper bound on the memory needed to store the terms in the inverted index.
- The construction of the term - term ID mapping is either done during the inverted index construction (single pass approach) or done separately before the actual index construction (2-pass approach)
- The steps involved in BSBI are:
  1. The algorithm parses documents into termID–docID pairs and accumulates the pairs inmemory until a block of a fixed size is full ($ParseNextBlock()$)
  2. The $BSBI\_Invert(block)$ function then sorts the termID–docID pairs and collects all termID–docID pairs with the same termID into a postings list where a posting is simply a docID.
  3. In the final step, the algorithm simultaneously merges the blocks into one large merged index

Algorithm 1 Blocked Sort-based Indexing Algorithm

procedure BSBI()
  $n \leftarrow 0$
  while all docs not processed **do**
    $n \leftarrow n + 1$
    $block \leftarrow ParseNextBlock()$
    $BSBI\_Invert(block)$
    $WriteToDisk(block, f_n)$
  $MergeBlocks(f_1, f_2, ..., f_n; f_{merged})$

- Time complexity of BSBI is $\Theta(TlogT)$ where T is number of terms.
- Merging process ($MergeBlocks()$) is outlined as follows:
  1. Open each block file and maintain one read buffer for each
  2. Open the output file and maintain a single write buffer for it
  3. For each iteration:
     (a) Read from all read buffers simultaneously
     (b) Select the lowest term ID that is not yet processed
     (c) Read and merge all posting lists for this term ID
     (d) Write this merged output using the write buffer
     (e) Refill all read buffers when needed
- This merge minimizes number of disk seeks as it moves linearly through each block.
- The limitations of BSBI are:
  – It assumes that document corpus fits on disk
  – It assumes that corpus is static
  – The term-term ID mapping has to be carried in memory throughout the execution

1.3 Single Pass In-Memory Indexing (SPIMI)

- BSBI scales well, but has disadvantages seen above. Also it involves the expensive step of sorting blocks and storing them, and storing the term-term ID mapping
- SPIMI uses the terms directly. The algorithm takes in a stream of ⟨ term ID, doc ID ⟩ pairs that are generated from the document corpus.
- The SPIMI algorithm comprises of the following steps:
  1. Till main memory is exhausted work with the current block
     (a) Sequentially access the documents. Let docID be the ID of the currently accessed document
        i. If the term is new, create a new entry for that term in the block dictionary
        ii. Else identify the list for the term in the dictionary
        iii. If the list is full double it's size
     (b) Add doc ID to the posting list for the term.
  2. Sort on terms for lexicographical ordering
  3. Write index block into disk and start a new block and its corresponding fresh dictionary.
- Posting lists here are dynamically growing, as against in BSBI where the posting list is constructed only once for all the term ID -doc ID pairs in a single block.
- Sorting of each block is done in order to facilitate the merging step

Algorithm 2 Single Pass In-Memory Indexing

```
procedure SPIMI(tokenStream)
    outFile = New_File()
    dict = New_HashTable()
    while memory is available do
        token ← getNextToken(tokenStream)
        if term(token) ∉ dict then
            PostingList = addToDict(dict, term(token))
        else
            PostingList = getPostingList(dict, term(token))
        if full(PostingList) then
            PostingList = doublePostingList(dict, term(token))
        AddToPostingList(PostingList, docID(token))
    sortedTerms ← SortTerms(dict)
    WriteBlockToDisk(sortedTerms, dict, outFile)
    return outFile
```

- Both the postings and the dictionary terms can be stored compactly on disk using compression.
- Compression increases the efficiency of the algorithm further because larger blocks can be processed, and because the individual blocks require less space on disk.
- Same limitations as BSBI exist here too

2 Distributed and Dynamic Indexing

2.1 Distributed Indexing

2.1.1 Map-Reduce and HDFS

- HDFS is a fault-tolerant distributed file system that offers high throughput access to data while being deployed on commodity hardware.
- The key ideas behind HDFS are:
  - Data **replicated across** multiple nodes for better fault tolerance
  - **Move computation to the data** to address data locality and minimize network latency due to data transfer (traditional approaches would move the data towards the computation)
  - Simple programming model that ensures minimal amount of data movement over the network
- MapReduce is a programming framework that enables large computations over a distributed cluster. The generic framework is summarized as follows:

Phase	**Input**	**Output**
Map	$\langle k_1, v_1\rangle, \langle k_2, v_2\rangle, \langle k_3, v_3\rangle, ..., \langle k_n, v_n\rangle$	$\langle\langle k_1, f(v_1)\rangle, \langle k_2, f(v_2)\rangle, ..., \langle k_n, f(v_n)\rangle\rangle$
Shuffle	$\langle\langle k_1, f(v_1)\rangle, \langle k_2, f(v_2)\rangle, ..., \langle k_n, f(v_n)\rangle\rangle$	$\langle k_1, \langle v_a, v_b, v_c\rangle, \langle k_2, \langle v_p, v_q, v_r, ..\rangle, ...\rangle$
Reduce	$\langle k_1, \langle v_a, v_b, v_c\rangle, \langle k_2, \langle v_p, v_q, v_r, ..\rangle, ...\rangle$	$\langle k_1, v_{out1}\rangle, \langle k_2, v_{out2}\rangle, ..$

- The **map** phase applies the map function on each value of the key-value pairs and outputs them.
- The **shuffle** phase takes input from map phase, and combines all the values under a single key into a list of values under that key.
- The **reduce** phase reduces the key, list of values pairs into key, value pairs.
- The optional **partitioner** phase takes place after map and before reduce. The data is partitioned according to the defined partition funciton, and number of partitions is same as number of reducers.

- The optional **combiner** phase pre-aggregates all the values of the same key within the mapper itself (rather than at the shuffle phase which has to take input from *all the mappers*).
- The partitioner and combiner are optimizations that increase performance and reduce the data transfer over network.

2.1.2 Index construction using Map-Reduce

- The document collection is split into blocks called *splits*. Splits are assigned to worker nodes by a master node.
- The master node is also capable of restarting a job on a different node if a particular split is not behaving properly (too slow or just failed).
- The **parser** (running on the mapper nodes) takes the documents and turns them into a stream of $\langle term, docID \rangle$ pairs. The parser writes them into the segment files (partitions)
- The segment files are made based on the first letter of the term and number of partitions j (eg: if $j = 3$ then one possible partition is a-f, g-p and q-z).
- The **inverter** (running on the reducer node) collects all the postings for a single term, aggregates them into a list and returns the posting list as output for the term.
- Each segment file requires only a sequential read as all data relevant to an inverter are only in written to a single segment file by the parser. This minimizes network traffic during indexing
- If term IDs are to be used instead of terms themselves, then the term-term ID mapping can be precomputed and stored on all the mapper nodes.
- The current result is a term-partitioned index (wherein each reducer only handles a subset of the terms) as against a document-partitioned index (each reducer handles a subset of documents) which is preferred for many search engines. This involves a transformation step to be discussed later.

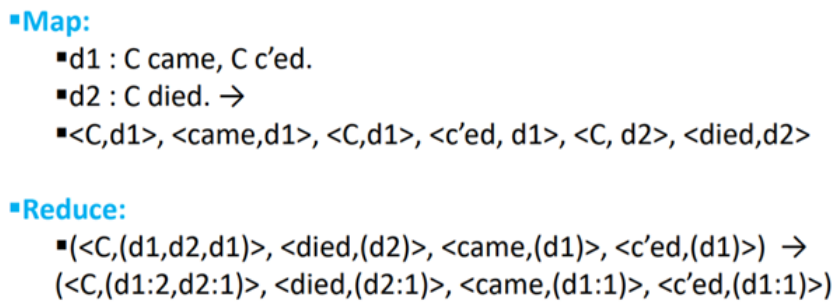

Figure 1: Map Reduce Index Construction

2.2 Dynamic Indexing

- Indexing method for dynamic corpora (changing with time).
- The simplest approach to dynamic indexing is as follows:
  1. Maintain a single main index on disk
  2. Each time a document is added add it to an auxiliary index maintained in main memory.
  3. Searching is carried out across both indices and the results are merged.
  4. Periodically, the main and auxiliary indices have to be merged when the auxiliary index becomes too large for the main memory.
  5. A bit vector is maintained for all documents. When a document is deleted, its bit is flipped. The set of documents returned by a query is filtered according to this bit vector.

- This strategy works well but has a few drawbacks:
  - Frequent merging operations can be costly depending on how the main index is stored on the file system:
    * If each posting list in the main index has its own file then merging is not very costly.
    * But each posting list in separate file means there are as many files as terms in the corpus. File system will not be able to handle so many open files.
    * The alternative is to merge efficiently, and maintain large index files. (in this discussion it is assumed that the index is a single large file)
- This approach has complexity $\Theta(T^2/n)$ where n is the size of the auxiliary index and T is the number of postings.

2.2.1 Logarithmic Merging

- Maintain indices $I_0, I_1, I_2, .., I_n$ on disk where the size of index I_i is $2^i \times n$ postings.
- Upto n postings are accumulated in the in-memory index called Z_0. When Z_0 becomes full, the n postings in Z_0 are transferred to index I_0 on disk.
- The next time that Z_0 becomes full, it is merged with I_0 to create Z_1. The size of Z_1 is $2^1 \times n$.
- If there is no index I_1 on disk then Z_1 is stored to disk as I_1. But if I_1 exists on disk then I_1 and Z_1 are merged to create Z_2, and so on.
- A query is serviced by checking Z_0 in memory as well as the indices I_i on disk. The results are merged from all these $log(T/n)$ indices.
- Overall index construction time is $\Theta(Tlog(T/n))$ because each posting is processed only once on each of the $log(T/n)$ levels.

2.2.2 Dynamic Indexing: Real world applications

- Having multiple indexes complicates the maintenance of collection-wide statistics
  - Affects spelling correction (querying multiple indexes + invalidation bit vector makes things more complicated)
  - Dynamic indexing is implemented at large search engines with frequent small increments.
  - Occasional complete rebuild of the index becomes harder with increasing size (e.g. Google would take several decades to rebuild its document index)
  - During the complete rebuild, query processing is switched to the new index and the old index is deleted

3 Index Compression

- Compression saves disk space, increases speed (by keeping more information in memory) and saves on disk seeks needed to read large amt of uncompressed data.
- Dictionary compression - keep in main memory, Index compression- reduce disk space and disk read latency
- Compression makes sense only if decompression algorithms are efficient.
- Lossy compression involves discarding some information. Techniques such as stop-word removal, stemming, downcasing, eliminating numbers are examples of lossy compression
- Lossless compression involves information being preserved even after the compressed data is decompressed.

3.1 Heap's Law and Zipf's Law

3.1.1 Heap's Law

- Heaps' law is that the simplest possible relationship between collection size and vocabulary size is linear in log–log space
- Heap's Law models the number of unique terms M as a function of the number of tokens in the collection T

$$M = kT^b \tag{1}$$

where $30 \leq k \leq 100$ and $b \approx 0.5$

- In the log-log space, the Heap's law equation transforms to

$$log(M) = k + b\ log(T) \tag{2}$$

which is linear.

- Heap's Law suggests that
  1. The dictionary size continues to increase with more documents in the collection, rather than a maximum vocabulary size being reached
  2. The dictionary size is large for large collections.

3.1.2 Zipf's Law

- Zipf's law models the frequency distribution of terms in a document corpus.
- Let cf_i be the collection frequency of the i^{th} most common term in the collection. Then Zipf's Law states that:

$$cf_i \propto \frac{1}{i} \tag{3}$$

- e.g. if the most frequent term has CF of k, then second most frequent will have CF $\frac{k}{2}$, third most frequent has $\frac{k}{3}$ and so on.
- Zipf's Law is also a power law as it can be written as

$$cf_i = \frac{c}{i} \tag{4}$$

$$log(cf_i) = log(c) - \ log(i) \tag{5}$$

- Key insight from Zipf's law: Few frequent terms, many rare terms

3.2 Dictionary Compression

3.2.1 Level 1

- Example corpus here is Reuters RCV1 with 400k terms.
- Assume dictionary implemented as an array of fixed-width entries (20 bytes for term, 4 bytes for document frequency, and 4 bytes for ptr to posting list). Then for RCV1, the size is 11.2 MB
- Giving 20 bytes for word wastes space for majority of words and doesn't handle words larger than 20 chars long.
- Average size of dictionary words is 8 chars.
- Solution: Store all terms as one long string, sorted in lexical order. The pointer to the next term is also used to demarcate the end of the current term.
- Pointer has to distinguish up to 400k $\times$ 8 = 3.2 million positions. Hence needs to be 22 bytes $\rightarrow$ 3 bits long.
- Now each entry is of size (4 for doc freq, 4 for posting ptr, 3 for term ptr, 8 for term) = 19 bytes, which leads to index size of 7.6 MB

3.2.2 Level 2: Blocked Compression

- Divide string into blocks of size k terms. Store only the pointer to the beginning of the block. In addition, at the beginning of each term in the string, store the length of the term in a single byte
- This saves $(k-1) \times 3$ bytes for term pointers, but adds k bytes for the term lengths. (saving for each block)
- Assume $k = 4$. Then saving per block is 5 bytes. Then saving overall is $\frac{400k \times 5}{4}$ which is 500kB. Hence new index size is 7.6-0.5 = 7.1 MB
- Disadvantage is that lookup speed decreases for large value of k (even though large k leads to more compression)

3.2.3 Level 3: Front Coding

- Since the terms are sorted lexicographically, there are sequences of consecutive terms in the string that share the same prefix.
- Once an unique prefix is identified, the end is denoted as *. In subsequent entries having that prefix it is denoted with ◊
- Experiments show that RCV1 dictionary size drops to 5.9 MB due to this
- e.g.: 8automat*a1◊e2◊ic3◊ion
- Above denotes "automata, automate, automatic, automation"

3.3 Posting List Compression

3.3.1 Level 1: Run Length Compression

- Instead of storing the posting list as a list of Doc IDs, store only the first doc ID and the rest of them as "gaps" between 2 doc IDs. (e.g. If the original list is 33, 47, 154, 159, 202 then the gap version is 33, 13, 107, 5, 43)
- Issue is that for rare words, gaps are large and need large space to store (almost as large as doc IDs themselves) but for frequent words this leads to lots of space saved.
- A variable encoding method is used that needs fewer bits for short gaps.

3.3.2 Variable Byte Codes

- The first bit (from the left) of each byte indicates whether that byte is the last byte of the current stream or not
- To convert any number from base 10 to its VB encoding, split into 7 bit substrings (from the rightmost string of 7 bits and then so on). Now at the head of each 7 bit string put 1 if it is the last byte or 0 if it is not
- This results in a sequence of 8-bit (byte) strings.
- e.g. 829 in base 10 is represented as 11 0011 1101.
  - Split into 7 bit strings from the right (pad 0s where needed) gives 0000110 0111101
  - Put 0 if byte is not last byte or 1 if it is last byte. This gives **0**0000110 **1**0111101. This is the VB encoded form of the number 829.

3.3.3 Bit-Level Code (Gamma Code)

- An **unary** code is a bit-level code. (Unary code for n is n ones followed by a 0. e.g. unary code for 5 is 111110)
- The γ code is a 2-tuple consisting of a length and an offset. For a value G,
  1. Offset is G in binary with the leading bit removed. e.g. length of 13 is 1101 without the leading bit so 101
  2. Length is the length of the offset string, represented in Unary code. For 13, offset is 101 so length is 3, in unary that is 1110
  3. Hence the γ code for 13 is (length, offset) which is (1110, 101)
- NOTE: 0 has no γ code, the γ code for 1 is 0.
- The γ code encodes a value G in $2\lfloor log_2(G) \rfloor + 1$ bits.
- Just like the VB code, the γ code is also uniquely prefix decodable. Read from L to R, as soon as you encounter a 0, you have reached the end of the length field.
- γ code can be used for any distribution (rare or common) and is independent of corpus size or any other parameters

3.3.4 Implementation Notes

- Computer systems have word boundaries as powers of 2 bits. As γ is a variable-bit length encoding scheme, it is impractical to use (operations that straddle a word bdy are slower)
- In this scenario VB code is more suitable as it is at the byte level, with only very small penalty in terms of space compared to γ code.

4 Term Frequency, Weighting and Scoring

- Boolean queries tend to either come up with too many irrelevant results, or too few good ones. It is hard to craft a good Boolean query in this case.
- In **ranked retrieval**, the system returns an ordering over the top documents that were retrieved for a query. A score (say in the range $[0, 1]$) is assigned to each document as a measure of its relevance to the query.
- Jaccard Similarity is useful as a scoring system. The drawbacks however are:
  - Biased against longer sentences
  - Doesn't consider term frequency information (rare terms contain more info than frequent terms)
- The Bag-of-Words model treats each document as a vector of term counts. But it does not encode any position information

4.1 TF-IDF Model

- **Term Frequency** $tf(t, d)$ is the number of times a term t appears in a specific document d.
- The weighted TF is given as

$$w_{t,d} = \begin{cases} 1 + log_{10}(tf_{t,d}) \text{ if } tf_{t,d} > 0 \\ 0 \text{ if } tf_{t,d} = 0 \end{cases} \tag{6}$$

- The Document Frequency df_t of a term t over a document corpus D is the number of documents that have atleast one occurrence of term t.

- The **inverse document frequency** is the quantity

$$idf_t = log_{10}\left(\frac{N}{df_t}\right) \tag{7}$$

where N is the number of documents.

- Hence the tf-idf weight is the product of tf and idf weights.

$$W_{t,d,D} = (1 + log_{10}(tf_{t,d})) \times log_{10}\left(\frac{N}{df_t}\right) \tag{8}$$

- For a query q, the tf.idf score for a given document d is

$$score = \sum_{t \in q \cap d} tf.idf(t,d) \tag{9}$$

4.2 Similarity Measures for Vector .Space models

- Euclidean distance is a bad idea because:
  - Euclidean distance is large for vectors of different lengths but they contain similar information with different magnitude
  - e.g. Append document d to itself to make d'. The euclidean distance between d and d' may be v large but they are v similar semantically (the **angle** between them is 0)
- Angles between vectors are measured in terms of cosine because cosine decreases as angle increases. Large angle leads to less similarity and less cosine, hence makes sense
- The cosine similarity between 2 vectors is given as

$$sim(v_1, v_2) = \frac{v_1.v_2}{||v_1|| \times ||v_2||} \tag{10}$$

- The two vectors are **length normalized** before the cosine similarity is computed. This is so that the effect of the lengths of the 2 vectors is nullified, but the intensity of the content still remains.

```
CosineScore(q)
 1  float Scores[N] = 0
 2  float Length[N]
 3  for each query term t
 4  do calculate w_{t,q} and fetch postings list for t
 5     for each pair(d, tf_{t,d}) in postings list
 6     do Scores[d] += w_{t,d} × w_{t,q}
 7  Read the array Length
 8  for each d
 9  do Scores[d] = Scores[d]/Length[d]
10  return Top K components of Scores[]
```

Figure 2: Algorithm for computing Cosine Similarity for a query q

5 Efficient Scoring and Ranking

- TF-IDF matrix can be computed :
  - One **Term at a Time** (TAAT). This involves summing over all the terms to get a document's score
  - One **Document at a Time** (DAAT). This involves calculating tf-idf for one document at a time.
- A **safe ranking** is a method that guarantees that the K documents returned are the absolute highest scoring documents.
- A non safe ranking does not provide any such guarantees. However from the user perspective even an approximation is sufficient to provide results that are usable.
- The generic approach is to select a set A of size $K < |A| << N$. The set A does not necessarily contain the top K, but has many docs from among the top K. Return the top K docs in A

5.1 Index Elimination

- One strategy is to *eliminate terms with low IDF* from the query, as they don't contribute much to the final score but still have large number of associated documents
- Another strategy is to only count *documents containing many query terms*, for multi-term queries. Imposes a "soft conjunction" (implicit conjunction) on queries seen on web search engines

5.2 Champion Lists

- For each dictionary term t, precompute the r highest weighted documents in the posting list of t. This list is the champion (aka fancy) list for t.
- It is possible that $r < K$ as the champion list is built during the index construction.
- At query time, only compute scores for docs in the champion list of some query term, and pick the K top-scoring docs from amongst these

5.3 Static Quality Score

- Documents can be ranked across the dimensions of relevance (using cosine similarity) and authority (using some query independent parameter)
- Measures of authority include number of Wikipedia links, number of bit.ly etc. links, number of citations, normalized into a range between 0 and 1.
- This authority measure for a document is denoted as a goodness score $g(d)$. The overall score for a term-document pair is $cosine(t, d) + g(d)$
- Ordering posting list for terms by $g(d)$ allows top K documents to be retrieved earlier, thus allowing concurrent traversal for intersection as well as cosine similarity computation
- The idea of champion list is also combined with the goodness score. • Maintain for each term a champion list of the r docs with highest $g(d) + tf - idf_{t,d}$
- Seek top-K results from only the docs in these ordered champion lists

5.4 Impact-Ordered Postings

- Sort each postings list by tf-idf weight $wf_{t,d}$.
- Follow the same algorithm but suitably modify for storing the partial scores for each doc as not all postings are in a common order

5.4.1 Early Termination

- When traversing t's postings, stop early after either
  - a fixed number of r docs
  - wft,d drops below some threshold
- Take the union of the resulting sets of docs (one doc list from each query term)
- Compute only the scores for docs in this union

5.4.2 idf Ordered Terms

- High idf terms are more likely to contribute to the highest ranking documents. Hence look at query terms ordered by their IDF.
- As the document score is updated for each term, if the score remains almost constant then stop traversing the query terms.

5.5 High and Low List

- For each term, the high list is the champion list.
- For each query term, traverse the docs in the high list first. If more than K documents are retrieved then get the top K and stop, else continue on to the low list
- This strategy can be used even for simple cosine scores, without global quality $g(d)$

5.6 Cluster Pruning

- Pick $\sqrt{N}$ documents as cluster centres or leaders. Cluster the documents about these leaders. By this arrangement now each leader has $\sqrt{N}$ followers.
- Given query Q, find its nearest leader L, and seek K nearest docs from among L's followers
- Leaders chosen at random are more likely to reflect the real data distribution.
- Modifications:
  - Bring more "fuzziness" into the method, have each follower attached to $b_1 = 3$ (say) nearest leaders.
  - From query, find $b_2 = 4$ (say) nearest leaders
  - Since each follower can attach to b_1 leaders, we are now looking at a larger set of followers.
  - Recursive leader-follower construction.

5.7 Parameterized and Zoned Indices

- A document is more than a stream of tokens. It contains field each having its own syntax (Author, Year, Title, Language, Format), and these fields comprise the metadata of the document
- Field or parametric index: postings for each field value
- A zone is a region of the doc that can contain an arbitrary amount of text, e.g., Title, Abstract, References. Build inverted indexes on zones as well to permit querying
- This can be implemented as:
  1. Having zone-specific tokens in the vocabulary (e..g "william.abstract" and "william.title" specify the term "william" occuring in the abstract and title of a dopcument)
  2. Single term but posting contains only the doc id but also the zone information

5.8 Tiered Index

- Break postings up into a hierarchy of lists (from most to least important), ordered by measure like $g(d)$
- Inverted index thus broken up into tiers of decreasing importance
- At query time use top tier unless it fails to yield K docs, if so drop to lower tiers

5.9 Free Text Queries

- Combination of phrase queries, sub-phrase (2 words at a time and so on) and vector space queries.
- The latter 2 are run only if the documents returned till that time are less than K
- Rank matching docs by vector space scoring
- Combination of multiple scoring algorithms (proximity, similiarity, static quality) done using expert-tuned approaches or using supervised machine learning approaches.

6 Components of an IR System

- Tokenization and normalization
- Processed output of step 1 in cache for generating snippets used in retrieval results
- Multiple indexes are built (can be more than four. Inexact means optimized and k-gram index is useful for spellingcorrection) in parallel
- Query parser submits query to the indexes
- Query parser also submits query after spelling correction
- Final score is aggregated using ML

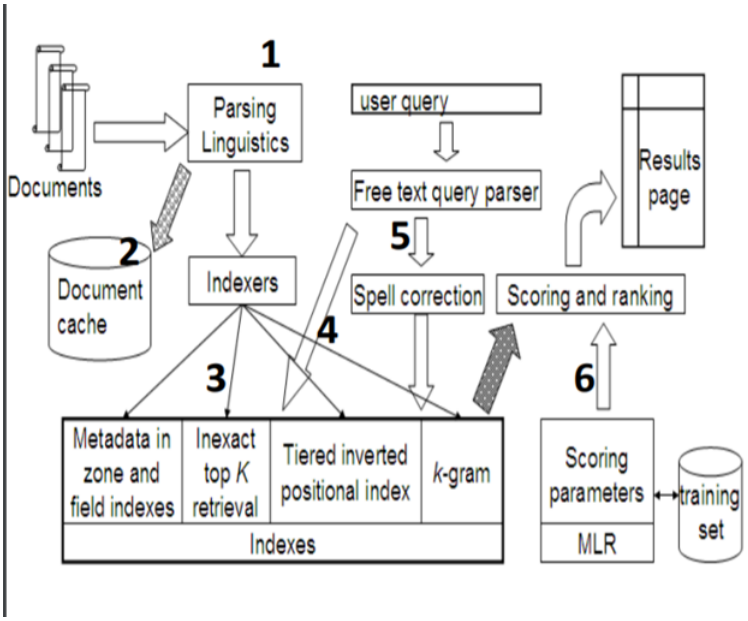

Figure 3: A complete IR System