The Index and the indexing process

The indexing process

Chapter 5

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2015 ; distribution outside this group of students is NOT allowed.*

1. An example of the indexing process requirements. Consider searching a small collection of HTML documents that describe Computer Science books. Each such document minimally includes the author(s) and title of the books it describes. Let us for the sake of this example we limit our exposure on few (approximately 10) books on say `data structures` and `algorithms`. For this collection the index-terms will be limited to this domain.

1.1 Tokenization and index terms. Through the document processing techniques of the previous section we decide that the index-terms are relevant words such as `algorithms, algorithm, data structures, design, analysis, computer, java, c++`. One might observe that **stemming** was not used and thus `algorithm` and `algorithms` are considered two different index-terms. One might also observe that a **noun-group** was identified and thus `data structures` became an index-term (not index-terms). For easiness we shall refer to each document in the remainder by the author(s) of the book that it describes rather than a generic and nameless `docID`.

1.2 Forward index: tuples utilizing wordIDs and docIDs. We mentioned earlier when we discussed the Google Search Engine Architecture (Subject 3) that during the building of the forward index a collection of tuples are generated that describe the information of each index-term in a document of the collection. Instead of a word, a `wordID` is used that is (usually) of fixed length (bytes), and the document is described in term of a (usually) fixed length `docID`; other information about the word might also be recorded. Auxiliary (data) structures allow for efficient conversion of `docID`s into `URL`, `Checksums` and pointers to the actual documents in the local repository of the web. Same applies to `wordID`s. Given that a document is scavenged from the beginning to the end, all tuples related to a given `docID` are next to each other ordered by word or character offset that describes the position of the word in the document with the given `docID`.

1.3 (Inverted) index: inversion (aka sorting) of the forward index. During the inversion phase, the tuples are getting sorted by index-term (i.e. wordID) and then by `docID` and offset, as needed, using a "stable" sorting algorithm that won't destroy in the second round of sorting for example (sorting by docID) the order implied by the first round sorting (sorting by wordID).

1.3.1 Hits list, Doclist. A simplified Hits array containing only docIDs is known as a `doclist`. The second phase sorting by `docID` (after the by `wordID` sorting) guarantees that this list is sorted by `docID`.

1.3.2 Building the index by grouping by wordID and building the Hits array of a wordID. Then we can easily (serially) group all tuples with the same `wordID`. This involves grouping of all tuples with the same wordID and docID into what is known as a `Hits list` for the `wordID` and the given `docID`. A Hits list (or array) also known as "hitlist" is an ordered sequence of hits ordered (usually) by offset. The same tuples can be grouped by `docID` if we eliminate duplicate tuples that contain multiple entries with the same `wordID, docID` and keeping only one instance (of each). The end result is that for a given `wordID` we have a list of all the unique documents (or occurrences if multiple entries for the same document are allowed) that contain the wordID (once or more).

1.4. Collecting Information. Thus the following information can be collected (and our terminology will be consistent with that used lated in Subject 8) but will deviate from the one used in Subject 6.

1.4.1 Corpus D, document d_j and corpus size n (number of documents). Let the number of documents of the corpus D be n, where d_j is the j-th document for $j = 1, \ldots, n$.

1.4.2 Index-terms: the i-th index-term is k_i, t is the number of index-terms, and n_i is the document frequency of k_i. The collection contains a number of index-terms and let this number of unique index-terms be t. The i-th index-terms is k_i and appears in n_i distinct documents; this is known as the **document frequency** of k_i. Thus the length of the `doclist` for the i-th index-term is also n_i (assuming that the doclists contains no duplicate `docID`s). (When we refer to a given Hits list or doclist, we might use **Ndocs** instead of n_i.)

1.4.3 Frequency of i-th index-term: The term frequency or tf-factor of k_i in d_j. The number of times the i-th index term k_i appears in an arbitrary document d_j of the collection is f_{ij}; this is known as the **term frequency** or **tf-**factor. (Sometimes we call it `Counts` or even `Nhits`, but see below other uses of the latter.)

1.4.4 Other information: $\sum_j f_{ij}$. The term $\sum_j f_{ij}$ give the total number of times the i-th index-term appears in D. Note that whereas n_i counts documents $\sum_j f_{ij}$ counts occurrences (the same document might contain multiple number of times the index term). This is the `Nhits` of the occurrence list for a given wordID.

1.4.5 Other information: $\sum_i f_{ij}$. The term $\sum_i f_{ij}$ gives the number of index-terms in document d_j. In other words this is the length of the Hits array, as duplicate occurrences are counted. (When we refer to a given Hits list of the forward index, we might use **Nhits** instead of $\sum_i f_{ij}$.)

1.4.6 Other information: $\sum_{i,j} f_{ij}$. Thus $\sum_{i,j} f_{ij}$ gives the total number of all index-terms (non-unique) in all the documents of the collection. (Sometimes we call it N.)

1.5. Preparing to answer queries. Queries that require the accessing of the index (aka doclist or Hits list for a given `wordID`) can vary depending on the requirements of the query: A **YES** or **NO** query response needs only access the doclist to determine whether a document satisfies or not a query. The we are talking about a **Boolean retrieval model** for query processing (more will have to wait for Chapter 8) as the response is binary (aka Boolean). If a ranking of the response in order of relevance is needed depending on say the number of times a keyword appears in a document or in the Corpus (rare or frequent word), then a more complex **Vector retrieval model** is to be used that will utilize n_i, f_{ij} and other collected information.

Indicator variable. An `indicator` variable for a condition is 1 if a condition is true and 0 otherwise.

1.6 Boolean Retrieval Model and the Index: (Boolean) Incidence vectors. For the simple query case where a YES or NO suffices for an anwer, one way to organize the information collected through document processing is to create for each index-term an **incidence vector**, i.e. a simplified form of a HITs list or doclist. An `incidence vector` for a term is a binary vector of indicator hits for a sequence of documents. In other words for an index-term i its incidence vector C_i is a vector that is n elements long (where n is the number of documents of D) and entry $C_i(j)$ is 1 if the j-th document of the collection contains term i and 0, if the j-th document of the collection does not contain i.

1.6.1 Term n_i appearing in d_j indicated by indicator variable $C_i(j)$. Thus $C_i(j)$ is the indicator variable for term i appearing in document j.

1.6.2 Incidence matrix. We can group the incidence vectors of all index-terms in the form of a matrix i.e. a collection of rows and columns of values (0s and 1s in our case). The rows in the matrix are the books i.e. `documents` of the collection, and the columns are terms and their incidence vectors. If we denote a matrix by C and reference it as $C(i, j)$ where index i references a row and index j references a column, then $C(i, j)$ is 1 if the i-th term appears in the j-th document and 0 otherwise. Thus the i-th row of C is the incidence row vector for the i-th term.

1.6.3 Example. An incidence matrix is depicted in Figure 1. The rows correspond to "incidence (row) vectors" (or their transposes) and describe the index-term appearing in the documents and the columns the documents. The term `algorithms` has an incidence vector of `0101111111` since all but the first and third books contain it. This incidence vector is the first row of matrix C since `algorithm` is the first index-term of the collection (not that they are sorted, but they are listed so).
If it sounds strange that an 'incidence vector' refers to a row of a matrix, take its transpose!

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
Algorithms	0	1	0	1	1	1	1	1	1	1
Algorithm	0	0	1	0	0	0	0	0	0	0
Data Structures	0	0	0	0	0	1	1	1	1	0
Java	0	0	0	0	0	0	0	1	0	0
C++	0	0	0	0	0	0	0	0	1	0
Computer	1	0	0	1	1	0	0	0	0	0
Design	0	0	1	1	0	0	0	0	0	1
Analysis	0	0	0	1	0	0	0	0	0	1

```
Corpus in detail
d1 = Knuth          'The Art of Computer Programming'
d2 = Cormen         'Introduction to Algorithms'
d3=  Tamassia et al 'Algorithm Design'
d4=  Aho et al      'Design and Analysis of Comp.Alg.'
d5=  Baase et al    'Computer Algorithms'
d6=  Lewis et al    'Data Structures and their Alg.'
d7=  Aho et al 2    'Data Structures and Algorithms'
d8=  Goodrich et al'Data Str. and Alg. in Java'
d9=  Goodrich*et al'Data Str. and Alg. in C++ '
d10= Levitrin       'Intro to Design and Analysis of A'
```

Figure 1: Incidence matrix example

1.6.4 Operations on incidence "vectors": Bitwise Boolean/Logical operators. Given an incidence matrix C for a corpus (document collection) query processing involves manipulating the rows of C i.e. its incidence vectors using traditional boolean or logical operators such as `AND, OR, NOT`. The **bitwise AND** or **bitwise conjunction** of two n-element row vectors x, y of 0s and 1s is an n-element row vector z of 0s and 1s denoted by $z = x\&y$ such that the i-th element of z is 1 if and only if the i-th elements of x and y are both 1, and 0 otherwise. (For a **bitwise OR** or **bitwise conjunction** denoted by $z = x|y$, the i-th element of z is 0 if the i-th elements of x and y are both 0, and 1 otherwise.)

1.6.5 Example. The conjunction

`algorithms` AND `data structures`

is the `bitwise AND` operation of the incidence row vectors of the two terms `algorithms` and `data structures`.

algorithms. The incidence vector of `algorithms` is `x = 0101111111`.

data-structures. The incidence vector of `data structures` is `y = 0000011110`.

Then $z = x\&y =$ `0000011110`.

This means that the sixth, seventh, eighth, ninth books contain both terms in their titles!

1.7 Google 1998. Consider the case of Google in 1998. In 1998 Google used to index as few as 25,000,000 books and thus $n = 25000000$, and the dictionary that contained the index-terms contained close to $t = 14,000,000$ words.

- **Corpus size.** $150GB$.
- **Number of documents** n**.** $n = 25,000,0000$.
- **Number of index-terms** t**.** $t = 14,000,000$.
- **Average document size.** $150GB/25000000 = 6KB$.
- **Average word length (with whitespace).** 6.
- **Average number of words per document.** $6KB/8 \approx 1000$.

1.8 An incidence matrix is an $t \times n$ **matrix.** A document is a column of the incidence matrix (see previous page). Thus an incidence matrix contains n columns. Given that the average document in Google@1998 contained 1000 words, each one of the 25,000,000 columns contained as few as 1000 ones with the remaining values being zeroes! The number of rows of the matrix is $t = 14,000,000$.

- **1.8.1 Incidence matrix** C **is an** $n \times t$ matrix where $t \times n = 14,000,000 \times 25,000,000$ matrix.
- **1.8.2 Number of non-zero (aka one) values.** On the average, a column of the matrix has 1000 non-zero values (ones) and $13,999,000$ zero values. Less than 1% of 1% of the values of a column or the matrix as a whole are ones!

1.8.3 Space for the incidence matrix C **is** $t \times n$**.** The total amount of space that one would have used to store the incidence matrix of Google@1998 (assuming one bit for a 0 or an 1) is $25 \cdot 10^6 \times 14 \cdot 10^6/8 = 4.3 \cdot 10^{13}$ bytes, or no more than $43TiB$ (i.e. **43 Terabytes**).

1.8.4 Google@1998 corpus size vs index size vs incidence vector size. Yet, Google@1998 indexed **150GiB** of documents, in around **60GiB**, an amount of space that is roughly one-thousandth of the size of the incidence matrix computed as **43TiB**.

This can only mean one thing:

An incidence matrix is inefficient.

1.8.5 Incidence matrix is space inefficient. The Google example shows that an incidence matrix is a very poor solution for storing index information. This is because it stores not only the 1s (useful information) but also the 0s. The zeroes tell us nothing: they indicate that an index-term does not appear in a document. It makes thus more sense to store only the 1s and forget the 0s. One example from algorithms and data structures that is used is in graph algorithms. If the graph has lots of zero relative to its ones an `adjacency matrix` which is an `incidence matrix` is used. But if the number of ones are 'sparsely' available relative to the zeroes, an `adjacency list` is used instead. So for indexing it is time to use an **incidence list** instead of an **incidence matrix**.

1.8.6 Incidence list (aka forward index). For every document we store the index-terms (i.e. the 1s of the incidence matrix) contained in it and ignore the 0s of the (columns of the) incidence matrix. This is an **orthogonal/inverse** solution!

1.8.7 (Inverted index): An orthogonal solution to the incidence list. Inversely (or orhtogonally) to an incidence list, for every index term, we store all the documents that contain that index term in a data structure aptly named **inverted index**!

1.8.8 Orthogonal solution, also known as the (inverted) index in Google 1998. An inverted index would thus store a total of $25,000,000 \times 1000$ document references, as the average document contained 1000 words for the Google 1998 corpus. A document reference (i.e. `docID`) needs approximately 4 bytes for a total of $100GB$. With some compression this space can easily be halved into 50GB, very close to Google's index size at that time!

```
                        d1  d2 d3 ....
1. Incidence Matrix   t1 0   1  1
                      t2 1   1  0....

2. Incidence List     d1  : t2      or  <d1,t2>
                      d2  : t1 t2   or  <d2,t1> <d2,t2>
                      d3  : t1      or  <d3,t1>
```

```
3. (Inverted) Index or (Inverted) Lists

   t1  :  d2 d3
   t2  :  d1 d2
```


2. Inverted index (Definition). An **inverted index** is a word-oriented mechanism for indexing a text collection and thus storing index-term or document-incidence information in order to speedup the search task. An inverted index consists of two components. (The second component has multiple names!)

- A **Vocabulary**, the set of all index-terms in the dictionary, and
- **Occurrence lists** or **inverted lists** (most detailed form) of **Hits list** or **Counts** or **doclist** (less detailed form) that contain the information necessary for each index-term of the vocabulary to be identified.

2.1 Book indexes. An **index of a book**, contains a list of important words (keywords), and for each such word a listing of the pages in which this word appears. If a word appears multiple times in a page, that page is listed once. If consecutive pages include the word, a compressed reference is used that includes the first page and the last page of the sequence separated with a dash −. The list of words in the index is a small subset of the words of the book, and potentially a smaller fraction of the dictionary of all English words (i.e. those of the Oxford English Dictionary, Second Edition of Figure 2).

2.2 Index of a corpus. The index might contain for each word, the document in which it appears (**doclist**), and/or positional information within that document. The level of granularity might depend on the use of the index.

2.3 Concordance. In several books or collections of books, a precursor to an inverted index is what are known as **concordances**, a primitive form of an occurrence list for a printed book.

General Info	: 1989 (Pub. date), 20 volumes, 21,730 pages, 62.6Kgr/137.82lbs.		
Amount of ink used to print complete run	: 2,830 kilos or 6,243 lbs.		
Number of words in entire text	: 59 Million	Number of printed characters	: 350 million
Number of megabytes required for text	: 540	Number of entries	: 291,500
Number of main entries for obsolete words	: 47,100	Number of main entries	: 231,100
Number of main entries for spurious words	: 240	Number of etymologies	: 219,800
Number of cross-references within entries	: 580,600	Number of cross-reference entries	: 60,400
Longest entry in Dictionary	: verb set ; 430 senses; ~60,000 words ; 326,000 characters		

Figure 2: OED: Oxford English Dictionary, Second Edition (1989)

3. Dictionary vs Vocabulary vs Lexicon. All three mean the same thing, a collection of unique words, but in the context of this discussion we are going to use them in a specific way.

3.1 Dictionary. Based on the discussion of Subject 6, a **dictionary** contains all the unique words in the corpus as identified by the tokenizer. For example a dictionary might contain `auto, car, vehicle, automobile`.

3.2 Vocabulary. The **vocabulary** is maintained by the index and contains all the unique index-terms that are indexed and thus included in the inverted index.

In the dictionary all four words that identify a car might all point to the same index-term of the vocabulary, say `car`. For example, for a query that includes the word `auto`, a lookup to the dictionary can resolve the correct spelling of this word (as opposed to say, `awto`) and also identify the correct index-term associated with the word i.e. `car`. Then index-term `car` will be used for `awto` or `autom` in query processing. It is often that dictionary and vocabulary are one and the same if only because of the overhead of maintaining two different data structures for them.

3.3 Lexicon. The **lexicon** is an efficient implementation of the dictionary/vocabulary by using a hash table more often than not. (See also Subject 4 for such an implementation.) It is possible that in this context not all of the words of a vocabulary (or dictionary) are stored into the hash table because of memory requirements (not enough available memory to store all the words in main memory). Thus in this case some rare words (of the vocabulary or dictionary) might be stored outside of the lexicon (i.e. outside of main memory).

3.4 Information to maintain in a Vocabulary (or Lexicon).

3.4.1. WordID. One entry per index-term. Rarely the word itself is stored in the vocabulary. A hash in the form of a word identifier called `wordID` is more useful to maintain and compact to store. A pointer to an area in memory where the string of the word itself is stored might also be available. (this is the Lexicon i.e. an implementation of the Vocabulary.)

3.4.2. n_i or Ndocs. The number of distinct documents containing the i-th index term k_i i.e. `wordID` can be recorded in this field. We call it n_i or generically `Ndocs`.

3.4.3. Nhits or $\sum_j f_{ij}$. The number of distinct occurrences of `wordID` i.e. k_i in all `Ndocs` documents or separately per document (f_{ij}) can be recorded in this field. The term f_{ij} indicates for the i-th index term k_i the number of times it appears in document d_j. Thus the `Nhits` entry stores $\sum_j f_{ij}$.

4. Inverted list or Occurence List. The two different names names of the same list describe the major component of an (inverted) index other than the vocabulary. For a given index-term it holds the relevant information for that index-term.

Each entry in the list is called a **posting** and the portion of a posting that refers to a specific document (eg. `docID`) or location (eg. offset) is called a **pointer**.

The posting minus its pointer might record additional information about the index-term. In the case of Google (1997) it records information such as relative font-size and capitalization.

4.1 Doclist. refers to a simplified form of an occurrence list (inverted list) in which only document references appear. That is, the posting only includes pointer information and the pointer information only includes a document reference. A doclist is usually sorted by `docID`.

(For example, Fig. 5.3 of the textbook on page 132 shows multiple instances of a doclist.)

4.2. Hits list or "Hitlist". It refers to the occurrence list of particular index-term in a particular document. Thus the Hits list of an index-term over all documents of the collection form its occurrence list.

For example, an index-term that is contained in 10 documents will consist of 10 Hits lists. Each one is going to be (or can be) of variable length depending on the times the index-term appears in the associated document. Note that in Google 1998 an occurrence list is a collection of doclists with associated Hits. Thus variations of the given explanations might be (and are) prevalent.

Auxiliary information can be maintained in the inverted/occurrence lists. This includes.

4.2.1 Counts or Nhits. For each document listed in the doc-list or in general the inverted list of a given index-term, a **Counts** field records the number of times the index-term occurs in that document. In most cases `Counts` provides additional information per posting of the `doclist`. **A single posting per document is available with the posting containing pointer and count information.**

4.2.2 Positions. This expands the information available through **Counts**. The number of postings for a particular document depends on the counts information and is as much as the counts field indicates. Each posting contains a pointer that includes document information plus an offset within the document. Such offset information can be in the form of a character offset (rarely used), a word offset, or a block offset if a document is split into blocks of coarser granularity (a block can be a number of consecutive characters or bytes, or a paragraph, or a section, or a chapter). The offset is from the beginning of the document. Note that a maximum size of an offset can be established. For example in Google (1997) a word offset can grow up to 4094 with 4095 used for words with offset greater than 4094.

4.2.3. Fields and Extents. Emails contain sender information and a subject line. Information located in those fields can be associated with these **fields** that are sections of the document that carry some specific semantic meaning. HTML documents contains title information. An **extent** is a consecutive area (i.e. consecutive words) of a document storing some specific information possibly about a field. For example an HTML title is stored between `<TITLE> and </TITLE>`.

4.2.4. Context Information. It records information about the structure around the term. This can include HTML tag information, font-size, capitalization that can be used in scoring processing (i.e. query-related document ranking).

4.2.5. Ordering of an inverted list. The usual ordering of a list is by document reference (i.e. `docID`). It is possible however, that a score can be assigned to a particular document. Then it might make more sense to order documents based on this score.

4.3 Example with `Doclist`, `Counts` and `Positions`. Figure 3 indexes two documents. A vocabulary is constructed of all the words of the two documents after capitalization is removed, and simple stemming is applied (plural elimination). The inverted list representation is shown under B1, B2, B3 that represents the inverted list part of an index using a Doclist, Count, and Positions respectively. Offsets given for Positions are word offsets. The vocabulary is sorted by word in this naive example and no wordID are used. The index (vocabulary and the corresponding inverted list) can be stored in the form of the two tables shown in Figure 4 and Figure 5 respectively.

```
Document 1
Sentences have words. Words have characters. Paragraphs have sentences.

Document 2
Sections have paragraphs. Paragraphs have sentences. Sentences have words.
                        [B1]             [B2]                    [B3]
Vocabulary              Inverted List    Inverted List           Inverted List
----------              [Doclist]        [Counts]                [Positions]
character               1                (1,1)                   (1,6)
have                    1,2              (1,3)(2,3)              (1,2)(1,5)(1,8)(2,2)(2,5)(2,8)
paragraph               1,2              (1,1)(2,2)              (1,7)(2,3)(2,4)
section                 2                (2,1)                   (2,1)
sentence                1,2              (1,2)(2,2)              (1,1)(1,9)(2,6)(2,7)
word                    1,2              (1,2)(2,1)              (1,3)(1,4)(2,9)
```

Figure 3: A simple example of an index: Two documents

4.3.1 Example continues on following page. Contrary to what we mentioned earlier, in Figure 4 on the following page the Vocabulary stores for illustration purposed words rather than `wordID`s. For each word (i.e. `wordID`) we store the number of documents containing the word: this is field `Ndocs`. In addition, pointer `LocP` points to the (first) table entry storing the inverted lists for the `wordID`. With reference to Figure 4 and Figure 5, we have that for `paragraph` the corresponding `Ndocs` is 2 and since `LocP` is 3 we conclude that lines `3,4` of the inverted lists table store the inverted lists for `paragraph`. In other words, the entries for a given `wordID` in the inverted lists table that are of interest are those starting at location `LocP` and ending just before `LocP+Ndocs` i.e. at location `LocP+Ndocs-1`.

WordID	Ndocs	LocP
character	1	0
have	2	1
paragraph	2	3
section	1	5
sentence	2	6
word	2	8

Figure 4: Inverted index: Vocabulary

Index	DocID	Nhits	Hits
0	1	1	6
1	1	3	2,5,8
2	2	3	2,5,8
3	1	1	7
4	2	2	3,4
5	2	1	1
6	1	2	1,9
7	2	2	6,7
8	1	2	3,4
9	2	1	9

Figure 5: Inverted index: Table of Inverted lists

4.3.1 Example continued. Indeed, line `4` has a `docID` field `2` which involves document 2 and the `Nhits` field is also 2 that indicates that `paragraph` appears twice in that document. The value of `Nhits` allows us to access the variable length array `Hits` that in this case has two elements, and records word offsets. Indeed, `paragraph` appears as the 3rd and 4th word in document 2.

In the inverted list table we store multiple entries for each index-term and one entry per document and index-term. The `Hits` table is of variable length and stores here offset information only. The compression methods discussed earlier directly apply to this example of `Hits`. An alternative would have been to store a single entry/array for each index-term similarly to the depiction in Figure 3 so that the number of rows in the inverted list table is the same as those of the vocabulary. This would require either a more complex representation of a Hits list or storing a pair of `(docID,offset)` in a posting. In that latter case a more involved compression is required since consecutive pairs might have the same first element that might grow slowly or slower relative to the second element of the pair that stores offset information.

4.3.2 Important Note. It is important that the `Hits` lines (i.e. hitlists) are sorted by `docID` i.e. that line `Index=3` that deals with `docID=1` in the Table of inverted lists comes before line `Index=4` that deals with `docID=2` for index term `wordID = paragraph`. The importance of this will become evident when we discuss query processing in pages 19-20.

4.4 Example. We give a more comprehensive example right now that shows all the steps performed from tokenization down to index creation.

Example 1 *Let the three documents of this example be the ones indicated in Figure 6. The whole indexing process consists of the following rounds.*

- **1. Tokenization and 2. Linguistic Analysis.** *This phase first generates for each document a stream of tokens with Positions information in the form of* `(wordID,docID,offset)` *tuples. The tuples are then sorted based on* `wordID` *and secondarily (or by preserving previous order) based on* `docID, offset`*. This is depicted in Figure 7.*
- **3. Forming and Building the index.** *The sorted stream is processed to determine (i) the number and list of unique words, (ii)* `Ndocs`*, the number of documents containing a given* `wordID`*, (iii)* `Nhits` *the number of occurrences of a* `wordID` *in each one of the* `Ndocs` *documents. Parsing sequentially the left part of Figure 8 can collect this information. At the same time* `Nhits, Hits` *information is incorporated and depicted in the right part of Figure 8. This is the first step of building the index i.e. forming the Vocabulary and the inverted lists. Immediately afterwards, the vocabulary is built, and inverted list table entries are coalesced. By going through the table of the right part of Figure 8 we can decide the number of lines in the table of inverted lists and build the* `LocP` *pointers to that table. This is depicted in Figure 9.*

Document 1.	Document 2.	Document 3.
data structures and algorithms in java	data structures and their algorithms	algorithms in java

Figure 6: Example: Three short documents to index

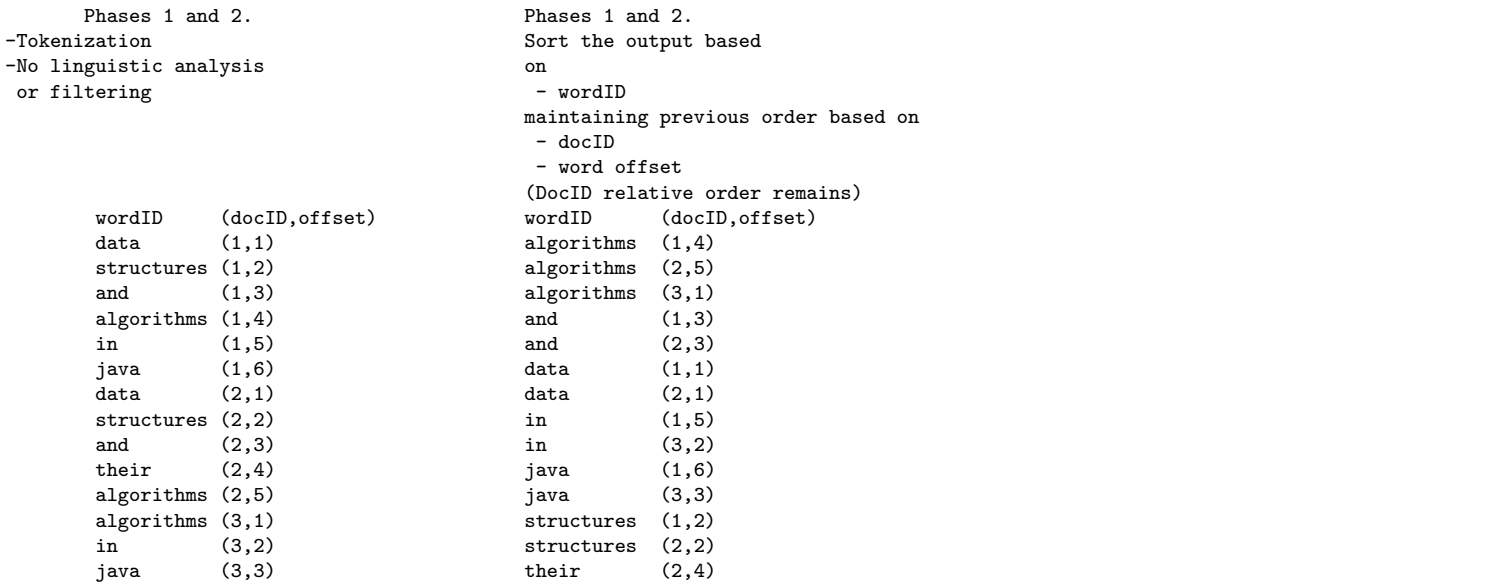


Figure 7: Phases 1 and 2: Tokenization and Linguistic Analysis

```
      Phase 3. Forming the index.

wordID       docID |  wordID      docID  Nhits Hits
algorithms  (1,4)  |  algorithms  1      1     4
algorithms  (2,5)  |  algorithms  2      1     5
algorithms  (3,1)  |  algorithms  3      1     1
and         (1,3)  |
and         (2,3)  |           Similarly.
data        (1,1)  |           (details
data        (2,1)  |            omitted)
in          (1,5)  |
in          (3,2)  |
java        (1,6)  |
java        (3,3)  |
structures  (1,2)  |
structures  (2,2)  |  structures  2      1     2
their       (2,4)  |  their       2      1     4
```

Figure 8: Phase 3: Forming the index

```
Phase 3. Building the index: Vocabulary and Inverted Lists.

Vocabulary              Inverted List Table

wordID     Ndocs LocP    docID Nhits Hits
algorithms  3     0    0  1    1     4
and         2     3    1  2    1     5
data        2     5    2  3    1     1
in          2     7    3  1    1     3
java        2     9    4  2    1     3
structures  2    11    5  1    1     1
their       1    13    6  2    1     1
                       7  1    1     5
                       8  3    1     2
                       9  1    1     6
                      10  3    1     3
                      11  1    1     2
                      12  2    1     2
                      13  2    1     4
```

Figure 9: Phase 3: Building the index

4.5 Simple Query Processing: Conjunctions and Disjunctions. With reference to the index depicted Figure 9 we show how simple queries can be satisfied.

4.5.1 Conjunction: Generating a single Hits list or doclist for each index-term. Let us have a simple conjunction of three terms `algorithms AND data AND structures`. In this example we use the standard convention that a capitalized `AND` is the conjunction operator (logical AND) rather than index-term `and`. The following steps are performed to generate the unified Hits lists for the three index-terms.

For each one of the index-terms of the conjunction, the following operations are repeated:
(a) the `wordID` is found for the index-term and its position located in the vocabulary,
(b) the number of documents `Ndocs` that contain `wordID` is then located and the corresponding lines of the table of inverted lists are then accessed using `LocP`, i.e. lines `LocP, ... , LocP+Ndocs-1` will be retrieved,
(c) for each line of the table of inverted lists retrieved, the `Nhits` is retrieved to determine the structure of `Hits`; `Hits` is then decompressed,
(d) the lists for each line are concatenated (unified) so that each term gives rise to one such list,
(e) further processing is performed to satisfy the conjunction (this example) or disjunction as applicable.

```
algorithms    3    0   --> 0  1  1  4   -->  (1,4)   -->  (1,4)(2,5)(3,1)
                           1  2  1  5   -->  (2,5)
                           2  3  1  1   -->  (3,1)
data          2    5   --> 5  1  1  1   -->  (1,1)   -->  (1,1)(2,1)
                           6  2  1  1   -->  (2,1)
structures    2   11   --> 11 1  1  2   -->  (1,2)  --->  (1,2)(2,2)
                           12 2  1  2   -->  (2,2)

algorithms  (1,4) (2,5) (3,1)
data        (1,1) (2,1)
structures  (1,2) (2,2)
```

Figure 10: Retrieving hits for the query index-terms

4.5.2 Conjunction: Processing the Hits list or doclists. Having retrieved the Hits lists for each index-terms of the conjunction and unifying them into one list per index-term we are then ready to satisfy the query.

The following algorithmic description deals with a two index-term conjunction. It can be generalized for this example as show in the Figure below. (Of course there is a tiny problem that has not discussed: when we evaluate $x \wedge y \wedge z$ do we do it $(x \wedge y) \wedge z$ or $x \wedge (y \wedge z)$?) The "scoring mechanism" that incorporates `Nhits` is for the moment excluded: all documents are ranked the same no matter how many times a given index-term is in them.

- **Naive Scoring Mechanism**
  - **Conjunction:** `x AND y` **is doclist(x)** $\cap$ **doclist(y)**. For a conjunction we need to find all documents (i.e `docIDs`) that contain both index-terms x and y. This is thus equivalent to finding the intersection of the Hits lists using only `docID` information and ignoring offsets or ther info.
  - **Disjunction:** `x OR y` **is doclist(x)** $\cup$ **doclist(y)**. For a disjunction we need to find all documents that contain one or the other (or both) of the index-terms. This is thus equivalent to finding the union of the Hits lists using only `docID`.
  - **Negation:** `NOT x` **or** `-x` is $\overline{\textbf{doclist}(x)} = \textbf{doclist}(\mathbf{x})^c$. We need to find all the documents that do not contain x. For this we subtract from the documents of the corpus those containing x. This is the set theoretic complement operation.

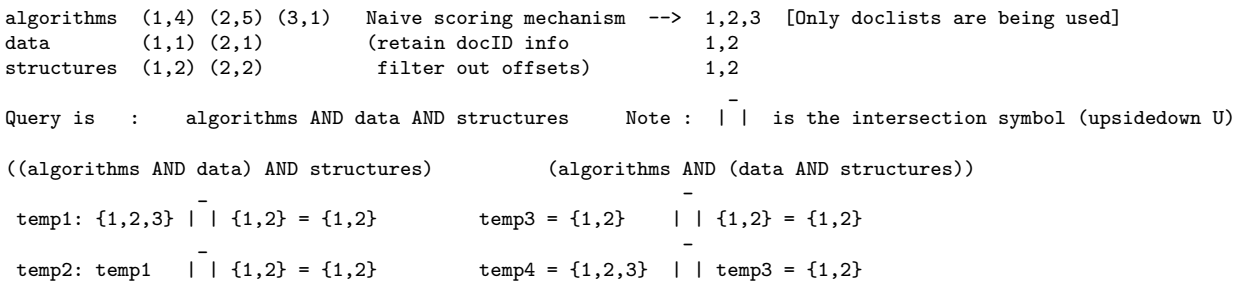


Figure 11: Conjunction of three index-terms: Associativity?

4.6 Conjunction is Intersection and Disjunction is Union: Intersection and Disjunction through merging!. In order to find the intersection (or union) of two sets or two sequences quickly, the two sequences must be in sorted order (increasing or non-decreasing order by `docID`). Then the steps outlined in page 19 would generate a sequence also ordered by `docID`. This is indeed the case in Figure 11.

4.6.1. Naive Method: Merge First. Say we are given two sorted (doc)lists $A = \{1, 2, 3\}$ and $B = \{1, 2\}$ of `docID`s for terms A and B. For illustration purposes, we write the elements of the two lists with subscripts to denote the set/term to which they belong. The merging of A and B would generate as output $\{1_A, 1_B, 2_A, 2_B, 3_A\}$.
$A = \{1_A, 2_A, 3_A\}$ and
$B = \{1_B, 2_B\}$.

4.6.2.a Intersection of A and B: Clean-up (pruning) of merged list The clean-up is performed through a linear scan of the merged-list. With a linear scan we can (i) eliminate non-duplicates, (ii) keep one copy of duplicates, and (iii) remove subscripts so that we finally get $\{1, 2\}$. This is the intersection of A and B.

4.6.2.b Union of A and B: Clean-up (pruning) of merged list. For the union, the clean-up is different. In a linear scan as before we only keep one copy of duplicates without eliminating non-duplicates and then remove subscripts. Thus we get $\{1, 2, 3\}$. This is the union of A and B.

4.6.3 Running time of merging and conjunction. For an a element sorted list A and b element sorted list B, the running time of merging is $a + b$. The pruning costs $a + b$; the **total cost** of the conjunction computation is $2a + 2b$.

4.7 Queries involving `NOT`. What if the query involves a `NOT`? What is the result of `NOT data`? It is all the documents in the collection that do not include `data` that is for our example this is $\{3\}$. Thus `algorithms AND NOT data` requires the intersection of the Hits list of `algorithms` with the Hits list not of `data` but the complement of `data` i.e. all the documents of the collection minus those containing `data`. Thus the set theoretic `complement` operation allows for the implementation of operator `NOT`. The cost is n (corpus size).

4.8 Conjunction and Disjunction by merging: Optimizing the previous method. A drawback of the method is that it generates the merged list before it is being cleaned-up. In Figure 12 on page 22 we show how to get the intersection/union without creating a merged list before the clean-up thus reducing the total time to $a + b - 1$ instead for conjunction and $a + b$ for a disjunction.

4.9. Is $(x \wedge y) \wedge z$ more efficient than $x \wedge (y \wedge z)$? An operator $\oplus$ is **associative** if and only for all a, b, c we have $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. Operators such as $+, *$ and `AND, OR, MIN, MAX` are all associative. If the cost of applying an operator is unit, then whether $a \oplus (b \oplus c)$ is performed or $(a \oplus b) \oplus c$ does not matter: 2 operations will be performed at a cost of two units. However in order to perform $x \wedge y$ i.e. `x AND y` we need to form the intersection of $\mathbf{doclist(x)} \cap \mathbf{doclist(y)}$. The operations Intersection i.e. $\cap$ and Union i.e. $\cup$ are more costly: the cost depends on the length of the two sets. Thus $(A \cap B) \cap C$ can be more or less costly or has the same cost as $A \cap (B \cap C)$). The rule of thumb is to follow the rule that says.

Rule 1. Intersect the shorter lists first! And if merging, merge the shorter lists first as well!

4.10 Revisiting Merge-First and Prune-later (4.6.1-4.6.3 method): MergeIntersect. When we consider the intersection of two sets A and B we know about intersection: the intersection is a subset of the smaller of the two sets and the cardinality of the intersection is no larger than the cardinality of the smaller of the two sets. (Thus the intersection of one set with 1000 elements and one with 2 elements can be no larger than 2). Moreover, while we are merging the two lists (i.e. in step 4.6.1) we also perform the pruning, If element x of A is equal to element y of B then $x = y$ should appear in the intersection of A and B and also in the union of A and B (i.e. we copy them in the output of the intersection or the union); if however $x < y$ or $x > y$ for the union we move the smaller into the output and move to the next element (of the smaller's list), while for the intersection we do not generate an output and move to the next element (of the smaller's list). The pseudocode is available in Figure 12.

4.10.1 Example 2. Intersection of three sets A and B and C of sizes n, $2n$ and $3n$.

We first find the intersection D of A and B in $n+2n-1$ operations. D can have no more than n elements. The we find the intersection E of D and C in $n+3n-1$ operations. Set E has no more than n elements; total cost is $(3n-1)+(4n-1) = 7n-2$.

If however we decide to find the intersection F of B and C this would cost $2n + 3n - 1 = 5n - 1$; set F can have no more than $2n$ elements. The merging of F and A into E would take $n + 2n - 1 = 3n - 1$ additional operations. The total costs is now $5n - 1 + 3n - 1 = 8n - 2$ higher than before.

This confirms Rule 1 and also shows that MergeIntersect is better than Merge-First Prune-Later.

Rule 2. MergeIntersect is better than Merge-First Prune-Later.

4.10.2 MergeIntersect.

```
 MergeIntersect (L,R)  // Logical x AND y  is MergeIntersect(L=doclist(x),R=doclist(y))
0.   i=1;
1.   s=Ndocs(L);      % length of left  sequence L is s
2.   t=Ndocs(R);      % length of right sequence R is t
3.   S=1;              % Runs through L
4.   T=1;              % Runs through R
5.   while i <=  (s+t) % Build output one element at a time
6.     if  S >  s
7.        return(AND,i)     % Size of AND is i
8.     elseif  T > t
9.           return(AND,i) % Size of AND is i
10.    elseif L(S) <  R(T)
11.          S=S+1;        % Advance L by one
12.    elseif L(S) >  R(T)
13.          T=T+1;        % Advance R by one
14.    else                 % L(S)==R(T)
15.          AND(i)=L(S);S=S+1;T=T+1;i=i+1;
16.    end
17.  end
```

Figure 12: Implementing logical AND through MergeIntersect

In other words while we scan one element $l = L(S)$ from one list and one element $r = R(T)$ from the other list if

- (i) $l < r$ we advance to the next element of the left list,
- (ii) $l > r$ we advance to the next element of the right list,
- (iii) $l == r$ we have determined an intersection element and store it in array `AND`.

Conjunction i.e. Intersection. For two lists A, B the total time/comparison time is $a + b - 1$, and the output is no more than $\min\{a, b\}$.

Disjunction i.e. Union. For two lists A, B the total comparison time is $a + b - 1$, total time is $a + b$ as the output can be at most $a + b$.

4.11 Further Optimizations are possible: Binary Search. If the sizes of the doclists are not comparable to each other but one is much smaller than the other, binary-search of the elements of the smaller of the two lists into the larger one can speed thing even more.

Consider the case of having a conjunction `x AND y` where the doclist X of x has 20 `docID`s and the doclist Y of y has 1000 `docID`s. The two lists X and Y are sorted already by `docID`. A binary search of each element of X into Y takes $\lceil \lg(1000+1) \rceil = 10$ operations per element of X. Since X has only 20 elements, in $20 \cdot 10 = 200$ comparisons (operations) we can find out which of the elements of X are in Y, i.e. establish the intersection of X and Y in that many 200 operations.

Merging however, of X and Y would alternatively take $20+1000-1 = 1019$ operations. The former method (binary-search) is 5 times faster than the latter method (merging)!

4.12 Estimating the size of a conjunction, union or negation. Therefore before we evaluate an expression that contains `OR, AND, NOT` it makes sense to estimate using information available through `Ndocs` the size of the Hits lists of the terms involved, and then compute bounds on the size of the Hits lists of intermediate results and use that information to develop a schedule that will minimize the total overall cost. (Search engines such as Google and Bing are possibly using that information when they announce `About **a number of** results` under the textbok of the engine.)

4.12.1 Conjunction. For an operation `x AND y` the doclist of the result will be at most (aka "about")
`MIN(Ndocs(x),Ndocs(y))`

4.12.2 Disjunction. For an operation `x OR y` the doclist of the result will be at most (aka "about")
`Ndocs(x)+Ndocs(y)`.

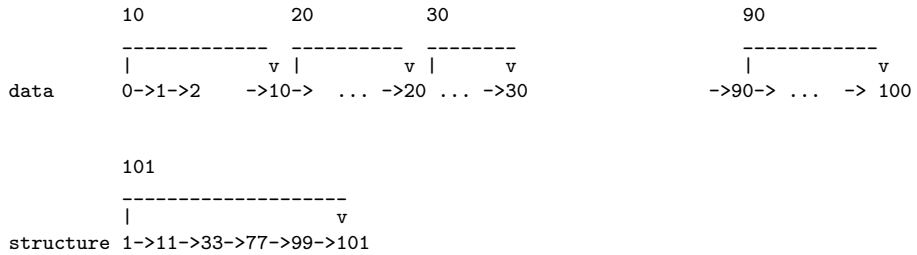
4.12.3 Negation. For operation `NOT x` the doclists of the results will be `n-Ndocs(x)` where n is the number of documents of the corpus.

5. Skip Lists: Need of. With Reference to MergeIntersect consider the case where array L has as its `docID`s all the integers from 1 through 100,000,001 in its `doclist` and in sorted order. And let also R have in its doclist every 10,000-th element or so in that range i.e. $1, 10001, \ldots, 100,000,001$. Then during the course of the algorithm above the operation $S = S + 1$ of Line 11 of Figure 12 will be performed $100,000,000 - 10,000$ times for no apparent reason. If during the first iteration(s) element 1 was included in the intersection, the next time element 2 is considered from L and element $10,001$ is considered from R. Since $2 < 10,001$ an $S = S + 1$ will be performed then and $9,998$ additional times or so. We could skip all those elements of L to speed up the process to read the document with `docID` 10001 in L. This is possible by maintaining the doclists not in the form of ordinary arrays but as **skiplists**.

5.1 Skip List. With skiplists few elements can have extra pointers. For a linked list of length n, each element has a `next` pointer and thus we need n pointers total. Let us call the 0-th level of pointers. In a skip list, at the 1-st level of pointers, the first element points to the third, the third to the fifth and so on. This way half of the elements get a second pointer, i.e. there are $n/2$ additional pointers overall at that level. Likewise in the i-th level, $n/2^i$ elements get an extra pointer. Overal we have doubled the number of pointers overall yet some elements can have up to $\lg n$ pointers. Search tranversal works not only horizontally for a given level but also vertically across the pointers of a given node. If skip by $2, 4, 8$, seems too little, in doclist processing skip by about $\sqrt{Ndocs(.)})$ can be used instead.

5.2 Example. In the example below the doclist of `data` contains about 101 document. The skip factor is 10 thus at level 1, 0 points to 10, 10 to 20, and so on, 90 points to 100. These pointers are in addition to the k points to $k+1$, the next element in the list. The doclist of `structure` is limited and thus the first element points to the last of the list. The intersection works along the 0-th level pointers and it continues as long as the value of data is less than the value of the 1-st level pointer of structure (i.e. 101) and also as long as the value of the 0-th level pointer of structure is less than then value of the 1-st level pointer of data (i.e 10).

Thus, given that ($0 < 101$ and $1 < 10$) 0 is compared to 1 and is smaller, i.e. data's list advances by. Then 1 is equal to and still ($1 < 101$ and $1 < 10$) and thus 1 is output for the intersection. Both lists advance to 2 and 11 respectively. However $11 > 10$ i.e. the 1-st level pointer of data can skip $2, 3, \ldots, 10$ and move to the next element through the 1-st level pointer. Thus data's doclist advances past 10, to 11. The doclist of structure is still at 11. Given that ($11 < 101$ and $11 < 20$) and both values are 11, 11 is output and pointers advance to the next element which is 12 for data and 33 for structure. Given that $33 > 20$ again data can use the 1-st level pointer not just once but twice to skip first $12, \ldots, 20$ and then $21, \ldots, 30$ and traverse the 0-th level pointer of data to 33



6.1 Tokenization. During tokenization, a stream of tuples of the form
`(docID,wordID,offset,otherinfo)`
are generated. These tuples form the forward index: they are ordered in the sense that for a given `docID` the tuples are generated in sequence and in increasing `offset` from document start to end. These tuples will be **inverted** during the indexing process: inversion means sorted by and grouped by `wordID` to generate the **posting** entries of the **occurrence/inverted** lists. In any such posting the portion identified by the pair `docID,offset` will form the **pointer** portion of the posting; the remainder will be other non-pointer information such as `otherinfo` in our case. This stream of tuples is also known as the **forward index**.

6.2 Indexing Process. During indexing this stream is sorted by `wordID` to generate the inverted lists (inverted/occurrence lists for all `wordID`s). This **sorting operation** inverts the forward index to generate an **inverted index**. Secondarily, for every `wordID` we need those tuples also sorted by `docID` so that the Hits lists for each `docID` of the given `wordID` can also be generated. (A `wordID` might appear multiple times within a `docID` and we want these tuples to be close together and in fact sorted of `offset` as well as `docID`.) If the tuples are indeed sorted by `docID`s, then `doclists` for the `wordID` can also be generated and also the accounting information discussed earlier involving `Nhits, Ndocs, LocP` can thus be easily collected. The `Hits list` for a given `wordID,docID` is also desirable to maintain order by `offset`; the original tokenization stream had for a given `docID` all its word ordered by `offset` anyway. Thus it is desirable that **sorting** will not destroy previous orderings. A **sorting algorithm** that behaves like that is said to **sort in-place**.

6.3 Indexing and Vocabulary formation. The length of the stream generated by the tokenizer gives the number of words (rather, index-terms) N in the collection . (This $\sum_i \sum_j f_{ij} = \sum_{i,j} f_{ij}$.) If the stream is sorted by `wordID` (as it should be after 6.2 above), we can easily form a dictionary by eliminating duplicate `wordID` entries to determine the size of the vocabulary in terms of number of distinct words, i.e. compute t. We can also determine `n` the size of the documents in the collection if we eliminate duplicate entries with the same `docID`. (Note that tuples with the same `docID` are consecutively listed in the tokenizer's output even if or even if not a global ordering by `docID` is available.) The only issue might be that new words might need to be assigned a new `wordID`, if during a previous indexing process, that word did not appear in the corpus. This can be done at this time! During indexing this stream can be sorted by `wordID,docID` in that order. Consecutive counting of `(wordID,docID)` pairs in the resulting stream can compute `Nhits` of the collection (the one maintained in the Vocabulary) for each `wordID`. Elimination of `(wordID,docID)` duplicates and counting can compute `Ndocs`, compute `LocP` or also compute `Nhits` information for a given Hits list. Thus the determination of the length of the Hits table for the appearance of `wordID` in a given `docID` can be established.

6.4 Sort and Scan operations. Thus all this activity can be determined from the initial stream of tuples, by performing a sequence of **sorting operations** followed by a sequence of **scanning operations** that might involve duplicate elimination or counting of duplicate entries (where duplicates are `wordID`s, `docID`s or pairs of `(wordID,docID)`s). (Consider Google in 1997 in which the sorter operated on the forward index stored in the barrels to generate the inverted index to be stored in the barrels as well.) Sometimes these `scanning` operations are known as `reductions`.

6.5 Parallel Execution on Massive amounts of Data. All those tasks involve streams of tuples in the billions or trillions. It is rather impossible for all these operations to be performed by a single computer. They have to be performed **in parallel** by a collection (i.e. cluster) of several (tens or hundreds or thousands) machines. Both **sorting** and **scanning / reduction** operations are amenable to efficient parallelization.

6.6 Map operations. Whether to count or not, eliminate or not, and how to count (duplicate or unique occurrence counting) involves a `mapping` of the original stream into a `mapped` set.

Thus a **MAP** operation is essential. The end result of the `mapping` is a **Reduction** operation, or **REDUCE** that will eliminate duplicates or count tuples in a given way as determined by the `MAP` operation. In addition if several machines are being used for this task, the `MAP` operation also works so that the tuples are distributed evenly to the machines of the collection based on the mapped values. Thus tuples with a given `(wordID,docID)` value might be destined for machine X instead of machine Y, and thus the `REDUCE` operation will be applied to them then (after they move to that machine X). Similarly the `REDUCE` operation will also collect partial results per machine X, to generate global results per collection rather than per machine that will be then sent to the `query server or GWS`.

6.7 MAP-REDUCE. This is the fundamental idea behind Google's **MAPREDUCE** system: a convenient way to automate several similar-looking repetitive and time-consuming tasks, and also the full and efficient parallelization of this activity due to the sizable amount of information involved. (And by parallelization we mean the splitting of data evenly among the machines of the cluster, the transmission of data to the machines based on this splitting, the schedule and synchronization of the computations involved, the transmission and collection of partial/local results, and finally the generation of global results with their dispatch to one machine, if they are small in size, or more than one machines otherwise.)

7. Index Size estimation. Using information from Google@1998, roughly the size (in bytes) of the index is one quarter to one fifth of the corpus size.

If we try to work around this let

- **7.1 Corpus size (docs):** n**.**
- **7.2 Text size per document:** s**.**
- **7.3 Corpus volume (B):** ns**.** (Assuming ASCII endocing i.e. one byte per character.)
- **7.4 Word size (words):** $N = sn/6$**.** A rough estimate of word length is 5, and thus accounting for white-space this estimate is derived.
- **7.5 Distinct words :** $t = 40\sqrt{sn}$**.** Heaps' Law with $k = 100$ and $\beta = 1/2$, gives $t = 100\sqrt{N} \approx 40\sqrt{sn}$.
- **7.6** F **frequency of most frequent word:** $F \approx N/\ln t = sn/120$**.** This is roughly one percent of the corpus volume! Zipf's Law gives that $N = F + F/2 + \ldots + F/t = F \ln t$. Thus $F = N/\ln t$. Given that $N = sn/6$ and $t = 40\sqrt{sn}$, we get that $F \approx sn/(3(\ln s + \ln n + 7.5)) \approx sn/120$. For the latter we assume that $10^9 \leq n \leq 10^{11}$, and $10^4 \leq s \leq 10^5$, i.e. $\ln n \approx 22.5$ and $\lg s \approx 10$.
- **7.7 Occurrence list length (B) of most frequent word:** 5% **of corpus size!** Since by (7.6) $F = sn/120$, assuming 2B for a posting and 4B for a docID we need roughly 5% of corpus volume for maintaining the most frequent word without any compression. With compression this can go down significantly to around 1.6% (eg. using Elias-γ). This is because $2sn/120 = sn/60$ is used for postings, and the n docIDs use roughly insignificant amount of space.
- **7.8 Top-100 words' index:** $\approx 10\%$ **of corpus volume.** Since by (7.6) $F = sn/120$, and $10^4 \leq s \leq 10^5$ we get that $F \geq 100n$. Thus the top-100 words have frequency at least $F_{100} = F/100 \geq n$. The doclist might include all docs of the corpus. Thus the 100 doclists use $(100 \cdot n \cdot 1)$B altogether. The postings however (2B per posting) use $2(F + \ldots + F/100) = 2F \ln 100 = 9.2sn/120 = 8sn/100$ bytes. Adding the two together we get roughly 9-10% of corpus volume!
- **Conclusion: One third of the index relates to the Top-100 popular words!**

Exercise 1 *Show that for a collection of D documents of average size $6KB$ and average word length 6, Heaps' Law can be used with $k = 100$ and $\beta = 0.5$ to show that the number of dictionary words is approximately $3000\sqrt{n}$.*

Exercise 2 *Reconsider the calculations of page 6 for a collection of $D = 1,000,000,000$ documents by using the following considerations.*

- **Web-page size.** *Assume that each web-page is on the average about 6KB long.*
- **Number of words.** *The average size of a word is around 6 characters/bytes (including surrounding spaces). What is the (average) number of words per document and what is the total number of words for the collection?*
- **Number of unique words.** *The English language has about 300,000 words including derivatives the total goes up to 750000 (See Figure 2). However Google in 1997 used to store close to 14,000,000 words. Use Heaps Law to determine the size of the dictionary in unique words. Use $k = 100$ and $\beta = 1/2$.*
- **Incidence Matrix size.** *How many rows and how many columns in the incidence matrix?*
- **Sparseness.** *What fraction of the matrix is non-zero? On the average what fraction of an incidence vector is non-zero?*
- **What is a better storage solution?** *How much space will it occupy? Express size in bytes. (Assume 1 byte is 8 bits.)*

