Notes of Information Retrieval course

Marco Natali

INDICE

ELENCO DELLE FIGURE

1 INTRODUCTION TO INFORMATION RETRIEVAL

In this course we will strongly consider *search engines*, but *information retrieval* is not only consider on search engines, so a definition of information retrieval is the following

Def. Information retrieval is finding material (usually documents) of unstructured nature that satisfies an information need from within large collections

An information need is the topic about which the user desires to know more, and is differentiated from a query, which is what the user conveys to the computer in an attempt to communicate the information need.

As defined in this way, information retrieval used to be an activity that only a few people engaged in: reference librarians, paralegals, and similar professional searchers, but now the world has changed, and hundreds of millions of people engage in information retrieval every day when they use a web search engine or search their email.

Information retrieval systems can also be distinguished by the scale at which they operate, and it is useful to distinguish three prominent scales. In web search, the system has to provide search over billions of documents stored on millions of computers. Distinctive issues are needing to gather documents for indexing, being able to build systems that work efficiently at this enormous scale, and handling particular aspects of the web, such as the exploitation of hypertext

In between is the space of enterprise, institutional, and domain-specific search, where retrieval might be provided for collections such as a corporation's internal documents, a database of patents, or research articles on biochemistry. In this case, the documents will typically be stored on centralized file systems and one or a handful of dedicated machines will provide search over the collection

Data available and considered can be from different nature, so we have the following classification:

STRUCTURED DATA: are data that tends to refer to tables and typically allows numerical range and exact match (for text) queries, like for example "Salary < 60000 AND Manager = Smith".

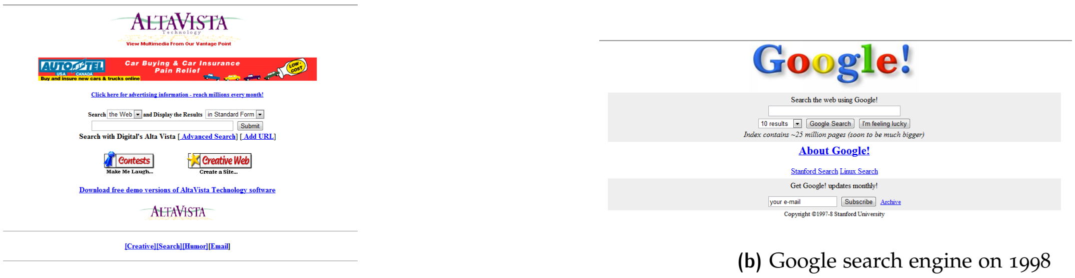
SEMISTRUCTURED DATA (XML/JSON): type of data where are available some structured aspects, like know the name of a document, chapter or paragraph, so it facilitate some semi-structured search like "Title contains data AND Bullets contain search".

UNSTRUCTURED DATA: Typically refers to free text, and allows keyword queries including operators and more sophisticated "concept" queries like find all web pages dealing with drug abuse.

Is the classic model for searching text documents, so we more concentrate on this type of data.

We consider now the search engines, that are defined as

Def. A search engine is a software system that is designed to carry out search, which means to search page in a systematic way for particular information specified in a textual search query.
The search results are generally presented in a line of results, often referred to as search engine results pages and the information may be a mix of links to pages, images, videos, infographics, articles, research papers, and other types of files.


ALTAVISTA
Technology
View Multimedia From Our Vantage Point
AUTO-BY-TEL
USA CANADA
Buy and insure new cars & trucks online
Car Buying & Car Insurance
Pain Relief
LOW-COST
Click here for advertising information - reach millions every month!
Search
the Web
and Display the Results
in Standard Form
Submit
Search with Digital's Alta Vista [Advanced Search] [Add URL]
Contests
Make Me Laugh...
Creative Web
Create a Site...
Download free demo versions of AltaVista Technology software
ALTAVISTA
[Creative][Search][Humor][Email]

(a) Altavista search engine web page

Google!
Search the web using Google!
10 results
Google Search
I'm feeling lucky
Index contains ~25 million pages (soon to be much bigger)
About Google!
Stanford Search Linux Search
Get Google! updates monthly!
your e-mail
Subscribe
Archive
Copyright ©1997-8 Stanford University

(b) Google search engine on 1998

Figura 1: Some example of Search Engine

Search engines are not only web search engine but contains also social network (Facebook), streaming site (Netflix), Maps (OpenStreetMap) and work finder (Linkedin) and we have 5 generations of search engines:

ZERO GENERATION: introduced on 1991, where was used only metadata added by users

FIRST GENERATION: introduced on 1995-1997 and used only on-page web-text data, as can be seen on figure 1a.

SECOND GENERATION: introduced on 1998 by Google, use off-page, web-graph data, where it use *anchor texts* (how people refer to the page) and *links*, that strongly improve the usability and utility of a search engine. An example of second generation search engine can be viewed on figure 1b.

THIRD GENERATION: introduced on 2005, it start to answer "the need behind the query" and are added more sources, like maps, images, news, wikipedia and so on.

FOURTH GENERATION: introduced on 2012, it strongly concern about *knowledge graph* defined as

Def. Knowledge graph is a knowledge base used by Google and its services to enhance its search engine's results with information gathered from a variety of sources.
The information is presented to users in an infobox next to the search results.

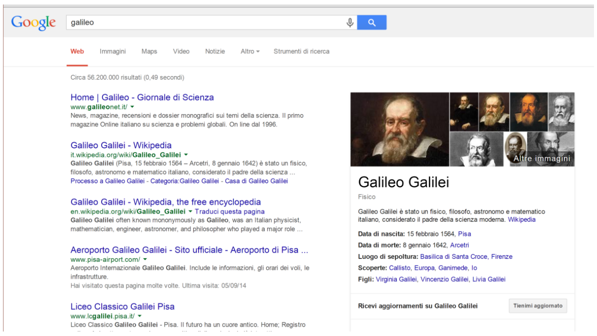
An example of knowledge graph can be viewed on figure 2 and knowledge graph also permit to transform word to concept, that can provide more information and provide more accurate results on user query, as we can see on figure 3 as we can viewed the phrase "Leonardo is the scientist that has painted the Mona Lisa".
With Knowledge graph is also possible to consider *polysemy* and *sinonimy* word, so at example is possible recognise that "Microsoft's browser" and "Internet explorer" represent the same concept.

In figure 4 is possible to notice which was the evolution on Google search engine.

Now are available "devices 2.0", that have their IDs, communication capacity, computing and storage, like for example car with maps, where the driver can asks (using text or voice audio) the route for a place.

1.1 BOOLEAN RETRIEVAL MODEL

We consider now the first model of Information retrieval, that was mainly used on first generation of search engine, but are also used currently in email, library catalog and so on.



Google galileo

Web Immagini Maps Video Notizie Altro Strumenti di ricerca

Circa 56.200.000 risultati (0,49 secondi)

Home | Galileo - Giornale di Scienza
www.galileonet.it/
News, magazine, recensioni e dossier monografici sui temi della scienza. Il primo magazine Online italiano su scienza e problemi globali. On line dal 1996.

Galileo Galilei - Wikipedia
it.wikipedia.org/wiki/Galileo_Galilei
Galileo Galilei (Pisa, 15 febbraio 1564 – Arcetri, 8 gennaio 1642) è stato un fisico, filosofo, astronomo e matematico italiano, considerato il padre della scienza ...
Processo a Galileo Galilei - Categoria:Galileo Galilei - Casa di Galileo Galilei

Galileo Galilei - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Galileo_Galilei Traduci questa pagina
Galileo Galilei often known mononymously as Galileo, was an Italian physicist, mathematician, engineer, astronomer, and philosopher who played a major role ...

Aeroporto Galileo Galilei - Sito ufficiale - Aeroporto di Pisa ...
www.pisa-airport.com/
Aeroporto Internazionale Galileo Galilei. Include le informazioni, gli orari dei voli, le infrastrutture.
Hai visitato questa pagina molte volte. Ultima visita: 05/09/14

Liceo Classico Galilei Pisa
www.lcgalilei.pisa.it/
Liceo Classico Galileo Galilei - Pisa. Il futuro ha un cuore antico. Home; Registro

Altre immagini

Galileo Galilei

Fisico

Galileo Galilei è stato un fisico, filosofo, astronomo e matematico italiano, considerato il padre della scienza moderna. Wikipedia

Data di nascita: 15 febbraio 1564, Pisa

Data di morte: 8 gennaio 1642, Arcetri

Luogo di sepoltura: Basilica di Santa Croce, Firenze

Scoperte: Callisto, Europa, Ganimede, Io

Figli: Virginia Galilei, Vincenzio Galilei, Livia Galilei

Ricevi aggiornamenti su Galileo Galilei

Tienimi aggiornato

Figura 2: A Google search with knowledge graph

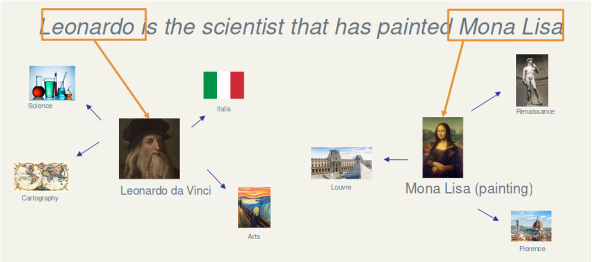


Figura 3: an example of transition from word to concept with the phrase "Leonardo is the scientist that has painted the Mona Lisa"

Figura 4: Google search evolution

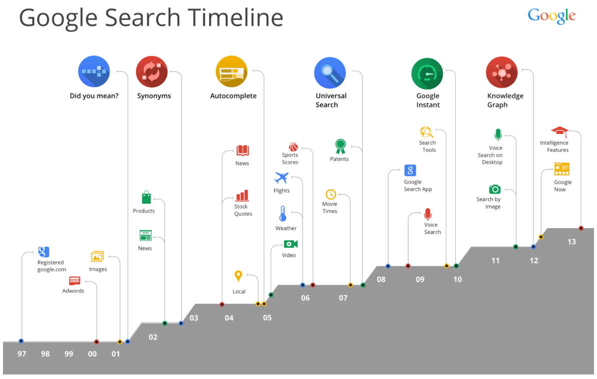
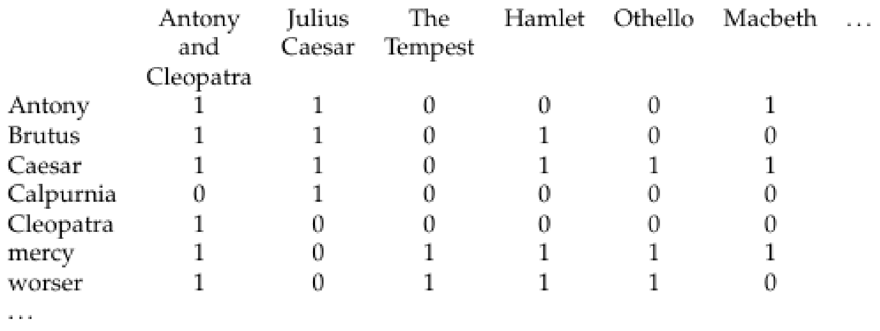


Figura 5: A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise



	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

Def. The boolean retrieval model is a model able to asks query formed by a boolean expression (query where we use AND, OR, NOT operators to join terms) where we views each document as a set of words and it is a precise model, because document matches condition or not.

We consider as example to determine which plays of Shakespeare contain the words "Brutus AND Ceaser AND NOT Calpurnia" and the simplest form is do a sort of linear scan, but this type of text processing does not enable to rank results, consider some similarity measures and so on, so we use our boolean retrieval model defined above.

We consider as *documents* whatever units we have decided to build a retrieval system over, so they might be individual memos or chapters of a book.
We will refer to the group of documents over which we perform retrieval as the (document) *collection* and it is sometimes also referred to as a *corpus* (a body of texts).

To assess the effectiveness of an IR system (the quality of its search results), a user will usually want to know two key statistics about the system's returned results for a query:

PRECISION: What fraction of the returned results are relevant to the information need?

RECALL: What fraction of the relevant documents in the collection were returned by the system?

In figure 5 is possible to note a term-document incident matrix, where we record for each document, here a play of Shakespeare's, whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000 different words).

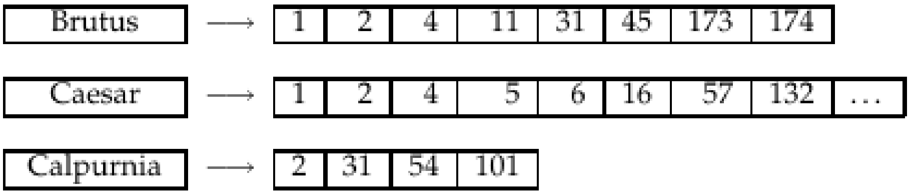
The problem of Term-document incident matrix is that the matrix could be very big and it is also a *sparse* matrix, so we waste a lot of spaces to represent useless data.

To solve this problem we introduce *inverted index*, where for each term t, we must store a list of all documents that contain t and we identify each by docID, a document serial number.
It is called inverted because usually in a document we have a list of word instead in our index we have for a word a list of documents where it occurs.
In figure 6 is possible to note an example of inverted index, where we only store the docID where a word appear in a document.

The advantages of inverted index is that query requires just a scan and also we can store smaller integers, using *gap coding*, that will enable to use less amount of memory.

To execute an AND query the first approach consist to check each element of the two postings list that will cause $n * m$ operations, where n and m are the length of the two postings list, and we can achieve a better result if we sort the two postings list, so at least if we compare 1 and 2 we can avoid to consider to compare the element 1

Figura 6: an example of Inverted index for Brutus, Caesar and Calpurnia



Brutus → 1 2 4 11 31 45 173 174

Caesar → 1 2 4 5 6 16 57 132 . . .

Calpurnia → 2 31 54 101

Figura 7: Pseudocode of Intersection between two postings list

```
INTERSECTION$(p_1, p_2)$
  $answer = <>$
  while $p_1 \neq$ NIL and $p_2 \neq$ NIL
      if DOCID$(p_1)$ == DOCID$(p_2)$
          ADD$(answer,$ DOCID$(p_1))$
          $p_1 =$ NEXT$(p_1)$
          $p_2 =$ NEXT$(p_2)$
      elseif DOCID$(p_1) <$ DOCID$(p_2)$
          $p_1 =$ NEXT$(p_1)$
      else $p_2 =$ NEXT$(p_2)$
      return $answer$
```

with element that are greater than 2, so we need only $n + m$ comparison that will improve the performance of AND query.

The pseudocode to intersect two postings list in a AND query can be found on figure 7 and in case we have to compute an AND query between 3 operands, it is better to first compute an AND query between operands with smallest length and then compute an AND query with the other operand, so to compute the query "Brutus AND Caesar AND Calpurnia" in figure 8 we first compute "Brutus AND Calpurnia" and then compute "(Brutus AND Calpurnia) AND Caesar"

In figure 9 it can be find the pseudocode to compute the intersection between n terms, that use the intution we have explain briefly previously.

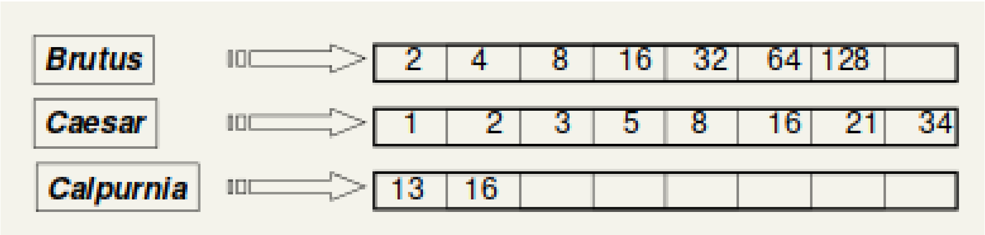
To gain the speed benefits of indexing at retrieval time, we have to build the index in advance and the major steps in this are:

1. Collect the documents to be indexed

2. Tokenize the text, turning each document into a list of tokens

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms

Figura 8: And query between Brutus, Calpurnia and Caesar



Brutus → 2 4 8 16 32 64 128

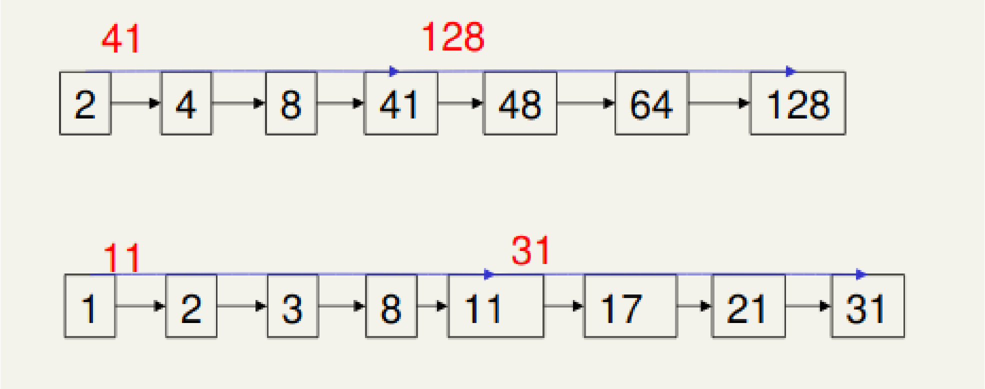
Caesar → 1 2 3 5 8 16 21 34

Calpurnia → 13 16

Figura 9: Pseudocode of Intersection operation between n terms

```
INTERSECTION($< t_1, \ldots, t_n >$)
terms = SortByIncreasingFrequency($< t_1, \ldots, t_n >$)
result = postings(first(terms))
terms = rest(terms)
while terms $\neq$ NIL and result $\neq$ NIL
    result = intersect(result, postings(first(terms)))
    terms = rest(terms)
return result
```

Figura 10: Postings lists with skip pointers. The postings intersection can use a skip pointer when the end point is still less than the item on the other list.



4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings

The first three steps will consider later in the course and the last one is considered now when we talk about inverted index.

To represent the index we have to done some choice about data structure to use, infact a fixed length array would be wasteful as some words occur in many documents, and others in very few.
For an in-memory postings list, two good alternatives are singly linked lists or variable length arrays: singly linked lists allow cheap insertion of documents into postings lists (following updates, such as when recrawling the web for updated documents), and naturally extend to more advanced indexing strategies such as skip lists, which require additional pointers.
Variable length arrays win in space requirements by avoiding the overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches and extra pointers can in practice be encoded into the lists as offsets.

If updates are relatively infrequent, variable length arrays will be more compact and faster to traverse and we can also use a hybrid scheme with a linked list of fixed length arrays for each term.
When postings lists are stored on disk, they are stored (perhaps compressed) as a contiguous run of postings without explicit pointers to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

An improvement to our intersection can be obtained with *skip pointers*, where we put a pointer head to some element that will permits to avoid to consider some elements, as it can be viewed on figure 10 and in figure 11 there is the pseudocode of intersection with skip pointers.

Figura 11: Pseudocode of Intersection with Skip Pointers

```
INTERSECTWITHSKIPS($p_1, p_2$)
answer $= <>$
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
    if DOCID($p_1$) $=$ DOCID($p_2$)

```

With skip we can logically divide elements in block, where we can access to the first item of a block and we can avoid maybe avoid to consider other elements of a block.
The important thing that we have to record is that more is the size of a block less is the number of skip pointers and more element is possible to purge from our operation and we have as worst case when all elements are in the last block so we have to consider $\frac{n}{L}$ blocks and L to scan the last block and we achieve

$$min\frac{n}{L} + L$$

when $L = \sqrt{n}$.

Typically we use a dynamic programming based skip pointers, where we assume that exist some documents very frequent as a result of a query so whenever we do an intersection it is more likely that they are occuring, so we divide block with frequent document as start of a block.
With Dynamic Programming approach the append of a new element will be difficult because we have to recompute the distribution of skip pointers instead on equal size skip pointers we only append new block, with the new element.

Another improvement is *Recursive Merge*, where we take a pivot (the median element) and we do a binary search to retrieve if the pivot occurs in the other lists and then we do something similar to Quicksort.
The time complexity of this approach is the following:

BEST CASE: we have that the median is always out of the other list so we have $O(\log n * \log m)$, because you always remove the upper bound of the list so we consider only the lower bound of median.

WORST CASE: we have that median is always inside the list so we have

$$T(n, m) = O(\log n) + 2T(n/2, m/2) = O(m \log n/m)$$

where the last equation is obtained by master theorem.

If $m \sim n$ we have $O(m) = O(n + m)$, in case $m << n$ we obtain $O(m \log n)$ that means for every element in list of size m we do a binary search.

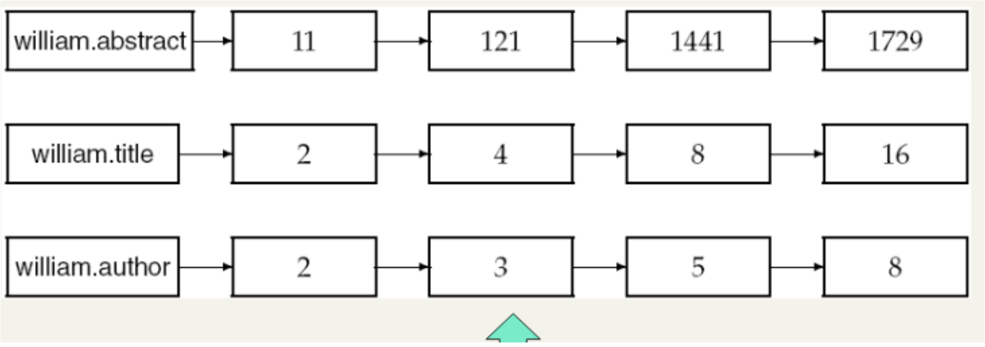
In case we have an OR query we do an union between n posting list without problem, instead with an OR NOT query there will be a problem because if we have a list with 2 element the complementation of a list has $N - 2$ elements and that will be a huge problem.

Information Retrieval is not only limited to boolean model so we want to consider phrases, proximity operators like "Gates near Microsoft" where we need index to capture term position in docs and sometimes we are also interested to consider zones in documents like "Find documents with (author = Ullman) and (text contains automata)".

We define zone indexes as

Def. Zone indexes: is a region of the doc that can contain an arbitrary amound of text, like title, abstract or references.

Figura 12: Inverted index on zone fields



We build inverted indexes on fields and zones to permit querying, as can be noted on figure 12.

An *search engine* it comes of several components that can be viewed in figure **??**, we will start talking about crawling and then we will introduce the other components in the following chapter.

2 SEARCH ENGINE

An *search engine* it comes of several components that can be viewed in figure **??**, we will start talking about crawling and then we will introduce the other components.

In figure **??** there is a bow tie that exploits some consideration about crawler where we can see that web pages that search engine consider are a small amount of all pages.

2.1 CRAWLING

Crawling is a graph visit of the web graph, run 24h each days, in order to discover new web pages and we have a direct graph $G = (N, E)$ with N that indicate N changes in nodes (usually trillion of nodes) and E indicate a link between two nodes.

In crawling we have to choose between several issues:

- How to crawl? we can choose between quality ("Best" pages first), efficiency (Avoid duplication) and also about malicious pages (Spam pages, Spider traps) including dynamically generated.
- How much to crawl and thus index? Coverage and Relative Coverage (coverage comparated with competitor)
- How often to crawl? Freshness: How much has changed?

Actually is difficult to decide how to implement and design a crawler that should respect the issues that we have introduced before.

In figure 13 it is possible to note the general structure of crawler process, in figure 14 it possible to note the component used to implement a crawler and in the end in figure **??** there is an pseudocode implementation of Crawler's component.

In visiting the URL frontier we have to define how "good" a page is and there exists several metrics (BFS, DFS, RANDOM, PAGERANK and so on) and also now we will introduce *Mercator*, an example of search engine released in 1999, where are present 3 assumpionts:

1. Only one connection per host is open at a time.
2. a waiting time of a few seconds occurs between successive requests to the same host.
3. high-priority pages are crawled preferentially.

The structure of Mercator can be viewed in figure 16 and we have that *Front queues* manage prioritization: prioritizer assigns to an URL an integer priority (refresh, quality, application specific) between 1 and K and appends URL to corresponding queue, according to priority.
Back queues enforce politeness: each back queue is kept non-empty and contains only URLs from a single host; in a back-queue request it select a front queue randomly, biasing towards higher queues.

The *min-heap* contains one entry per back queue and the entry is the earliest time t_e at which the host corresponding to the back queue can be "hit again": this earliest time is determined from last access to that host and any time buffer heuristic we choose.
The *crawl thread* consist that a crawler seeks a URL to crawl: extracts the root of the heap, waits the indicate time t_{url}, parses URL and adds its out-links to the Front

Figura 13: Diagramm of Crawler operation

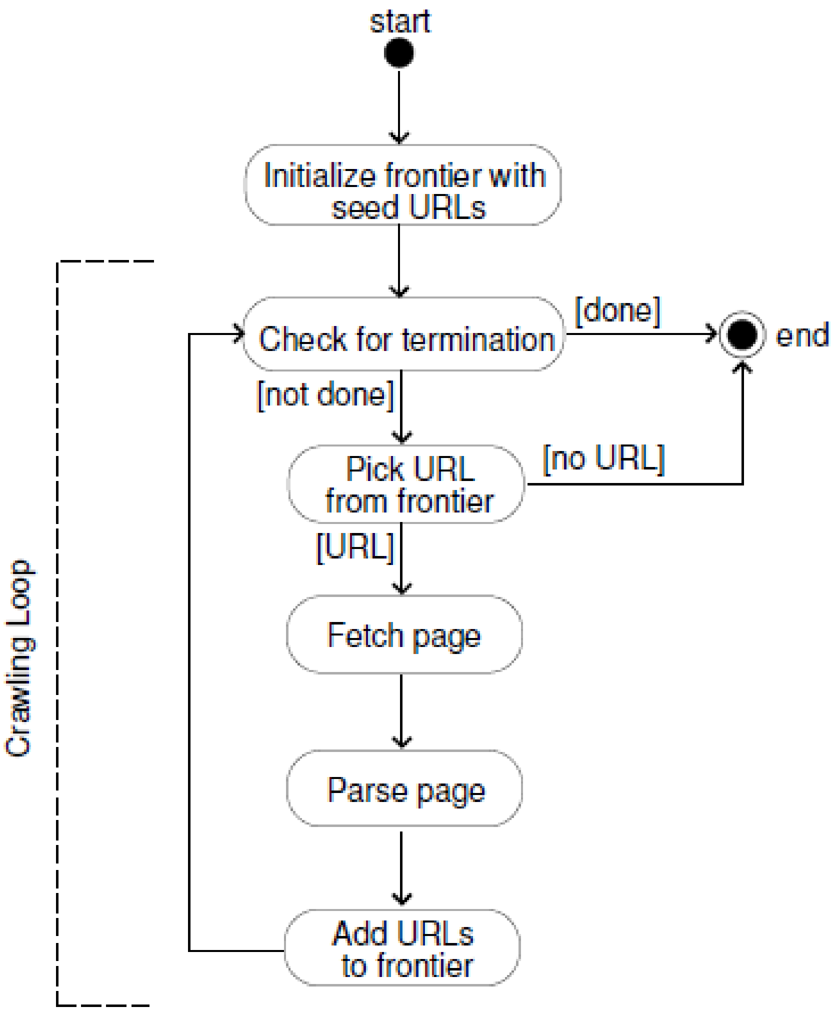


Figura 14: Component of Crawler

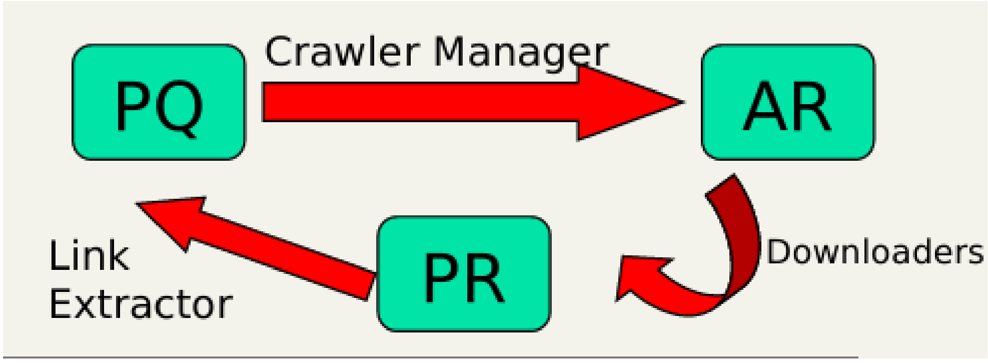
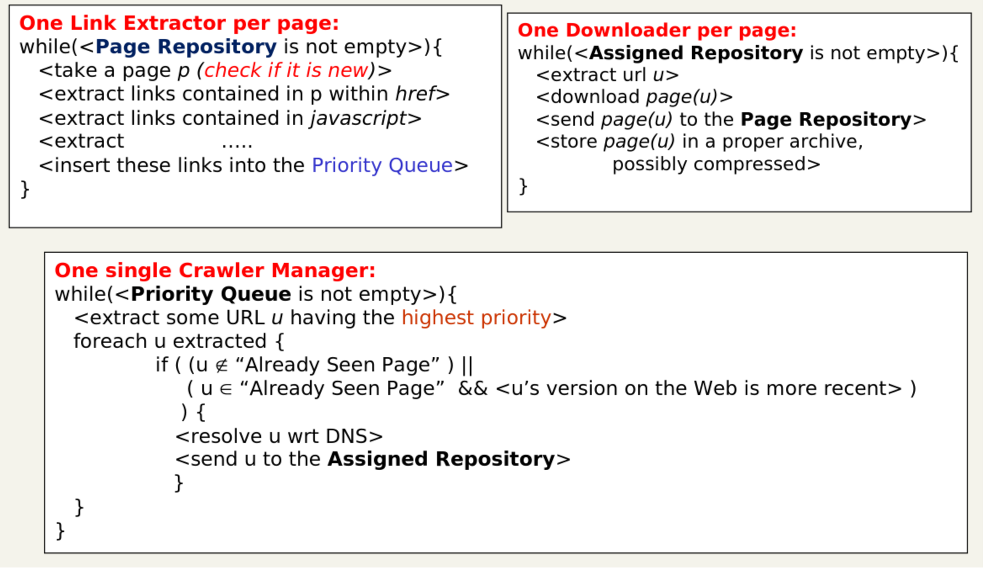


Figura 15: Pseudocode of Crawler components

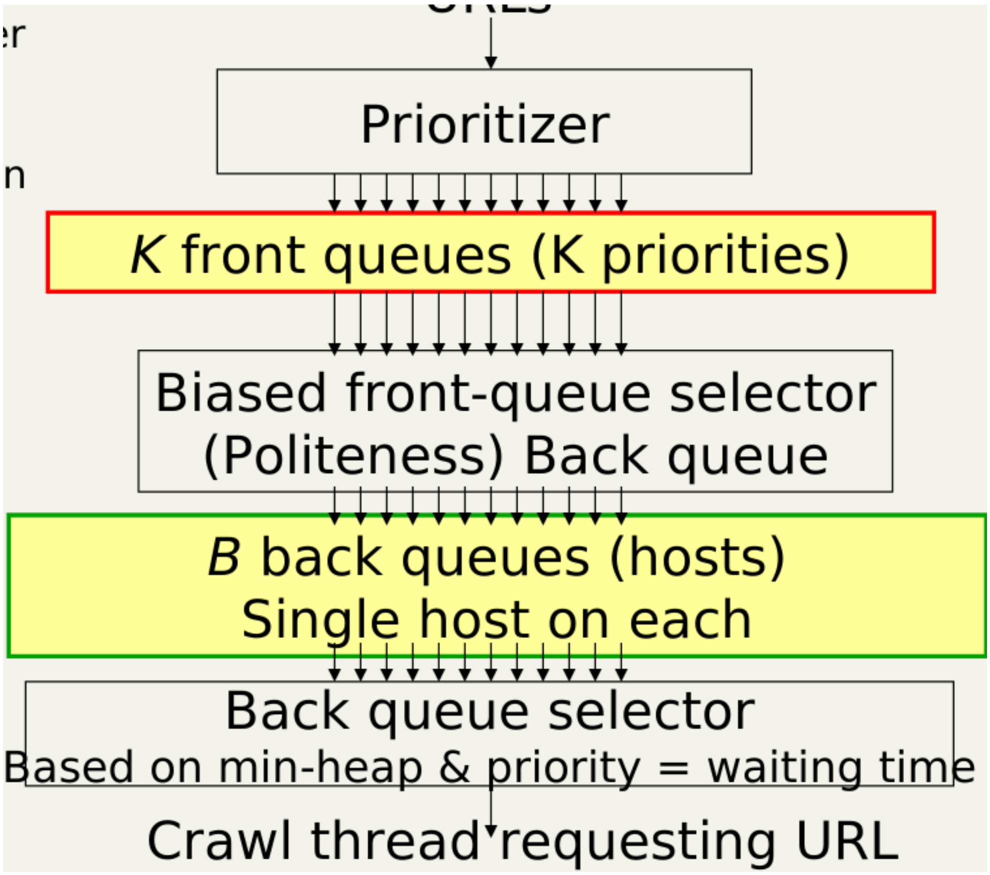


```
One Link Extractor per page:
while(<Page Repository is not empty>){
   <take a page p (check if it is new)>
   <extract links contained in p within href>
   <extract links contained in javascript>
   <extract               .....
   <insert these links into the Priority Queue>
}
```

```
One Downloader per page:
while(<Assigned Repository is not empty>){
   <extract url u>
   <download page(u)>
   <send page(u) to the Page Repository>
   <store page(u) in a proper archive,
            possibly compressed>
}
```

```
One single Crawler Manager:
while(<Priority Queue is not empty>){
   <extract some URL u having the highest priority>
   foreach u extracted {
            if ( (u ∉ "Already Seen Page" ) ||
                 ( u ∈ "Already Seen Page"  && <u's version on the Web is more recent> )
               ) {
               <resolve u wrt DNS>
               <send u to the Assigned Repository>
               }
   }
}
```

Figura 16: Structure of Mercator search engine



URLs
er
n
Prioritizer
K front queues (K priorities)
Biased front-queue selector
(Politeness) Back queue
B back queues (hosts)
Single host on each
Back queue selector
Based on min-heap & priority = waiting time
Crawl thread requesting URL

queues.
If back queue q gets empty, pulls a URL v from some front queue (more prob for higher queues): if there's already a back queue for v's host, append v to it and repeat until q gets not empty, else make q the back queue for v's host.
If back queue q is non-empty, pick URL and add it to the min-heap with $priority = waitingtimet_{url}$.

To check if the page has been parsed/downloaded before URL match, duplicate document match and near-duplicate document match we have several solutions:

- Hashing on URLs: after 50 bln pages, we have "seen" over 500 bln URLs and each URL is at least 1000 bytes on average so in overall we have about $500.000Tb (= 500Pb)$ for just the URLS

- Disk access with caching (e.g. Altavista): > 5 ms per URL check and $> 5ms * 5 * 10^{11}$ URL-checks ($80years/1PC$ or $30gg/1000PCs$).

- *Bloom Filter* (Archive): for 500 bln URLs we have about $500Tbit = 50Tb$

2.2 BLOOM FILTER

An empty Bloom filter is a bit array of m bits, all set to 0 and there must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions, generating a uniform random distribution.
Typically, k is a small constant which depends on the desired false error rate ϵ, while m is proportional to k and the number of elements to be added and to add an element, feed it to each of the k hash functions to get k array positions and set the bits at all these positions to 1.

To query for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions and if any of the bits at these positions is 0, the element is definitely not in the set; if it were, then all the bits would have been set to 1 when it was inserted and if all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive.

In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem and the requirement of designing k different independent hash functions can be prohibitive for large k, so for a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields.
Alternatively, one can pass k different initial values (such as $0, 1, \ldots, k-1$) to a hash function that takes an initial value, or add (or append) these values to the key.

Removing an element from this simple Bloom filter is impossible because there is no way to tell which of the k bits it maps to should be cleared and although setting any one of those k bits to zero suffices to remove the element, it would also remove any other elements that happen to map onto that bit, so since the simple algorithm provides no way to determine whether any other elements have been added that affect the bits for the element to be removed, clearing any of the bits would introduce the possibility of false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed, however, false positives in the second filter become false negatives in the composite filter, which may be undesirable and in this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.

Assume that a hash function selects each array position with equal probability, so if m is the number of bits in the array, the probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element is

$$1 - \frac{1}{m}$$

so if k is the number of hash functions and each has no significant correlation between each other, then the probability that the bit is not set to 1 by any of the hash functions is

$$(1 - \frac{1}{m})^k$$

and using the well-known identity for e we can obtain for large m

$$(1 - \frac{1}{m})^k = ((1 - \frac{1}{m})^m)^{k/m} \approx -e^{k/m}$$

If we have inserted n elements, the probability that a certain bit is still 0 is

$$(1 - \frac{1}{m})^{kn} \approx -e^{kn}m = 0.62^{m/n}$$

It minimize prob. error for $k = (m/n)ln2$and it is advantageous when $(m/n) <<$ $($ key-length in bits $+ \log n)$

Pattern maching is a set of objects, whose keys are complex and time costly to be compared (URLs, matrices, MP3 and so on) so we use Bloom Filter to reduce the explicit comparison and it is effective in hierarchical memories (Example on Dictionary matching).

Another example is *Set intersection*, where we have two machines M_A and M_B, each storing a set of items A and B and we wish compute $A \cup B$ exchanging small number of bits.
A solution consist to compute $B - A$ by exchanging small amount of bits ($|A| \log \log |B|$), time depending on $B - A$ and with only 1 communication round and we consider now *Patricia Tree*, a special variant of the radix binary trie, in which rather than explicitly store every bit of every key, the nodes store only the position of the first bit which differentiates two sub-trees.
During traversal the algorithm examines the indexed bit of the search key and chooses the left or right sub-tree as appropriate, and an example can be viewed in figure 17.

Given PT_A and PT_B at machine M_B we procede as follows:

1. Visit PT_B top down and compare a node of PT_B against the correspective node in PT_A.

2. If there is a match, the visit backtracks, otherwise proceeds to all children.

3. If we reach a leaf, then the correspective of B is declared to be in $B - A$

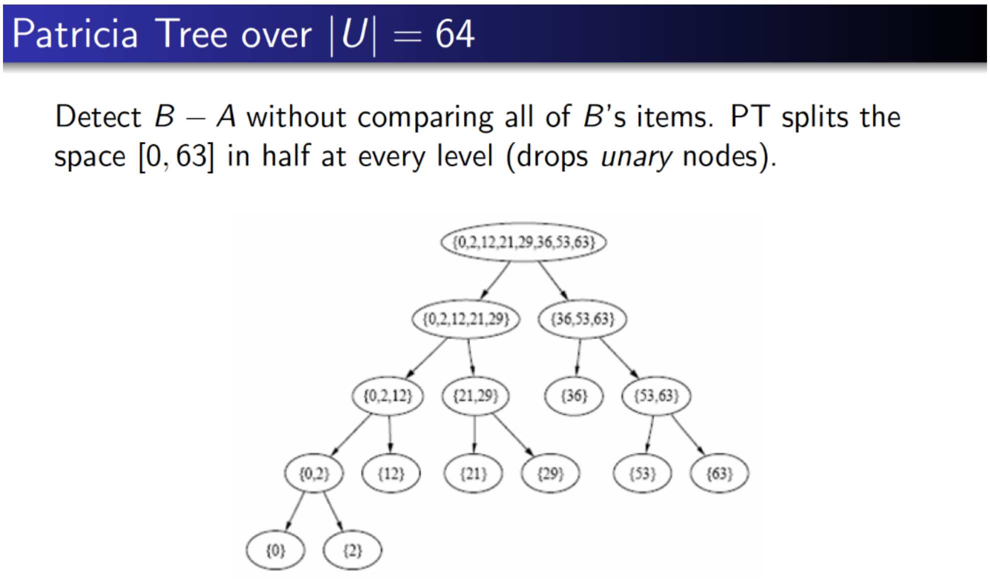
In figure 18 is possible to note Merkle Tree, that are Patricia Tree with hashing and we consider an approximate algorithms, visible in figure 19, that use $BF(MT_A)$ to send MT_A in less bits and bookkeeping for its structure, but this of course introduce false-positive errors.

Given $BF(MT_A)$ and MT_B, the machine M_B proceeds as follows:

1. Visit MT_B top-down and, for each node, check its hash in $BF(MT_A)$.

2. IF there is a match the visit backtrack, otherwise proceeds to their children.

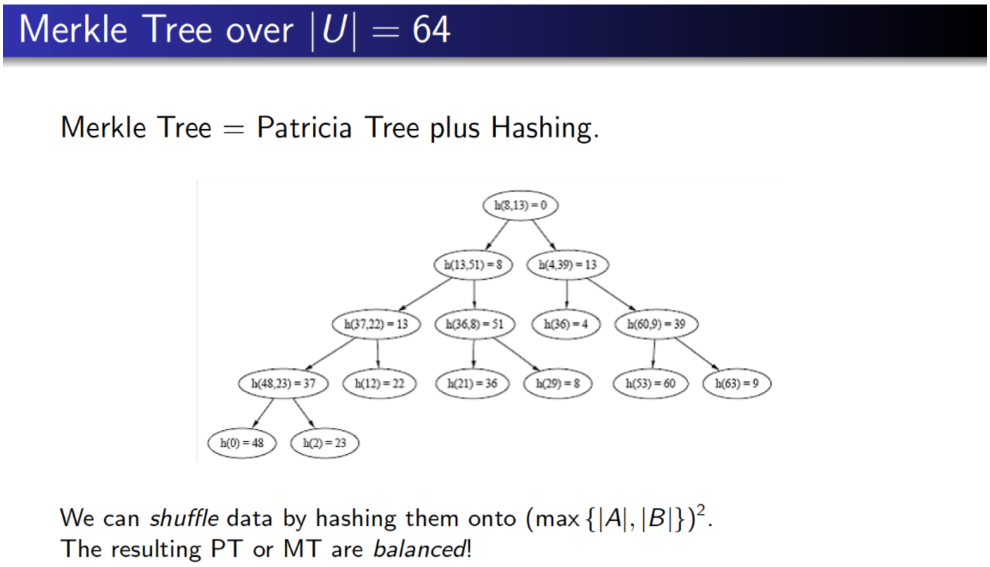
3. If we reach a leaf, then the correspective of B is declared to be in $B - A$



Patricia Tree over $|U| = 64$

Detect $B - A$ without comparing all of B's items. PT splits the space $[0, 63]$ in half at every level (drops *unary* nodes).

Figura 17: Example of Patricia Tree

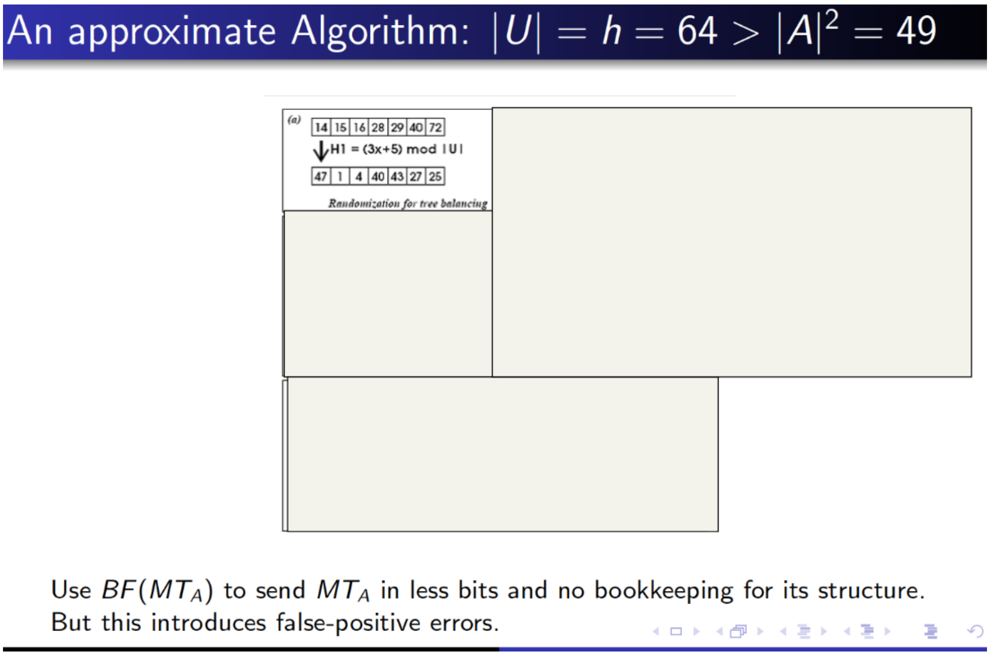


Merkle Tree over $|U| = 64$

Merkle Tree = Patricia Tree plus Hashing.

We can *shuffle* data by hashing them onto $(\max\{|A|, |B|\})^2$. The resulting PT or MT are *balanced*!

Figura 18: Example of Merkle Tree



An approximate Algorithm: $|U| = h = 64 > |A|^2 = 49$

Use $BF(MT_A)$ to send MT_A in less bits and no bookkeeping for its structure. But this introduces false-positive errors.

Figura 19: Approximative algorithm to compute Set intersection

Let $m_A = \Theta(|A| \log\log|B|)$ and the optimal $k_A = \Theta(\log\log|B|)$ we send m_A bits for the $BF(A)$ and we have

$$\epsilon_A = (1/2)^{k_A} = O(1/\log|B|)$$

that is the error of $BF(A)$.
The probability of a success for a leaf is given by $(1-\epsilon_A)^d = \Theta(1)$ and for a correct leaf we have visited its downward path of length $\Theta(\log|B|)$ computing $\Theta(\log\log|B|)$ hash functions per node.
This needs a round, $O(|A|\log\log|B|)$ bits, and $O(B - A\log|B|\log\log|B|)$ of reconciliation time.

2.2.1 Spectral Bloom Filter

We define now an evolution of Bloom Filter, used not only in URL match, with the following definition

Def (Spectral Bloom Filter)**.** We have a multiset $M = (S, f_X)$ where S is a multiset and f_X is a count function that return the number of occurrences of x in M

Comparated with Bloom Filter we have an slightly large usage of space, but we achieve better performance, and also can be built incrementally for streaming data.
Applications of this data structure is to answer to two common query:

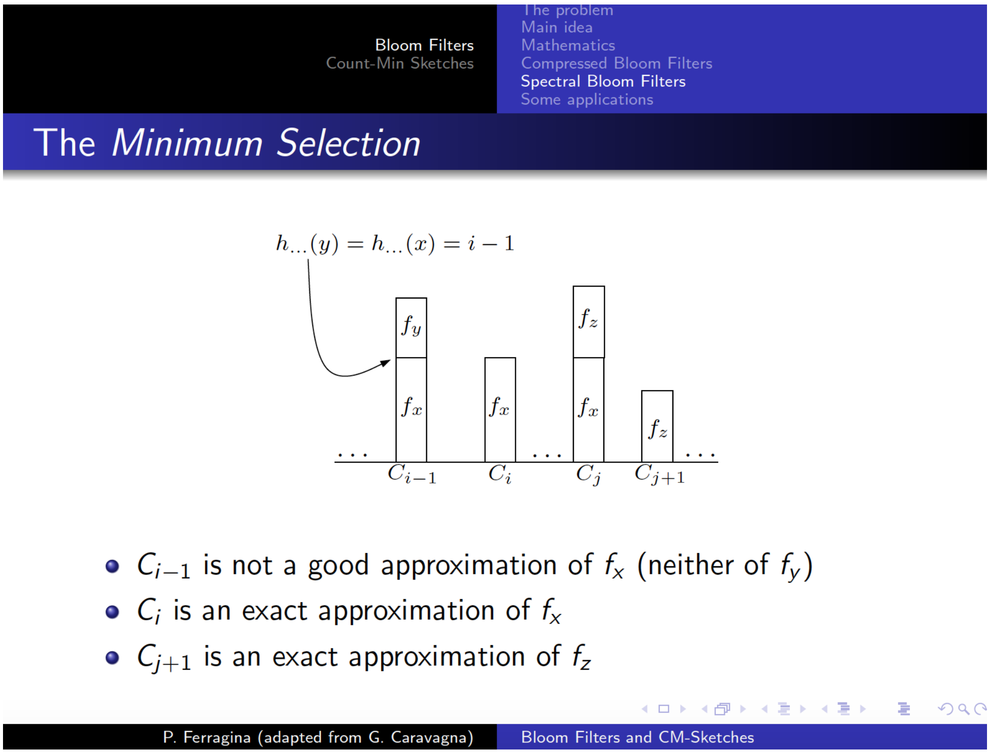
ICEBERG QUERY: given x check if $f_X > T$, where T is a dynamically threshold

AGGREGATE QUERY: $SELECTcount(a1)FROMRWHEREa1 = v$

B vector is replaced by a vector of counters $C_1, C_2, \ldots, C_m$, where C_i is the sum of f_X values for elements $x \in S$ mapping to i, and approximations of f_X are stored into $C_{h_1(x)}, C_{h_2(x)}, \ldots, C_{h_k(x)}$, but due to conflicts C_i provide only an approximation.
In figure 20 is possible to note what is good approximation or a bad approximation of f_X, and insertion and deletion are quite simple because we have only to increase/decrease each counter by 1, instead the search operation return the minimum selection (MS) value defined as

$$m_X = \min\{C_{h_1(x)}, \ldots, C_{h_k(x)}\}$$

Figura 20: Example of approximation between C_i and f_X



Bloom Filters
Count-Min Sketches

The problem
Main idea
Mathematics
Compressed Bloom Filters
Spectral Bloom Filters
Some applications

The *Minimum Selection*

- C_{i-1} is not a good approximation of f_x (neither of f_y)
- C_i is an exact approximation of f_x
- C_{j+1} is an exact approximation of f_z

P. Ferragina (adapted from G. Caravagna) Bloom Filters and CM-Sketches

The error rate is the same as bloom filter and we will now prove it

Thm 2.1. For all x it is $f_X \leq m_X$ and we have $f_X \neq m_X$ with probability $E_{SBF} = \epsilon \sim (1-p)^k$

Dimostrazione. The case that $m_X < f_X$ can not even happen instead the case $m_X > f_X$ happen when all the counter have a collision, that correspond to the event of a false positive in Bloom Filter □

Mainly we have two challenges: allow insertion/deletion while we keeping low E_{SBF} and dynamic array of variable-length counters and to solve the first problem we will use *Recurring Minimum (RM)* that is defined as

Def. An element has a RM iff more than one of its counters has value equal to the minimum

An item which is subject to a Bloom Error is typically less likely to have recurring minimum among its counters, because we have the following basic idea, using two SBF:

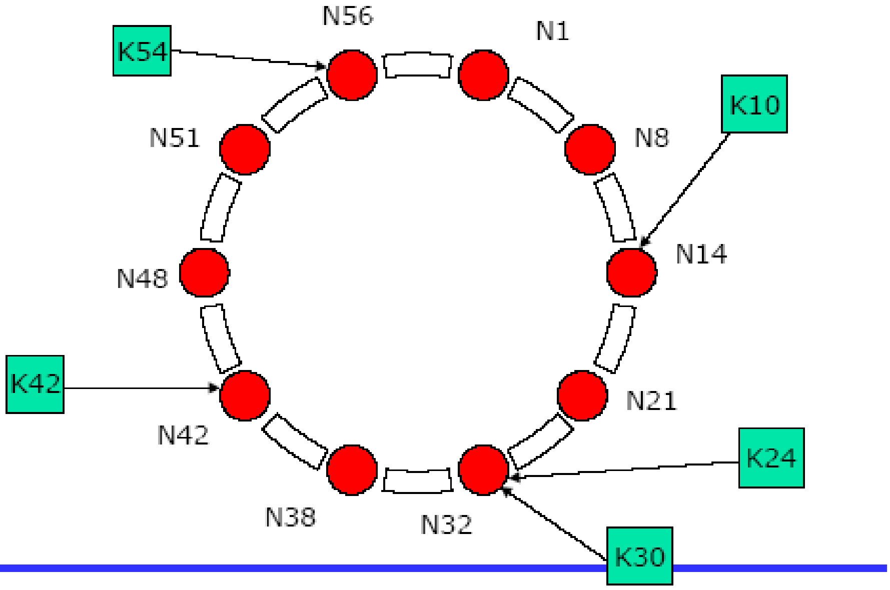
1. For item x with RM we use m_X as estimator, which is highly probable to be correct and hence $E_{SBF_1} < \epsilon$
2. For items with a SM we use a secondary SBF which is $|SBF_2| << |SBF_1|$ and thus can guarantee $E_{SBF_2} << \epsilon$.

With this approach we use more space which could be used for enlarging the single BF, but experiments show that improvements may be remarkable.

The insertion handles potential future errors, because we increase all counters of x in SBF_1 and if x has a SM in SBF_1 we look for x in SBF_2 and if yes we increase all counters of x in SBF_2, otherwise we set x in SBF_2 to be the minimum value in SBF_1.

The deletion is the inverse of insertion, so we decrease all counters of x in SBF_1 and if x has a SM in SBF_1 we decrease all counters of x in SBF_2.

Figura 21: Consistent Hashing Example



In lookup we have that if x has a RM in SBF_1 we return it otherwise we set m_x^2 as the value of x in SBF_2 that if it is > 0 we return it otherwise we return the min value of x in SBF_1.

2.3 PARALLEL CRAWLERS

Web is too big to be crawled by a single crawler, work should be divided avoiding duplication so we need several crawlers that works in parallel and assignment between different crawlers can be done in two ways:

DYNAMIC ASSIGNMENT: central coordinator dynamically assigns URLs to crawlers and it needs communication between coordinator/crawl threads.

STATIC ASSIGNMENT: web is statically partitioned and assigned to crawlers and crawler only crawls its part of the web, no need of coordinator and thus communication

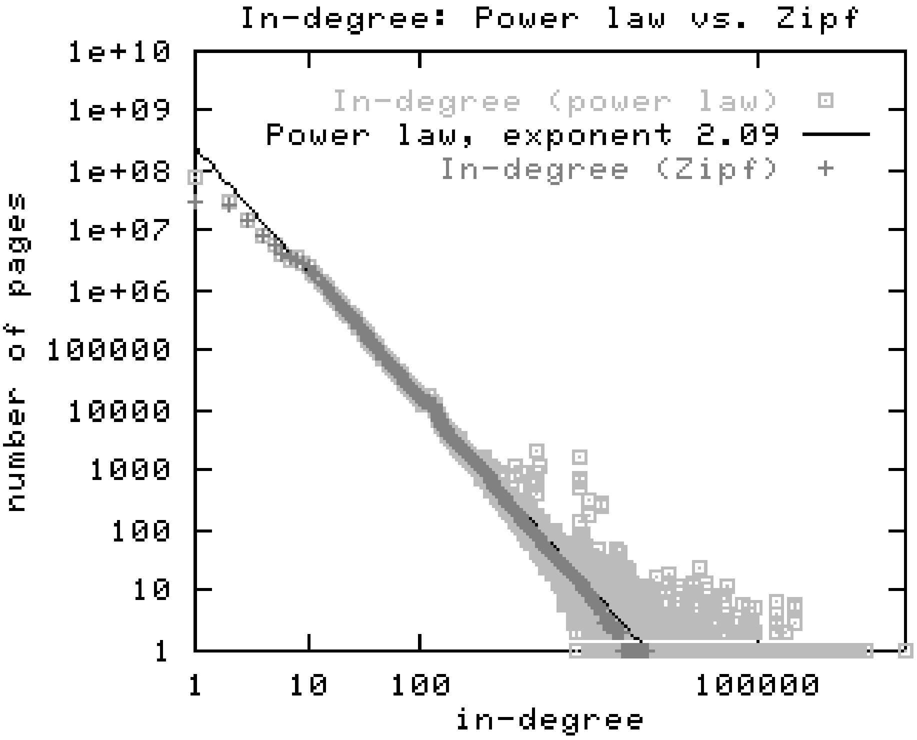
The Dynamic assignment is problematic because it is computationally expensive and may be complicated, anyway also static assignment has two problem:

- Load balancing the number of URL assigned to crawler because static schemas based on hosts may fail and dynamic assignment may be complicated

- Managing the fault-tolerance so in case we have a death of crawler or we have a new crawler we have to recompete the hash function and choose which crawler to assign.

A nice technique to solve this problem consist in *consistent hashing*, a tool for Spidering, Web Cache, P2P, Routers Load Balance and Distributed FS.
It consist that item and servers are mapped to unit circle via hash function ID() and item K are assigned to first server N such that $ID(N) \geq ID(K)$, as we can see in figure **??**.
Each server gets replicated $\log S$ times, adding a new server moves points between an old server to the new one, only, we have that in average a server gets $\frac{n}{s}$ element.

Figura 22: In-degree value in Altavista Crawl in 1997



2.4 COMPRESSED STORAGE OF WEB GRAPH

Given a directed graph $G = (V, E)$, where V are URLs and $E = (u, v)$ if u has an hyperlink to v, also isolated URLs are ignored (they do not have IN and/or OUT) and we have three key properties:

SKEWED DISTRIBUTION: probability that a node has x links is $1/x^{\alpha}$ with $\alpha \approx 2.1$, so in-value degree follows power law distribution, as we can see in figure 22 and 23.

LOCALITY: usually, most of the hyperlinks from URL u point to other URLs that are in the same host of u (about 80%), so hosts in the same domain are close to each other in the lexicographically sorted order, and thus they get close docIDs.

SIMILARITY: if URLs u and v are close in lexicographic order, then they tend to share many hyperlinks, so we have that each bit of the copy list informs whether the corresponding successor of y is also a successor of the reference x and the reference index is the one in $[0, W]$ that gives the best compression, as we can see in figure 24.

To consider these properties we now introduce *copy lists*, to compress information and exploits locality and similarity, but also consider the *copy block*, visible in figure 25, where the first bit specifies the first copy block and last block is omitted because we know the length from Out_d.

2.5 LOCALITY-SENSITIVE HASHING AND ITS APPLICATIONS

Given U users, described with a set of d features, the goal is to find (the largest) group of similar users, and to find these group we have three approaches:

1. Try all groups of users and, for each group, check the (average) similarity among all its users.

Figura 23: In-degree value in WebBase crawl in 2001

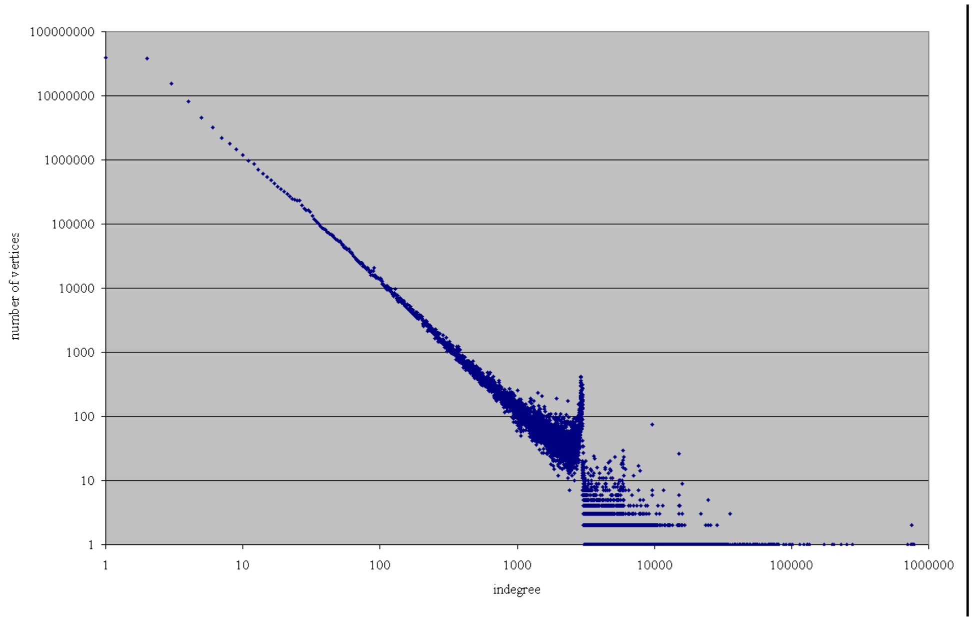
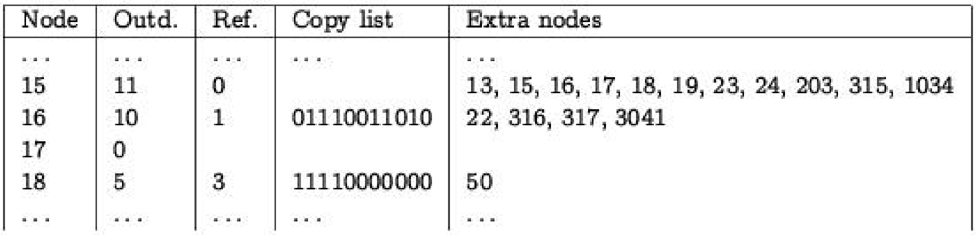
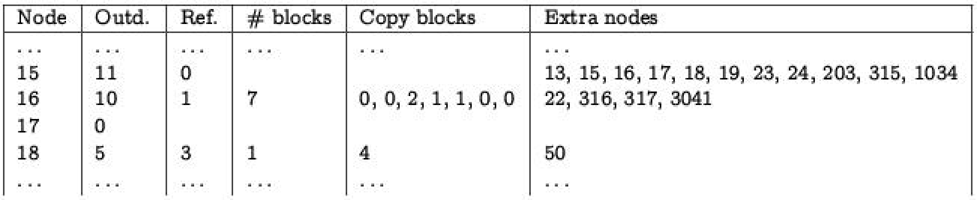


Figura 24: Example of Copy List



Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Figura 25: Example of Copy Block



Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, 317, 3041
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

The problem of this approach is that it requires $2^U * U^2$ and also if we limit groups to have a size $\leq L$ we have anyway $U^L * L^2$ that is computationally infeasible with large U.

2. Interpret every user as a point in a d-dim space, and then apply a clustering algorithm, where each iterations require $K * U$ and iterations are relatively small.
   This approach is locally optimal, comparing users/points costs $O(d)$ in time and space and iterate $k = 1, \dots, U$ costs $U^3 < U^L$, that are in order of years, so in T time we can manage $U = T^{1/3}$ users.

3. Generate a fingerprint for every user that is much shorter than d and allows to transform similarity into equality of fingerprints.
   It is randomized, correct with high probability and it guarantees local access to data, which is good for speed in disk/distributed setting.

   We consider two vectors $p, q \in \{0,1\}^d$ and we define the *hamming distance* $D(p,q)$ as the number of bits where p and q differ; we define also a *similarity* measure as
   $$s(p,q) = s = \frac{d - D(p,q)}{d} \quad 0 \leq s \leq 1$$
   We define now hash functions h by choosing a set l of k random coordinates and we have that the propability to x random such that $p(x) = q(x)$ is defined as
   $$P[\text{picking } x \text{ random such that } p(x) = q(x)] = \frac{d - D(p,q)}{d} = s$$
   The probability that the hash function h_I has the same value in p and q is defined as
   $$P[h_I(p) = h_I(q)] = s^k = (\frac{d - D(p,q)}{d})^k$$
   In case we have larger k we have small false positive, instead if we have a large l we have small false negative.

   We can iterate L times the k projections $h_I(p)$ and we set $g(p) =< h_1(p), h_2(p), \dots, h_L(p) >$, so we declare "$p$ matches q" if exist a I such that $h_I(p) = h_I(q)$ and we have that probability of a match defined as
   $$\begin{aligned} P[g(p) \approx g(q)] &= 1 - P(h_{I_j}(p) \neq h_{I_j}(q) \, \forall j) & (1) \\ &= 1 - [P(h_{I_j}(p) \neq h_{I_j}(q)]^L & (2) \\ &= 1 - (1 - s^k)^L & (3) \end{aligned}$$
   This probability follow the aspect described in figure **??** and we have that we have to scan all L hash function to create a L fingerprint with a problem in time complexity.

   To solve this problem we define for every p_i element $g(p_i) =< h_{I_1}(p_i), h_{I_2}(p_i), \dots, h_{I_L}(p_i) >$ and to compute we follow this algorithm

   (a) Sort by I_1, scan $g(p_i)$ and find group of continuous vector that have the same first component
   (b) Sort by I_2, scan $g(p_i)$ and find group of continuos vector that have the same second component
   (c) Repeat this approach L times until you sort for I_L

   This approach is done by offline search engine, where it is possible to compute statistically connected components, instead online search engine, like databases, given a query w compute $h_{I_1}(w), \dots, h_{I_L}(w)$ and check the vectors in the buckets $h_J(w)$.

   This approach of LSH(Locality-sensitive hashing) finds correct clusters with high probability, compares only very short (sketch) vectors, does not need to know the number of clusters and sorts U short items, with few scans.

2.6 DOCUMENT DUPLICATION

The web is full of duplicated content, and exist only few exact duplicate documents but many cases of near duplicates docs (differ for Last modified date, malicious, spam and so on) so in this section we will analyze how to determine if two document are duplicates.

To determine an exact duplication there are several approaches:

- Obvious (slow) technique, like *checksum* (no worst-case collision probability guarantees) or *MD5* (cryptographically-secure string hashes)

- Karp-Rabin (fast) scheme: it is a *Rolling hash* (split doc in many pieces), it use arithmetic on primes, it is efficient and has other nice properties.
  We consider an m bit string $A = 1a_1 \dots a_{m-1}a_m$ and we choose a prime p in the universe U, such that $2p$ uses few memory-words (hence $U \approx 2^64$), so we define the fingerprints $f(A) = A \mod p$, that has a nice properties that if $B = 1a_2 \dots a_{m-1}a_m a_{m+1}$ we have that

  $$f(B) = [2(A - 2^m - a_1 2^{m-1}) + a_{m+1} + 2^m] \mod p$$

  and the probabilities that we have a false positive is defined as

  $$\begin{cases} P[\text{false hit to } A \text{ and } B \text{ on same window}] & = \text{Probability } p \text{ divides } (A - B) \\ & = \frac{\#div(A-B)}{\#prime(U)} \\ & \approx \frac{(\log(A+B)}{\#prime(U)} \\ & = \frac{m \log U}{U} \end{cases}$$

Now we consider the problem to given a large collection of documents identify the near-duplicate documents and this aspect is important because it has been found that in 199730% of web-pages was near-duplicates.

A common approach used is the *shingling*, where from docs we obtain sets of shingles that are a dissection of document in $q-$gram (shingles), with usually $4 \le q \le 8$, and the near-duplicate document detection problem reduces to set intersection among integers (shingles), but this naive approaches is computationally expensive so we consider now a better approach that use the *Jaccard similarity*, defined as

$$sim(S_A, S_B) = \frac{|A \cap B|}{|A \cup B|}$$

but also this approach has the problem that we have to compute the similarity of both A and B, so a solution consist to use the min hashing, where we define a permutation function, we apply them and we take the min element in the permutation of A and B.
An heuristic consist to use 200 random permutation or pick the 200 smallest item using only a single permutation, so we obtain a 200 vector per set which we compare using Hamming distance, but of course we can choose how many k element to consider to estimate the Jaccard similarity; the importance of this approximation yields to this important proposition, that will stated and proved

Prop 2.1. $P(\alpha, \beta)$ is exactly the Jaccard similarity $JS(S_A, S_B)$

Dimostrazione. We give the proof in a slightly more general setting: consider a family of sets whose elements are drawn from a common universe and view the sets as columns of a matrix A, with one row for each element in the universe.
The element $a_{ij} = 1$ if element i is present in the set S_j that the jth column represents and let Π be a random permutation of the rows of A; denote by $\Pi(S_j)$ the column that results from applying Π to the jth column and finally, let x_{Π_j} be the index of the

first row in which the column $\Pi(S_j)$ has a 1, now we prove that $P(\alpha, \beta) = JS(S_A, S_B)$ and if we can prove this, the theorem follows.

Consider two columns j_1, j_2 and the ordered pairs of entries of S_{j_1} and S_{j_2} partition the rows into four types, as we can note in figure **??**, denote by C_{00} the number of rows with 0's in both columns, C_{01} the second, C_{10} the third and C_{11} the fourth, then

$$JS(S_{j_1}, S_{j_2}) = \frac{C_{11}}{C_{01} + C_{10} + C_{11}}$$

To complete the proof by showing that the right-hand side equals to $P(x^{\Pi}_{j_1} = x^{\Pi}_{j_2})$, consider scanning columns j_1, j_2 in increasing row index until the first non-zero entry is found in either column, and because Π is a random permutation, the probability that this smallest row has a 1 in both columns is exactly the right-hand side of Equation. □

Another important similarity function is the *cosine Similarity* defined as

$$\cos\alpha = \frac{p * q}{||p||||q||}$$

The computation of the scalar product between p and q is huge, so to solve this problem we use an approximation that consist to construct a random hyperplane r of dimension d and unit norm, so we define a sketch vector $h_r(p) = sign(p * r) = \pm 1$ and in a similar way we define a sketch vector for q so we have this proposition

Prop 2.2. $P(h_r(p) = h_r(q)) = 1 - \frac{\alpha}{\pi}$ and also $P(h_r(p) \neq h_r(q)) =$ hyperplane falls between p and q

With this probabilistic interpretation we have $O(nk)$ time for each scalar product with an overall time of $O(D * n * k)$, instead if we use $sort(D)$ (I/O efficient) we have $O(D^2)$ that do not scale very well.

3 INDEX CONSTRUCTION

In these chapter we will analyze how we construct inverted index and storage in memory/disk, we will consider *SPIMI* (Single-pass in-memory indexing) and *Multi-way Merge-sort*, but also distributional caching.

3.1 SPIMI APPROACH

SPIMI is an approach to storage inverted index using a single pass in memory and has two key ideas:

1. Generate separate dictionaries for each block of docs (no need for term map to termID)

2. Accumulate postings in lists as they occur in each block of docs, in internal memory.

With this approach we generate an inverted index for each block, where also compression is possible and in figure 26 there is the pseudocode of SPIMI approach.

There are some problems with this approach, like we decide always to double dimension of block when is full, also we assign TermID, create pairs $< termID, docID >$ and sort pairs by TermID.

Given a query q we require N/m queries whose results have to be combinated, where N is the number of items and m is the dimension of main memory.

To sort n inverted index we accumulate terms, and in a certain time we will encounter again, so we assume $|\text{dictionary}| \leq M$ and we have the following steps:

1. Scanning and build dictionary of distinct tokens.

2. Sort the tokens, assign lexicografic IDs such that $T_1 \leq_L T_2$ **to** $ID(T_1) \leq ID(T_2)$

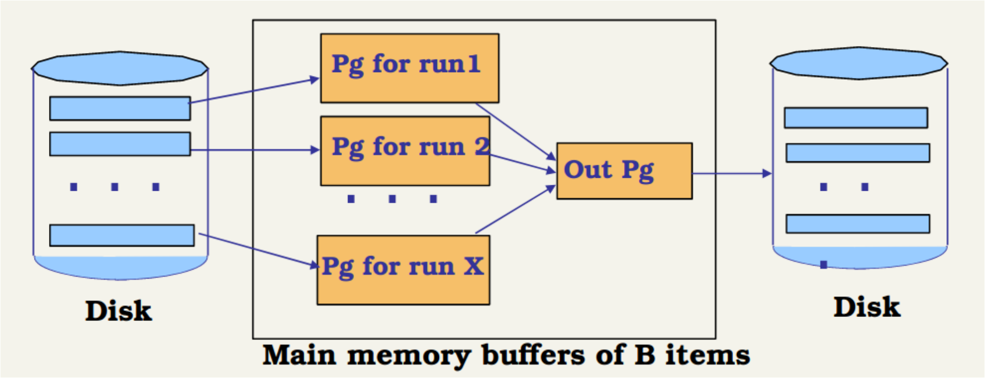
3. Scan documents and we create pair $< termID, docID >$.

4. Sort by first component and then to second component and since the order of terms are lexicografically sort of first component is this correct.

Figura 26: SPIMI pseudocode

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token $\leftarrow$ next(token_stream)
    if term(token) $\notin$ dictionary
        then postings_list = ADDTODICTIONARY(dictionary, term(token))
        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms $\leftarrow$ SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

Figura 27: Multiway Merge-sort merging



5. Decode termID, such that scanning pair in substituting termID with terms, by using the internal memory dictionary.

This sorting is stable, a properties that means that we keep reciprocal order of equal items.

3.2 MULTI-WAY MERGE SORT

We will now consider the multi-way merge-sort, called also *BSBI* (Blocked sort-based Indexing), that consist that we map term to termID to be kept in memory for construction the pairs and needs two passes, unless we use hashing and thus with some probability of collision.

This merge-sort consist in particular in two phases:

1. Scan input and divide on block of size M, where we have for each block $2M/B$ I/Os where B is the size of block.
   The total cost of this step is $\frac{2M}{B} * \frac{n}{M} = O(\frac{n}{B})$ I/Os.
2. Merge $X = M/B - 1$ runs, given a $\log_X N/M$ passes, as we can see in figure 27

   We have to compare k minimum comparison to find the smallest and write in output and in case output is full we have to flush on memory harddisk/SSD, so we have $O(\frac{X}{B})$ I/Os to find a list of X items in k sorted rows, and we have $\log_k \frac{n}{M}$ levels, yields to a total cost of $O(\frac{n}{B} \log_k \frac{n}{M})$.

3.3 DISTRIBUTED INDEXING

For web-scale indexing we must use a distributing computing cluster of inverted index, and since 2004 Google use *Map Reduce*, that we will introduce later, but we now introduce the distributed indexing.

We maintain a master machine directing the indexing job, considered "safe" and we break up indexing into sets of (parallel) tasks, where master machine assigns tasks to idle machines and other machines can play many roles during the computation. We will use two sets of parallel tasks, Parsers and Inverters, so we break the document collection in two ways:

TERM-BASED PARTITION: one machine handles a subrange of terms, as we can note in figure 28.

DOC-BASED PARTITION: one machine handles a subrange of documents, as we can note in figure 29.

Figura 28: Term-based Distributed indexing

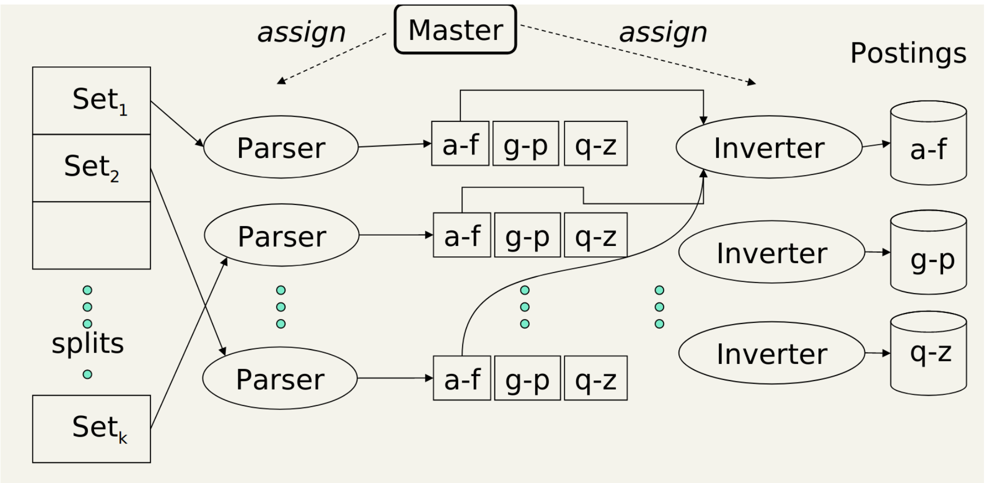
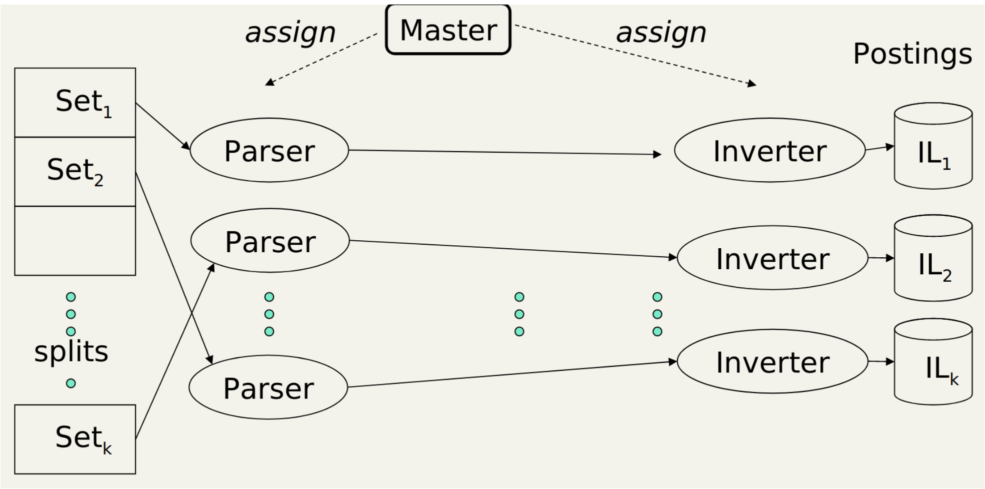


Figura 29: Doc-based Distributed indexing



MapReduce is a robust and conceptually simple framework for distributed computing, without having to write code for the distribution part and Google indexing system (ca. 2004) consists of a number of phases, each implemented in MapReduce.

Up to now, we have assumed static collections, now more frequently occurs that documents come in over time and documents are deleted and modified, so this induces postings updates for terms already in dictionary and new terms added/deleted to/from dictionary.

A first approach is to maintain "big" main index, and new docs go into "small" auxiliary index, where we search across both, and merge the results.
In case of deletions we use an invalidation bit-vector for deleted docs, so we filter search results by the invalidation bit-vector and periodically, we re-index into one main index.

The problem is this approach is that has poor performance: merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list and merge is the same as a simple append [new docIDs are greater] but this needs a lot of files so is inefficient for O/S, anyway in reality we use a scheme somewhere in between, like split very large postings lists, collect postings lists of length 1 in one file and so on.

We introduce now *Logarithmic merge*, where we maintain a series of indexes, each twice as large as the previous one ($M, 2M, 2^2M, 2^3M, \ldots 2^iM$) and we keep a small index Z in memory of size M and we store $I_0, I_1, I_2, \ldots$ on disk and if Z gets full, we write to disk as I_0 or merge with I_0 (if Io already exists).
Either write $Z + I0$ to disk as I_1 (if no I1) or merge with I_1 to form I_2, and so on.

Some analysis, with C =total collection size) we have that auxiliary and main index has that each text participates to at most (C/M) mergings because we have 1 merge of the two indexes (small and large) every M-size document insertions, instead in logarithmic merge each text participates to no more than $\log(C/M)$ mergings because at each merge the text moves to a next index and they are at most $\log(C/M)$.

Most search engines now support dynamic indexing (news items, blogs, new topical web pages), but (sometimes/typically) they also periodically reconstruct the index, query processing is then switched to the new index, and the old index is then deleted.

4 COMPRESSION

In this chapter we will consider how to compress documents, that deal the problem to reduce the amount of data that are travel across network.

Snappy is a compression/decompression library that implement it, usable with several language and Google has implement recently *Brotli*, a new compression algorithm for internet.

4.1 LZ77 COMPRESSION METHOD

LZ77 is a compression method, where given a input in the form "past_knowledge | string_to_comprex" so LZ77 start from string_to_comprex and using past knowledge we encode the document with triples of form $< dist, len, next - char >$, that represent the pattern that was founded in previous string and we advance in string by $len + 1$.
Usually it used a buffer "window" used to find repeted occurencies to compress and has to be the same for encoding and decoding.

The decoding operation do the inverse process, so decoder keeps the same dictionary window as encoder and finds substring $< dist, len, next - char >$ in previously decoded text and insert a copy of it.

Esempio 1. Given the document aacaacab | caaaaaaac we compress the document as following

$$< 6, 3, a >, < 3, 4, c >$$

4.2 COMPRESSION AND NETWORKING

Compression is also important in networking, because it helps to make able the sender and receiver to share more and more data, to reduce battery usage and so on.

There are 2 standard techniques used to achieve these results:

CACHING: we want to avoid to send the same object again and it only works if objects are unchanged.

COMPRESSION: remove redundancy in trasmitted data, so avoid repeated substring in trasmitted data and can be extended with history of past trasmission.

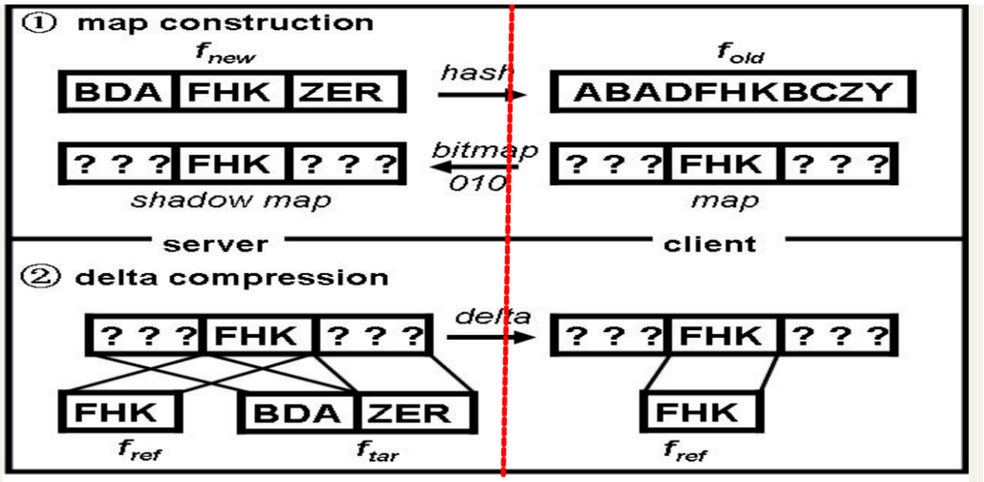
These two standard techniques used two types of situation can happen:

- Common knowledge between sender and receiver, and it used with unstructured file using *Delta Compression* (de/compress f given f').
- Partial knowledge between sender and receiver, where in unstructured data it is used *File Synchronization* and in record based data we use *set reconciliation*.

4.3 Z-DELTA COMPRESSION

We have two files f_known (known to both parties) and f_new (is known only to the sender) and the goal is to compute a file f_d of minimum size such that f new can be derived by the receiver from f_known and f_d; it assume that block moves and copies are allowed, LZ77 decomprension scheme provides an efficient, optimal

Figura 30: Zsync Computation steps



solution, so we only compress f_new based on f_known and we decompress f_d using f_known .

An example of Z-delta compression importance comes from Dual proxy architecture: pair of proxies (client cache + proxy) located on each side of the slow link use a proprietary protocol to increase performance and we use zdelta to reduce traffic: so we restricted the number of pages we have to resend it.

We wish also to compress a group of files F useful on a dynamic collection of web pages, back-ups, and so on.
To do it we apply pairwise zdelta: find a good reference for each $f \in F$, we reduce to the Min Branching problem on DAGs and we build a complete weighted graph G_F , where nodes are files and weights are the zdelta-size.

We insert a dummy node connected to all, and weights are gzip-coding so we compute the directed spanning tree of min tot cost, covering G's nodes.

Constructing G is very costly, n^2 edge calculations, so we wish to exploit some pruning approach, like *shrinkling* to detect similar documents given $O(n \log n)$ using min-hashing.

4.4 FILE SYNCHRONIZATION

In File Synchronization we have a Client request to update an old file, who sends a sketch of the old one to the server.
Server has these new file but does not know the old file, so it sends an update of f_old given its received sketch and f_new

We will briefly analyze two different approach:

RSYNC: file synch tool, distributed with Linux and it is simple, widely used and use single roundtrip.
It uses 4-byte rolling hash +2-byte MD5, gzip for literals, choice of block size problematic (default: $max\{700, \sqrt{n}\}$ bytes) and also there is a high load on the server.

ZSYNC: minimze server load, where computation are done in Client, and has three communication steps visible in figure 30.

The hashes of the blocks can be precalculated and stored in .zsync file and Server should be not overloaded, so it is better suited to distribute multiple-files through network, given one .zsync.

In our example we have a bitmap of 3 bits because of #blocks in f_new are only three.

5 DOCUMENT PREPROCESSING

In previous chapters we have analyzed how to construct query, how to compress result, now we will explain how to process document that has to be indexed.

Given documents to parse them we have to discover format, language and character set used and each of them is a classification problem, but usually we use some heuristics.

After we have parsed document we tokenize them and we have the following definitions:

Def. Token is an istance of a sequence of characters

Each such token is now a candidate for an index entry, after further processing, that we will now present.

Sometimes there are some issues like "San Francisco" should be 1 or 2 tokens, or also how to deal with hypens, apostrophe and usually this is done based with word and is language dependent.

Another preprocessing step is to remove *Stop words*, most common words in document, and the intuition behind is that they have little semantic content (the, a, and, to, be) and there are a lot of them ($\sim$ 30% of postings for top 30 words.
Nowadays good compression techniques has reduced the space necessary to include also stopwords and with good query optimations it is required only a small more time also to consider it, so usually the stop words are not deleted.

We need also to "normalize" terms in indexed text and query words into the same form so we want to match U.S.A. and USA, so we most commonly implicitly define *equivalence classes* of terms; another preprocessing is to reduce all letters to lower case (there is an exception to consider upper case in midsentence but often best to lowercase everythin, since users will use lowercase regardless of 'correct' capitalization).

Thesauri is how to handle synonyms and homonyms and there are two ways to consider it: by hand-constructed equivalence classes we can rewrite to form equivalence-class terms and when the document contains automobile, index it under car-automobile (and vice-versa) or we can expand a query, so when the query contains automobile, look under car as well.

Stemming is the process to reduce terms to their "roots" before indexing and suggest crude affix chopping (it is language dependent and for example automate(s), automatic, automation all reduced to automat).

Lemmatization reduce inflectional/variant forms to base form (so for example am, are,is became be)and Lemmatization implies doing "proper" reduction to dictionary headword form.

Many of the above features embody transformations that are language-specific and often, application-specific and these are "plug-in" addenda to indexing.

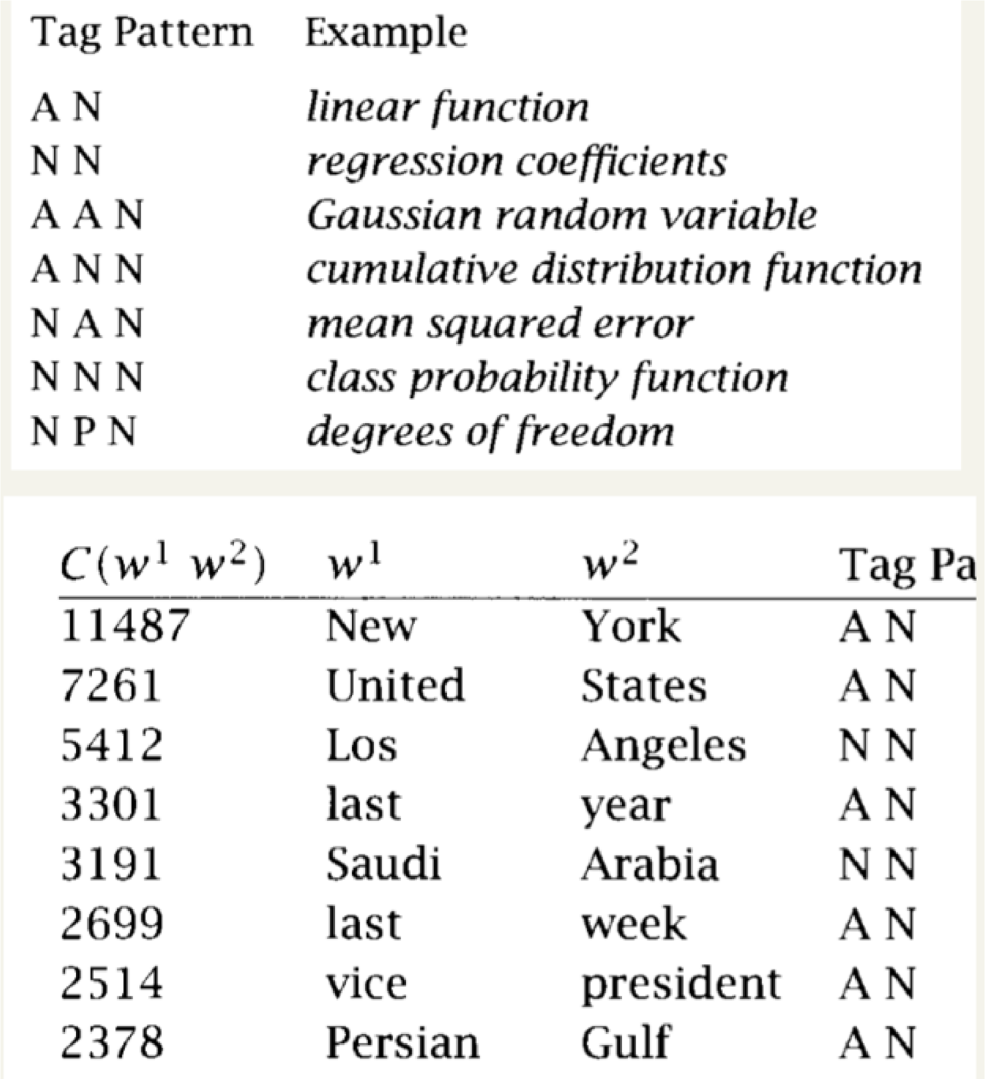
5.1 STATISTICAL PROPERTIES OF TEXT

Tokens are not distributed uniformly, they follow the so called *Zipf Law*, so few tokens are very frequent, a middle sized set has medium frequency and many are rare; the first 100 tokens sum up to 50% of the text, and many of them are stopwords.

k-th most frequent token has frequency $f(k)$ approximately $1/k$, equivalently, the product of the frequency $f(k)$ of a token and its rank k is a constant

$$k * f(k) = c$$

Figura 31: Frequency and POS tagging approach



Tag Pattern	Example
A N	*linear function*
N N	*regression coefficients*
A A N	*Gaussian random variable*
A N N	*cumulative distribution function*
N A N	*mean squared error*
N N N	*class probability function*
N P N	*degrees of freedom*

$C(w^1\ w^2)$	w^1	w^2	Tag Pa
11487	New	York	A N
7261	United	States	A N
5412	Los	Angeles	N N
3301	last	year	A N
3191	Saudi	Arabia	N N
2699	last	week	A N
2514	vice	president	A N
2378	Persian	Gulf	A N

The number of distinct tokens grows, following with the so called *Heaps Law*

$$T = kn^b$$

where b is typically 0.5 and n is the total number of tokens.
The average token length grows as $\Theta(\log n)$ and interesting words are the ones with medium frequency (Luhn).

5.2 KEYWORD EXTRACTION

To extract keyword that should be consider as a single token we have several approaches that we can consider:

1. Use frequency and POS (Part of Speech) tagging to obtain keywords, as can be viewed in figure 31.

2. Often the words are not adjacent to each other and to find keyword we compute the mean and the variance of the distance, by restricting within a window, as it is possible to note in figure 32 and 33. If s is large, the collocation is not interesting, instead if $d > 0$ and s very small we have interesting new keyword.

Figura 32: Mean and Variance between Strong and some words

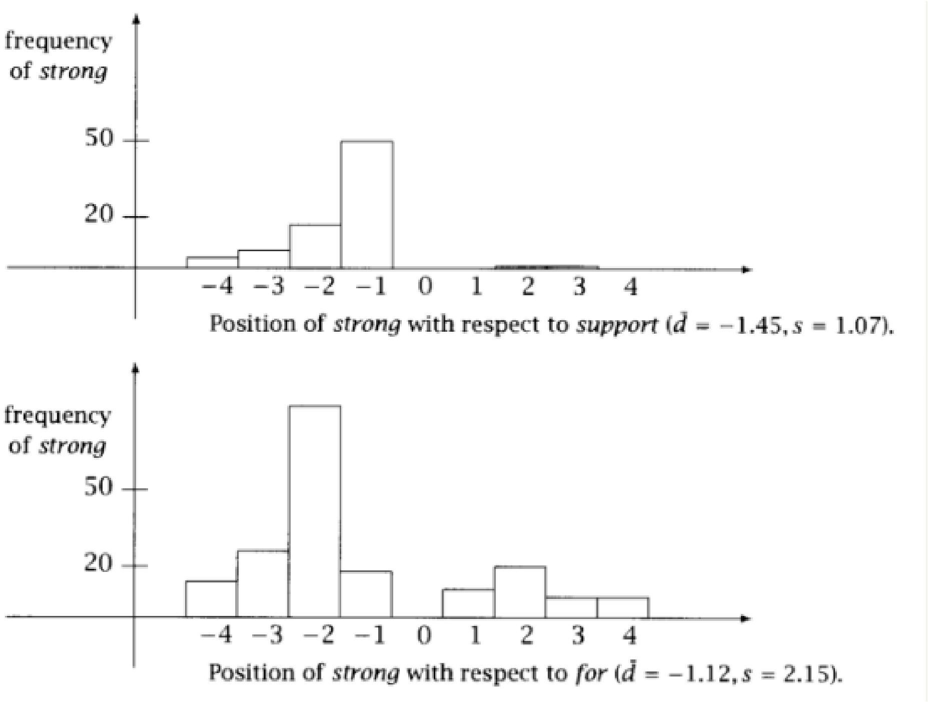
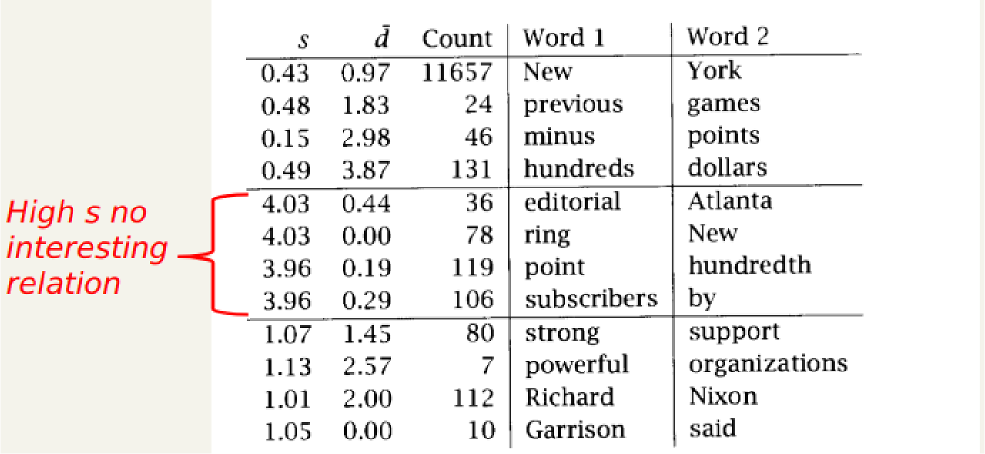


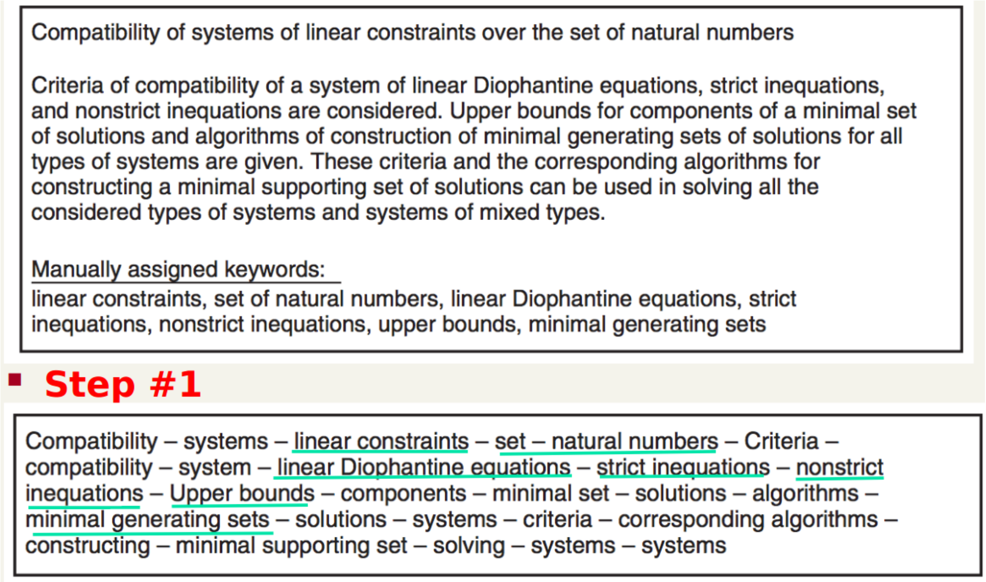
Figura 33: Example of Mean and Variance distance



s	$\bar{d}$	Count	Word 1	Word 2
0.43	0.97	11657	New	York
0.48	1.83	24	previous	games
0.15	2.98	46	minus	points
0.49	3.87	131	hundreds	dollars
4.03	0.44	36	editorial	Atlanta
4.03	0.00	78	ring	New
3.96	0.19	119	point	hundredth
3.96	0.29	106	subscribers	by
1.07	1.45	80	strong	support
1.13	2.57	7	powerful	organizations
1.01	2.00	112	Richard	Nixon
1.05	0.00	10	Garrison	said

High s no interesting relation

Figura 34: Extraction of Candidate keywords using RAKE



Compatibility of systems of linear constraints over the set of natural numbers

Criteria of compatibility of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered. Upper bounds for components of a minimal set of solutions and algorithms of construction of minimal generating sets of solutions for all types of systems are given. These criteria and the corresponding algorithms for constructing a minimal supporting set of solutions can be used in solving all the considered types of systems and systems of mixed types.

Manually assigned keywords:
linear constraints, set of natural numbers, linear Diophantine equations, strict inequations, nonstrict inequations, upper bounds, minimal generating sets

- **Step #1**

Compatibility – systems – linear constraints – set – natural numbers – Criteria – compatibility – system – linear Diophantine equations – strict inequations – nonstrict inequations – Upper bounds – components – minimal set – solutions – algorithms – minimal generating sets – solutions – systems – criteria – corresponding algorithms – constructing – minimal supporting set – solving – systems – systems

3. Pearson Chi-Square test statistics that follow a chi-squared distribution arise from an assumption of independent normally distributed data, which is valid in many cases due to the central limit theorem and we use p-value to reject the null hypothesis.

4. *RAKE* (Rapid Automatic Keyword Extraction) works on single (not much long) documents and is easily applicable to new domains, fast and unsupervised. The Key observation of this approach is that keywords frequently contain multiple words but rarely contain punctuation or stop words.

   The input parameters are the following:

   - a set of word delimiters
   - a set of phrase delimiters
   - a list of stop words (or stoplist).

   The RAKE approach has the following 4 step:

   STEP #1: document is split into an array of words by the specified word delimiters and this array is split into sequences of contiguous words at phrase delimiters and then stop word.

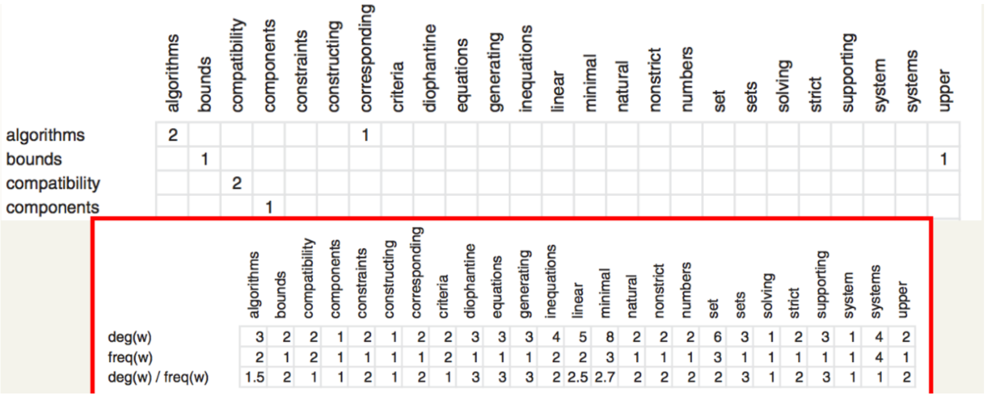
   Words within a sequence are considered a candidate keyword, as we can see in figure 34.

   STEP #2: compute the table of co-occurrences and we use few metrics: $freq(w)$ = total frequency on diagonal and $deg(w)$ sum over row.
   Final score is the sum of $deg(w)/freq(w)$ for the constituting words w of a keyword and in figure 35 is possible to know how to compute the scoring Candidate keywords.

   STEP #3: identifies keywords that contain interior stop words such as axis of evil and looks for pairs of keywords that adjoin one another at least twice in the same document and in the same order.
   The score for the new keyword is the sum of its member keyword scores, as can be viewed in figure 36.

   STEP #4: we select the top one-third of sorted list of scoring obtained in previous steps and in figure **??** is possible to compare results obtained by RAKE and by manual keyword extraction.

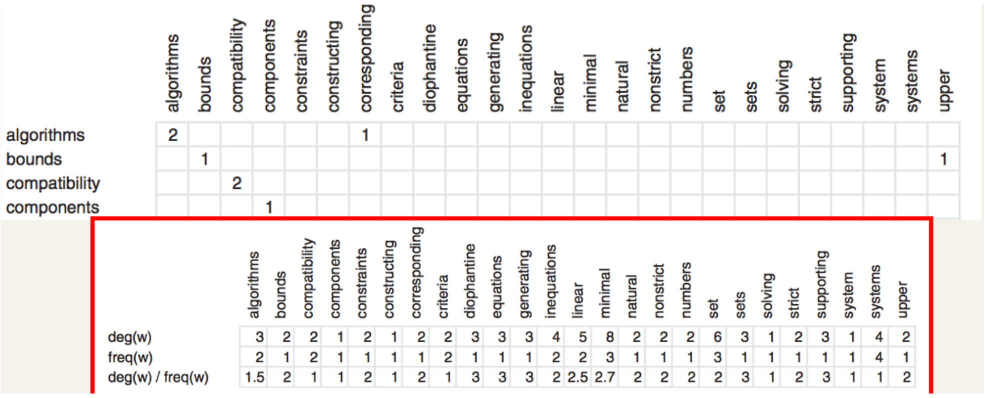
Figura 35: Calculation of Scoring of Candidate keywords



	algorithms	bounds	compatibility	components	constraints	constructing	corresponding	criteria	diophantine	equations	generating	inequations	linear	minimal	natural	nonstrict	numbers	set	sets	solving	strict	supporting	system	systems	upper
algorithms	2						1																		
bounds		1																							1
compatibility			2																						
components				1																					

	algorithms	bounds	compatibility	components	constraints	constructing	corresponding	criteria	diophantine	equations	generating	inequations	linear	minimal	natural	nonstrict	numbers	set	sets	solving	strict	supporting	system	systems	upper
deg(w)	3	2	2	1	2	1	2	2	3	3	3	4	5	8	2	2	2	6	3	1	2	3	1	4	2
freq(w)	2	1	2	1	1	1	1	2	1	1	1	2	2	3	1	1	1	3	1	1	1	1	1	4	1
deg(w) / freq(w)	1.5	2	1	1	2	1	2	1	3	3	3	2	2.5	2.7	2	2	2	2	3	1	2	3	1	1	2

Figura 36: Sorted Scoring of RAKE approach



	algorithms	bounds	compatibility	components	constraints	constructing	corresponding	criteria	diophantine	equations	generating	inequations	linear	minimal	natural	nonstrict	numbers	set	sets	solving	strict	supporting	system	systems	upper
algorithms	2						1																		
bounds		1																							1
compatibility			2																						
components				1																					

	algorithms	bounds	compatibility	components	constraints	constructing	corresponding	criteria	diophantine	equations	generating	inequations	linear	minimal	natural	nonstrict	numbers	set	sets	solving	strict	supporting	system	systems	upper
deg(w)	3	2	2	1	2	1	2	2	3	3	3	4	5	8	2	2	2	6	3	1	2	3	1	4	2
freq(w)	2	1	2	1	1	1	1	2	1	1	1	2	2	3	1	1	1	3	1	1	1	1	1	4	1
deg(w) / freq(w)	1.5	2	1	1	2	1	2	1	3	3	3	2	2.5	2.7	2	2	2	2	3	1	2	3	1	1	2

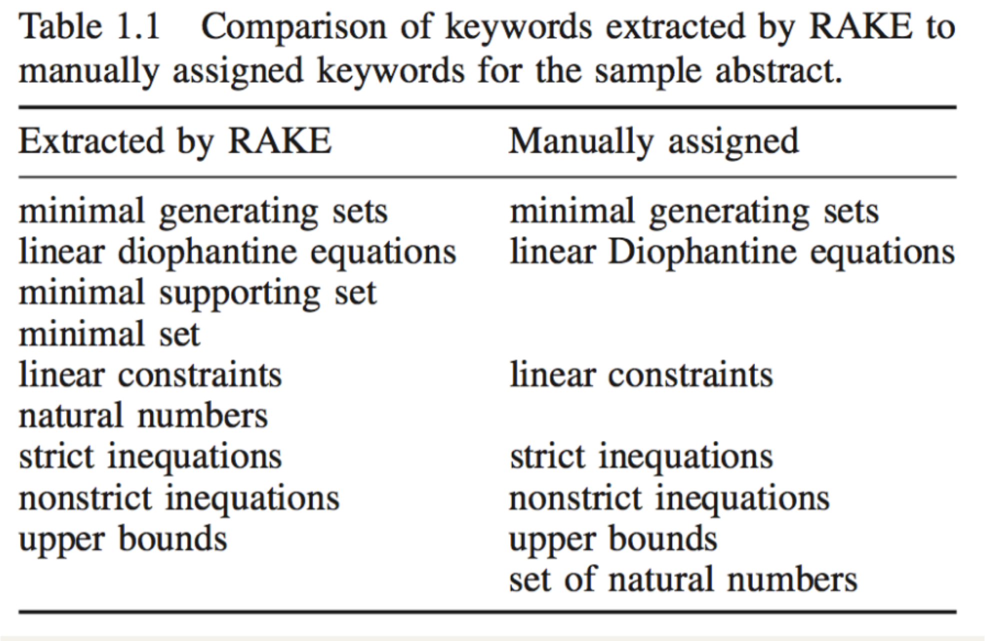


Table 1.1 Comparison of keywords extracted by RAKE to manually assigned keywords for the sample abstract.

Extracted by RAKE	Manually assigned
minimal generating sets	minimal generating sets
linear diophantine equations	linear Diophantine equations
minimal supporting set	
minimal set	
linear constraints	linear constraints
natural numbers	
strict inequations	strict inequations
nonstrict inequations	nonstrict inequations
upper bounds	upper bounds
	set of natural numbers

6 DATA STRUCTURES FOR INVERTED INDEX

In this chapter we will analyze which data structures are used for Inverted Index and a naive approach consist in save in a dictionary but this cause a waste in memory and not helps in retrieve elements quickly.

To improve exact and prefix search are usually used the following data structures:

- Hashing
- Tree
- Trie, also called *prefix tree*, is an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings.
  All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string; keys tend to be associated with leaves, though some inner nodes may correspond to keys of interest and hence, keys are not necessarily associated with every node.

  Solves the prefix problem, but has $O(p)$ time, with many cache misses and from 10 to 60 (or, even more) bytes per node.

To improve our search we exploits 2-level caching indexing, that improve search, typically 1 I/O + in-mem comparison and improve also space requirement, because we use a trie built over a subset of string and front-coding over buckets.
A disadvantage is the trade-off between speed and space, caused by bucket size.

Front-coding is a type of delta encoding compression algorithm whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated and this algorithm is particularly well-suited for compressing sorted data, as we can see in figure 37.

6.1 CORRECTION QUERIES

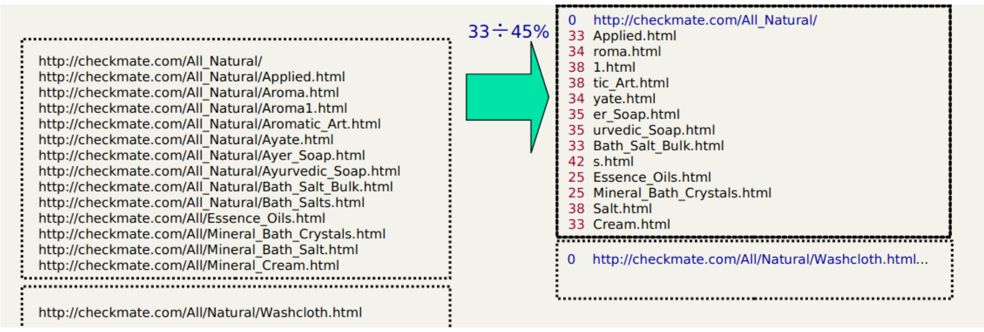
Spell correction has 2 principal uses:

1. Correcting document(s) to be indexed.
2. Correcting queries to retrieve "right" answers.

There are two approaches that can be used:

1. Isolated word: check each word on its own for misspelling.

Figura 37: Example of Front Coding



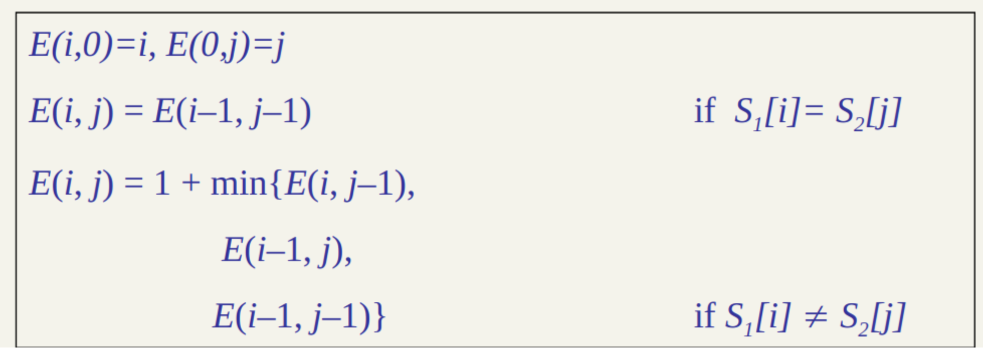
```
http://checkmate.com/All_Natural/
http://checkmate.com/All_Natural/Applied.html
http://checkmate.com/All_Natural/Aroma.html
http://checkmate.com/All_Natural/Aroma1.html
http://checkmate.com/All_Natural/Aromatic_Art.html
http://checkmate.com/All_Natural/Ayate.html
http://checkmate.com/All_Natural/Ayer_Soap.html
http://checkmate.com/All_Natural/Ayurvedic_Soap.html
http://checkmate.com/All_Natural/Bath_Salt_Bulk.html
http://checkmate.com/All_Natural/Bath_Salts.html
http://checkmate.com/All/Essence_Oils.html
http://checkmate.com/All/Mineral_Bath_Crystals.html
http://checkmate.com/All/Mineral_Bath_Salt.html
http://checkmate.com/All/Mineral_Cream.html

http://checkmate.com/All/Natural/Washcloth.html
```

```
0  http://checkmate.com/All_Natural/
33 Applied.html
34 roma.html
38 1.html
38 tic_Art.html
34 yate.html
35 er_Soap.html
35 urvedic_Soap.html
33 Bath_Salt_Bulk.html
42 s.html
25 Essence_Oils.html
25 Mineral_Bath_Crystals.html
38 Salt.html
33 Cream.html

0  http://checkmate.com/All/Natural/Washcloth.html...
```

Figura 38: Equation for computing Edit Distance



$$E(i,0)=i,\ E(0,j)=j$$

$$E(i,j) = E(i-1, j-1) \qquad \text{if } S_1[i]= S_2[j]$$

$$\begin{aligned} E(i,j) = 1 + \min\{&E(i, j-1),\\ &E(i-1, j),\\ &E(i-1, j-1)\} \qquad \text{if } S_1[i] \neq S_2[j] \end{aligned}$$

2. Context-sensitive is more effective and look at surrounding words.

To correct isolated word there is a lexicon from which the correct spellings come and two basic choices for this are a standard lexicon such as Webster's English Dictionary or an "industry-specific" lexicon, that is specific for a field and where we can use mining algorithms to derive possible corrections.

Isolated word correction consist that given a lexicon and a character sequence Q, return the words in the lexicon closest to Q; for estabilish what's closest we will study several measures (we will study Edit distance, Weighted edit distance and n-gram overlap).

Edit Distance is generally found by dynamic programming and consist given two strings $S1$ and $S2$, to find the minimum number of operations to convert one to the other (Operations are typically character-level insert, delete, replace, with possibility of also transposition.

In figure 38 is possible to find how is computed the edit distance and we compute the table of distance from bottom to top, using equation described in the figure.

We introduce now *Weighted edit distance*, where the weight of an operation depends on the character(s) involved and meant to capture keyboard errors, as for example m is more likely to be mis-typed as n than as q.
Therefore, replacing m by n is a smaller cost than by q and requires weighted matrix as input.

We create two dictionaries $D_1 = \{strings\}$ and $D_2 = \{strings of D1 with one deletion\}$; for a query we have to do 1 search in D_1 in perfect match, 1 query in D_2 to find 1-char less, p queries to find P with 1-char less from D_2 to D_1 and in the end p queries to find substitution in D_2 from P with 1-char less.

We need $2p+2$ hash computations for P and the positive aspects are that is CPU efficient, no cache misses for computing P's hashes, but $O(p)$ cache misses to search in D_1 and D_2, instead negative aspects are large space because of the many strings in D_2 which must be stored to search in the hash table of D2, unless we avoid collision and the presence of false matches.

A better approach consist to use overlap distance, where we use the k-gram index contains for every k-gram all terms including that k-gram and we append $k-1$ symbol $ at the front of each string, in order to generate a number L of k-grams for a string of length L

We select terms by threshold on matching k-grams and if the term is L chars long (it consists of Lk-grams) and if E is the number of allowed errors ($E*k$ of the k-grams of Q might be different from term's ones because of the E errors) and so at least $L\check{}E*k$ of the k-grams in Q must match a dictionary term to be a candidate answer and if ED is required, post-filter results with dynamic programming.

We enumerate multiple alternatives and then need to figure out which to present to the user for "Did you mean?" and we use heuristics: the alternative hitting most docs and query log analysis + tweaking (done for especially popular, topical queries), anyway spell-correction is computationally expensive and is run only on queries that matched few docs.

We introduce now how to deal with *wildcard queries* and to deal we use *permuterm index*, where we have the following possible queries:

- X lookup on X$
- X* lookup on $X*
- *X lookup on X$*
- *X* lookup on X*
- X*Y lookup on Y$X*

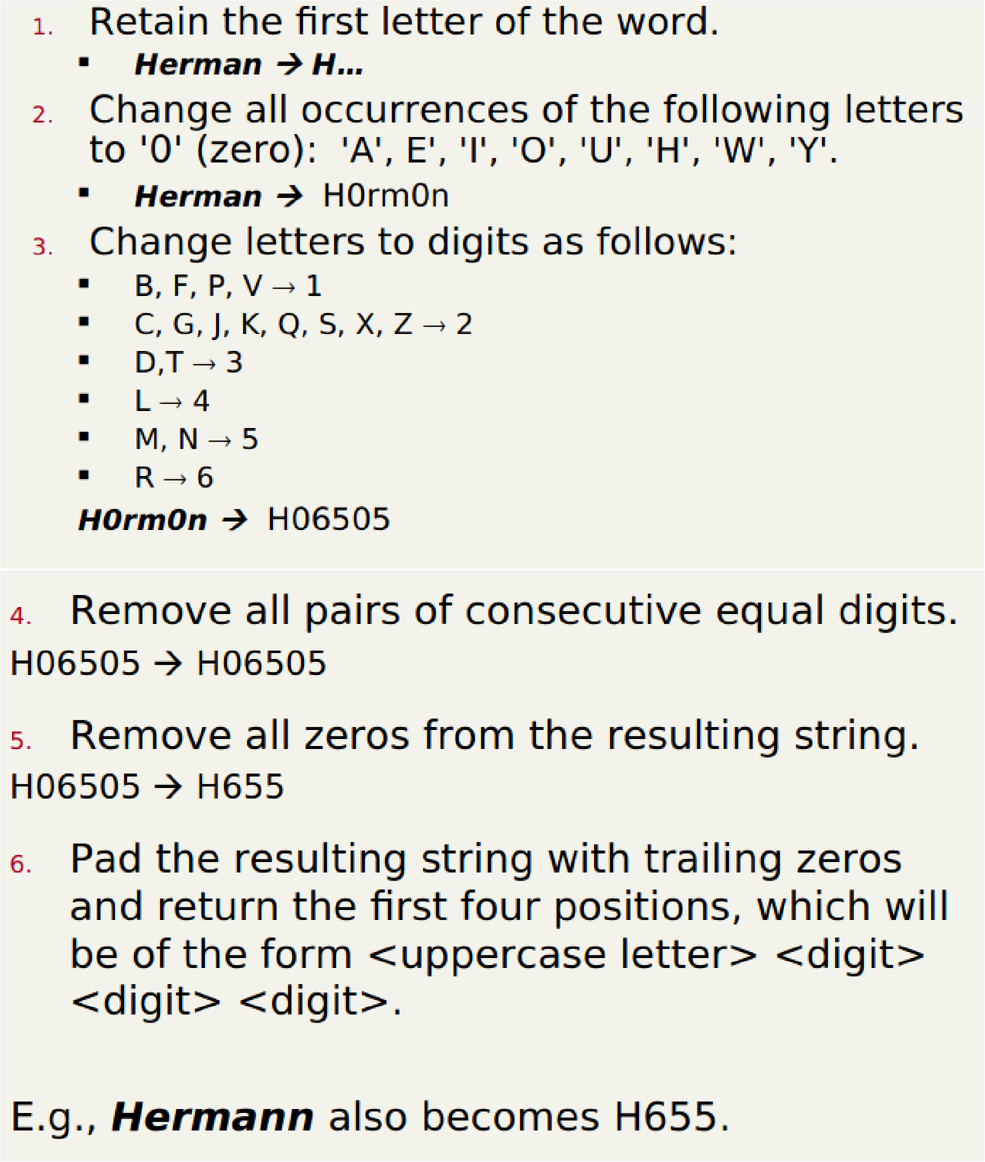
The permuterm query processing consist to rotate query wild-card to the right so P*Q become Q$P* so now we use prefix-search data structure and a problem of Permuterm is that has $\approx 4x$ lexicon size (an empirical observation for English).

Soundex is a class of heuristics to expand a query into phonetic equivalents, it is language specific used mainly for names and was invented for the U.S. census in 1918.

We introduce now the original algorithm that consist to turn every token to be indexed into a reduced form consisting of 4 chars and do the same with query terms, so we build and search an index on the reduced forms (in figure 39 are indicated all steps of basic algorithm).

Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, and so on) but is not very useful for information retrieval; is okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities, so other algorithms for phonetic matching perform much better in the context of IR.

Figura 39: Soundex basic Algorithm



1. Retain the first letter of the word.
   - ***Herman → H…***
2. Change all occurrences of the following letters to '0' (zero): 'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
   - ***Herman →*** H0rm0n
3. Change letters to digits as follows:
   - B, F, P, V → 1
   - C, G, J, K, Q, S, X, Z → 2
   - D,T → 3
   - L → 4
   - M, N → 5
   - R → 6

   H0rm0n → H06505

4. Remove all pairs of consecutive equal digits.

H06505 → H06505

5. Remove all zeros from the resulting string.

H06505 → H655

6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Hermann*** also becomes H655.