Seminar report

Set Intersection Problem

Marco Bellò

April 24, 2025

Faculty of Science
University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	

Tekijä — Författare — Author

Marco Bellò

Työn nimi — Arbetets titel — Title

Set Intersection Problem

Ohjaajat — Handledare — Supervisors

Prof. J. Kärkkäinen, Prof. V. Mäkinen, Dr. D. Diaz

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminar report	April 24, 2025	25 pages

Tiivistelmä — Referat — Abstract

ACM Computing Classification System (CCS)

General and reference → Document types → Surveys and overviews

Software and its engineering → Software creation and management → Designing software

Applied computing → Document management and text processing → Document searching

Avainsanat — Nyckelord — Keywords

algorithms, data structures, set intersection, inverted index

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Seminar in Algorithm Engineering

Contents

1 Introduction

The act of *searching* has become so deeply ingrained in the modern society that we tend to take it for granted, not only assuming it normal to have immediate and easily accessible information on the tip of our thumbs, but expecting it: a study from 2004 showed that users were not willing to wait more than ten seconds for a page to load (Nah, 2004). Fast forward twenty years and nowadays even a couple seconds holdup would be unacceptable, thus query retrieval needs to be fast. Blazingly fast in fact, since we need to account for all the delays typical of a gargantuan structure as big as the modern web, and, as the reader probably knows, it is *not* a good idea to rely on memory's performances increasing over time: the smart way to tackle this problem is via research and development of efficient algorithms, and exactly which type should be self-evident from the title of this document. The problem of the set intersection constitutes the backbone of every query resolver in a (web) search engine, since every word in a query is interpreted as a collection of documents' IDs which contains it.
In this survey-style paper we will first explain what searching (i.e., querying) entails, show how a document (e.g., a web page) can be transformed into word tokens which are then further processed into inverted indexes, and, finally, we will see a collection of algorithms that concern themselves with intersecting sets, meaning finding common elements between two or more comparable collections.

1.1 How Do We Search?

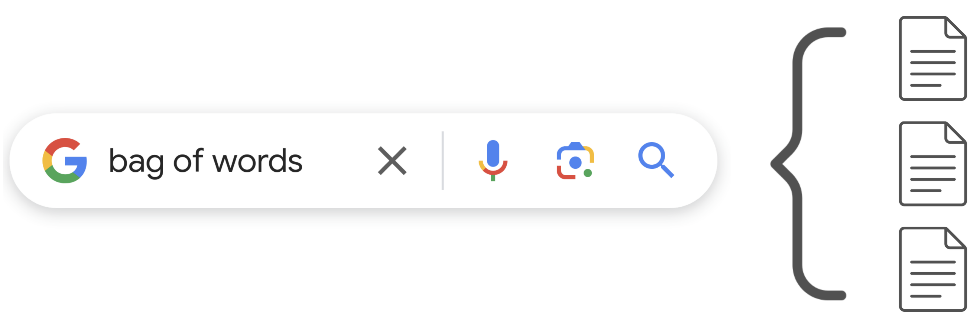

Figure 1.1: From a bag of words to a set of documents

Generally speaking, a query is called a *bag of words*, and finding its result means computing which documents contain all word tokens that are being searched for [1.1]. Let's make and example: word `abiura` is contained in documents number `[31, 42, 127]`, while word `bitonto` is contained in documents number `[20, 42, 72]`.
Thus `query = (abiura,bitonto)` will return the result `42`.

Dictionary	Posting List (ASC)	Relevance
abaco	1, 7, 136	0.6, 0.3, 0.8
abiura	31, 42, 127	0.12, 0.5, 0.77
bitonto	20, 42, 72	0.8, 0.1, 0.03

Figure 1.2: Table of word tokens

Both The example above [1.2] and all the algorithms we will see in this survey consider the problem of searching as the problem of complete intersection, but modern search engine (e.g., Google) leverage input relevance and filter unneeded outputs to obtain faster and better results. Unfortunately finding information about how they do it is near impossible, since everything is covered by trade secret.
Let's now see what inverted indexes are and how we can obtain them starting from a document corpus.

2 Inverted Indexes

Most of the information present in this chapter is thanks to Mahapatra and Biswas "Inverted indexes: Types and techniques" (Mahapatra and Biswas, 2011).

What we will need for the algorithms presented in the rest of this documents are inverted indexes (also called posting lists). To get them we first need to process documents into lists of words (called *word tokens*), then for each token compute a list of `IDs` that refer to the documents which contain that specific token. Let's see each step in order.

2.1 Document Pre-Processing

Documents go through a series of processing steps before being indexed: they get converted into token in the lexing phase, which are then possibly normalized, stemmed or even pruned (removed) entirely.

2.1.1 Lexing

The process of transforming a document into a list of tokens, each of which is a single word, si called *lexing* [2.1]. There often is a maximum length for a single token, as to prevent unbounded index growth in edge cases, and all input is generally first converted into lower-case to normalize it. All non-punctuation characters are added to the list of tokens one by one, and those that exceed a certain size are often pruned (removed from the corpus). It is not entirely clear how Google and other big companies do this step, and it certainly feels strange to think they employ a simple *brute force*, single scan approach, but as mentioned be-

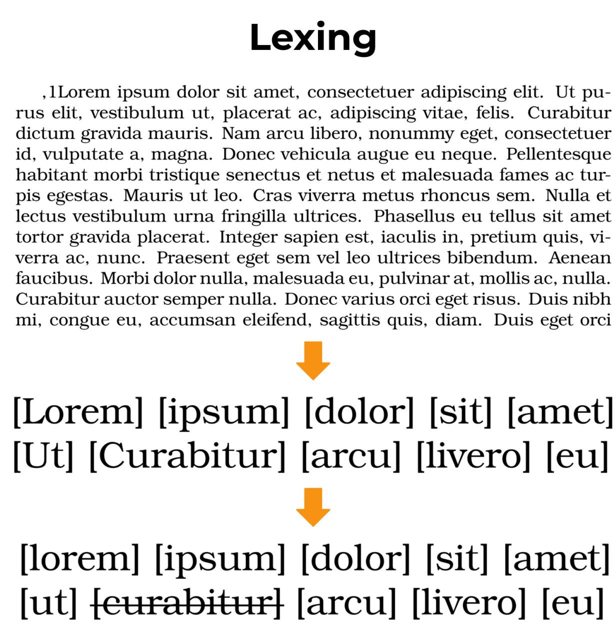

Figure 2.1: Lexing: from text to word tokens

fore it is not easy to find information about
it.
All of the above works only with alphabetic languages, ideographic ones (e.g., Chinese) need specialized search techniques.

2.1.2 Stemming

We can consider this step deprecated, since nowadays memory, especially for things like text and arrays (which inverted index basically are), is cheap and bountiful.
The idea is to find a sort of *root* (stem) of the words, and indexing that instead. To make an example: fishnet, fishery, fishing, fishy, fishmonger, can all be boiled down to their stem *fish* [2.2].

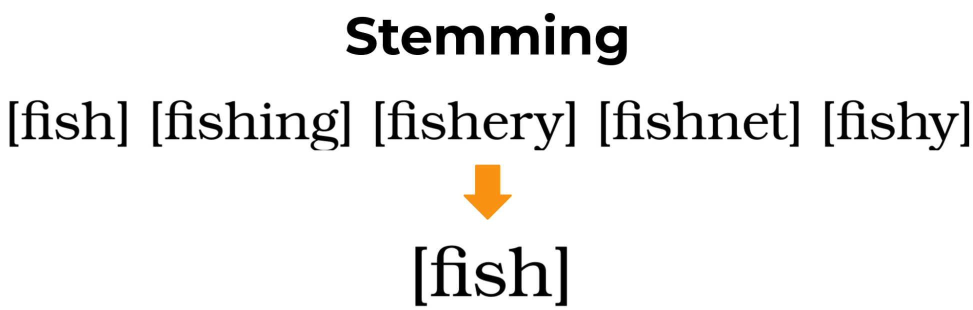

Figure 2.2: Stemming to stem "fish"

In the example above should be clear already that stemming carry some problems: a user searching for "fishnet" is likely not shopping for fishing equipment, thus most modern search engine skip this normalizing step, and most stemming algorithms (most famous of which is Porter's) are complex, full of exceptions and exceptions to the exceptions, while still failing to unite together the correct words. This step basically reduces query precision while providing very little in return.

2.1.3 Stop Words

Stop words are words that work as connectives of sorts, like *and*, *the*, *is*, *of*, *to*, etc.
Their quantity is language dependent (e.g., in English they could be around 500 words) and they are often removed from the corpus which, for normal queries, does not worsen the results while saving space in the index. However in some cases like searching for *to be or not to be* stop words are actually essential, and removing them would make the search

fail.

Thankfully they are so common that if saved as differences between consecutive different values, both their document number and word position lists can be compressed to save space. Because of this, the overhead is not as big as one might think, thus modern search engines (like Google) do not seem to remove them from the index, since doing so put them at a competitive advantage at the expense of a slightly bigger index.

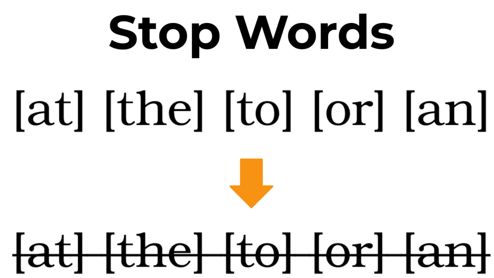

Figure 2.3: Stop words pruning

2.2 Inverted Indexes

Now that we have a set of word tokens, we can start building our inverted indexes (or posting lists): documents are often stored as lists of words, but we invert (hence the name) this concept by storing for each word the list of documents that contain it. There are several variants of this data structure, but at minimum you need to store for each word the list of documents that contain that specific word.
We can change the granularity by adding the frequency of the word in the document, which can be useful for query optimization, or by adding the word position in the document, allowing for in-document queries.
Space used by inverted indexes varies wildly in the range of five to one hundred percent (5-100%) of the total size of the document indexed, and this is because implementations come in many different variations: some store word positions and some don't, some aggressively pre-process documents and some don't, some dynamically update themselves and some don't, some use complex and powerful compression methods and some don't, and so on.

Table [2.1] show some sample documents, while table [2.2] shows some examples of inverted indexes, with different levels of granularity.

ID	Contents
1	The only way not to think about money is to have a great deal of it
2	When I was young I thought that money was the most important thing in life; now that I am old I know that it is.
3	A man is usually more careful of money than he is of his principles.

Table 2.1: Sample document collection

Word	Doc List	Frequency	Positions
a	1, 3	1:1, 3:1	1:(12), 3:(1)
About	1	1:1	1:(7)
am	2	2:1	2:(19)
Careful	3	3:1	3:(6)
deal	1	1:1	1:(16)
great	1	1:1	1:(13)
have	1	1:1	1:(11)
...	...	...	...
money	1, 2, 3	1:2, 2:1, 3:1	1:(8), 2:(8), 3:(9)
more	3	3:1	3:(5)
...	...	...	...
when	2	2:1	2:(1)

Table 2.2: Inverted lists example, most words omitted

3 Intersection Algorithms

In this chapter we are going to see a collection of algorithms to compute the intersection of two **sorted** lists, taken from the chapter six of "Pearl of Algorithm Engineering" by Paolo Ferragina, published by Cambridge University Press (Ferragina, 2023).
We will first look at two of the most commonly used search algorithms, since we cannot intersect without searching.

3.1 Search Algorithms

3.1.1 Binary Search

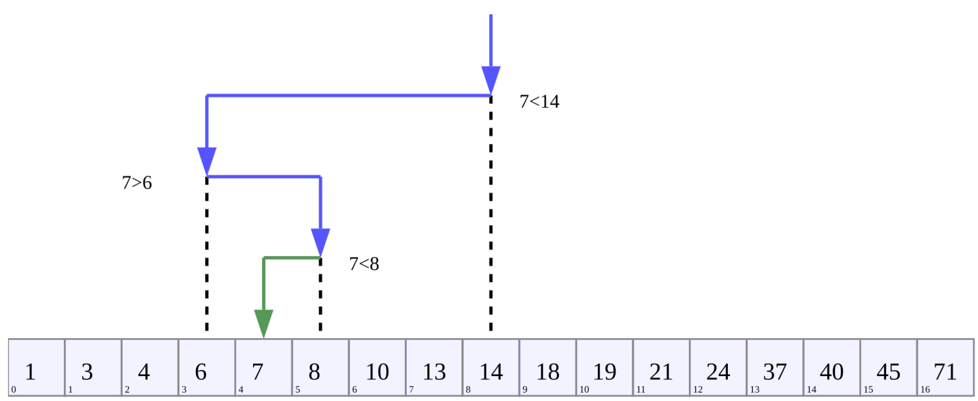

Figure 3.1: Binary search algorithm, source: Wikipedia

Binary search, also known as logarithmic search or binary chop, is a search algorithm that locates the position of a target value within a sorted array: it compares the target to the middle element of the array and, if they are not equal, it eliminates half of the search space by discarding either the left or right half, depending on whether the target value is less than or greater than the middle element. This process is repeated by iteratively searching into the remaining sub-array until the target value is found or the search space is empty. The pseudocode for the algorithm can be seen at *Algorithm* [1].
Binary search runs in logarithmic time in the worst case, doing $O(\log n)$ comparisons,

where n is the number of elements in the array, making it much faster than linear search with large arrays thanks to its scaling.

Algorithm 1

Pseudocode for binary search algorithm

Looking for element key
Let $L = 0$ ▷ First half
Let $R = n - 1$ ▷ Second half
while $L \leq R$ **do**
    $m = \lfloor (L + R)/2 \rfloor$
    if $A[M] < key$ **then**
        let $L = m + 1$
    else if $A[M] > key$ **then**
        $R = m - 1$
    else
        return m ▷ Found
    end if
end while
return False ▷ Not found

3.1.2 Exponential Search

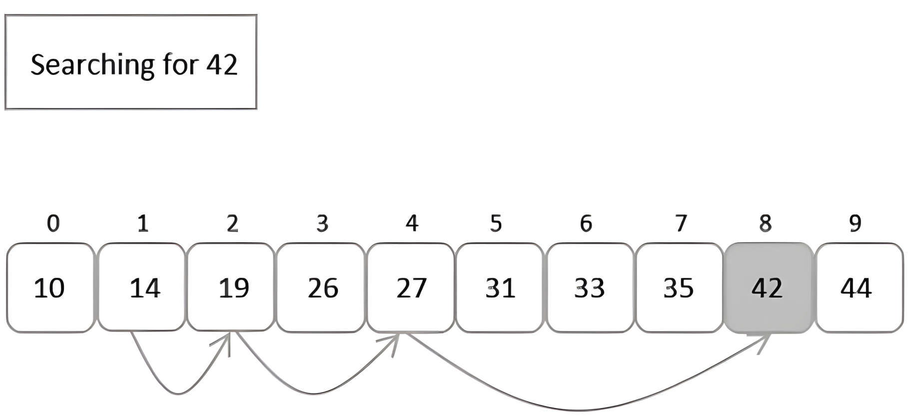

Figure 3.2: Exponential search algorithm, source: Tutorialspoint

Exponential search, also called doubling or galloping search, is an algorithm for searching sorted, unbounded lists: there are numerous implementations, most common being determining a sub-array into which the `key` may resides in and performing a binary search [3.1.1] within its range.
To be more precise: we examine the list in exponentially increasing steps, with a factor of 2^k such that we first look into `list[0]`, then `list[1]`, then `list[2]`, `list[4]`, `list[8]`, following with *16*, *32*, *64*, *128* and so on until we find a value that is greater than the `key`. Once we find it, we perform a binary search between the previous step and the current (or the end of the array): $2^{k-1} \leq key \leq min(2^k, n)$.

The algorithm can be more efficient than binary search, as it runs in $O(\log i)$ time, where i is the index of the element being searched for, which could be half if not less than n. The pseudocode can be seen at *Algorithm* [2].

Algorithm 2

Pseudocode for exponential search algorithm

Looking for element key
Let $i = 0$
Let $k = 0$
while $(key > list[i + 2^k]$ and $i < n)$ **do**
    $i = i + 2^k$ ▷ Gallop to next step
    $k = k + 1$ ▷ Increment exponent
end while
if $i < n$ **then**
    binary_search($list$, key, i, $min(i + 2^k, n)$)
else
    return false ▷ Not found
end if

3.2 Intersection Algorithms

3.2.1 Brute Force

The first idea that would come to mind when thinking about intersecting two lists is to simply iterate through both of them and check for matching elements: this is the *brute*

force approach, which is simple but inefficient.
With a time complexity of $O(m \cdot n)$, assuming lists sizes n and m to be around 10^6, and assuming a modern computer able to do 10^9 operations per second, this algorithm would need ten minutes to compute a 2-words query, which is less than ideal.
The (very short) pseudocode can be seen at *Algorithm* [3].

Algorithm 3

Pseudocode for brute force algorithm

```
for all i = 0 to n − 1 do
    for all j = 0 to m − 1 do
        if A[i] == B[j] then
            add A[i] to result
        end if
    end for
end for
```

3.2.2 Bunny Race

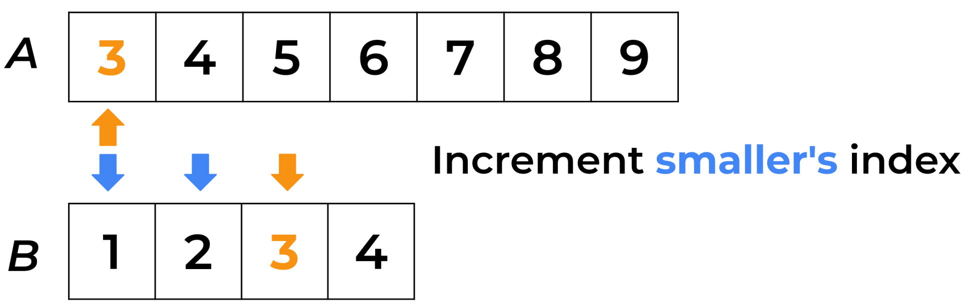

Figure 3.3: Bunny race algorithm

This approach, often called merge-based, is simple, elegant and fast: the main idea is to have two indices pointing at the two list running after each other by comparing elements each time and incrementing the index pointing at the smallest one (or incrementing both if they are equal).
To clarify: lets say we have two lists, A and B, of size n and m respectively. We start with two pointers, i and j, both set to zero.
We compare the elements at these indices, $A[i]$ and $B[j]$: if they are equal, we add the

element to the result and increment both pointers.
If $A[i] < B[j]$, we increment i, while if $A[i] > B[j]$, we increment j.
This process continues until one of the pointers reaches the end of its respective list.
The correctness can be proven inductively, exploiting the following observation: if $A[i] < B[j]$ then $A[i]$ is smaller than all elements following $B[j]$ in B since its ordered, so $A[i] \notin B$. The other case is symmetric.
In regards to time complexity, we just need to note that at each step the algorithm executes one comparison and advances at least one iterator, thus, given that $n = |A|$ and $m = |B|$, the algorithm runs in no more than $O(n + m)$ time.
This time complexity is significantly better than the *brute force* [3] approach, since it can compute a 2-word query in 10^{-3} seconds.
The pseudocode can be seen at *Algorithm* [4].

In the case that $n = \Theta(m)$ this algorithm is optimal, because we need to process the smallest set, thus $\Omega(min(n, m))$ is an obvious lower bound. Moreover, this procedure is also optimal in the disk model since it takes $O\left(\frac{n}{B}\right)$ I/Os.
In the case that $n \ll m$ the classic *binary search* can be helpful since we can design an algorithm that search in A for each elements of B in $O(m \log n)$ time, which is better than $O(n + m)$ when $m = o\left(\frac{n}{\log n}\right)$.

Algorithm 4

Pseudocode for bunny race algorithm

Let $i = 0$
Let $j = 0$
while $i < n$ and $j < m$ **do**
    if $A[i] < B[j]$ **then**
        $i = i + 1$ ▷ Increment first
    else if $A[i] > B[j]$ **then**
        $j = j + 1$ ▷ Increment second
    else
        Add $A[i]$ to result ▷ Found
        $i = i + 1$ ▷ Increment both
        $j = j + 1$
    end if
end while

3.2.3 Divide and Search

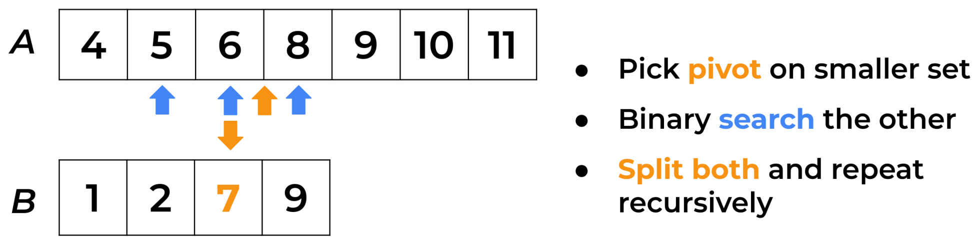

Figure 3.4: Divide and search algorithm

Also called *mutual partitioning*, this approach adopts a classic algorithmic paradigm, namely *divide and conquer*, famously used to design the *quick sort* algorithm (which can be visualized here).
Let us assume $m = |B| \leq n = |A|$. We select the median element of B, $b_{m/2}$, as a *pivot* and search for it in the longer sequence A using the *binary search* [3.1.1] algorithm. Two cases may occur:

i. *pivot* is one of the elements fo the intersection;

ii. $b_{m/2} \notin A$, e.g, $A[j] < b_{m/2} < A[j+1]$.

In both cases the algorithm proceeds *recursively* by calling itself on the two sub-lists in which each list (A and B) has been split according to the *pivot* element, thus computing the following intersections:

- $A\Big[1, j\Big] \cap B\Big[1, \frac{m}{2} - 1\Big]$
- $A\Big[j+1, n\Big] \cap B\Big[\frac{m}{2} + 1, m\Big]$.

In simpler terms: we pick the middle element of the smaller list, search for it in the bigger list, split both lists and repeat the process recursively on the remaining sub-lists, then again, then again, then again, until we reach the base case where each list is of size one.
The pseudocode of the algorithm can be seen at *Algorithm* [5].

Correctness follows, while for evaluating time complexity we need to identify the worst case.
Let us start with the case where *pivot* falls outside A, meaning that one of the two parts is

empty and thus the corresponding half of B can be ignored. So, one *binary search* [3.1.1] over A, costing $O(\log n)$ time, has discarded half of B.
If this keeps occurring in all recursive calls, the total number of them will be $O(\log m)$, which leads us to a time complexity of $O(\log m \log n)$.
On the other hand, if we have both balanced partitions so that $b_{m/2}$ not only falls inside A but coincides with the median element $a_{n/2}$, the time complexity can be expressed via the recurrence relation $T(n,m) = O(\log n) + 2T\left(\frac{n}{2}, \frac{m}{2}\right)$, with the base case $T(n,m) = O(1)$ whenever $n, m \leq 1$.
This recurrence has the solution $T(n,m) = O\Big(m\Big(1 + \log\frac{n}{m}\Big)\Big)$ for each $m \leq n$, which is an optimal time complexity in the comparison model.

That being said, despite its optimal time complexity, the mutual-partitioning paradigm is heavily based on recursive calls and binary searching, and both paradigms offer poor performance in a disk-based setting when sequences are long hence requiring a large number of both dynamic memory allocations (recursive calls) and random memory access (*binary search* steps).

Algorithm 5

Pseudocode for divide and search algorithm

Let $m = |B| \leq n = |A|$
Pick *pivot* $p = b_{\lfloor m/2 \rfloor}$
Binary search for p in A ▷ Say $a_j \leq p < a_{j+1}$
Divide and search on $A\Big[1, j\Big] \cap B\Big[1, \frac{m}{2} - 1\Big]$
if $p = a_j$ **then**
    Add p to result
end if
Divide and search on $A\Big[j+1, n\Big] \cap B\Big[\frac{m}{2} + 1, m\Big]$

3.2.4 Doubling Search

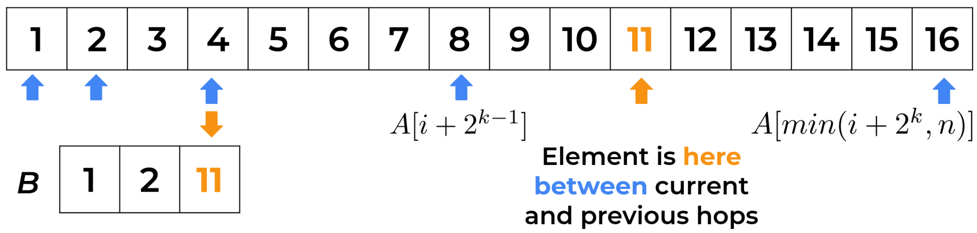

Figure 3.5: Doubling search algorithm

Also called *exponential search* or *galloping search*, this paradigm is more or less what we presented in the search algorithm *exponential search* [3.1.2]: assuming $m = |B| \leq n = |A|$, we search of each element b_j of B into A by jumping trough it with exponentially bigger steps that increase by a factor of 2^k, meaning that we compare b_j with $A[0]$, $A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, $A[32]$, and so on, until we find that either $b_j < A\left[i + 2^k\right]$ for some k, or we have jumped out of the array since $i + 2^k > n$.
Then we perform a *binary search* [3.1.1] for b_j between $A\left[i + 2^{k-1}\right]$ and $A\left[min\left(i + 2^k, n\right)\right]$.
$A\left[0, i + 2^{k-1}\right]$ will be discarded from the subsequent searches.
The pseudocode of the algorithm can be seen at *Algorithm* [6], which is a bit different from the one we saw in *exponential search* section [3.1.2], so it can be seen as an extra resource. Both works very similarly.

Correctness is again immediate, while deriving time complexity will require some reasoning: we denote with Δ_i the size of the sub-array of A where b_j could be found. We then say that:

- $b_j \geq i + 2^{k-1}$ i.e. previous step
- $b_j < min\left(i + 2^k, n\right)$ i.e. current step or end of A

We can therefore write $2^{k-1} \leq i - (i - 1)$ where i and $(i - 1)$ are current and previous step respectively, and combining this inequality with Δ_i we get $\Delta_i \leq 2^{k-1} \leq i - (i - 1)$.
At this point we can estimate the total length of search sub-arrays of A: $\sum_{i=1}^{m} \Delta_i \leq \sum_{i=1}^{m}(i - (i - 1)) \leq n$, because the latter is a telescopic sum in which consecutive terms cancel out.
For every i, the algorithm executes $O\left(1 + \log \Delta_i\right)$ steps, thus summing for $i = 1, 2, ..., m$ (since $m = |B|$) we get a total time complexity of:

$\sum_{i=1}^{m} O\left(1+\log\Delta_i\right) = O\left(\sum_{i=1}^{m}\left(1+log\Delta_i\right)\right) = O\left(m + m\log\sum_{i=1}^{m}\frac{\Delta_i}{m}\right) = O\left(m\left(1+\log\frac{n}{m}\right)\right)$. This is the same time complexity we got from the *divide and search* [5] algorithm, except with a iterative paradigm, which thus does not require dynamic memory allocation. Moreover, it calls the *binary search* [3.1.1] on a sub-array of A needing less disk accesses. Unfortunately, it falls short with very large lists, since galloping through them may require moving them in chunks back and forth from memory. It would thus be ideal to *compress* in some way the vector A.

Algorithm 6

Pseudocode for doubling search algorithm

Let $m = |B| \leq n = |A|$
Let $i = 0$
for all $j = 0$ to $m - 1$ **do**
  Let $k = 0$
  while $B\left[j\right] > A\left[i + 2^k\right]$ and $i + 2^k \leq n$ **do**
    $k = k + 1$ ▷ Increment exponent
  end while
  $i' =$ *binary search* into $A\left[i + 2^{k-1} + 1, min\left(i + 2^k\right), n\right]$
  if $a_{i'} = b_j$ **then**
    Add b_j to result
  end if
  $i = i'$ ▷ Update i to the last position
end for

3.2.5 Two-Level Storage Approach

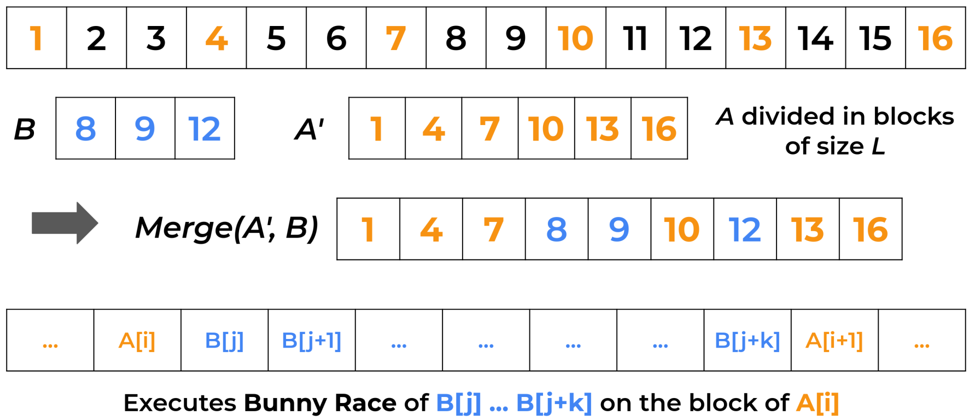

Figure 3.6: Two-level storage search algorithm

Since most of the time we can consider physical memory as divided in two, namely cache and RAM, engineers took advantage of this by adopting a two-level storage approach: the main idea is to preprocess a collection of lists, logically partitioning each of them into blocks of size L (final block may be shorter), and copying the first element of each block into an auxiliary sequence. Then, merge them and compute the intersection: elements of block $merged\_list[i+1]$ will be contained (if present) in block $merged\_list[i]$.
Let's see a simplified implementation where we preprocess only the longer sequence: say we have two lists such that $m = |B| \leq n = |A|$, we then divide A in $\left\lceil \frac{n}{L} \right\rceil$ blocks of size L and we save the first element of each block, called *guides*, in a new list A', effectively compressing the original list.
We then use the *merge procedure* (which can be seen here) to merge A' and B into a new list *Merged*, which will contain each *guide* of A and all elements of B interspersed between them. The list is, of course, sorted.
Now we can find the elements of B in the respective preceding *guide* of A. Lets clarify, we have:

- $Merged[k] = A'[i]$
- $Merged[k+1 \ldots k+1+\alpha] = B[j \ldots j+\alpha]$
- $Merged[k+\alpha+2] = A'[i+1]$

Thus the elements of B, $b_j \dots b_{j+\alpha}$ can all be found in the block of *guide* $A'[i]$.

Regarding time complexity, creating the list of *guides* takes linear time over the size of A. We then apply the *merge procedure* to fuse together A' and B in $O\left(\frac{n}{L} + m\right)$ time, and finally we search for $b_j \dots b_{j+\alpha}$ in the block of *guide* $A[i]$ using the *bunny race* [3.2.2] algorithm in time $O\left(|A_i| + |B_j|\right) = O\left(L + |B_j|\right)$ where $|A_i|$ is the size of the block of $A'[i]$, and $|B_j| = \left|B[j \dots j+\alpha]\right|$.
This algorithm is executed over all non-empty pairs (A_i, B_j), which are no more than m since $B = \cup_j B_j$, thus the total time taken by this step is $O(Lm + m)$.
Summing up the time complexity of all steps we get that the algorithm has:

- $O\left(\frac{n}{L} + mL\right)$ Time complexity
- $O\left(\frac{n}{LB} + \frac{mL}{B} + m\right)$ I/Os

Were B is the disk-page size of the two-level memory model.

3.3 Melding Algorithms

"The algorithms discussed so far focus on intersecting two lists, even though they are easily extendable to k lists intersections. Now we will see a collection of algorithms that can be used to intersect k sets. Most of the algorithm shown are taken from the article "An Experimental Investigation of Set Intersection Algorithms for Text Searching" by Birbay and López-Ortiz (Barbay et al., 2009).

3.3.1 Baeza-Yates and Baeza-Yates Sorted

One of the oldest algorithms presented in this survey, it is commonly found in the literature as a baseline for comparison, although it does not find much use in the real-world anymore. It was originally intended for the intersection of two sorted lists: it takes the median element of the smaller list and searches for it in the larger list, and it adds it to the result if it finds it.
Since it will always find either and element (e.g. b_j) or a position where it could have been (e.g. $a_i \leq b_j \leq a_{i+1}$), it recursively calls itself on the two sub-lists in which each list can be split according to the median element. This approach is very similar to the *divide and search* [3.2.3] algorithm.

To adapt this algorithm for k sets, Baeza-Yates suggests to intersect the lists two-by-two, starting with the first two smallest ones. Since the result of the intersection may not be sorted, the result set needs to be sorted before going to the next list. The pseudocode can be seen at *Algorithm* [7].

To avoid the cost of sorting each intermediate result, a minor variant can be used, called *Baeza-Yates Sorted*, which move the elements to te result only at the last recursive step, ensuring they are added in order.

Algorithm 7

Pseudocode for Baeza-Yates algorithm

function BAEZA-YATES(set,k)
    Sort sets by size $\Big(\Big|set[0]\Big| \leq \ldots \leq \Big|set[k]\Big|\Big)$
    for all $i = 1$ to k **do**
        $set[0] = BYinteresct\Big(set[0], set[i], 0, \Big|set[0]\Big| - 1, 0, \Big|set[i]\Big| - 1\Big)$
        Sort $set[0]$
    end for
end function

function BYINTERSECT(setA, setB, minA, maxA, minB, maxB)
    if $setA = \emptyset$ or $setB = \emptyset$ **then**
        return $\emptyset$
    end if
    Let $m = \frac{(minA+maxA)}{2}$
    Let $median\_A = setA[m]$
    Search for $median_A$ in $setB$
    if $median\_A$ found **then**
        Add $median\_A$ to result
    end if
    Let r be the position of $median\_A$ in $setB$
    Solve the intersection recursively on both sides of r and m in each set
end function

3.3.2 Sequential and Random Sequential

Introduced by Barbay and Kenyon, it takes a collections of k sorted sets ordered by size (ascending). Then it picks elements from the smallest one (S_0) and searches for them in the other sets ($S_{1\ldots k-1}$) via *galloping search* [3.2.4], one set at a time. It progressively check whether the element can be found in the set its looking into: if $key = 1$ and $S_i[j] = 7$ there is no need to keep checking S_i. There exists a randomized variants which instead of going through the sets in order, it does so randomly (of course, it only chooses not-yet-checked sets).

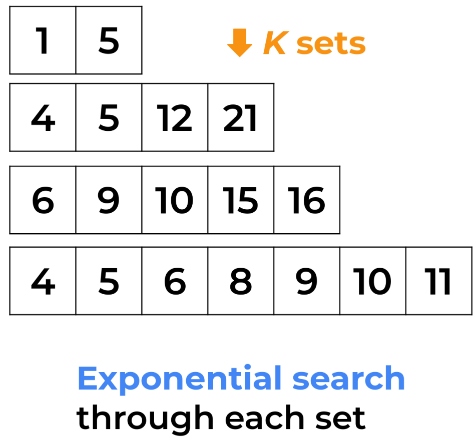

Figure 3.7: Sequential melding algorithm

The pseudocode of a simplified version can be seen at *Algorithm* [8], which has a time complexity of $O\left(k \cdot m^2 \left(1 + \log \frac{n}{m}\right)\right)$ since it performs *k-galloping searches* for each element of the smallest set. We can let n to be the size of the biggest list.
Another version of the algorithm, which considers also the eliminator and therefore is more efficient, can be seen at *Algorithm* [9].

Algorithm 8

Pseudocode for Sequential melding algorithm

for all $i = 0$ to $|S_0|$ **do**
    Let $key = S_0[i]$
    Let $counter = 0$
    for all $j = 1$ to $k - 1$ **do**
        Galloping search for key in S_j
        if key found **then**
            $counter = counter + 1$
        end if
    end for
    if $counter = k - 1$ **then**
        Add key to result
    end if
end for

Algorithm 9

Pseudocode for Sequential with eliminator algorithm

Let eliminator be $e = S_0[0]$
Let $i = 1$
Let $counter = 0$
while $e \neq \infty$ **do**
    Search for e in S_i
    if e found **then**
        $counter = counter + 1$
        if $counter = k - 1$ **then**
            Add e to result
        end if
    else
        $e = S_i\,[succ(e)]$
    end if
    $i = i + 1 \bmod k$
end while

3.3.3 SvS and Swapping SvS

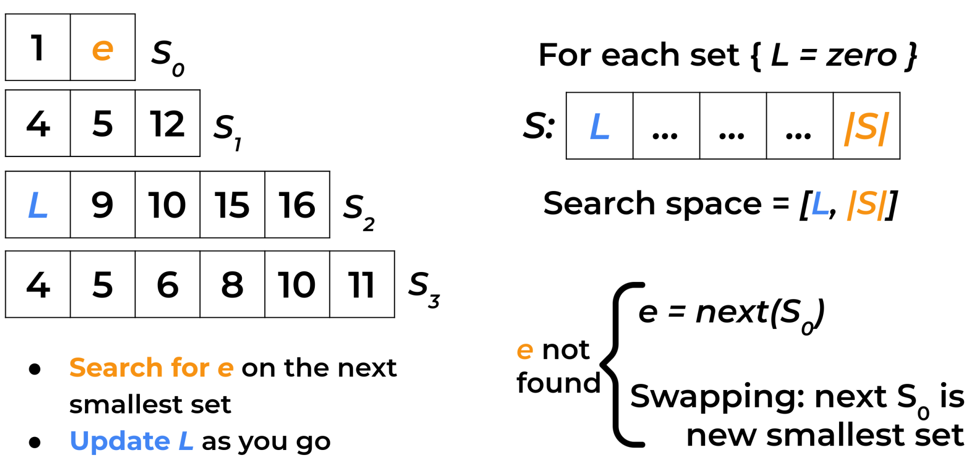

Figure 3.8: SvS melding algorithm

SvS is a widely used algorithm which, starting from a collection of sorted sets $S_0 \ldots S_k$, ordered by size (ascending), computes the intersection by iteratively reducing the search space.

We first instantiate a value $\ell = 0$ for all sets, which will be used to keep track of the last checked element in each set, acting as a sort of logical starting position, then, we pick elements from the smallest set S_0 and search for them in the other sets, updating ℓ as we go: if $key = 10$ and $S_j[\alpha] = 9$ then we won't need to ever check $S_j[0 \dots \alpha]$. We can use any search algorithm we want.

Its variant picks the next search element from the new smallest set instead than going through all the elements of the first smallest set (the "original" S_0).

The pseudocode of the algorithm can be seen at *Algorithm* [10], and there is one thing worth noting: at the end of its run, the resulting intersection is contained in the smallest *candidate* set S_0, wether such intersection exists (thus $S_0 = [s_1, \dots, s_\alpha]$) or not (thus $S_0 = \emptyset$).

Once again, complexity is $O\left(k \cdot m^2 \left(1 + \log \frac{n}{m}\right)\right)$ if we decide to use the *galloping search* [3.2.4], where $m = |S_0|$ and $n = |S_k|$, but since we update ℓ as we go, thus progressively reducing the search space, the algorithm is more efficient than *Sequential* [3.3.2] in a real-world implementation.

Algorithm 10

Pseudocode for SvS melding algorithm

Sort sets by size $\left(\left|set[0]\right| \leq \dots \leq \left|set[k]\right|\right)$
for all $i = 0$ to $k - 1$ **do**
    Let $\ell[i] = 0$ ▷ Set all starting positions to zero
end for
for all key in S_0 **do**
    for all $i = 1$ to $k - 1$ **do**
        Search for key in S_i in the range $\ell[i]$ to $|S_i|$
        Update $\ell[i]$ to the position of key in S_i
        if key not found **then**
            Delete key from S_0 ▷ S_0 is the result in the end
        end if
    end for
end for

3.3.4 Small Adaptive

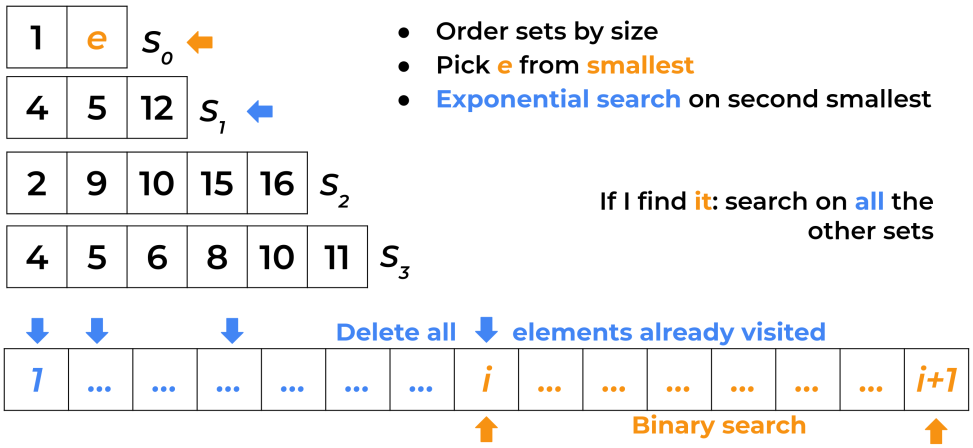

Figure 3.9: Small Adaptive melding algorithm

Proposed by Birbay et al. in their article (Barbay et al., 2009), *Small Adaptive* is a hybrid algorithm which combines properties from various others: we start from a collection of sets ordered by size (ascending), then we pick an element to search for (say e) from the smallest set S_0 and we perform an *exponential search* [3.1.2] on the second smallest set S_1. If we find it, we look for it in the remaining $k-2$ sets, deleting already-seen elements as we go, thus progressively reducing the search space.
The algorithm, after each search on a set S_i, picks the next smallest one and keeps ordering the sets so that, after a full search, it may be that S_0 and S_1 are different sets from before.

Pseudocode can be seen at *Algorithm* [11], while the time complexity is almost identical to *SvS* [3.3.3] or *Sequential* [3.3.2] (except for the ordering), but thanks to the ideas of: *i.* searching first on the second smallest list before looking trough the rest; *ii.* re-ordering the sets so we always first search into the two smallest and; *iii.* deleting already-seen elements; *Small Adaptive* can be more efficient in practice (although *SvS* still outperforms it often).

A comparison of the algorithms seen in [3.3.1], [3.3.2], [3.3.3] and *Small Adaptive*, implemented and run on a collection of both synthetic and real-world-data datasets, can be seen in the tables: 3.10, 3.11, 3.12.

Algorithm 11

Pseudocode for Small Adaptive melding algorithm

```
while No set is empty do
    Sort sets by size (|set[0]| ≤ ... ≤ |set[k]|)
    Look for key = S_0[0]
    Delete S_0[0]
    Let couter = 1                                  ▷ At least e ∈ S_0
    Search for key in S_1
    if key found then
        counter = counter + 1
        for all i = 1 to k − 1 do
            Search for e in S_i
            Delete already-seen elements
            if key found then
                counter = counter + 1
            end if
        end for
    end if
    if counter = k then
        Add key to result
    end if
end while
```

	SvS		Swapping_SvS		Sequential		BaezaYates		So_BaezaYates		Small_Adaptive	
	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu
Total_Binary	2815	262397	2815	254008	4397	457540	2811	250018	4501	402544	2815	677318
Adaptive_Binary	2469	255064	2469	230916	2632	327075	1620	**188258**	1620	218156	2469	444476
Rounded_Binary	2623	242986	2623	246871	3997	436438	2629	242773	4190	391347	2623	443064
Galloping	2087	245333	2087	244216	2237	332311	2410	255945	2373	286040	2087	435828
Interpolation	1067	279127	1067	280624	1242	374779	1066	275463	1064	304616	1067	466446
Extrapolation	1281	375585	1281	371444	1444	464203	1261	373947	1262	401933	1281	547751
Extrapol_Ahead	**1024**	413209	**1024**	404841	1198	576109	1085	426452	1073	506075	**1024**	584941

Figure 3.10: Total number of comparisons and CPU times performed by each algorithm over the Random data set. In bold, the best performance in terms of the number of comparisons, for various melding algorithms in combination with *Extrapol_Ahead*, and the best performance in terms of CPU: *BaezaYates* using *Adaptive_Binary*. (Barbay et al., 2009)

	SvS		Swapping_SvS		Sequential		BaezaYates		So_BaezaYates		Small_Adaptive		RSequential	
	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu	cmp	cpu
Total_Binary	58217	5.142	58209	4.976	93087	8.674	57594	5.426	83710	7.140	58217	8.325	94400	15.446
Adaptive_Binary	39221	3.762	39221	3.937	55817	6.704	18543	3.284	15689	3.113	39225	7.208	54210	13.401
Rounded_Binary	54674	4.684	54671	4.831	87267	8.260	54286	5.327	78511	6.908	54679	7.995	88509	14.873
Galloping	16884	**2.791**	16884	2.874	25440	4.808	24285	3.953	20935	3.769	16884	5.980	24518	11.525
Interpolation	12184	3.338	12184	3.434	17843	5.640	15352	4.182	12386	4.046	12185	6.577	17398	11.992
Extrapolation	13426	4.229	13426	4.248	19672	6.617	17455	5.426	14428	5.258	13427	7.493	19100	13.104
Extrapol_Ahead	**12125**	5.480	**12125**	5.424	17701	8.641	16179	6.637	13145	7.279	12126	8.614	17279	15.036

Figure 3.11: Total number of comparisons and CPU times (in millions of cycles) performed by each algorithm over the Google data set. In bold, the best performance in terms of number of comparisons, *SvS* and *Swapping_SvS* using *Extrapol_Ahead*, and in terms of CPU times, *SvS* using *Galloping*. (Barbay et al., 2009)

	SvS		Swapping_SvS		Sequential		BaezaYates		So_BaezaYates		Small_Adaptive	
	INTEL	SUN	INTEL	SUN	INTEL	SUN	INTEL	SUN	INTEL	SUN	INTEL	SUN
Adaptive_Binary	117303	153887	57686	159169	901254	409576	53363	112401	36273	98411	180957	230258
Total_Binary	360526	180854	81227	182974	598387	354558	93341	184239	88081	227041	320692	244521
Rounded_Binary	64910	175343	63693	180150	169797	348563	75730	182170	83717	223368	108728	241526
Galloping	33255	**96907**	**30686**	102197	132245	219816	55088	125904	40462	111422	59081	162243
Interpolation	47883	134960	49060	140272	127338	327509	67066	157669	54331	142653	75162	200471
Extrapolation	49694	142385	50570	147886	136946	328316	77592	185944	63244	171270	78606	208057
Extrapol_Ahead	61731	158138	62021	163545	155396	338525	87303	194108	81922	192490	88674	223195

Figure 3.12: Total CPU time performed by each algorithm over the TREC GOV2 data set. In bold, the smallest CPU times on the INTEL platform, obtained using *Swapping_SvS*; and on the SUN platform, obtained using *SvS*, both in combination with *Galloping* search. (Barbay et al., 2009)

4 Conclusions

As we were able to see in the previous chapters, especially in the comparison tables in section [3.3.4], out of the many algorithms available the best ones are often the simplest, keeping true to the KISS principle: in both tables [3.11] and [3.12] the best performing solutions are *SvS* or its variations, and frequently the preferred searching algorithm is either *Binary search* or *Galloping*, both of which are not particularly exotic.

Moreover, straightforward solutions offer a significant advantage: they are easy to implement since they are easy to understand, which is just as important as being efficient and effective. This pattern can be observed in other domains, such as distributed systems, where the *Raft* algorithm for consensus (Ongaro and Ousterhout, 2014) won over its predecessor *Paxos* (Lamport, 1998) thanks to its simplicity and ease of understanding.

Bibliography

Barbay, J., López-Ortiz, A., Lu, T., and Salinger, A. (Dec. 2009). "An Experimental Investigation of Set Intersection Algorithms for Text Searching". In: *ACM Journal of Experimental Algorithmics*, 14. DOI: 10.1145/1498698.1564507.

Ferragina, P. (2023). *Set Intersection*. Cambridge University Press, pp. 72–81.

Lamport, L. (May 1998). "The part-time parliament". In: *ACM Trans. Comput. Syst.*, 16(2), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229.

Mahapatra, A. K. and Biswas, S. (July 2011). "Inverted indexes: Types and techniques". In: *International Journal of Computer Science Issues*, 8.

Nah, F. (Jan. 2004). "A study on tolerable waiting time: how long are Web users willing to wait? Citation: Nah, F. (2004), A study on tolerable waiting time: how long are Web users willing to wait? Behaviour & Information Technology, forthcoming". In: *Behaviour & IT*, 23.

Ongaro, D. and Ousterhout, J. (2014). "In search of an understandable consensus algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, pp. 305–320. ISBN: 9781931971102.