Signature Files

- Unlike the case of inverted files where, most of the time, where is a tree structure underneath, signature files use hash tables
- The main idea is to divide the document into blocks of fixed size and each block has assigned to it a signature (also fixed size), which is used to search the document for the queried pattern
- The block signature is obtained by OR'ing the hashed bitstrings of each term in the block
- Signature files pose a low overhead (10% to 20% over the text size), at the cost of forcing a sequential search over the index.
- The search complexity is linear instead of sublinear as with the previous approaches.
- However the constant of the complexity is rather low: the technique is suitable for not very large texts.
- Inverted files outperform signature files for most applications

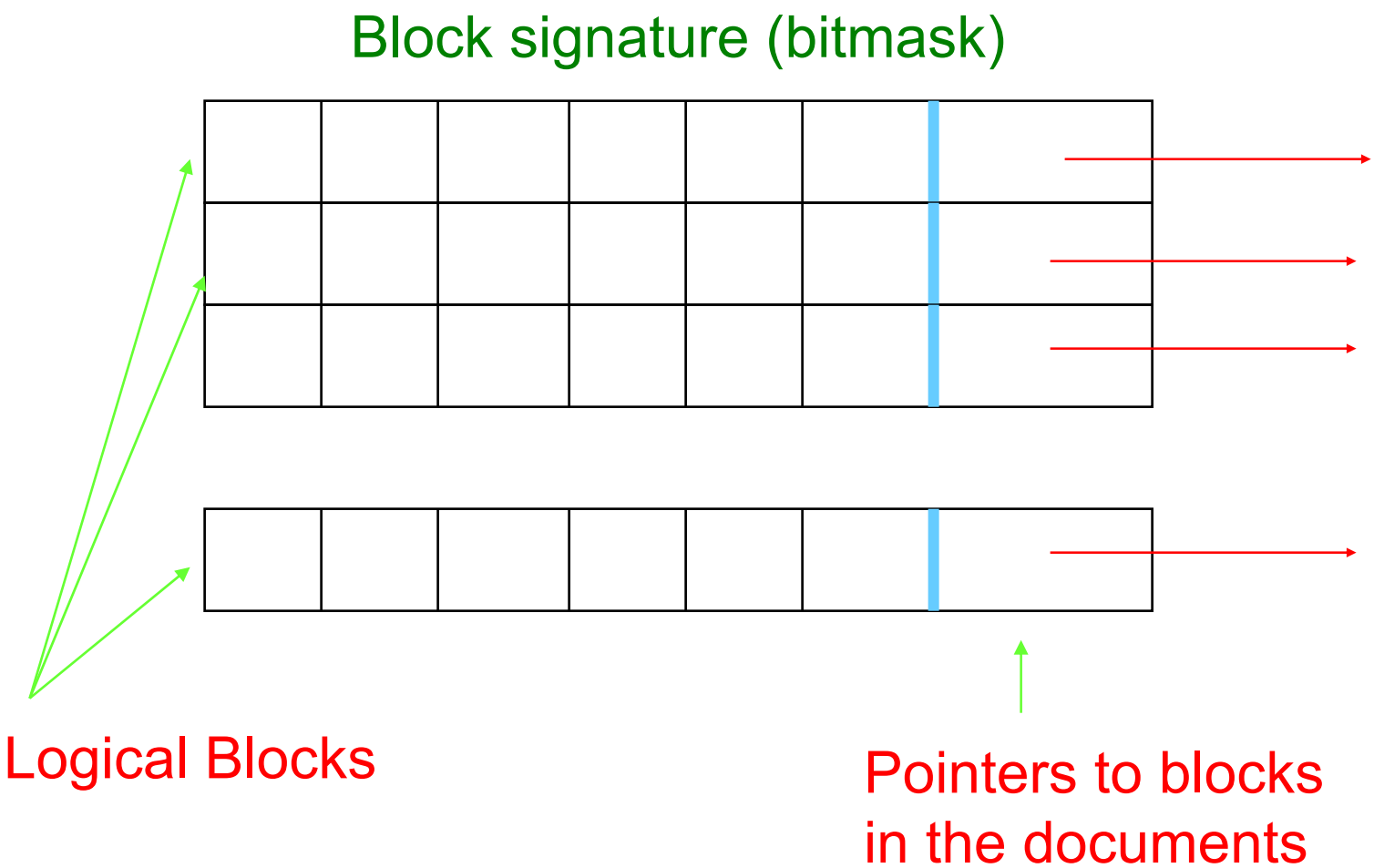
Signature Files

- Consider:
  - H(information) = 010001
  - H(text) = 010010
  - H(data) = 110000
  - H(retrieval) = 100010
  - The block signatures of a document D containing the text “textual retrieval and information retrieval” (after removing stopwords and stemming) for a block size of two terms – would be:
    - B1D = 110010 and
    - B2D = 110011

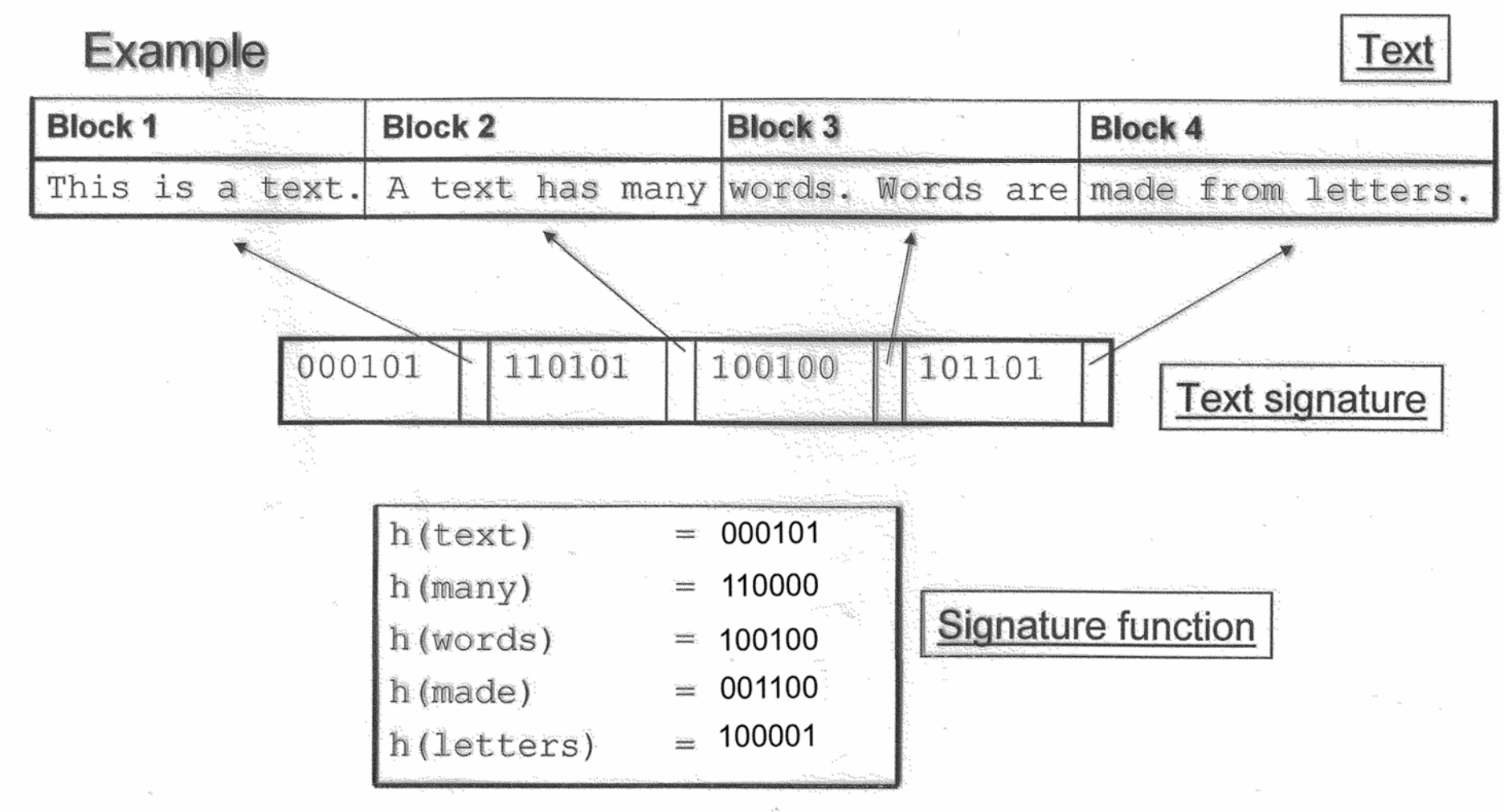
Signature Files - searching

- To search for a given term we compare whether the term's bitstring could be "inside" the block signatures
- Consider we are searching for "text" in document D
  - H(text) = 010010 and B1D = 110010
  - H(text) bit-wise-AND B1D = 010010 = H(text)
  - Therefore "text" **could** be in B1D (it is in this particular case)
- Consider we are now searching for "data"
  - H(data) bit-wise-AND B1D = 110000 = H(data)
  - H(data) bit-wise-AND B2D = 110000 = H(data)
  - Though "data" is not in either block !
- Signature files may yield false hits …

Signature File - layout



Signature File - layout



Signature Files – design issues

- How to keep the probability of a false alarms low ? How to predict how good a signature is ?
- Consider:
  - B: the size (number of bits) of each term's bitstring
  - b: number of terms per document block
  - l: the minimum number of bits set (turned on) in B, this depends on the hashing function
- A good model assumes that l bits are randomly set in B
- Since each of the b terms sets l bits at random, the probability that a given bit of the mask is set in a block signature is:

$$1-(1-1/B)^{bl} \approx 1-e^{-bl/B}$$

- Hence, the probability that the l random bits set in the query are also set in the mask of any given text block is

$$\left(1-e^{-bl/B}\right)^{l}$$

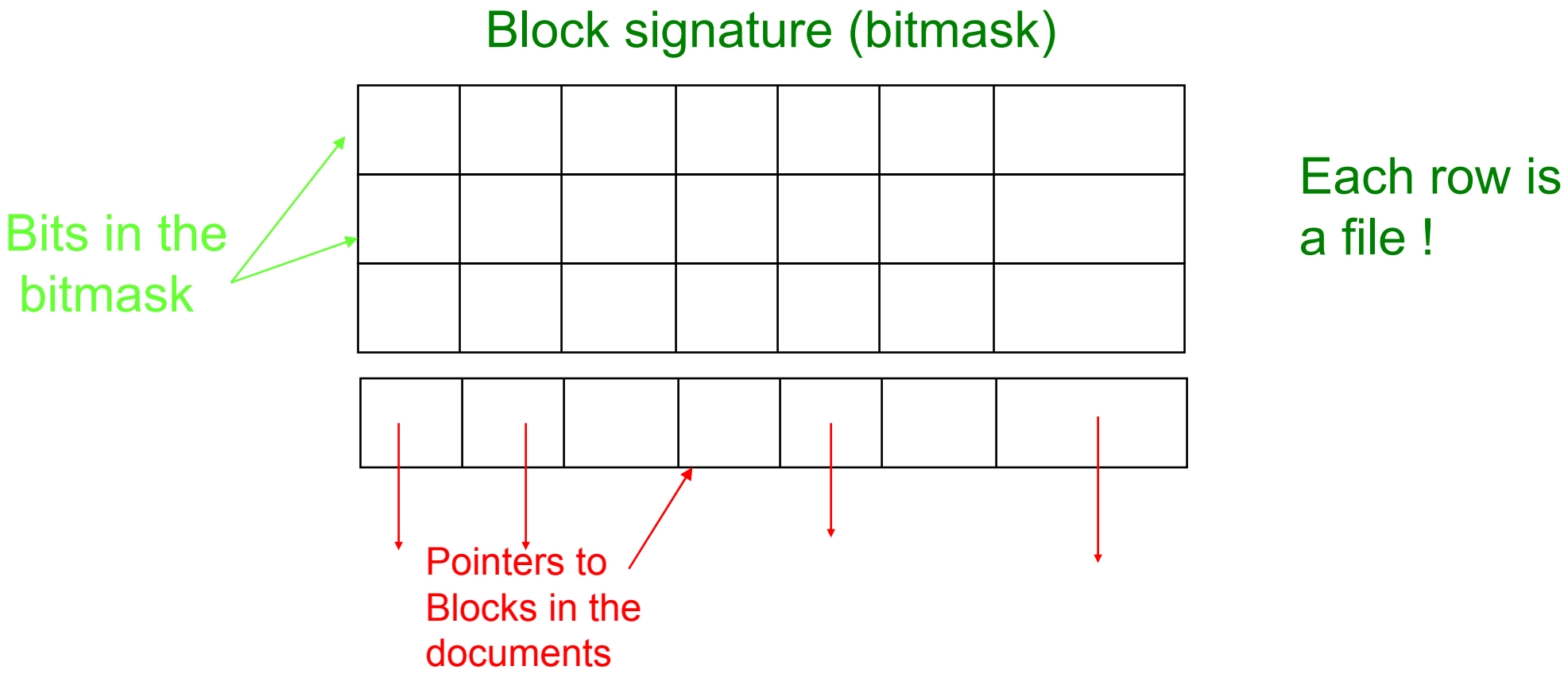
- Now, the expected number of the false matches is $N \bullet \left(1-e^{-bl/B}\right)^{l}$ where N is the total number of document blocks
- This expression is minimized for l = B ln(2)/b: the probability of false hits equal to $2^{-l.}$
- B/b indicates the space overhead to pay

Signature Files - alternatives

- There are many strategies which can be used with signature files
- Signature compression
  - Given that the bitstring is likely to have many 0s a simple run-length encoding technique can be used
  - 0001 0010 0000 0001 is encoded as “328” (each digit in the code represents the number of 0s preceding a 1)
  - The main drawback is that comparing bitstrings require decoding at search time
- Bitsliced signature files
  - Using the original signature file layout and a bitstring (queried term signature) all logical blocks are checked as to whether that block could contain that bitstring
  - This may take a long time if the signature file is long (this depends on the block size)
  - Could we check only at the bits we are interested in?

    - Original signature file layout:
      - Each row has a bitmask corresponding to a block
      - Each column tells whether that bit is set of not
    - Bit-sliced signature files layout:
      - Each row correspond to a particular bit in the bitmasks
      - Each column is the bitmask for the blocks
    - The signature file is “transposed”

Signature Files - alternatives



- Advantages of bit-slicing:
- Given a term bitstring one can read only the lines (files) corresponding to those set bits and if they are set in a given column, then the whole column may be retrieved.
- Empty answers are found very fast

Bitsliced Signature File - Example

Term	Hash string
cold	1000 0000 0010 0100
days	0010 0100 0000 1000
hot	0000 1010 0000 0000
in	0000 1001 0010 0000
it	0000 1000 1000 0010
like	0100 0010 0000 0001
nine	0010 1000 0000 0100
old	1000 1000 0100 0000
pease	0000 0101 0000 0001
porridge	0100 0100 0010 0000
pot	0000 0010 0110 0000
some	0100 0100 0000 0001
the	1010 1000 0000 0000

Document	Text	Descriptor
1	Pease porridge hot, pease porridge cold,	1100 1111 0010 0101
2	Pease porridge in the pot,	1110 1111 0110 0001
3	Nine days old.	1010 1100 0100 1100
4	Some like it hot, some like it cold,	1100 1110 1010 0111
5	Some like it in the pot,	1110 1111 1110 0011
6	Nine days old.	1010 1100 0100 1100

Signature file

Pos.	Slice	Pos.	Slice	Pos.	Slice	Pos.	Slice
1	111111	5	111111	9	000110	13	001001
2	110110	6	111111	10	011011	14	101101
3	011011	7	110110	11	110110	15	000110
4	000000	8	110010	12	000000	16	110110

Bitsliced Signature file

Suffix Trees/Arrays

- Suffix trees are a generalization of inverted files
- For traditional queries, i.e., those based on simple terms, inverted files are the structure of choice
- However, complex queries like phrase queries are expensive to solve
- This type of index treats the text to be indexed as a finite, but long string.
- Thus each position in the text represent a suffix of that text (i.e. a string that goes from that text position to the end of the text.
- Each suffix is uniquely identified by its position
- Not all text points need to be indexed. Index points are selected from the text, which point to the beginning of the text positions which will be retrievable.

Suffix Trees

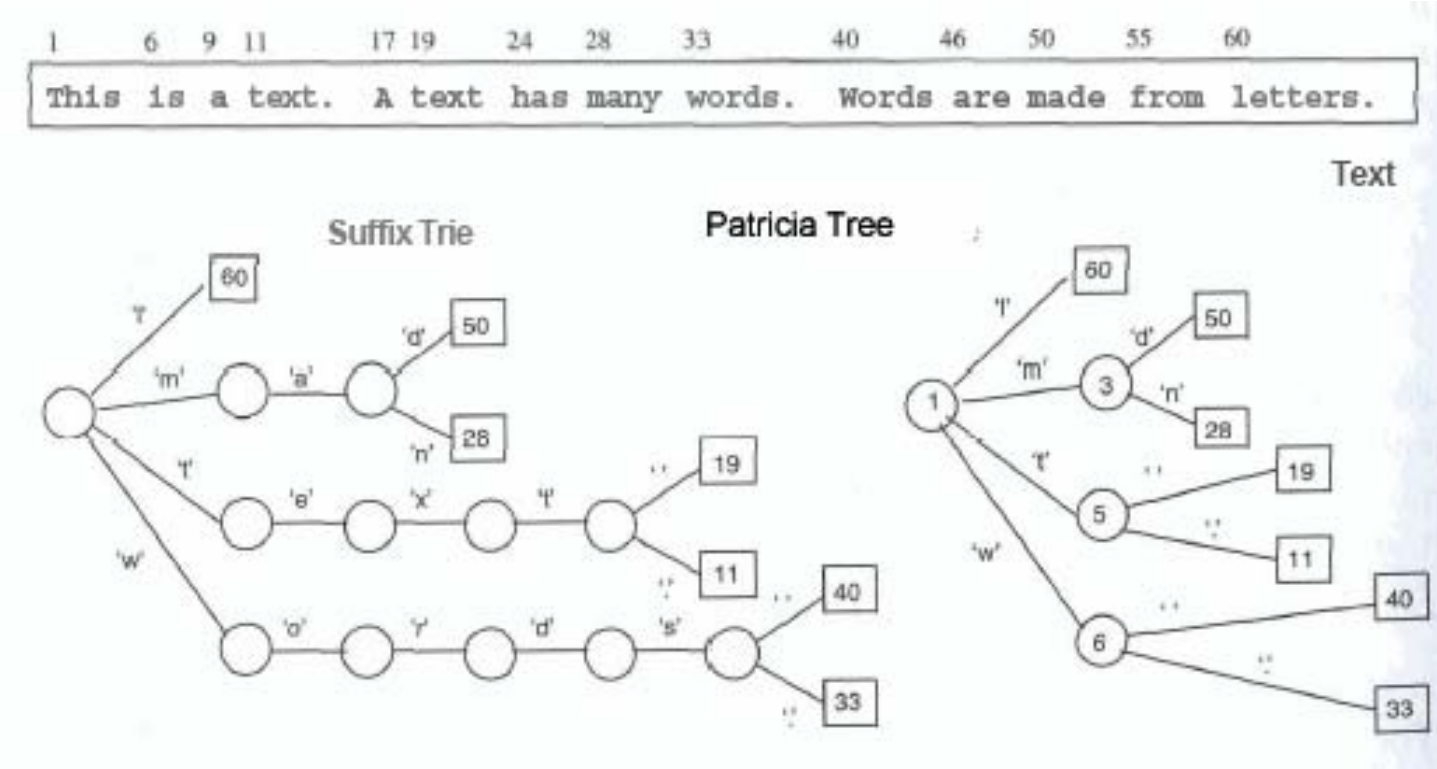
- Note that the choice of index points is crucial to the retrieval capabilities

1 6 9 11 17 19 24 28 33 40 46 50 55 60
This is a text. A text has many words. Words are made from letters.

…and the index points/suffixes:

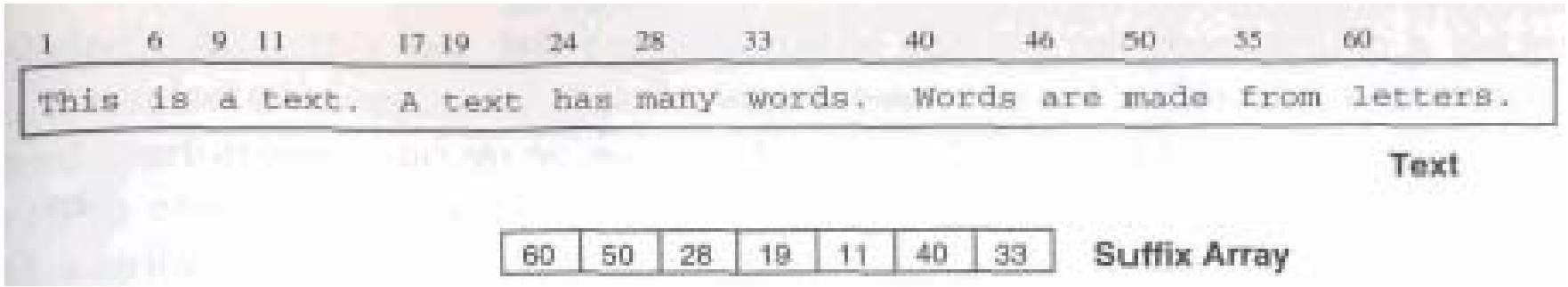
(suffix: a string that goes from that text position to the end of the text)

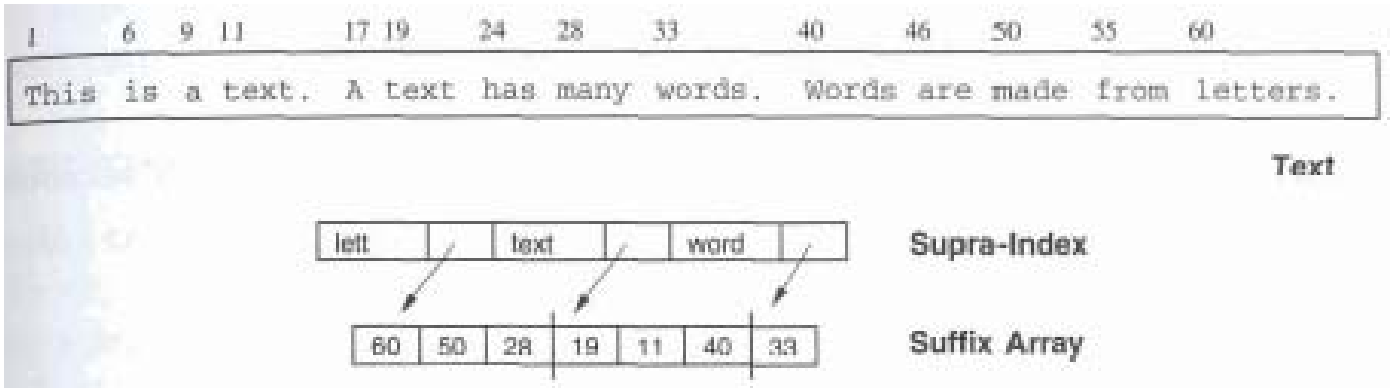
Suffix	Index point
text. A text has many words. … letters.	11
text has many words. … letters.	19
many words. … letters.	28
words. … letters.	33
Words … letters.	40
made … letters.	50
letters.	60



- Suffix tree is a trie data structure built over the suffixes of the text.
- The pointers to the suffixes are stored at the leaf nodes
- To improve space utilization, the trie is compacted into a Patricia tree.
- This involves compressing unary paths, i.e. paths where each node has just one child.
- Once unary paths are not present the tree has O(n) nodes instead of the worst-case $O(n^2)$ of the Trie.
- The problem with this structure is its space:
  - Depending on the implementation, each node of the trie takes 12 to 24 bytes, and therefore even if only beginnings are indexed, a space overhead of 120% to 240% over the text size produced.

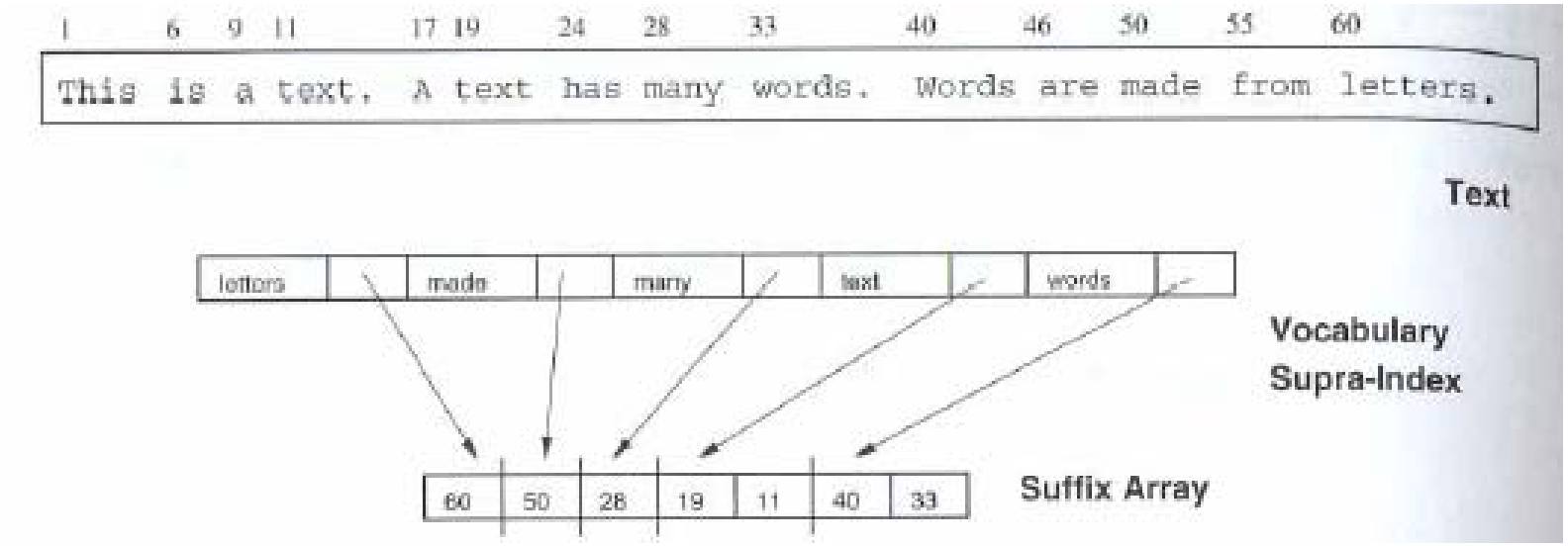
Suffix arrays





- Suffix arrays provide essentially the same functionality as suffix trees with much less space requirements.
- If the leaves of the suffix tree are traversed in left-to-right order (top to bottom in the figure), all the suffixes of the text retrieved lexicographical order.
- A suffix array is simply an array containing all the pointers to the text suffixes listed in lexicographical order
- Since they store one pointer per indexed suffix, the space requirements are almost the same as those for inverted indices, i.e. close to 40% overhead over the text size
- Suffix arrays are designed to allow binary searches done by comparing the contents of each pointer.
- If the suffix array is large (the usual case), this binary search performs poorly because of the number of random disk accesses.
- To remedy this situation, one idea is to use of supra-indices over the suffix array.
- The simplest supra-index is no more than a sampling of one out of b suffix array entries, where for each sample the first l suffix characters are stored in the supra index.
- This supra-index is used as a first step of the search to reduce external access

Suffix trees/arrays - Searching



- Many basic patterns such as words, prefixes, and phrases can be searched in O(m) time by a simple trie search.
- However, suffix trees are not practical for large texts, as explained.
- Suffix arrays can perform the same search operations in O(log n) time by doing a binary search instead of a trie search.
- This is achieved as follows:
  - the search pattern originates two ‘limiting patterns’ P1 and P2, so that we want any suffix S such that P1 ≤ S < P2.
  - We binary search both limiting patterns in the suffix array.
  - Then, all the elements lying between both positions point to exactly those suffixes that start like the original pattern (i.e., to the pattern positions in the text).
  - For instance, in the example, in order to find the word 'text' we search for 'text’ and 'texu', obtaining the portion of the array that contains the pointers 19 and 11.
  - All these queries retrieve a subtree of the suflix tree or an interval of the suffix array.
  - The results have to be collected later, which may imply sorting them in ascending text order.
  - This is a complication of suffix trees or arrays with respect to inverted indices.
  - Simple phrase searching is a good case for these indices.
  - A proximity search has to be solved element-wise: the matches for each element must be collected and sorted and then they have to be intersected as for inverted files.

Construction in Main Memory

- A suffix tree for a text of *n* characters can be built in *O(n)* time.
- The algorithm performs poorly if the suffix tree does not fit in main memory, which is especially stringent because of the large space requirements of the suffix trees.

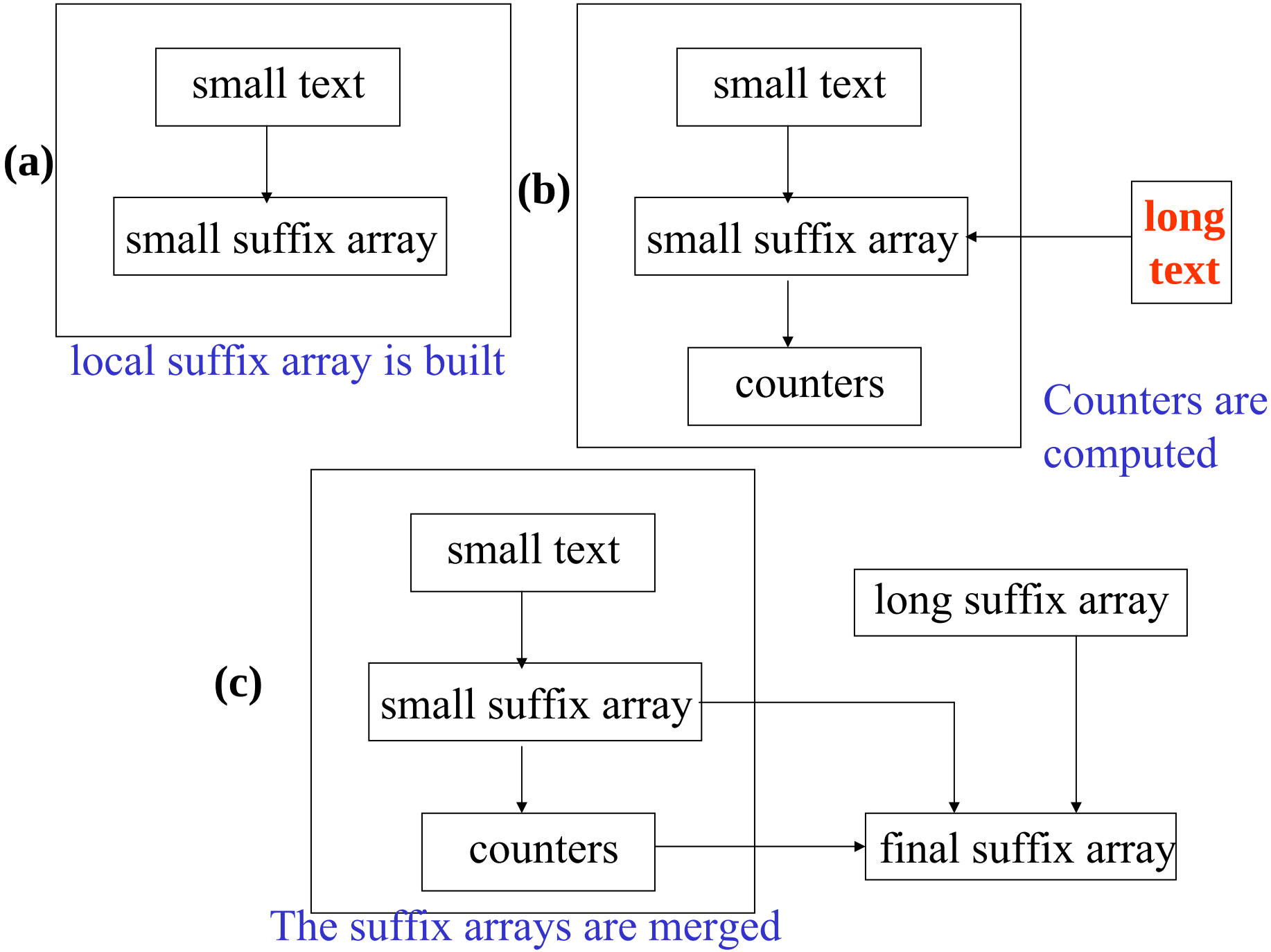
- Suffix array construction:
  - Suffix array is a set of pointers lexicographically sorted.
  - The pointers are collected in ascending text order and then sorted by the text they point to.
  - In order to compare two suffix array entries the corresponding text positions must be accessed.
  - These accesses are basically random.
  - Hence, both the suffix array and the text must be in main memory.
  - This algorithm costs O(*n* log *n)* string comparisons.

- Suffix array construction with *O(n* log *n)* character comparisons:
  - All the suffixes are bucket-sorted in *O(n)* time according to the first letter only.
  - Then, each bucket is bucket-sorted again, now according to their first two letters.
  - At iteration i, the suffixes begin already sorted by their 2^{i-1} first letters and end up sorted by their first 2^{i} letters.
  - As at each iteration the total cost of all the bucket sorts is O(*n),* the total time is O(*n* log *n).*
  - *S*orting the strings in the i-th iteration:
    - *all* suffixes are sorted by their first 2^{i-1} letters,
    - to sort the text positions *Ta...* and *Tb... belonging to* in the same bucket, it is enough to determine the relative order between text positions $T_{a+2^{i-1}}$ and $T_{b+2^{i-1}}$ in the current stage of the search.
    - This can be done in constant time by storing the reverse permutation.

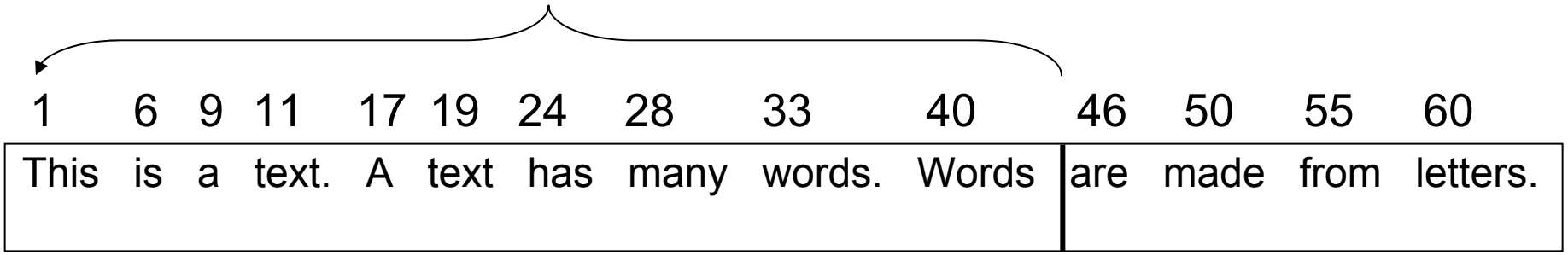
Construction of Suffix Arrays for Large Texts

- Use of an external memory sorting algorithm:
  - each comparison involves accessing the text at random positions on the disk.
  - This will severely degrade the performance of the sorting process.
- An algorithm especially designed for large texts:
  - Split the text into blocks that can be sorted in main memory.
  - For each block, build its suffix array in main memory and merge it with the rest of the array already built for the previous text.
- How to merge a large suffix array (already built) with the small suffix array (just built)?
  - determine how many elements of the large array are to be placed between each pair of elements in the small array,
  - Then use that information to merge the arrays without accessing the text.
- Hence, the information that we need is how many suffixes of the large text lie between each pair of positions of the small suffix array. We compute counters that store this information:
  - The text corresponding to the large array is sequentially read into main memory.
  - Each suffix of that text is *searched* in the small suffix array (in main memory).
  - Once we find the inter-element position where the suffix lies, we just increment the appropriate counter.
  - *O(M) the* main memory to index: *O(nlM)* text blocks.
  - Each block is merged against an array of size *O(n),* where all the *O(n)* suffixes of the large text are binary searched in the small suffix array.
  - This gives a total CPU complexity of $O(n^2 \log(M)lM)$.

Construction of Suffix Arrays for Large Texts (Cont.)



Construction of Suffix Arrays for Large Texts (Cont.)

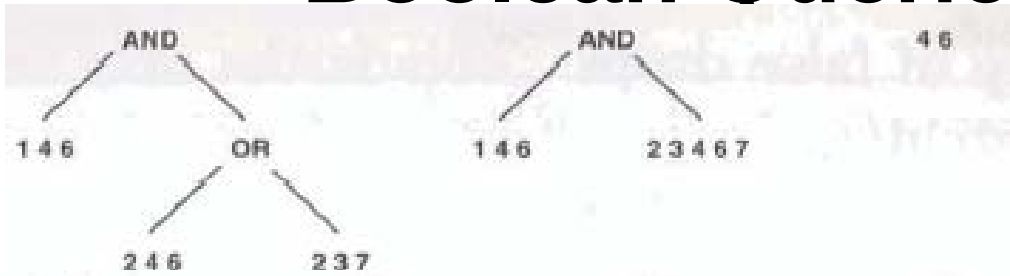


A text ….	17
a text. …	9
has …	24
is …	6
many …	28
text …	19
text. …	11
This …	1
Words …	40
words. …	33

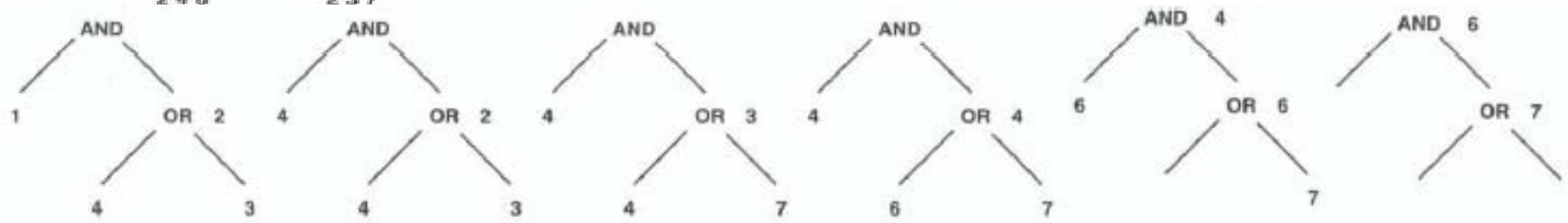
counter	**2**
are …	46
counter	**0**
from …	55
counter	**2**
letters …	60
counter	**0**
made …	50
counter	**6**

Boolean Queries

Full evaluation



lazy evaluation



- Set manipulation algorithms: these algorithms are used when operating on sets, which is the case in Boolean queries
- Once the leaves (basic queries) of the query syntax tree are solved, the relevant documents must be worked on by composition operators.
- As all operations need to pair the same document in both their operands, it is good practice to keep the sets sorted, so that operations like intersection, union, etc. can proceed sequentially on both lists and also generate a sorted list.
- Evaluation of the syntax tree:
    - full evaluation - both operands are first completely obtained and then the complete result is generated.
    - lazy evaluation - results are delivered only when required, and to obtain that result some data is recursively required to both operands.
- Full evaluation allows some optimizations to be performed because the sizes of the results are known in advance:
    - for instance, merging a very short list against a very long one can proceed by binary searching the elements of the short list in the long one.
    - Lazy evaluation allows the application to control when to do the work of obtaining new results, instead of blocking it for a long time.
- The complexity of solving these types of queries, apart from the cost of obtaining the results at the leaves, is normally linear in the total size of all the intermediate results.
- This time may dominate the others, when there are huge intermediate results. This is more noticeable to the user when the final result is small.