INFORMATION RETRIEVAL

Laura Nenzi

lnenzi@units.it

Lecture 4a

LECTURE OUTLINE

Counting disk accesses

Postings Lists and dictionary compression

index construction and updating

Boolean Queries Optimisation

$(a \wedge b) \wedge c$

$a \wedge (b \wedge c)$

NEW DOCUMENTS
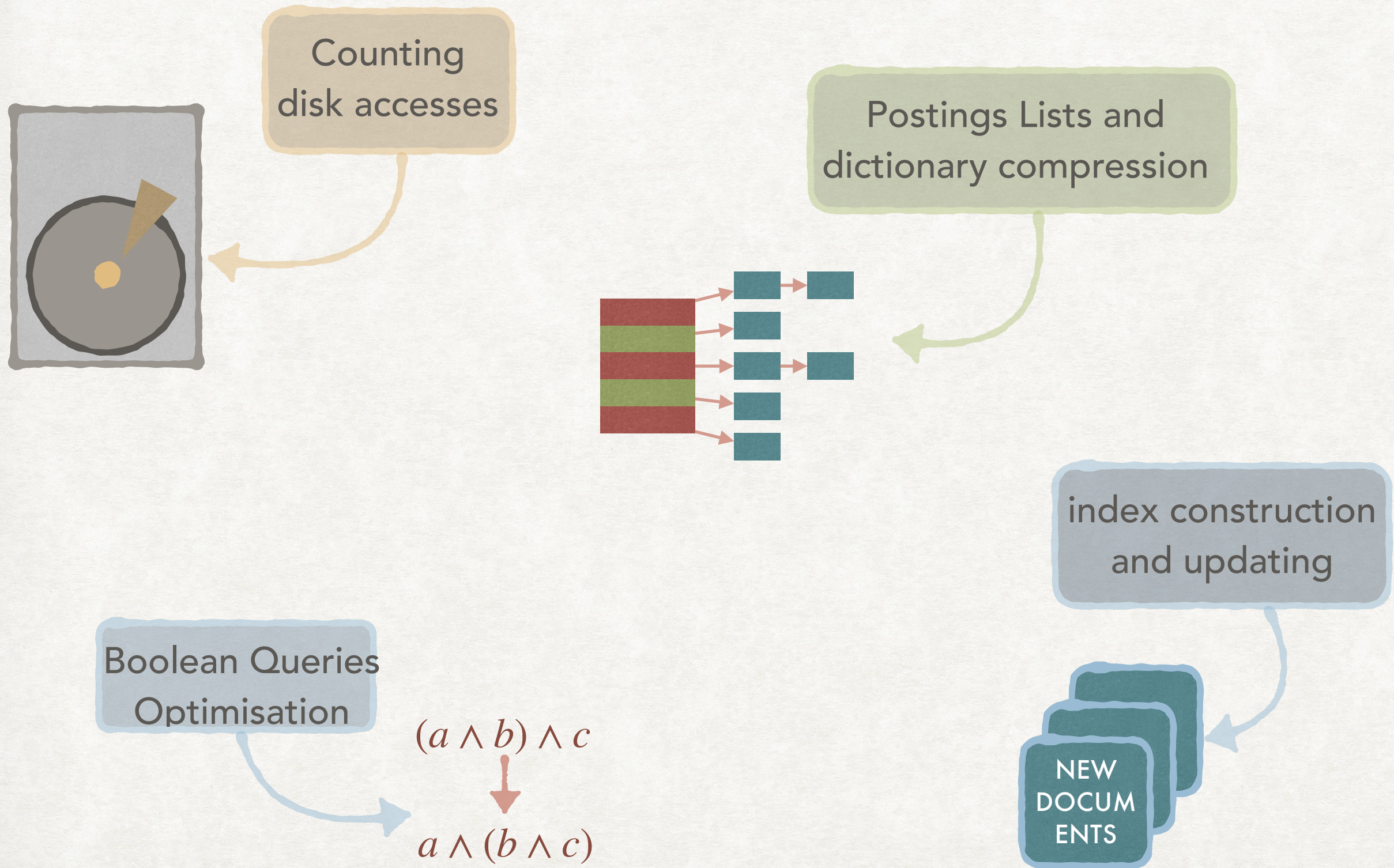
OPTIMISATION OF BOOLEAN QUERIES

WHICH ONE IS BETTER?
SOMETIMES ORDER IS IMPORTANT

Query: Monty AND Python AND Grail

Can be evaluated in three ways:

(Monty AND Python) AND Grail

(Python AND Grail) AND Monty

(Monty AND Grail) AND Python

The result is the same but the performances might differ

OPTIMISATION OF BOOLEAN QUERIES

- The main idea is to select the order the reduce the size of the intermediate results…

- …but we don't know the size of the intersection

- But we know that $|A \cap B| \leq \min(|A|, |B|)$, hence we use $\min(|A|, |B|)$ as an estimate.

- We evaluate the terms from the one with the shorter postings list to the largest.

- Similar considerations can be made with the union, using $|A| + |B|$ as an estimate

COUNTING DISK ACCESSES

EXTERNAL STORAGE
MAIN MEMORY AND EXTERNAL STORAGE

- When analysing the complexity of algorithms each step is considered as having the same cost.

- This is a reasonable assumption in many cases, especially if all the data fit in the main memory.

- When accessing storage this assumption is stretched too thin: the costs can be orders of magnitude greater than accessing the main memory.

- Similar considerations holds when using the network.

TIMING OF SOME STANDARD OPERATIONS
FROM NANOSECONDS DI MILLISECONDS

execute typical instruction	1 ns
fetch from L1 cache memory	0.5 ns
branch misprediction	5 ns
fetch from L2 cache memory	7 ns
Mutex lock/unlock	25 ns
fetch from main memory	100 ns
send 2K bytes over 1Gbps network	20,000 ns
read 1MB sequentially from memory	250,000 ns
fetch from new disk location (seek)	8,000,000 ns
read 1MB sequentially from disk	20,000,000 ns

From http://norvig.com/21-days.html#answers

SOME TERMINOLOGY

- **Caching**: keeping frequently used disk data in main memory

- **Seek time:** the time needed for the disk head to move to the part of the disk where the data are located (averages 5 ms for typical disks). No data are being transferred during the seek

- **Buffer**: the part of main memory where a block being read or written is stored. Reading single bytes from disk can take as much time as reading entire blocks

COMPLEXITY: COUNTING DISK ACCESSES

- Some effects on this choice:

  - We might want to transfer data "in block". Since each read is costly, we want to read more than strictly necessary.

  - While asymptotic results are important, we might want to do a finer analysis.

  - Since the access to external storage is expensive, we might decide to do more work "in memory" to minimise the number of accesses (e.g., choice of data structure, compress and decompress data).

  - The total time of reading and then decompressing compressed data is usually less than reading uncompressed data

DICTIONARY AND INDEX COMPRESSION

WHY COMPRESSION?
SPEED IMPLICATIONS

- We can compress two things: the dictionary and the postings

- Why using compression?

  - To save disk space.

  - To keep the entire dictionary in memory.

  - To keep more data in the main memory.

  - It might be faster to read less data from disk and decompress it in memory than to read the non-compressed data.

ESTIMATION OF THE NUMBER OF TERMS

HEAPS' LAW

In a collection with T tokens the estimated size of the vocabulary is:

$$M = kT^b$$

Typical values for k are between 30 and 100

Usually, $b \approx 0.5$

HEAPS' LAW SUGGESTS THAT THE SIZE OF THE DICTIONARY INCREASES WITH THE SIZE OF THE COLLECTION

DISTRIBUTION OF TERMS

ZIPF'S LAW

The i-th most common term in a collection appears with a frequency cf_i proportional to $\dfrac{1}{i}$:
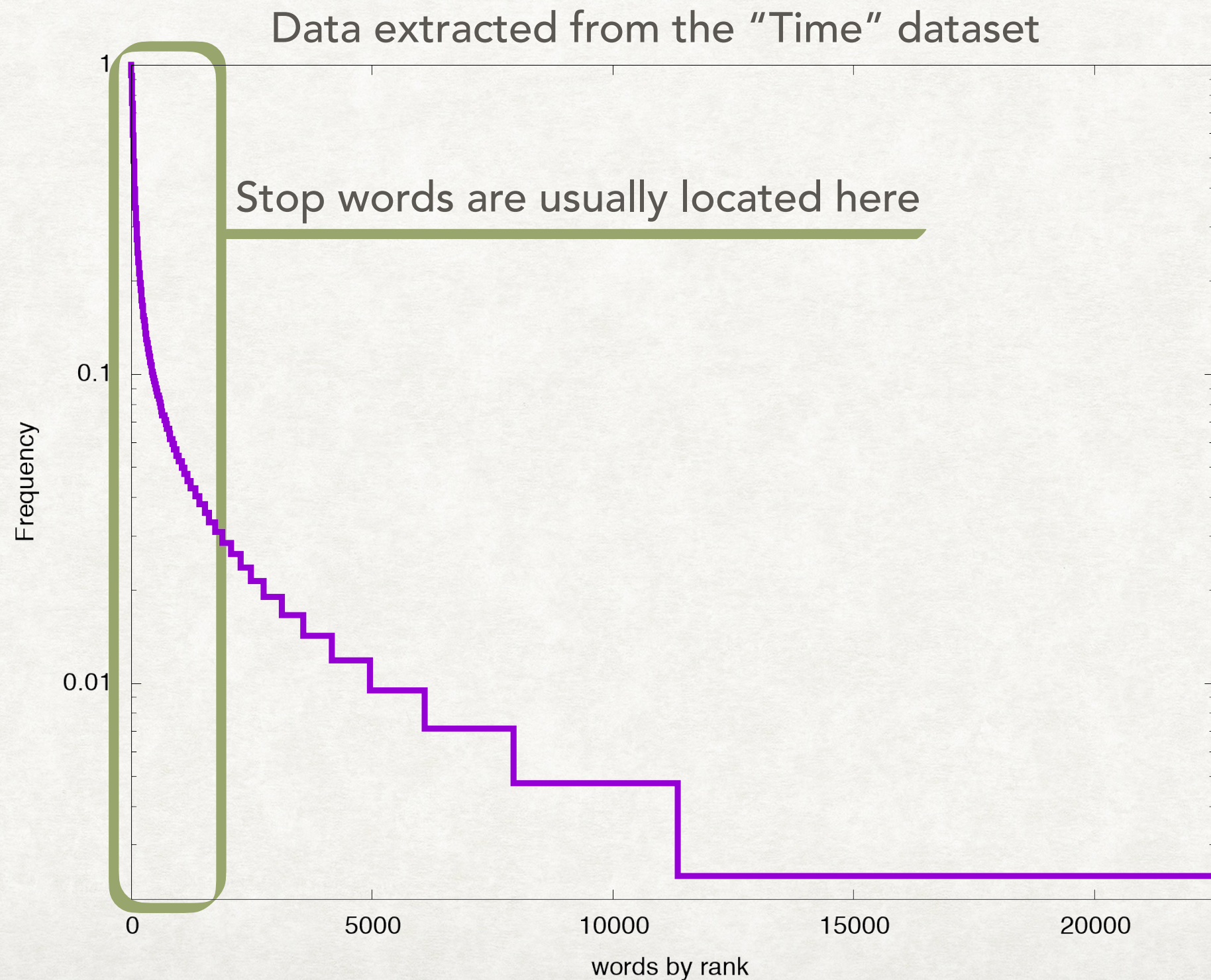
$$\text{cf}_i \propto \frac{1}{i}$$

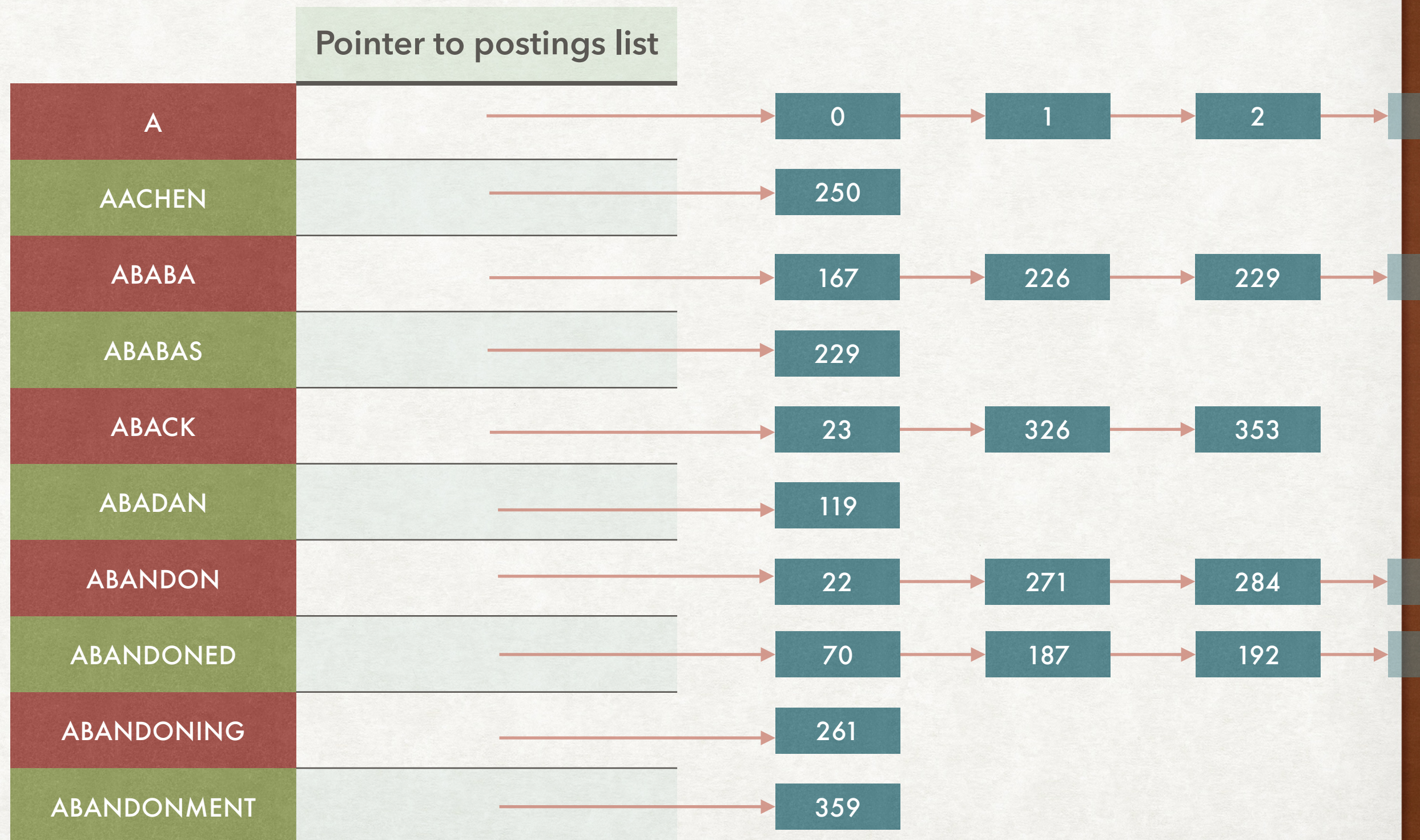In other words, frequency of terms decreases rapidly with the rank.

Equivalent formulation: $\text{cf}_i = ci^k$ for some constants c and for $k = -1$.

DISTRIBUTION OF WORDS
## FREQUENCIES OF WORDS IN A CORPUS

Data extracted from the "Time" dataset

Stop words are usually located here

DICTIONARY AS FIXED-WIDTH ENTRIES

	Pointer to postings list
A	→ 0 → 1 → 2 →
AACHEN	→ 250
ABABA	→ 167 → 226 → 229 →
ABABAS	→ 229
ABACK	→ 23 → 326 → 353
ABADAN	→ 119
ABANDON	→ 22 → 271 → 284 →
ABANDONED	→ 70 → 187 → 192 →
ABANDONING	→ 261
ABANDONMENT	→ 359

DICTIONARY AS FIXED-WIDTH ENTRIES
ADVANTAGES AND DISADVANTAGES

- Each entry consists of an entry of m characters and a pointer to the postings list.

- Words of length at most m can be stored.

- If we save a 40 characters string (i.e., $m \geq 40$) then every entry will require 40 characters, wasting a lot of space for short words.
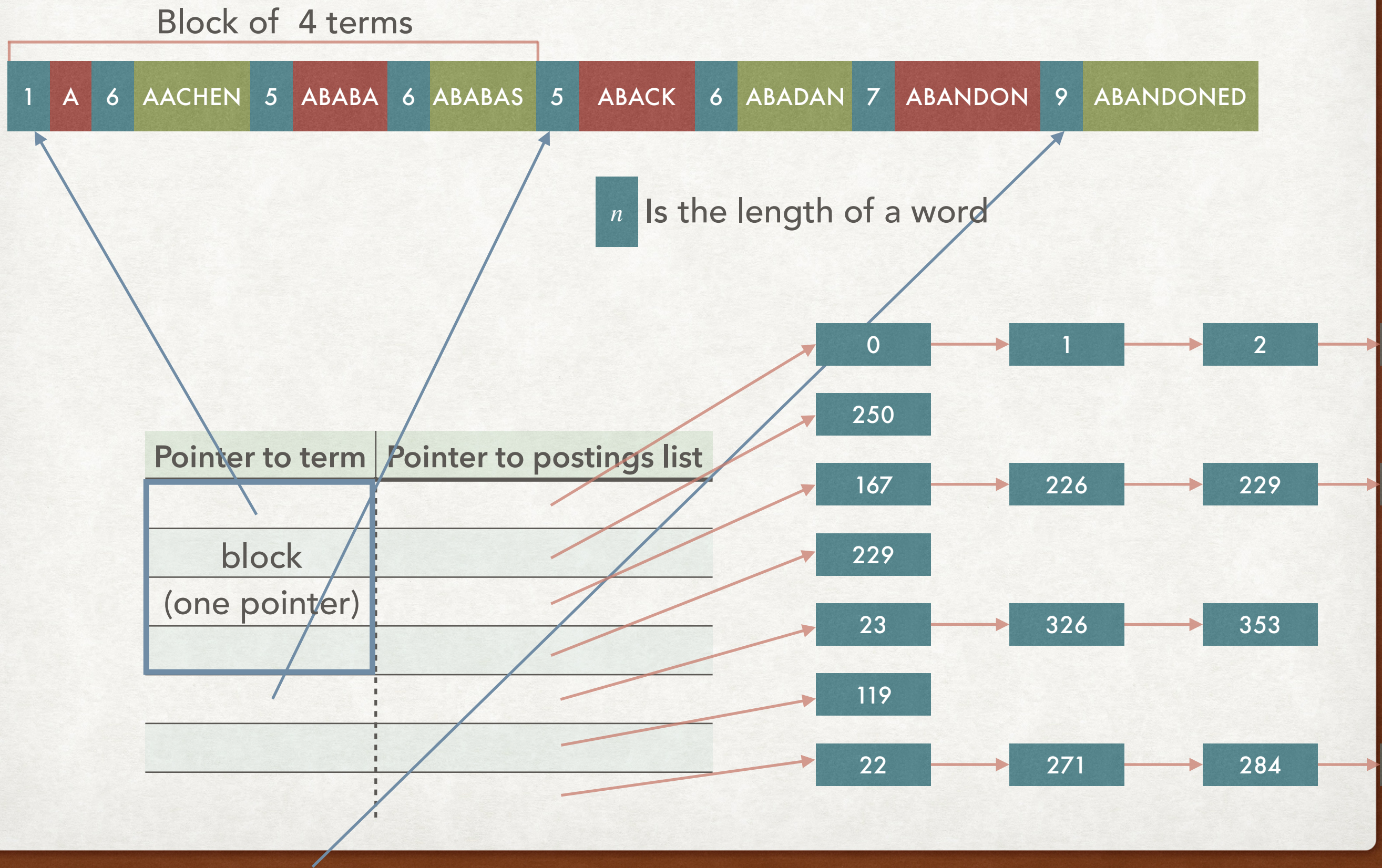
DICTIONARY AS A SINGLE STRING
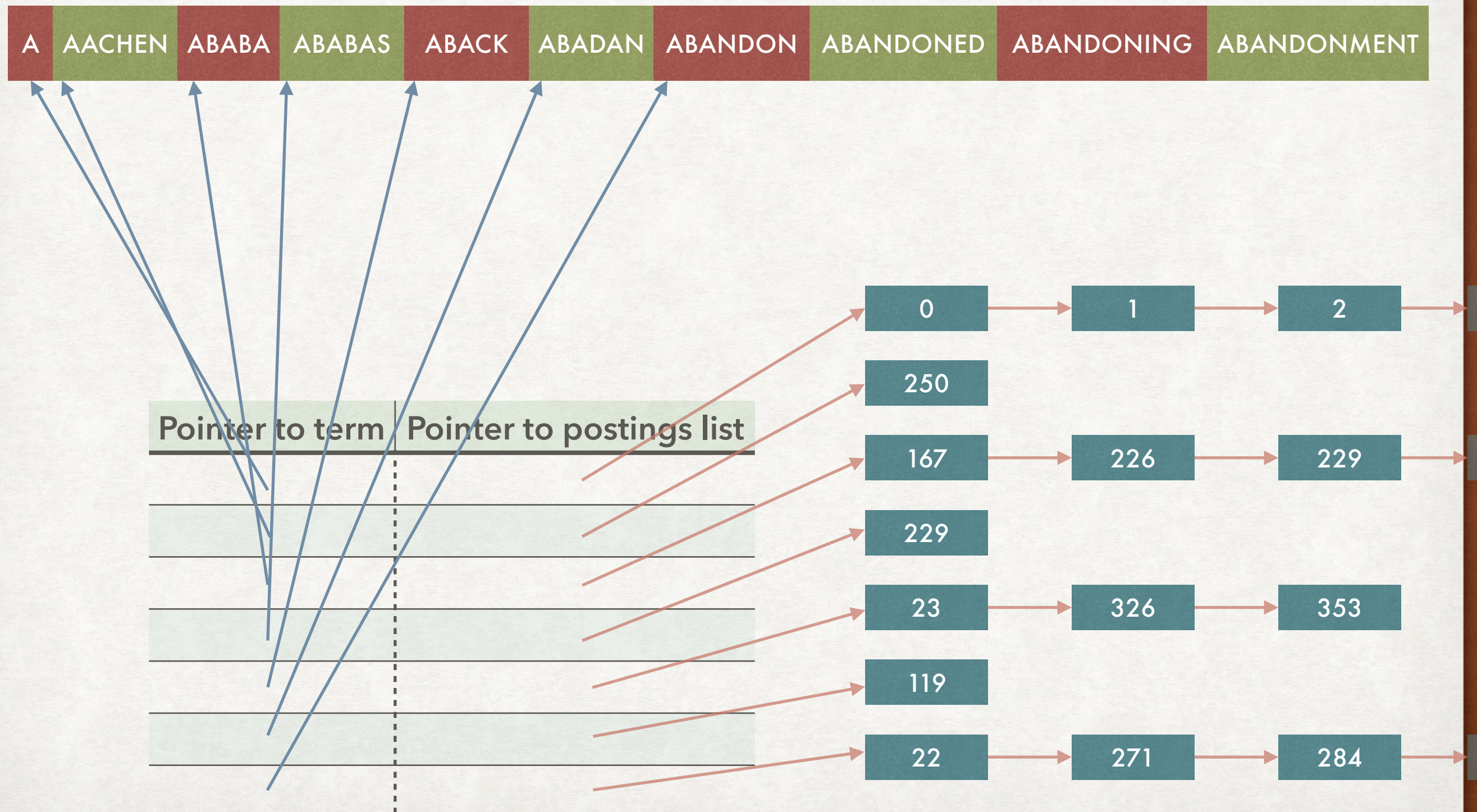
DICTIONARY AS A SINGLE STRING
DESCRIPTION AND ADVANTAGES

- Each entry consists of a pointer to the term that is part of a contiguous string and to the pointer to the postings list

- To know the end of the word it is necessary to look at the next pointer.

- There is no wasted space for the strings…

- …but it is necessary to keep an additional pointer for each entry.

- We can reduce the space used for the pointers by using *blocked storage.*

BLOCKED STORAGE

Block of 4 terms

| 1 | A | 6 | AACHEN | 5 | ABABA | 6 | ABABAS | 5 | ABACK | 6 | ABADAN | 7 | ABANDON | 9 | ABANDONED |

n Is the length of a word

Pointer to term	Pointer to postings list
block (one pointer)	

0 → 1 → 2 →

250

167 → 226 → 229 →

229

23 → 326 → 353

119

22 → 271 → 284 →

BLOCKED STORAGE
ADVANTAGES AND DISADVANTAGES

- Now only one every k entries (the block size) has a pointer.

- The value of the others entries is determined by a linear scan of the block.

- The addition of the length of the string is needed to know where a string end.

- It is a trade-off between space and access time (which is increased due to the linear scanning inside a block).

- We are still not using the fact that the words in the dictionary are ordered alphabetically.

FRONT CODING

Block of 4 terms   Block of 4 terms

| 1 | A | 6 | 1 | ACHEN | 5 | 1 | BABA | 6 | 5 | S | 5 | ABACK | 6 | 3 | DAN | 7 | 3 | NDON | 9 | 7 | ED |

Length of the prefix shared with the previous word

In red the prefix shared by each word
with the previous one in the block:

A   ABACK
AACHEN   **ABA**DAN
ABABA   **ABA**NDON
ABABAS   **ABANDON**ED

This works because the
dictionary is ordered alphabetically,
thus many words share a prefix

FRONT CODING
ADVANTAGES AND DISADVANTAGES

- Used inside a block reduces the size of the dictionary.

- Uses the fact that the dictionary is ordered.

- Requires, in addition to the linear scanning, a decoding phase.

- As before a trade-off between reducing size and incrementing the cost of retrieving a term.

POSTING FILE: ENCODING DIFFERENCES
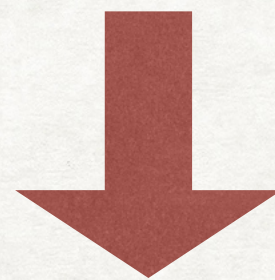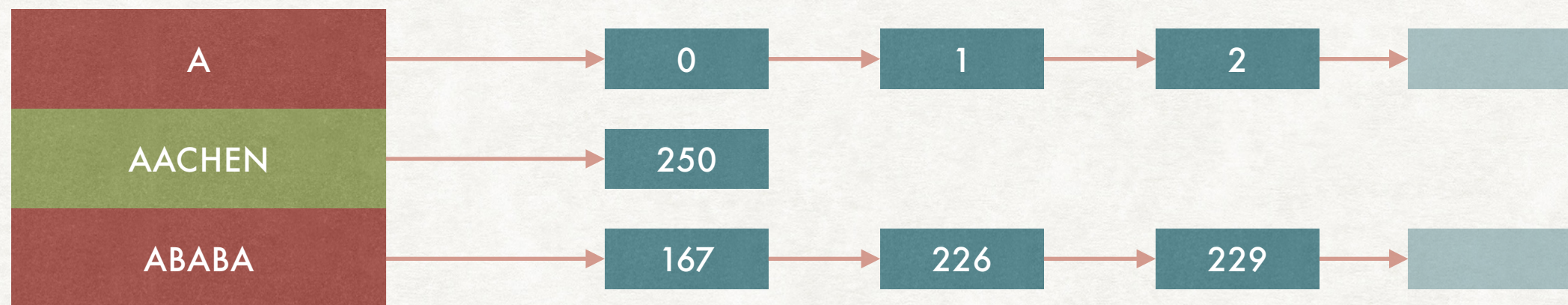
AND THE USE OF VARIABLE BYTE CODES

A	→	0	→	1	→	2	→	
AACHEN	→	250						
ABABA	→	167	→	226	→	229	→	
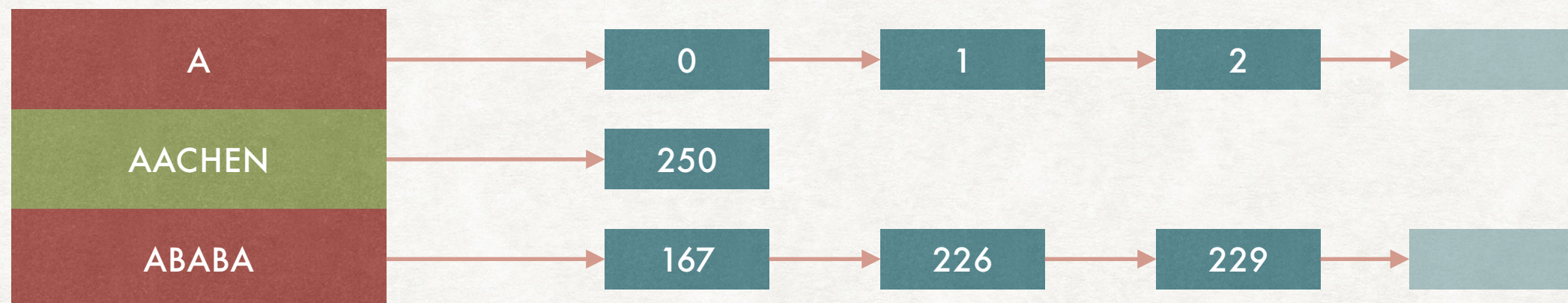
Can we use the fact that a postings list is ordered?

YES

If we have a sequence $(a_0, a_1, a_2, a_3, \ldots)$ with $a_i < a_{i+1}$ for all i, we can also encode it as a sequence of differences of consecutive terms: $(a_0, a_1 - a_0, a_2 - a_1, a_3 - a_2, \ldots)$

POSTING FILE: ENCODING DIFFERENCES
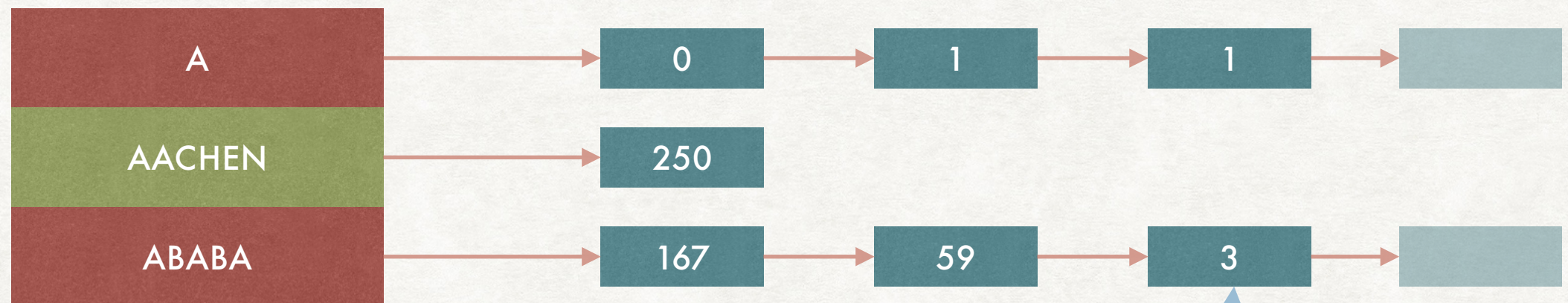## AND THE USE OF VARIABLE BYTE CODES

Encoding gaps instead of DocIDs

AND THE USE OF VARIABLE BYTE CODES

A	→	0	→	1	→	1	→	
AACHEN	→	250						
ABABA	→	167	→	59	→	3	→	
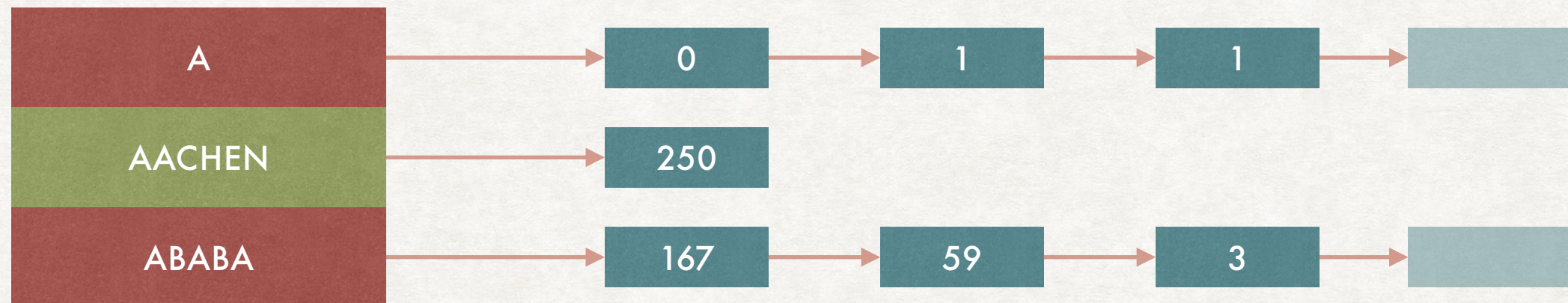
How can we recover this DocID?

$$167 + 59 + 3 = 229$$

In general, to recover the k-th DocID in list we sum all the values up to the k-th one:

$$a_0 + \sum_{i=0}^{k} (a_i - a_{i-1}) = a_0 + a_1 - a_0 + a_2 - a_1 + \ldots = a_k$$

POSTING FILE: ENCODING DIFFERENCES
MOTIVATIONS FOR ENCODING THE GAPS

- The DocID can be arbitrarily large…

- …but most of the gaps between two DocID will be small

- We can use *variable byte codes* to use less storage

- Still, recovering a DocID now is more complex.

- Most importantly, access to the list must be sequential: to recover a DocID we need to read all the previous ones.

- But the algorithms for union and intersection access the list sequentially.

VARIABLE BYTE CODES
ENCODE NUMBERS IN A VARIABLE NUMBER OF BYTES

0	1	0	0	1	0	0	0

$2^6 + 2^3 = 72$

One byte usually encodes $2^8 = 256$ different values

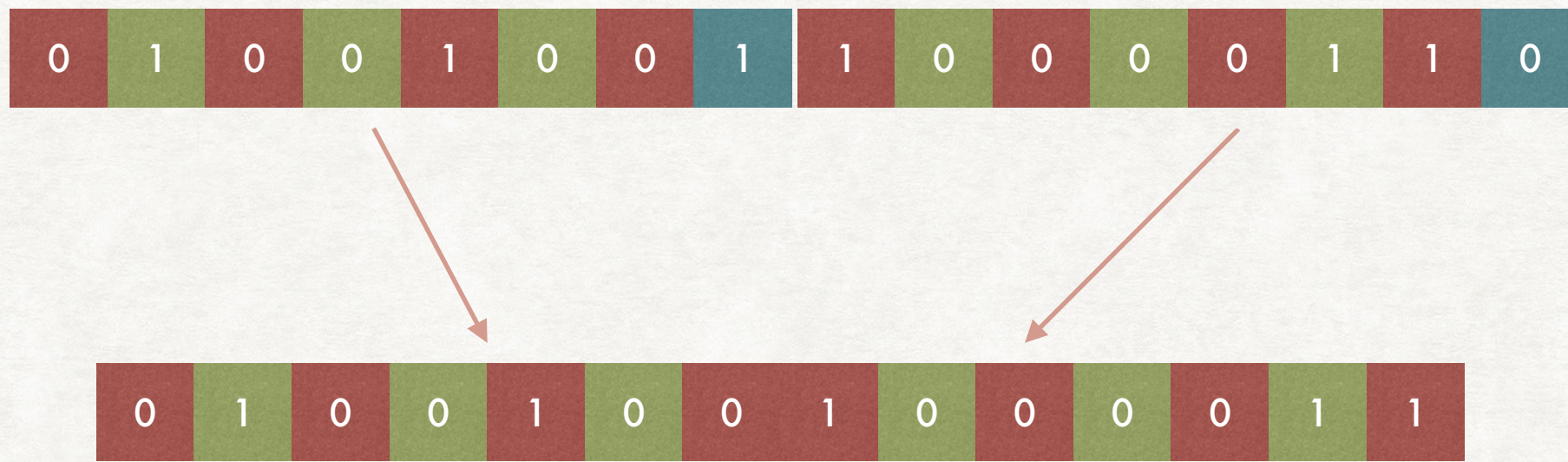| 0	1	0	0	1	0	0	0

$2^5 + 2^2 = 36$

We encode $2^7 = 128$ different values in the first seven bits the last bit is a *continuation bit*.
If it is 0 then we have completed reading the number otherwise we must continue to read the next byte

VARIABLE BYTE CODES
ENCODE NUMBERS IN A VARIABLE NUMBER OF BYTES

| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

$$2^12 + 2^9 + 2^6 + 2^1 + 2^0 = 4675$$

# of bytes	Max value
1	127
2	16383
3	2097152
4	268435456

The most frequent terms will have small gaps in their postings lists.

Hence, we can store the size of most gaps in only a few bytes

COMPRESSION + DATA STRUCTURES

Block of 2 terms

| 1 | A | 6 | AACHEN | 5 | ABABA | 6 | ABABAS | 5 | ABACK | 6 | ABADAN | 7 | ABANDON | 9 | ABANDONED | 10 | ABANDON |

Pointer to block
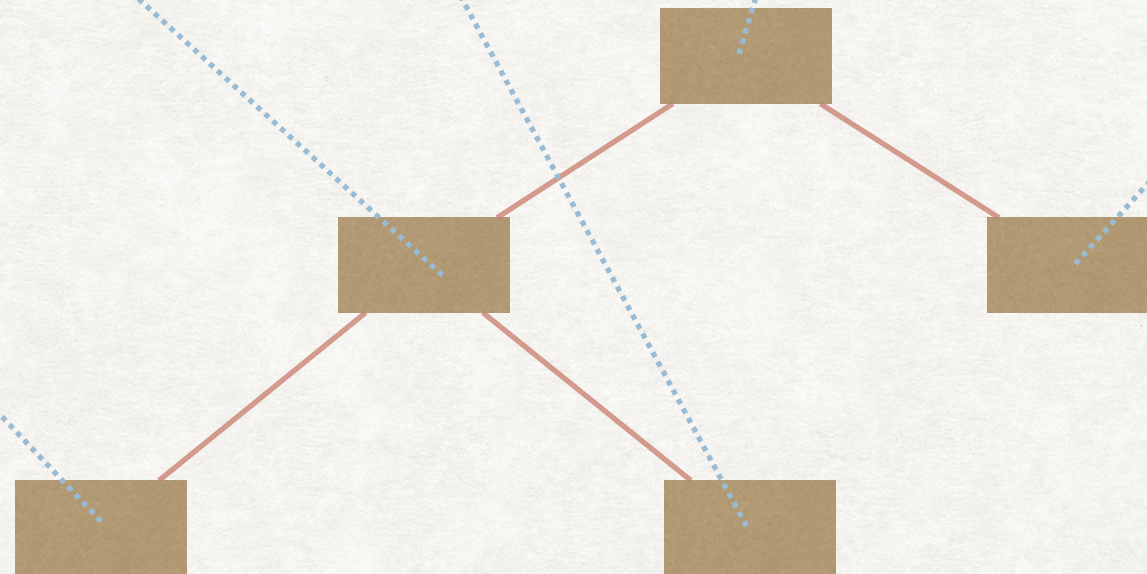
A binary search on a compressed dictionary requires a linear search inside each block

UPDATING THE INDEX

STORING POSTINGS
HOW THE POSTINGS LISTS ARE ORGANISED ON DISK

- How are the postings stored on-disk?

- One file per postings list can lead to too many files for a filesystem to manage efficiently

- One single large file containing all the postings can be better (here we select this solution)

- In reality we can have a combination of both, with multiple large files each storing part of the postings

DYNAMIC INDEXING
FOR COLLECTIONS THAT CHANGE WITH TIME

- Up to now, we have assumed that collections are static.

- They rarely are:

  - Documents come in over time and need to be inserted.

  - Documents are deleted and modified.

- This means that the dictionary and postings lists have to be modified

DYNAMIC INDEXING
FOR COLLECTIONS THAT CHANGE WITH TIME

- How can we insert new documents (or delete old ones) in an inverted index?

- We can rebuild the index:

  - Not very efficient.

  - Only useful when the number of changes is small.

  - To keep the system online while reindexing we need to keep the old index until new one is ready.
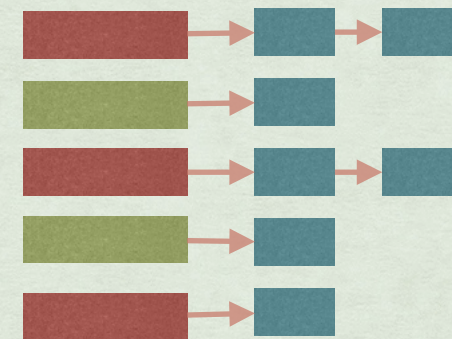
AUXILIARY INDEX
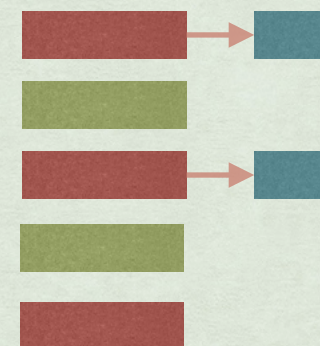## A MORE EFFICIENT SOLUTION

Queries merge
the results of
the two indices

+

Filtering using the
invalidation bit vector

Invalidation bit vector:
we save which documents
has been deleted
(one bit for each document)

Main index

**Auxiliary index:**
contains only
new document

AUXILIARY INDEX
CAN WE DO BETTER?

- When the auxiliary index becomes too big we need to merge it with the main index.

- We can improve the efficiency by keeping $\log_2(T/n)$ auxiliary indices (each one of size double the previous one) where T is the total number for postings and n the size of the smaller auxiliary index.

- This increase the complexity of all algorithms used to answer queries, so it is a trade-off.
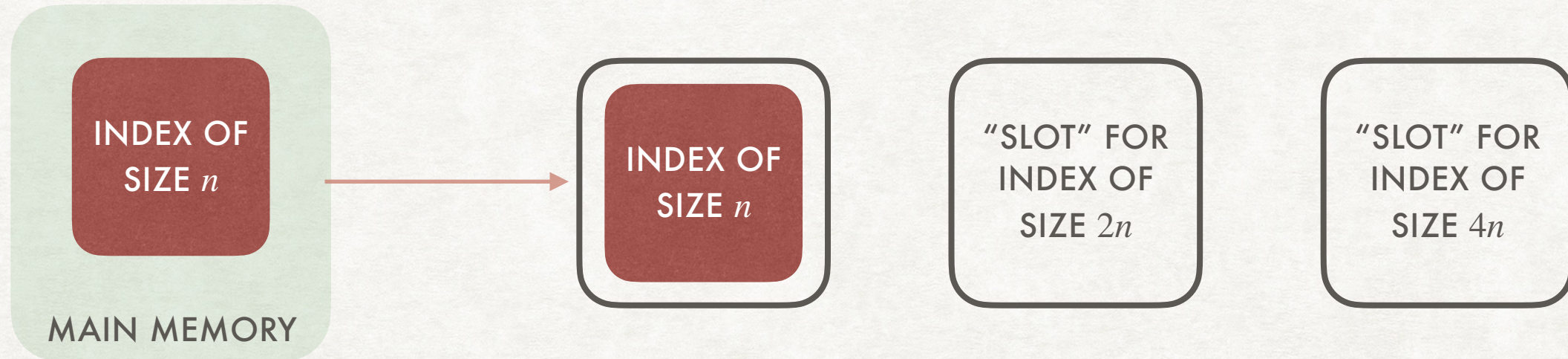
LOGARITHMIC MERGING
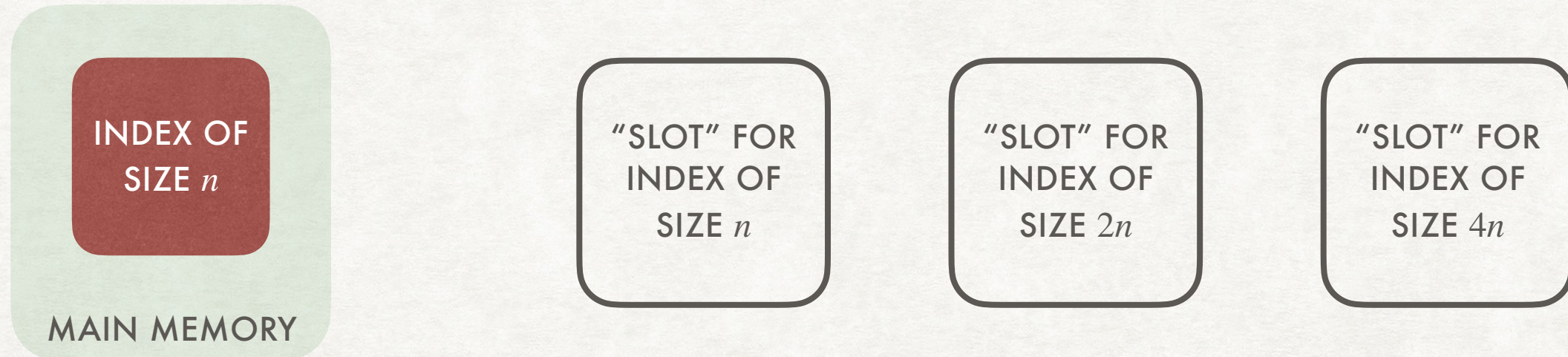
MAIN IDEAS

The smaller index is kept in memory

LOGARITHMIC MERGING
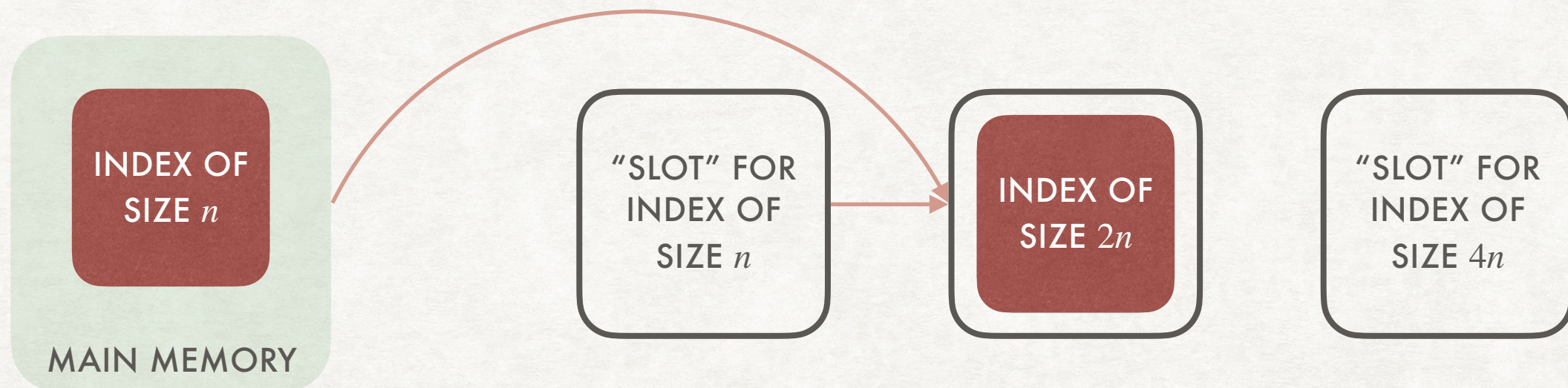
MAIN IDEAS

INDEX OF SIZE n	INDEX OF SIZE n	"SLOT" FOR INDEX OF SIZE $2n$	"SLOT" FOR INDEX OF SIZE $4n$

MAIN MEMORY

When full it is copied to disk.

A new (empty) index of size n is created in memory

LOGARITHMIC MERGING

MAIN IDEAS

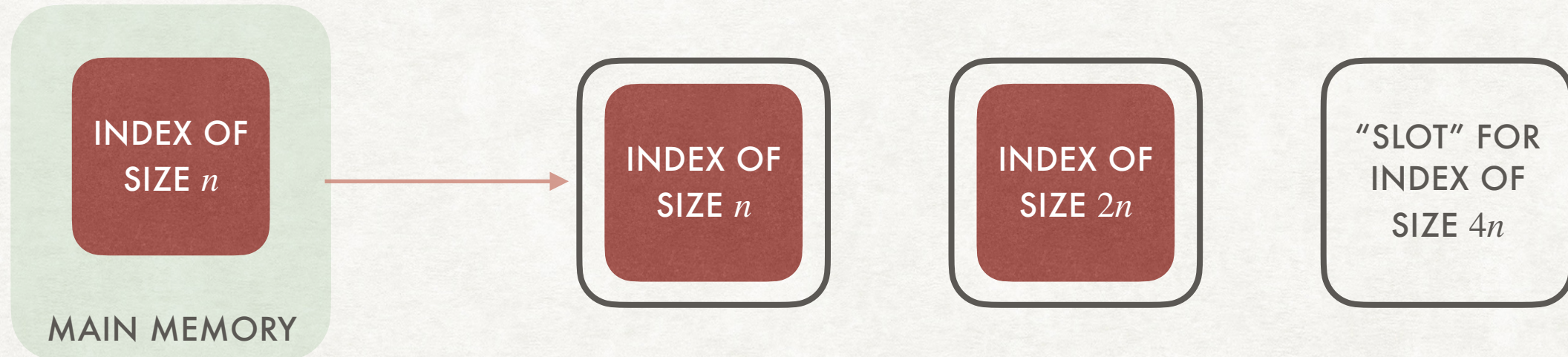|			
INDEX OF SIZE n	"SLOT" FOR INDEX OF SIZE n	**INDEX OF SIZE $2n$**	"SLOT" FOR INDEX OF SIZE $4n$

MAIN MEMORY

When it is full, since there is already an index of size n saved on disk, it is merged with it to form an index of size $2n$. A new (empty) index of size n is created in memory

LOGARITHMIC MERGING

MAIN IDEAS

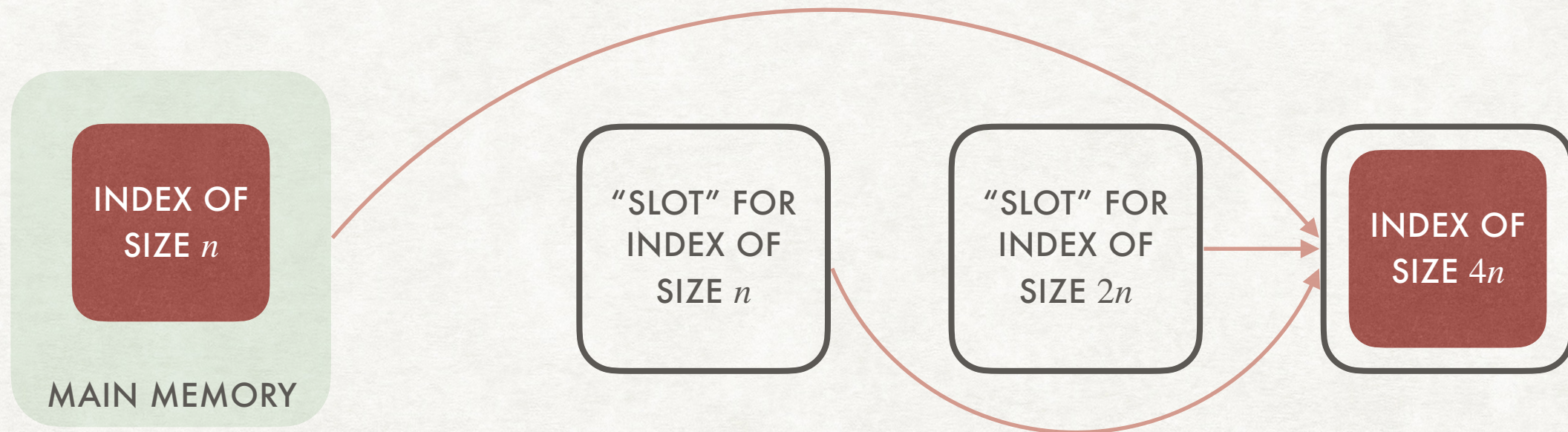|			
INDEX OF SIZE n MAIN MEMORY	**INDEX OF SIZE n**	**INDEX OF SIZE $2n$**	"SLOT" FOR INDEX OF SIZE $4n$

When full it is copied to disk. No merge is necessary (the "slot" is free)
A new (empty) index of size n is created in memory
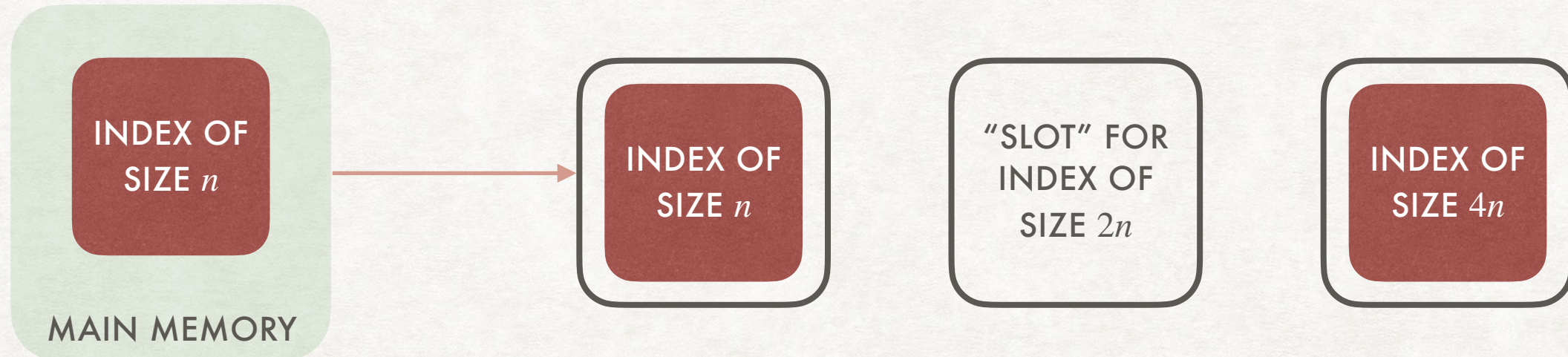
LOGARITHMIC MERGING

MAIN IDEAS

When it is full, since there is already an index of size n
saved on disk, it is merged with it to form an index of size $2n$.
Since there is already an index of size $2n$ saved on disk,
it is merged with it to form an index of size $4n$
A new (empty) index of size n is created in memory

LOGARITHMIC MERGING

MAIN IDEAS

| INDEX OF SIZE n | | INDEX OF SIZE n | "SLOT" FOR INDEX OF SIZE $2n$ | INDEX OF SIZE $4n$ |

MAIN MEMORY

When full it is copied to disk. No merge is necessary (the "slot" is free)
A new (empty) index of size n is created in memory

Most merges are of small indices, even if sometimes a series of more merge operations are necessary.