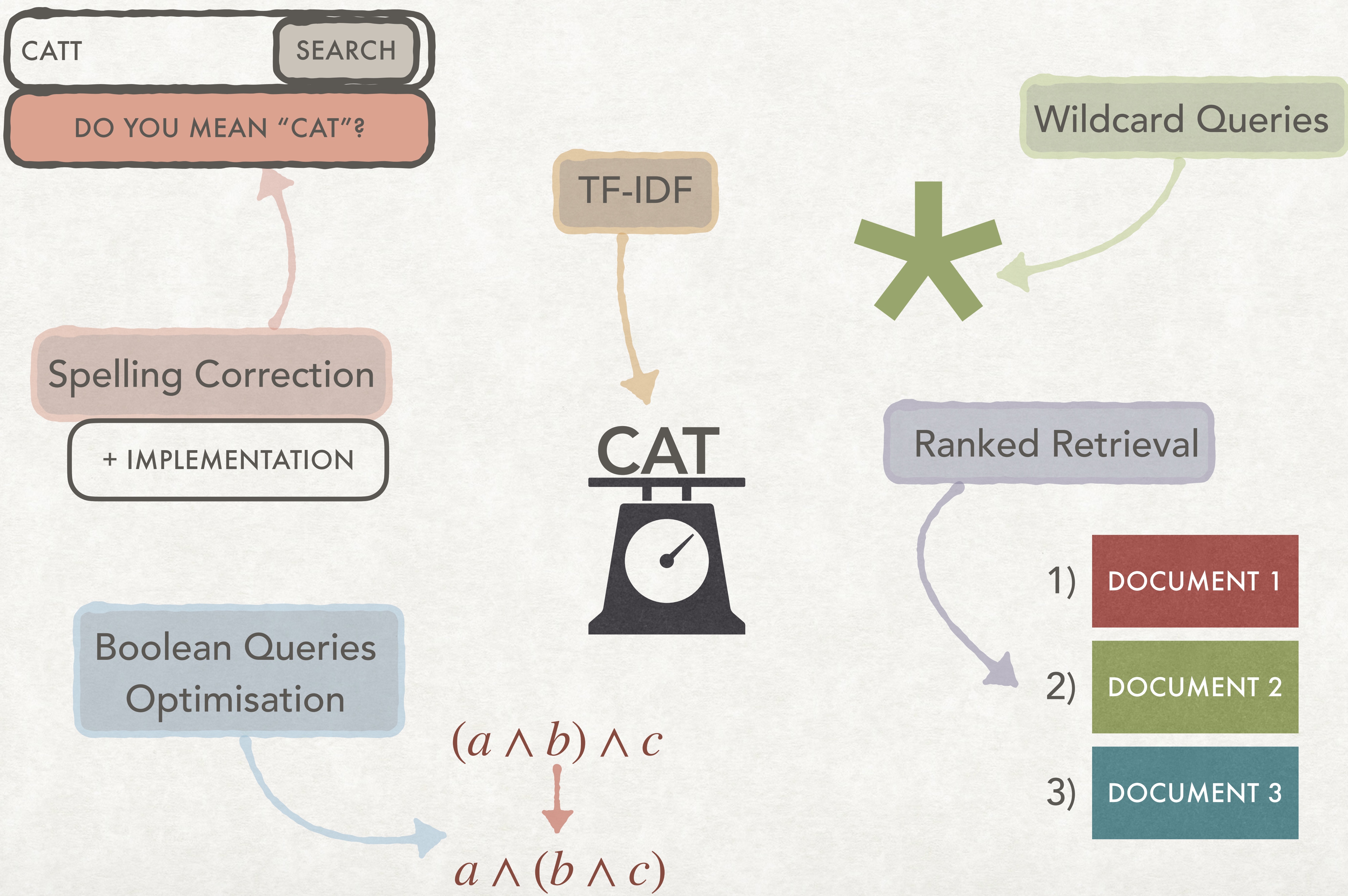
INFORMATION RETRIEVAL

Luca Manzoni
lmanzoni@units.it

Lecture 3

LECTURE OUTLINE

*MAY CONTAIN TRACES OF PEANUTS



WILDCARD QUERIES

WHAT ARE WILDCARD QUERIES?

SEARCHING AN ENTIRE SET OF WORDS

- Examples of wildcard queries:
  - **Car***: captures "**car**", "**cars**", "**cart**", "**carb**on", etc.
  - ***e*a***: captures "fl**ea**", "**ea**r", "h**ea**d", "**Eva**", etc.
- The uses might use wildcard queries when he/she:
  - Is uncertain of the spelling of a word.
  - Knows that a word has multiple spellings.
  - Want to catch all variants of term
    (which might also be "captured" by stemming).

TRAILING WILDCARDS

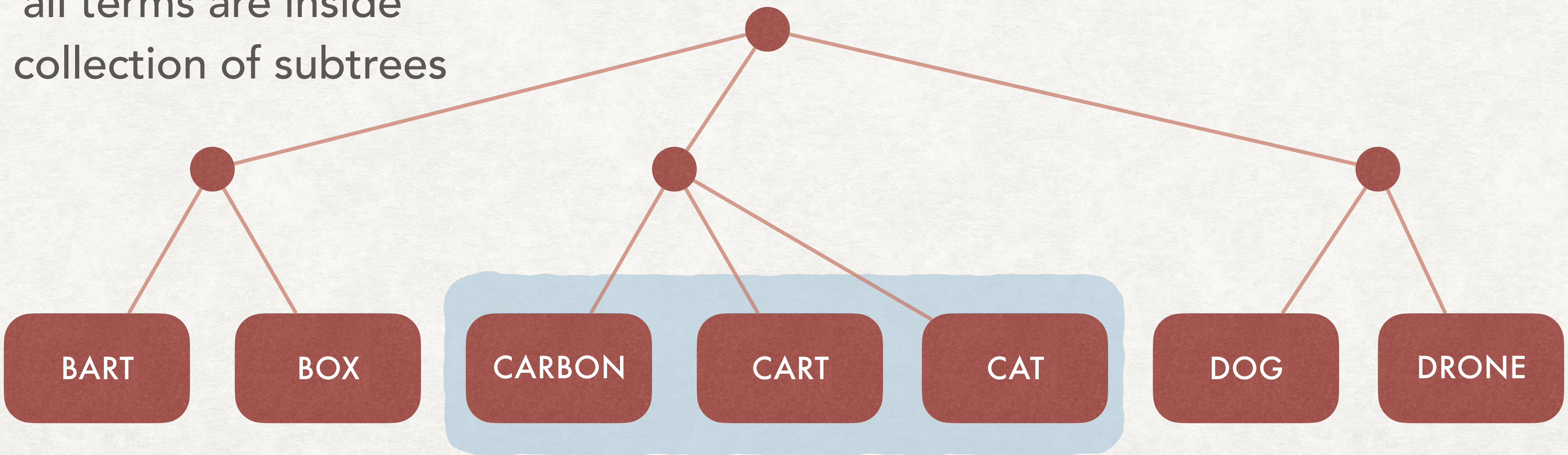
THE SIMPLEST CASE

term* — **Trailing wildcard**
there is only one wildcard
and it is at the end of the word

Let us consider the query **CA***

In a binary tree/b-tree or
a variant (as shown below)
all terms are inside
a collection of subtrees

We can retrieve the posting lists
of all of them and perform
a union of the results



LEADING WILDCARDS

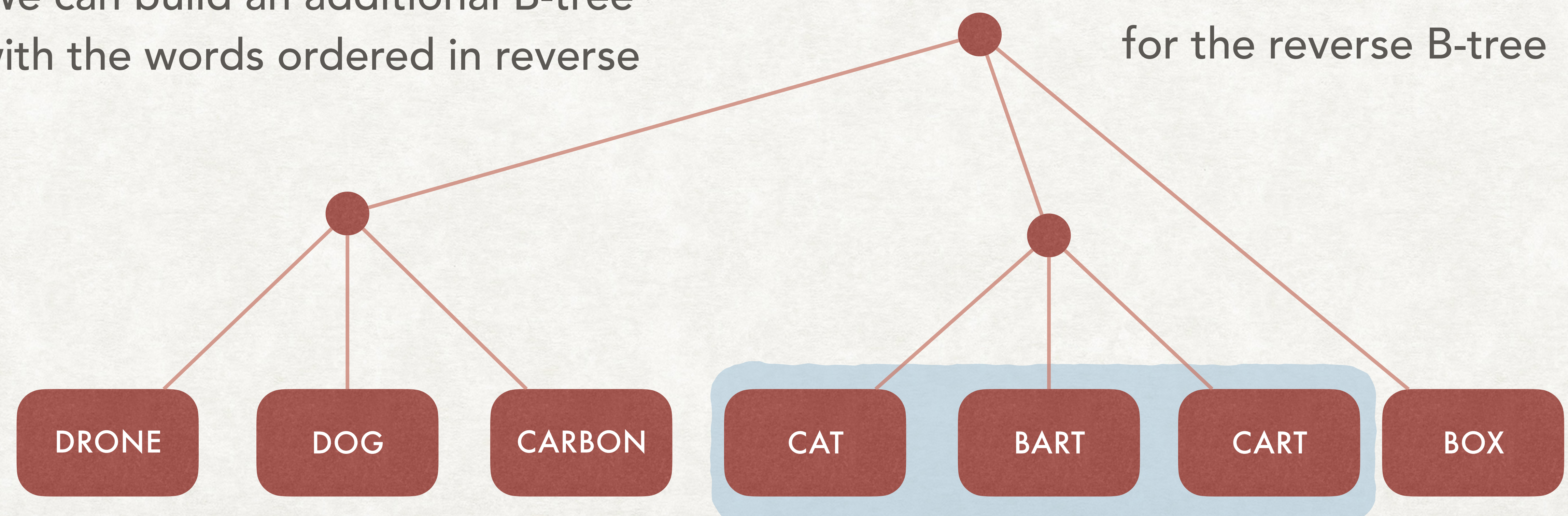
AND REVERSE (B-)TREES

***term** Leading **wildcard**
there is only one wildcard
and it is at the beginning of the word

Let us consider the query ***T**

We can build an additional B-tree
with the words ordered in reverse

Then the "leading wildcard" is
like an "inverse wildcard"
for the reverse B-tree



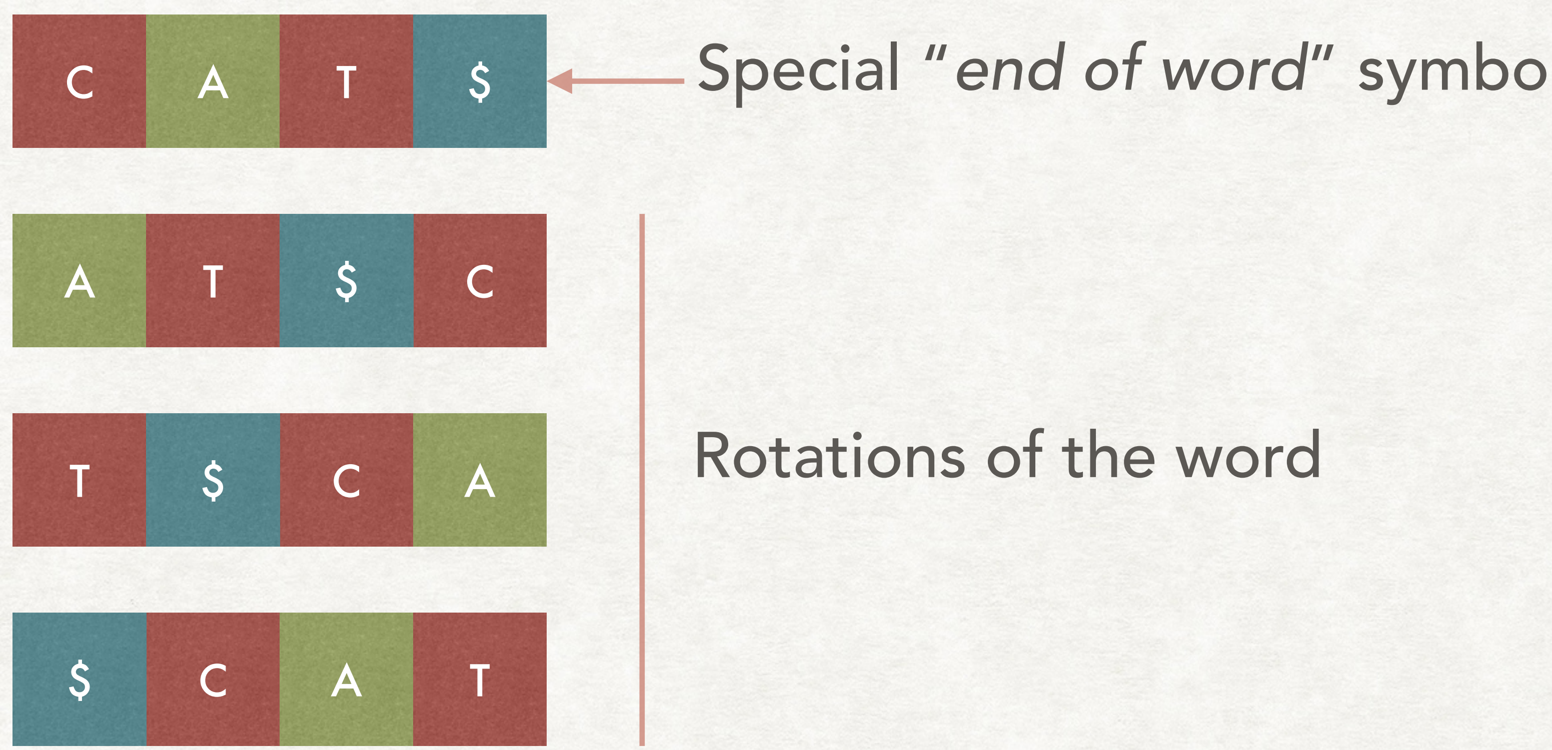
PERMUTERM INDEX

MANAGING GENERAL WILDCARD QUERIES

- Now we can answer all queries with leading and trailing wildcards.
- What about queries like "$word_1$*$word_2$"?
- Can we reformulate the problem of "one wildcard" as a leading or trailing wildcard problem?
- Yes, using the "permuterm index"
- We can also extend the solution to queries with more than one wildcard.

PERMUTERM INDEX

MANAGING GENERAL WILDCARD QUERIES



We insert all the rotations of the word (including the "end of word") in the dictionary.

All the rotations of the same word points to the **same** postings list

PERMUTERM INDEX

MANAGING GENERAL WILDCARD QUERIES

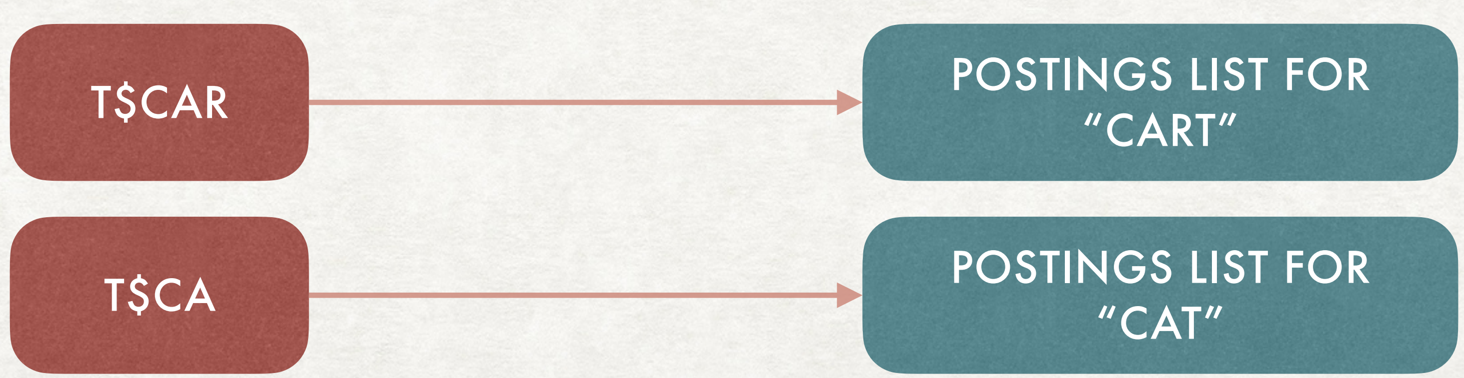
Our query: **C*T**

C*T$ Put the “end of word” at the end

T$C* Rotate the word to have the wildcard at the end

We can have a trailing wildcard, that we know how to solve!

Term in the dictionary



PERMUTERM INDEX

WHAT ABOUT MULTIPLE WILDCARDS?

Our query: ***A*T**

***A*T$** Put the "end of word" at the end

***T$** Consider the more general query where everything between the first and last wildcard is "folded" inside a single wildcard

T$* Rotate to have a trailing wildcard query

BART ~~BORT~~ CART CAT

Collect all the terms matching the simplified query

Scan the list to remove the ones **not** matching the original query

PERMUTERM INDEX

ADVANTAGES AND DISADVANTAGES

- We can now answer wildcard queries with any number of wildcards!
- Even if for more than one wildcard a linear scan of a list of terms is still needed.
- There is an interesting interplay between the algorithm that we use and the data structures employed.
- The main problem of permuterm indices: the amount of space needed to store all rotations of a word. A word with n letters will have $n+1$ rotations (due to the "end of word" symbol).

K-GRAM INDEXES

ANOTHER WAY TO MANAGE WILDCARD QUERIES

k-gram: a sequence of k characters

DRONE

DRO
RON
ONE

3-grams of "DRONE"

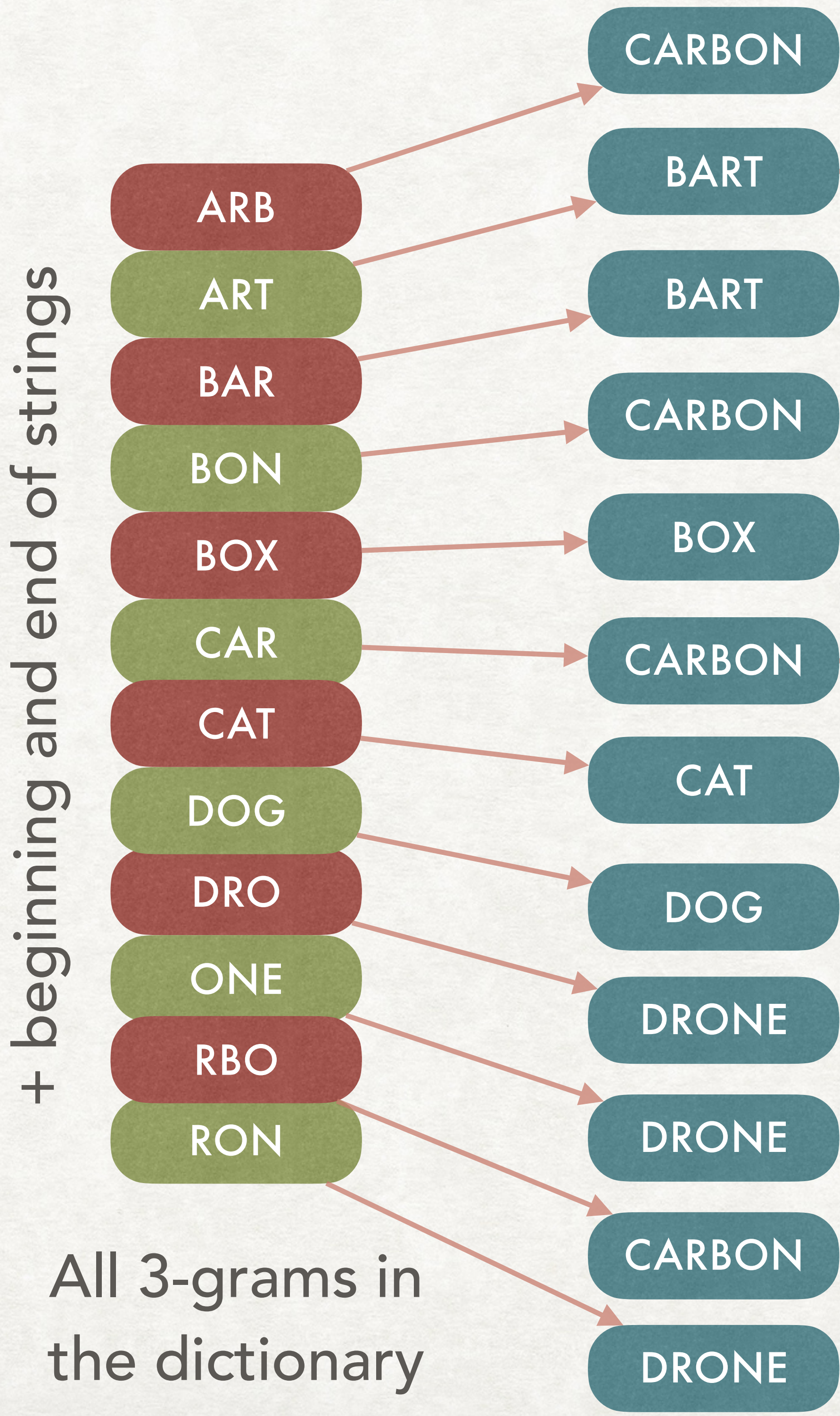
$DR
DRO
RON
ONE
NE$

We actually use the "$" symbol to denote the beginning and end of the word

We create a dictionary of k-grams obtained from all the terms

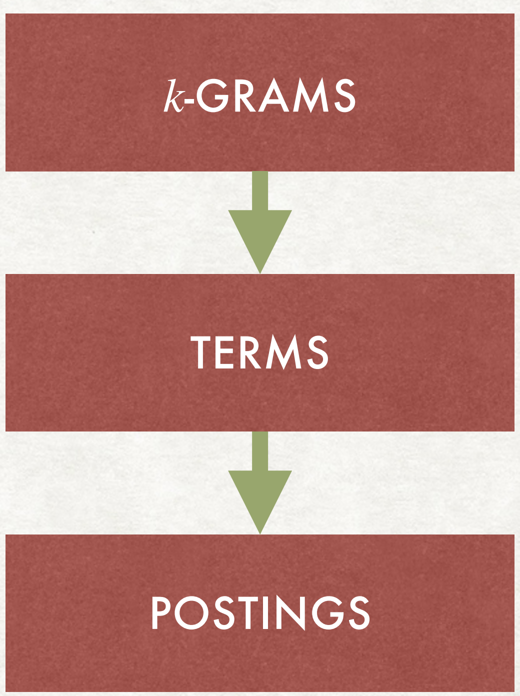
K-GRAMS INDEXES

AN EXAMPLE



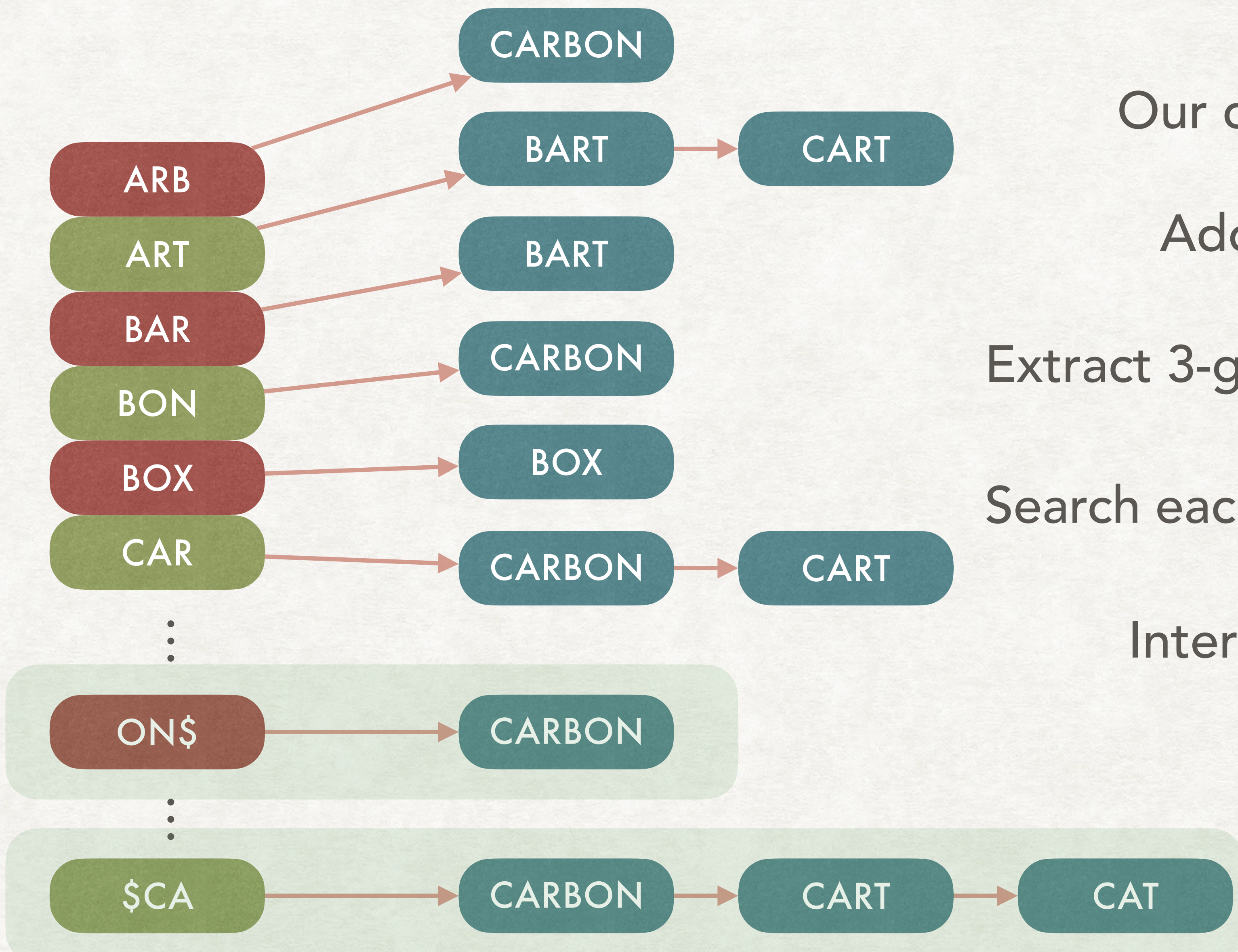
Each 3-gram points to the list of terms containing it.

The current structure of the system:



K-GRAMS INDEXES

HOW TO USE THEM TO ANSWER QUERIES



Our query: **CA*ON**

Add "$": **$CA*ON$**

Extract 3-grams: **$CA ON$**

Search each one of the 3-grams

Intersect the results: CARBON

K-GRAMS

ADVANTAGES AND DISADVANTAGES

- They allow to answer wildcard queries
- A filtering step might still be needed:
  - Query: **GOL***
  - 3-grams: **$GO** and **GOL**
  - Possible element of the intersection: GOGOL, which does not respect the original query.
- k-grams can also be used to help in spelling correction

SPELLING CORRECTION

BASICS OF SPELLING CORRECTION

- There are two main principle behind spelling correction:
  - If a word is misspelled, then find the nearest one.
  - If two or more words are tied (or nearly tied) select the most frequent word.
- Which means that we need to define what "nearest" means.
- Two main approaches:
  - Edit (or Levenshtein) distance
  - k-grams overlap

EDIT DISTANCE
AKA LEVENSHTEIN DISTANCE

- The idea is that the distance between two words w_1 and w_2 is given by the *smallest* number of edit operations that must be performed to transform w_1 in w_2.
- The possible edit operations are:
  - *Insert* a character in a string (e.g, from **brt** to **bart**).
  - *Delete* a character from a string (e.g., from **caar** to **car**).
  - *Replace* a character in a string (e.g., from **arx** to **art**).

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

- How to compute efficiently the edit distance?
- There is a classical dynamic programming algorithm the runs in time $O(|w_1| \times |w_2|)$, where $|\cdot|$ denotes the length of a word.
- We are now going to detail the idea formally and then with an example

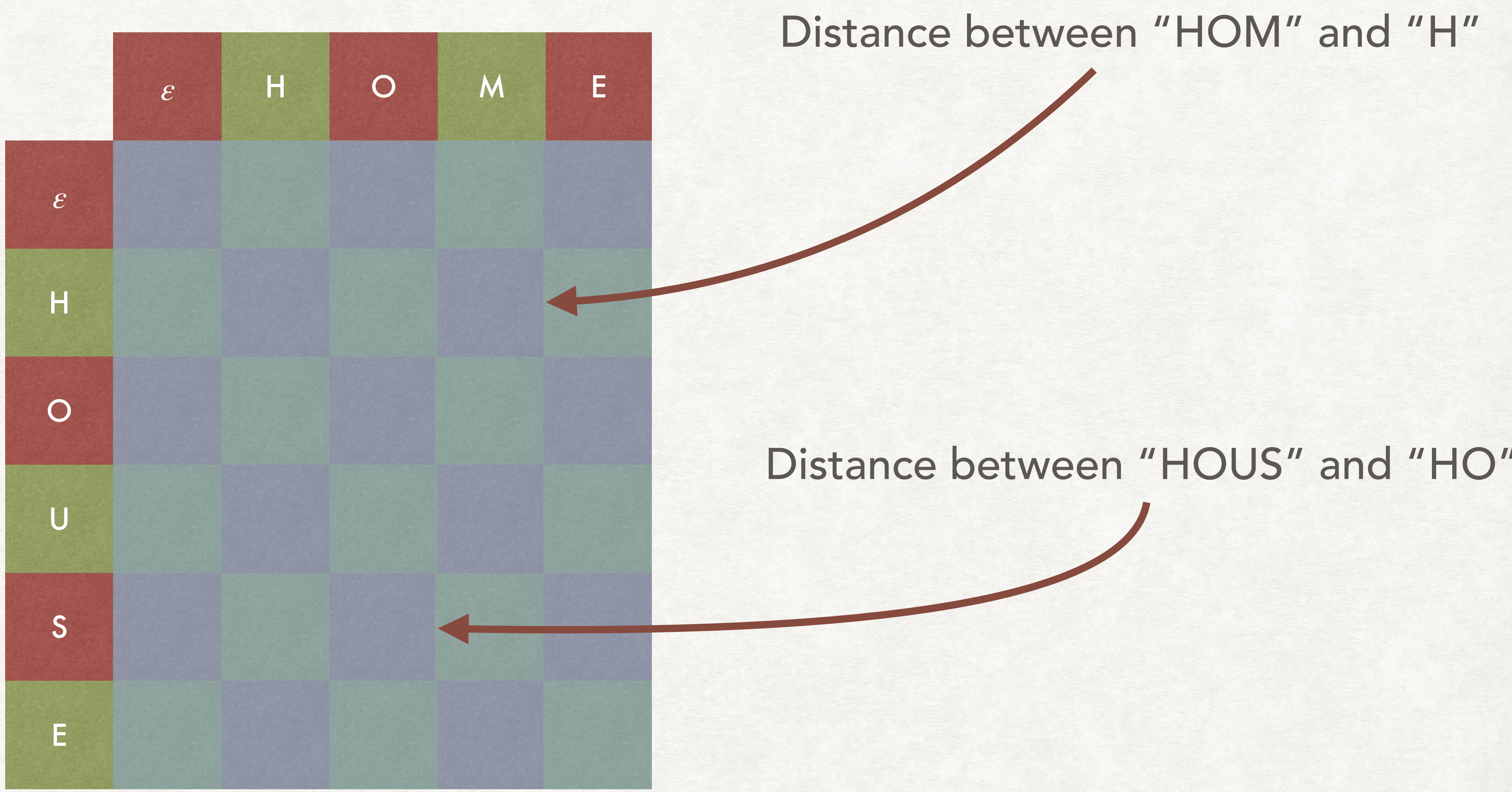
COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

- Let $w_1 = v_1 a$ and $w_2 = v_2 b$ with a, b characters and v_1, v_2 words.
- The main idea is that you know the edit distance $d(w_1, w_2)$ between w_1 and w_2 is the minimum between:
  - $d(v_1, v_2) + 1$ if $a \neq b$ (i.e., we replace a by b)
  - $d(v_1, v_2)$ if $a = b$ (i.e., the distance does not increase)
  - $d(v_1, v_2 b) + 1$ (i.e., we remove a from the first word)
  - $d(v_1 a, v_2) + 1$ (i.e., we add b in the second word)

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING



COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1				
O	2				
U	3				
S	4				
E	5				

The distance between a word and an empty string is simply the length of the word

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1	0			
O	2				
U	3				
S	4				
E	5				

This is the minimum between:

$$d(\varepsilon, H) + 1 = 2$$

$$d(H, \varepsilon) + 1 = 2$$

$$d(\varepsilon, \varepsilon) + 0 = 0$$

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1	0	1		
O	2				
U	3				
S	4				
E	5				

This is the minimum between:

$d(HO, \varepsilon) + 1 = 3$

$d(H, H) + 1 = 1$

$d(H, \varepsilon) + 1 = 2$

COMPUTING THE EDIT DISTANCE

WITH DYNAMIC PROGRAMMING

	ε	H	O	M	E
ε	0	1	2	3	4
H	1	0	1	2	3
O	2	1	0	1	2
U	3	2	1	1	2
S	4	3	2	2	3
E	5	4	3	3	2

We compute each element of the matrix

The result is in the bottom right corner of the matrix

Computing the value for one cell requires constant time…

…and there are $O(|w_1| \times |w_2|)$ cells

THE EDIT DISTANCE

ADVANTAGES AND DISADVANTAGES

- By computing the edit distance we can find the set of words that are the closest to a misspelled word.
- However, computing the edit distance on the entire dictionary can be too expensive.
- We can use some heuristics to limit the number of words, like looking only at words with the same initial letter (hopefully this has not been misspelled).
- Or we can use k-grams to retrieve terms with low edit distance from the misspelled word.

K-GRAM INDEXES

THIS TIME FOR SPELLING CORRECTION

- We can try to retrive terms with “many” k-grams in common with a word.
- We hypothesise that having “many” k-grams in common is indicative of a low edit distance.
- This might not be true. Consider the the word “*cata*”:
  - it has all of its 2-grams in common with “*catastrophic*”, but it is not a “good” correction.
  - “*cats*”, which has has fewer 2-gram in common, is a more reasonable correction

THE JACCARD COEFFICIENT

MEASURING THE OVERLAP OF TWO SETS

The Jaccard coefficient of two sets A and B is defined as:

$$\frac{|A \cap B|}{|A \cup B|}$$

We can use the Jaccard coefficient to select the terms obtained by looking at the k-grams in common.

In this "cata" and "catastrophe" have a Jaccard coefficient of $3/10$, while "cata" and "cats" of $1/2$.

CONTEXT-SENSITIVE CORRECTION

SOMETIMES CONTEXT IS IMPORTANT

- Sometimes all the words of a query are spelled correctly…
  …but one is actually the wrong word.
- Consider "Flights *form* Malpensa".
  The correct query should have been "Flights *from* Malpensa".
- How can we mitigate the problem?
- Substitute one at a time the words of the query with the most similar in the dictionary, perform the modified queries and look at the variants with most results.
- Can be expensive, but some heuristics can help (e.g., looking at common pairs of words)

PHONETIC CORRECTION

WHEN A WORD IS WRITTEN "AS IT SOUNDS"

- Sometimes the user does not know how to spell a word…
- …so he/she tries to write it based on the sound…
- …and gets the result wrong.
- We can try to correct this kind of error by using specific algorithms that tries to put similar-sounding words in the same equivalence class.
- These algorithms are language-specific (or, at least, non universal).
- For English we will see the **Soundex** algorithm.

SOUNDEX ALGORITHM

MARSHMALLOW	Keep the first letter unchanged through the algorithm
M0RS0M0LL00	Change all occurrences of A, E, I, O, U, H, W, Y to 0
M0620504400	Convert the letters according to the following table: 1) B, F, P, V 2) C, G, J, K, Q, S, X, Z 3) D,T 4) L 5) M, N 6) R
M6254400000	Remove all occurrences of 0 and pad the string with 0
M625	Return the first four positions (1 letter, 3 digits)

THE SOUNDEX ALGORITHM

HOW TO USE IT

- We can search for words with the same "phonetic hash" as the ones in the query.
- The mains ideas that make the Soundex algorithm work are:
  - Vowels are seen as interchangeable.
  - Consonants are assigned to different equivalence classes depending on how they sound.
- The algorithm, however, is not perfect. There can be words that sound similar with different "phonetic hashes" and vice versa.

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

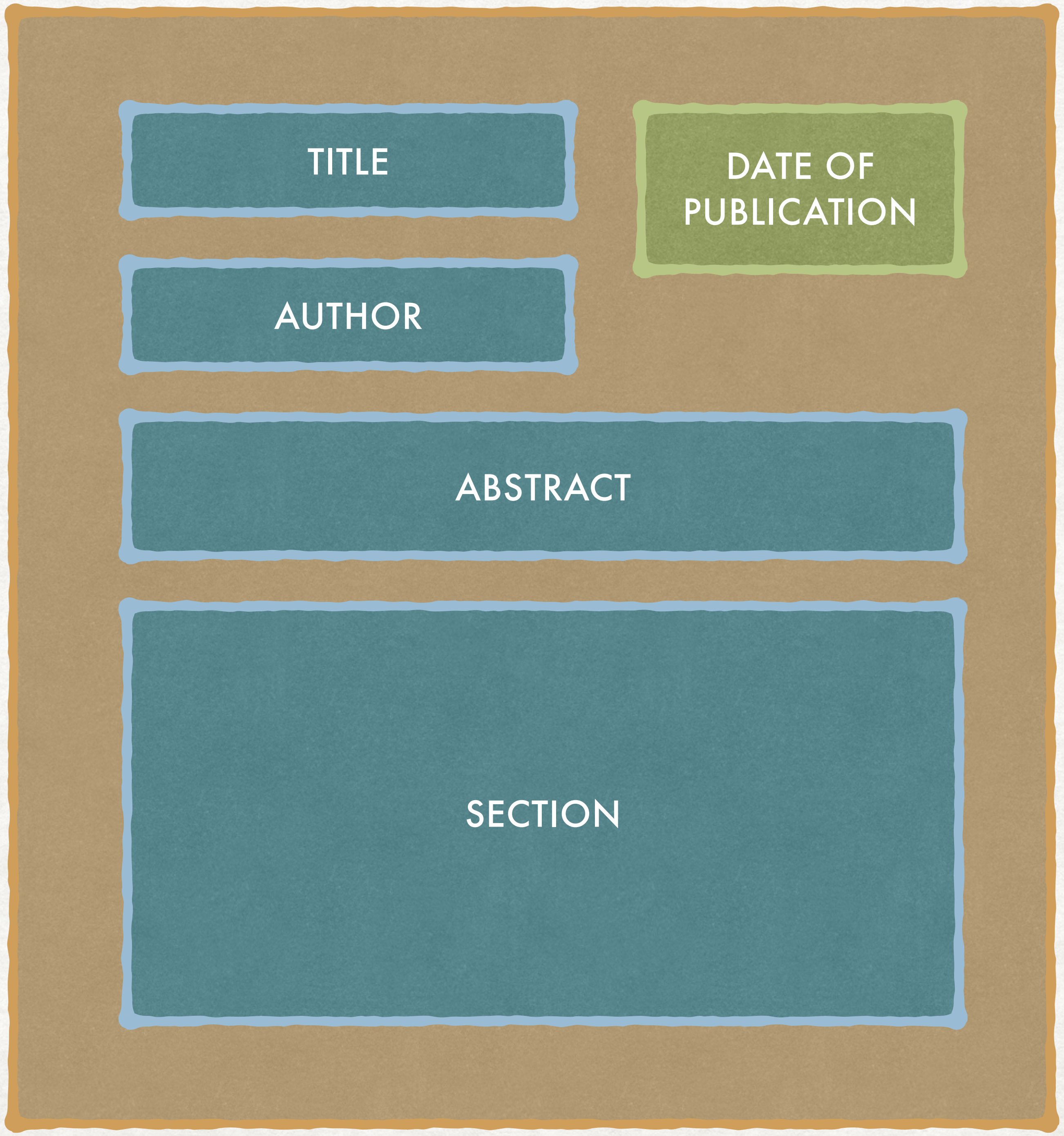
RANKED RETRIEVAL

MOTIVATIONS

- Until now we have returned all documents matching a Boolean query as a set.
- If many documents are returned then it might be important to rank them according to how relevant they are.
- A first way of ranking them is to "split" a document according to some structure and then weight different zones in different ways.
- We will then see how we can extend the idea of adding weights also to the terms of a document.

DOCUMENT STRUCTURE

METADATA, FIELDS, AND ZONES



- A text may have associated **metadata**.
- Some of them can be **fields**, with a set of values that can be finite, like publication dates.
- Others might be zones, arbitrary areas of free-form text (e.g., abstract, section, etc.).

PARAMETRIC INDEXES

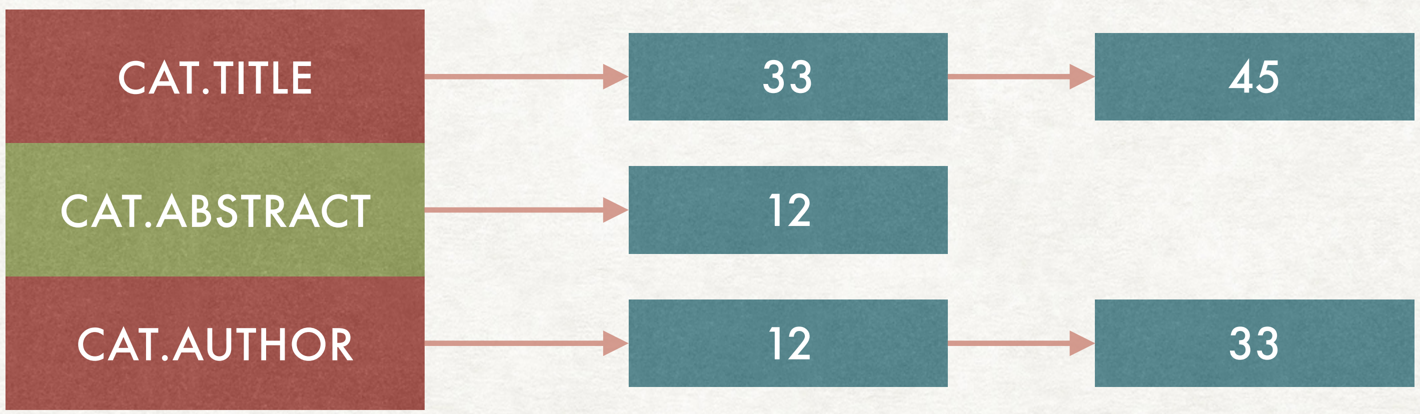
SEARCHING INSIDE FIELDS

- To allow for searching inside the fields we might want to build additional indexes, called **parametric indexes**.
- A parametric index can be thought as a standard index that only has information about a field (e.g., all the dates).
- If a query asks for “cat” in the title and “dog” inside the document we will retrieve the posting lists for dog from the ”standard” index e “cat” from the parametric index for the title.
- The operations of union and intersections works as usual.

ZONE INDEXES

POSSIBLE APPROACHES

Separate inverted index for each zone



Single inverted index in which the zones are part of the postings

WEIGHTED ZONE SCORING

AN ADDITIONAL USE FOR ZONES

- We now have a way of searching inside different parts of a document…
- …but different parts might carry different importance: e.g., a title vs inside the main text.
- We can rank retrieved documents according to where the term is found inside the document.
- We can do this via **weighted zone scoring** (also called **ranked Boolean retrieval**).

SCORING FUNCTION
DEFINITION

- Consider a pair (q, d) of a query q and a document d.
- A **scoring function** associates a value in $[0,1]$ to each pair (q, d).
- Higher scores are better.
- Suppose that a document has ℓ zones.
- Each zone has a weight $g_i \in [0,1]$ for $1 \leq i \leq \ell$.
- The weights sums to one:

$$\sum_{i=1}^{\ell} g_i = 1$$

SCORING FUNCTION

PART II

- Given a query q let s_i be defined as
$$s_i = \begin{cases} 1 & \text{if } q \text{ matches in zone } i \\ 0 & \text{otherwise} \end{cases}$$

- Actually, s_i can also be defined to be any function that maps "how much" a query matches in the i-th zone.

- The weighted zone score in then defined as:
$$\sum_{i=1}^{\ell} g_i s_i$$

WEIGHTED ZONE SCORING

A SIMPLE EXAMPLE

Query: **CAT**

	Title: 0.5	Author: 0.2	Body: 0.3	
TITLE: LIFE OF A CAT AUTHOR: JAMES CAT ONCE THERE WAS A CAT...	0.5	0.2	0.3	**1**
TITLE: DOGS AND OTHER PETS AUTHOR: ANONYMOUS DOGS AND CATS ARE THE...	0	0	0.3	**0.3**
TITLE: ORCHARDS MANAGEMENT AUTHOR: JAMES CAT THE MANAGEMENT OF ORCHARDS...	0	0.2	0	**0.2**

LEARNING WEIGHTS

OR SETTING THEM MANUALLY

- The new problem is now to find how to set the weights for the different scores.
- One possibility is to ask a domain expert.
- Another possibility is to have users label documents relevant or not with respect to a query…
- …and trying to learn the weights using the training data.
- In addition to the binary classification (relevant or not) more nuanced classifications might be used.

THE TRAINING SET

Example	DocID	Query	In the title	In the body	Judgment
e1	43	LISP	1	1	Relevant
e2	43	BASIC	1	0	Relevant
e3	76	LISP	0	1	Non-relevant
e4	76	BASIC	0	1	Relevant
e5	87	SMALLTALK	1	1	Relevant
e6	87	APL	1	0	Non-relevant

COMPUTING THE ERROR

HOW TO DECIDE IF OUR WEIGHTS WORKS

With only two zones, site score is computed as:

$$\text{score}(d, q) = g \cdot s_{\text{title}} + (1 - g) \cdot s_{\text{body}}$$

Since we know the queries and the real relevance of the documents in the training set we can compute the output that a weight g would give:

$$\text{score}(43,\text{LISP}) = g \cdot 1 + (1 - g) \cdot 1$$

$$\text{score}(43,\text{BASIC}) = g \cdot 1 + (1 - g) \cdot 0$$

$$\text{score}(76,\text{LISP}) = g \cdot 0 + (1 - g) \cdot 1$$

$$\vdots$$

COMPUTING THE ERROR

HOW TO DECIDE IF OUR WEIGHTS WORKS

If we decide that relevant is 1 and non-relevant is 0
we can compare the real score with the computed one
and compute an error:

$$\text{Err}(g, e1) = (1 - \text{score}(43,\text{LISP}))^2$$

$$\text{Err}(g, e2) = (1 - \text{score}(43,\text{BASIC}))^2$$

$$\text{Err}(g, e3) = (0 - \text{score}(76,\text{LISP})^2$$

$$\vdots$$

MINIMISING THE ERROR
(AND MAYBE IT CANNOT BE ZERO)

We now want to minimise the sum of the errors:

$$\sum_{i=1}^{n} \text{Err}(g, ei)$$

Notice that it might not be possible to reach an error of zero:

$$\text{score}(43,\text{BASIC}) = g \cdot 1 + (1 - g) \cdot 0 = g$$
$$\text{score}(87,\text{APL}) = g \cdot 1 + (1 - g) \cdot 0 = g$$

But:

$$\text{Err}(g, e2) = (1 - g)^2$$
$$\text{Err}(g, e6) = g^2$$

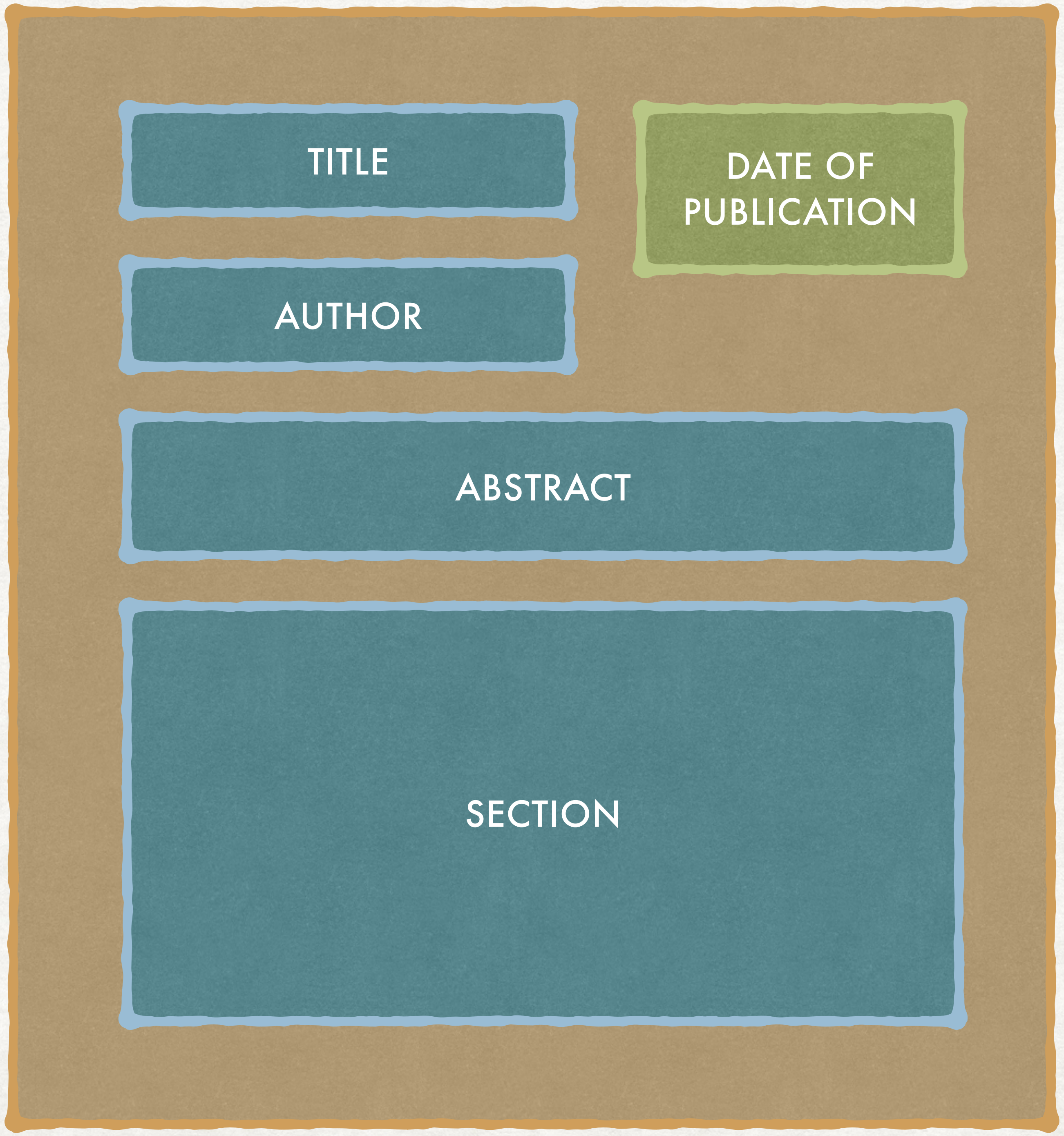
RANKED RETRIEVAL

MOTIVATIONS

- Until now we have returned all documents matching a Boolean query as a set.
- If many documents are returned then it might be important to rank them according to how relevant they are.
- A first way of ranking them is to “split” a document according to some structure and then weight different zones in different ways.
- We will then see how we can extend the idea of adding weights also to the terms of a document.

DOCUMENT STRUCTURE

METADATA, FIELDS, AND ZONES



- A text may have associated **metadata**.
- Some of them can be **fields**, with a set of values that can be finite, like publication dates.
- Others might be zones, arbitrary areas of free-form text (e.g., abstract, section, etc.).

PARAMETRIC INDEXES

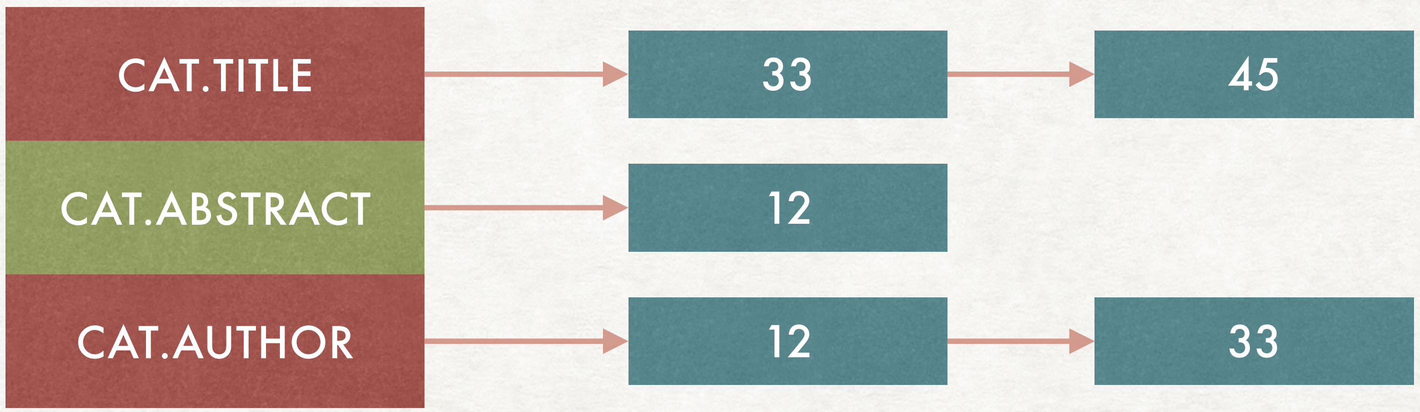
SEARCHING INSIDE FIELDS

- To allow for searching inside the fields we might want to build additional indexes, called **parametric indexes**.
- A parametric index can be thought as a standard index that only has information about a field (e.g., all the dates).
- If a query asks for “cat” in the title and “dog” inside the document we will retrieve the posting lists for dog from the ”standard” index e “cat” from the parametric index for the title.
- The operations of union and intersections works as usual.

ZONE INDEXES

POSSIBLE APPROACHES

Separate inverted index for each zone



Single inverted index in which the zones are part of the postings

WEIGHTED ZONE SCORING

AN ADDITIONAL USE FOR ZONES

- We now have a way of searching inside different parts of a document…
- …but different parts might carry different importance: e.g., a title vs inside the main text.
- We can rank retrieved documents according to where the term is found inside the document.
- We can do this via **weighted zone scoring** (also called **ranked Boolean retrieval**).

SCORING FUNCTION

DEFINITION

- Consider a pair (q, d) of a query q and a document d.
- A **scoring function** associates a value in $[0,1]$ to each pair (q, d).
- Higher scores are better.
- Suppose that a document has ℓ zones.
- Each zone has a weight $g_i \in [0,1]$ for $1 \leq i \leq \ell$.
- The weights sums to one:

$$\sum_{i=1}^{\ell} g_i = 1$$

SCORING FUNCTION
PART II

- Given a query q let s_i be defined as

$$s_i = \begin{cases} 1 & \text{if } q \text{ matches in zone } i \\ 0 & \text{otherwise} \end{cases}$$

- Actually, s_i can also be defined to be any function that maps "how much" a query matches in the i-th zone.

- The weighted zone score in then defined as:

$$\sum_{i=1}^{\ell} g_i s_i$$

WEIGHTED ZONE SCORING

A SIMPLE EXAMPLE

Query: **CAT**

	Title: 0.5	Author: 0.2	Body: 0.3	
TITLE: LIFE OF A CAT AUTHOR: JAMES CAT ONCE THERE WAS A CAT…	0.5	0.2	0.3	**1**
TITLE: DOGS AND OTHER PETS AUTHOR: ANONYMOUS DOGS AND CATS ARE THE…	0	0	0.3	**0.3**
TITLE: ORCHARDS MANAGEMENT AUTHOR: JAMES CAT THE MANAGEMENT OF ORCHARDS…	0	0.2	0	**0.2**

LEARNING WEIGHTS

OR SETTING THEM MANUALLY

- The new problem is now to find how to set the weights for the different scores.
- One possibility is to ask a domain expert.
- Another possibility is to have users label documents relevant or not with respect to a query…
- …and trying to learn the weights using the training data.
- In addition to the binary classification (relevant or not) more nuanced classifications might be used.

THE TRAINING SET

Example	DocID	Query	In the title	In the body	Judgment
e1	43	LISP	1	1	Relevant
e2	43	BASIC	1	0	Relevant
e3	76	LISP	0	1	Non-relevant
e4	76	BASIC	0	1	Relevant
e5	87	SMALLTALK	1	1	Relevant
e6	87	APL	1	0	Non-relevant

COMPUTING THE ERROR

HOW TO DECIDE IF OUR WEIGHTS WORKS

With only two zones, site score is computed as:

$$\text{score}(d, q) = g \cdot s_{\text{title}} + (1 - g) \cdot s_{\text{body}}$$

Since we know the queries and the real relevance of the documents in the training set we can compute the output that a weight g would give:

$$\text{score}(43, \text{LISP}) = g \cdot 1 + (1 - g) \cdot 1$$

$$\text{score}(43, \text{BASIC}) = g \cdot 1 + (1 - g) \cdot 0$$

$$\text{score}(76, \text{LISP}) = g \cdot 0 + (1 - g) \cdot 1$$

$$\vdots$$

COMPUTING THE ERROR

HOW TO DECIDE IF OUR WEIGHTS WORKS

If we decide that relevant is 1 and non-relevant is 0
we can compare the real score with the computed one
and compute an error:

$$\text{Err}(g, e1) = (1 - \text{score}(43, \text{LISP}))^2$$

$$\text{Err}(g, e2) = (1 - \text{score}(43, \text{BASIC}))^2$$

$$\text{Err}(g, e3) = (0 - \text{score}(76, \text{LISP})^2$$

$$\vdots$$

MINIMISING THE ERROR

(AND MAYBE IT CANNOT BE ZERO)

We now want to minimise the sum of the errors:

$$\sum_{i=1}^{n} \text{Err}(g, ei)$$

Notice that it might not be possible to reach an error of zero:

$$\text{score}(43, \text{BASIC}) = g \cdot 1 + (1 - g) \cdot 0 = g$$

$$\text{score}(87, \text{APL}) = g \cdot 1 + (1 - g) \cdot 0 = g$$

But:

$$\text{Err}(g, e2) = (1 - g)^2$$

$$\text{Err}(g, e6) = g^2$$

TF-IDF WEIGHTING

CHANGING SCORING

REFINING THE SCORING

- For now we have used a weight that is either 0 or 1 depending on wether a query term was present or not.
- We might want to assign different weight depending on the term and the number of times a term is present in the document.
- This works well with *free-form text* queries:
  - For each term in the query we compute a "match score"
  - The score of a document is the sum of the scores for each term

TERM FREQUENCY

A SIMPLE SCORE

Term frequency: $\mathrm{tf}_{t,d}$

Number of occurrences of the term t inside the document d.

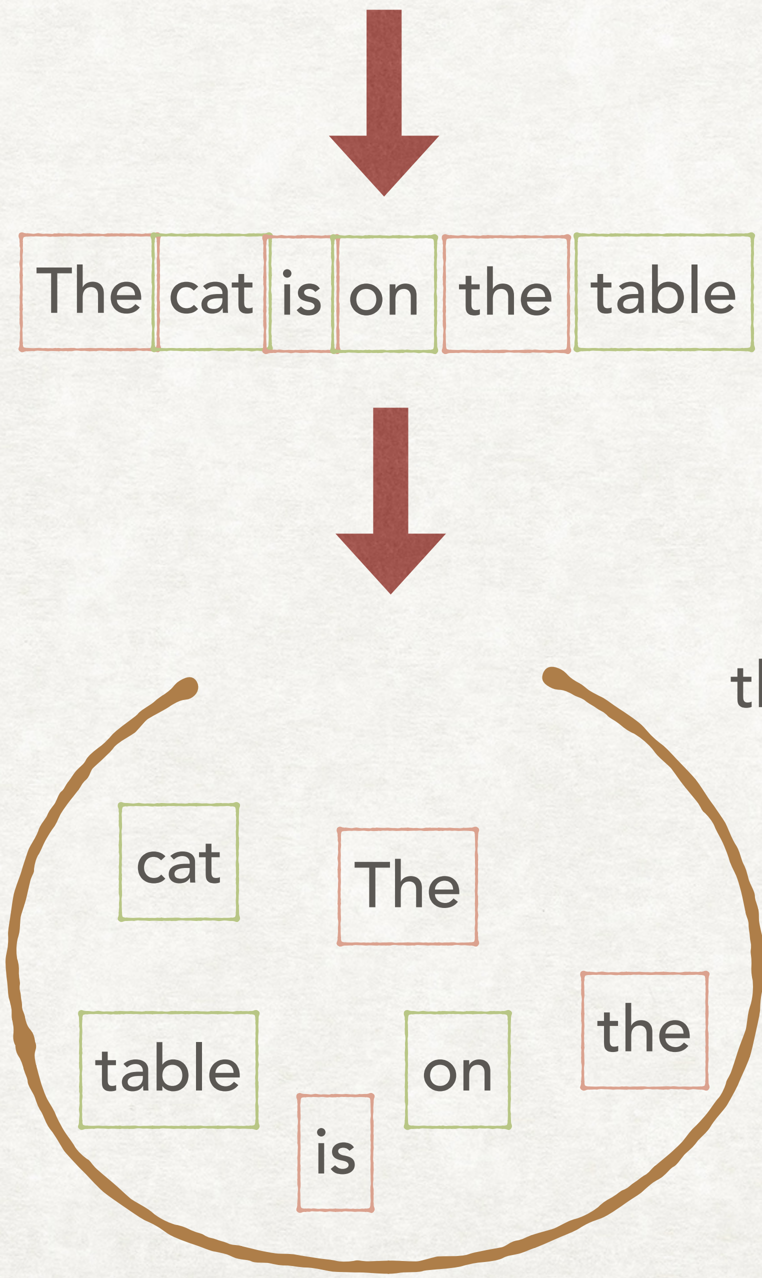
The main motivation is that the more a term is present inside a document the more we consider the document relevant with respect to that term.

But what about the order of the words?

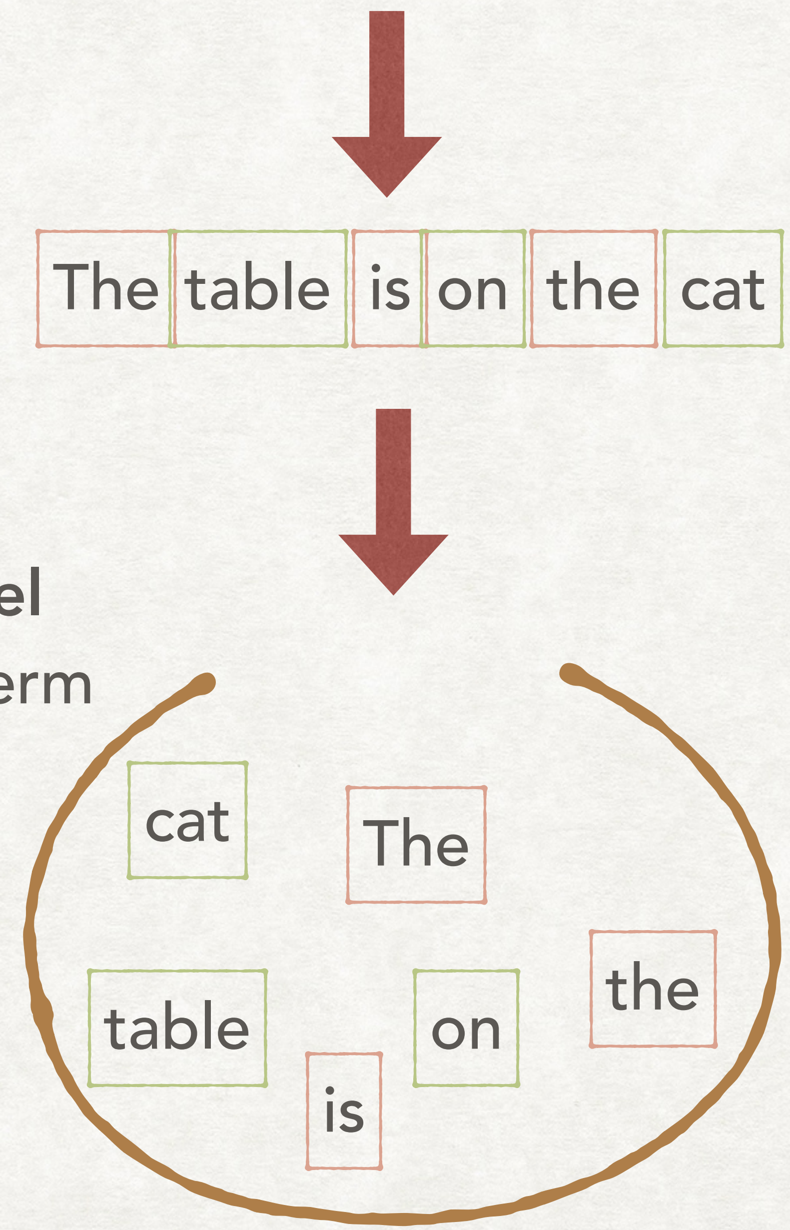
BAG OF WORDS

IGNORE THE ORDER!

The cat is on the table



The table is on the cat

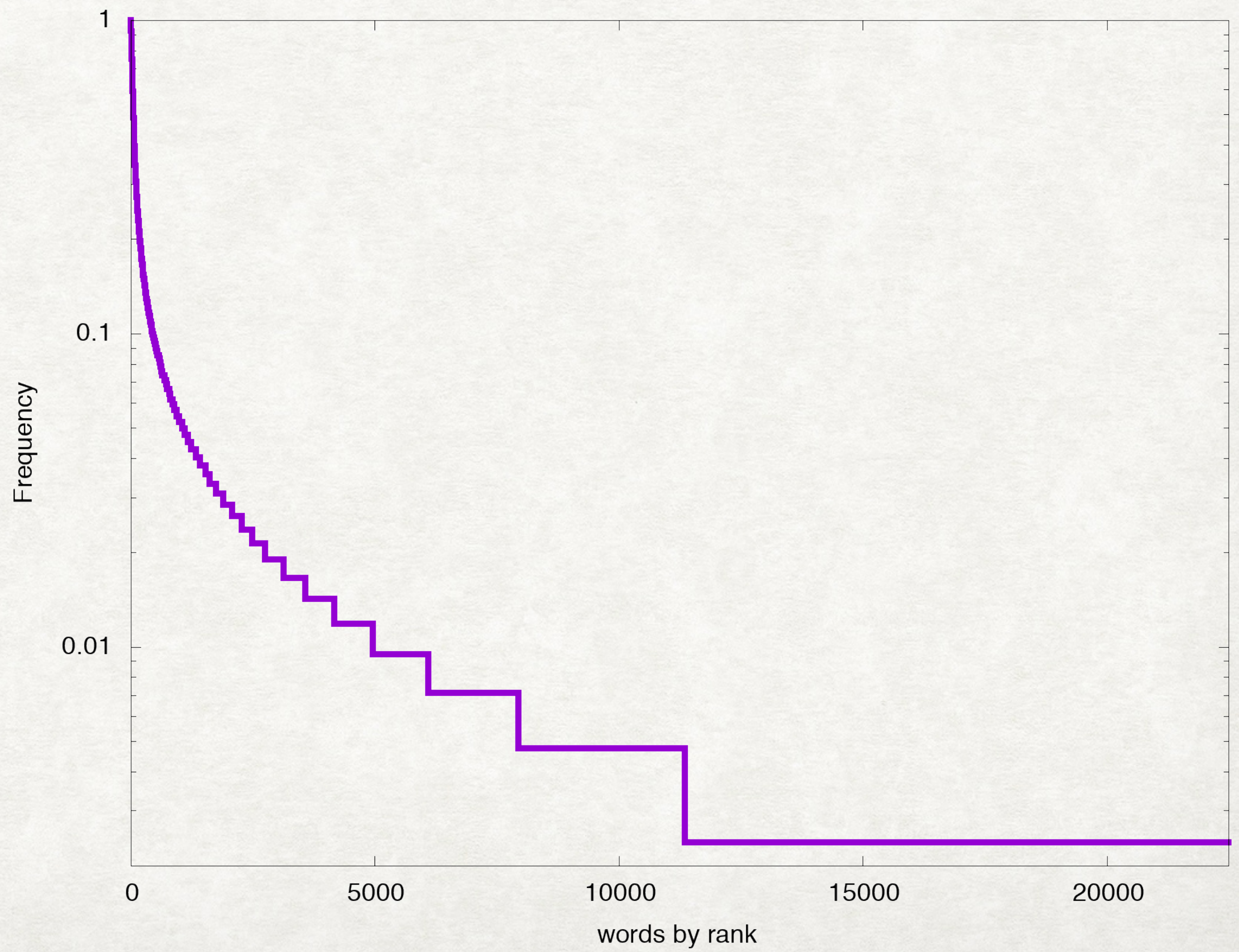


In the
bag of words model
the ordering of the term
is *immaterial* but
the amount
of occurrences
is *material*

TERM FREQUENCY

SOME LIMITATIONS

- Does the number of occurrences really represents the importance of a term?
- Which terms are more frequent?
- A small hint:
- Stop words!
- Not all terms carry the same weight in determining the relevancy of a document



COLLECTION AND DOCUMENT FREQUENCIES

RARE WORDS COUNT MORE

- The main characteristic of stop words is that they are present in most documents.

- Therefore, we might want to scale the importance of a word based on some measure of the frequency of the term:

- cf_t is the **collection frequency** of the term t:
  total number of occurrences of the term t in the collection.

- df_t is the **document frequency** of the term t:
  total number of document in which t appears in the collection.

COLLECTION AND DOCUMENT FREQUENCIES

RARE WORDS COUNT MORE

- The document frequency df_t of a term is usually preferred.
- We prefer to use a document-based measure to weight documents.
- cf_t and df_t can behave quite differently. For example:
  - A single document with 1000 instances of a term t_1 in a collection of 1000 documents.
  - Each one of 1000 documents contains a term t_2 exactly once.

INVERSE DOCUMENT FREQUENCY

MODIFYING DOCUMENT FREQUENCY

df_t is larger when we want the penalties to be larger

We use a modification of it:

Number of documents in the collection

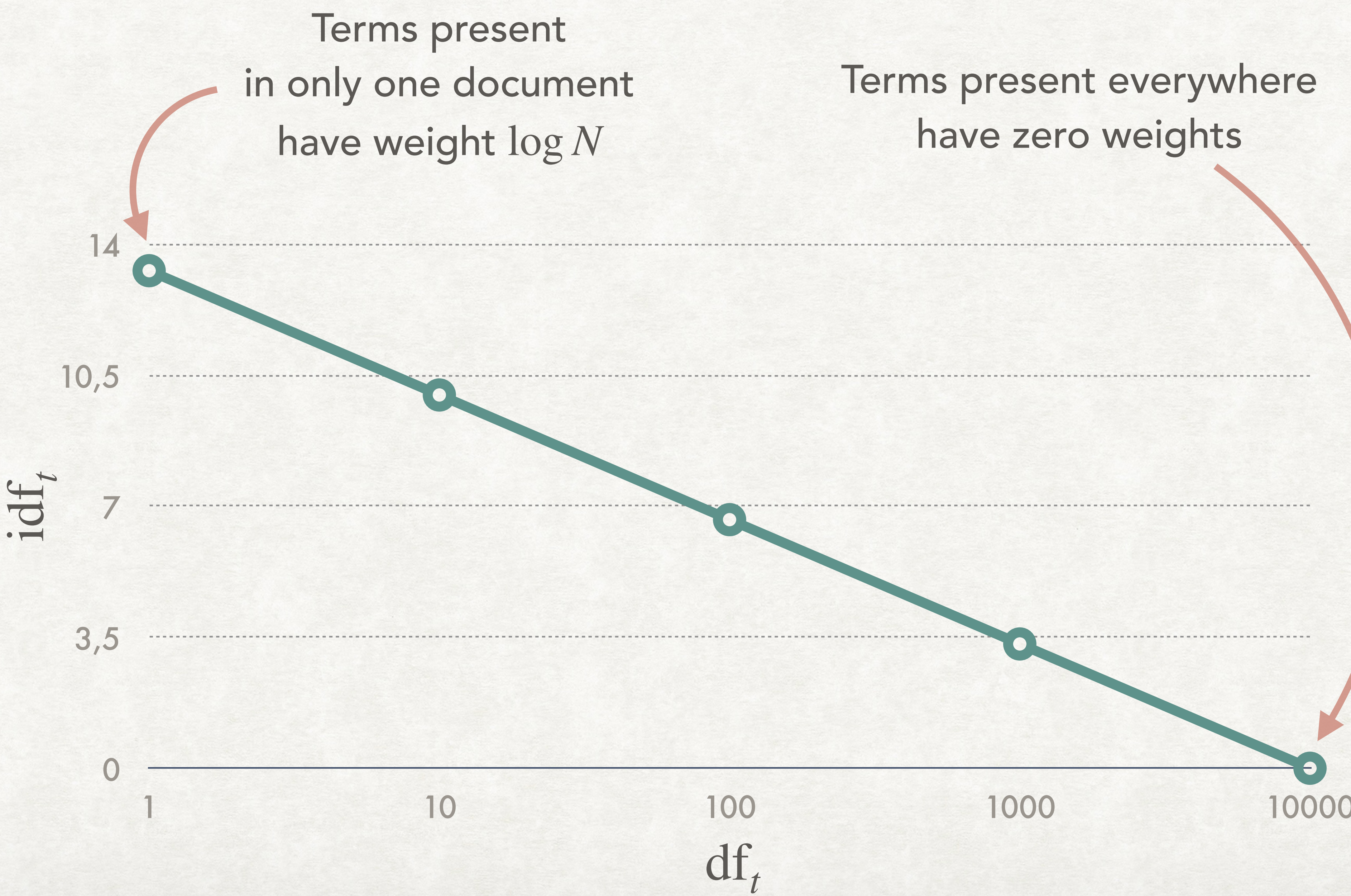
$$\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t}$$

Inverse document frequency

Document frequency

INVERSE DOCUMENT FREQUENCY

EFFECTS ON THE WEIGHTS



TF-IDF WEIGHTING

HOW TO COMBINE $\text{tf}_{t,d}$ AND idf_t

We now need to combine the two ideas:

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

- When a rare term is present a many times in a document then the value is high
- When a frequent term is present many times or a rare term is present only a few time the value is low
- When a very frequent term is present only a few times then the value is the lowest

SCORING A DOCUMENT

TOWARDS THE VECTOR SPACE MODEL

The cat is on the table

We can see a document as a vector with a components for each term in the dictionary and having as elements the $\text{tf-idf}_{t,d}$ of the term t in the document

cat	is	on	table	the
$\text{tf-idf}_{\text{cat},d}$	$\text{tf-idf}_{\text{is},d}$	$\text{tf-idf}_{\text{on},d}$	$\text{tf-idf}_{\text{table},d}$	$\text{tf-idf}_{\text{the},d}$

$\text{tf-idf}_{t,d} = 0$ for all terms not in the document

SCORING A DOCUMENT

TOWARDS THE VECTOR SPACE MODEL

To score a document for a query q
we can simply sum the $\text{tf-idf}_{t,d}$ values
for all terms appearing in q:

$$\text{Score}(q, d) = \sum_{t \in q} \text{tf-idf}_{t,d}$$

Notice that in this way a document where a term
does *not* appear might still have a positive score.
The "penalty" will depend on which term is not present

VARIANTS OF TF-IDF

AND WHEN TO USE THEM

- There are some possible alternative in using directly tf-idf.
- One first consideration is that not all instances of a term inside a document carry the same weight.
- There is the idea of "diminishing returns": is a document with 20 occurrences really twice as important as one with 10 occurrences?
- Another observation is that we might be interested in the frequency of a term relative to the other terms in the document.

SUBLINEAR TF SCALING

We can scale the $\mathrm{tf}_{t,d}$ value to have
the influence of additional terms reduced:

$$\mathrm{wf}_{t,d} = \begin{cases} 1 + \log \mathrm{tf}_{t,d} & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The new value can be replaced where $\mathrm{tf}_{t,d}$ is used:

$$\text{wf-idf}_{t,d} = \mathrm{wf}_{t,d} \times \mathrm{idf}_t$$

TF NORMALIZATION

We can scale the $\mathrm{tf}_{t,d}$ value to be dependant
on the maximum term frequency in the document $\mathrm{tf}_{\max}(d)$:

$$\frac{\mathrm{tf}_{t,d}}{\mathrm{tf}_{\max}(d)}$$

Another possibility is to normalise according to the
number of terms in the entire document:

$$\frac{\mathrm{tf}_{t,d}}{\sum_{t' \in d} \mathrm{tf}_{t',d}}$$

In both cases there are drawbacks and some smoothing might be
applied to limit large swings in the normalised value