INFORMATION RETRIEVAL

Luca Manzoni

lmanzoni@units.it

Lecture 4

LECTURE OUTLINE

*MAY CONTAIN TRACES OF PEANUTS

CATT — SEARCH

DO YOU MEAN "CAT"?

Spelling Correction

+ IMPLEMENTATION

Wildcard Queries

*

Boolean Queries Optimisation

$$(a \wedge b) \wedge c$$

$$a \wedge (b \wedge c)$$
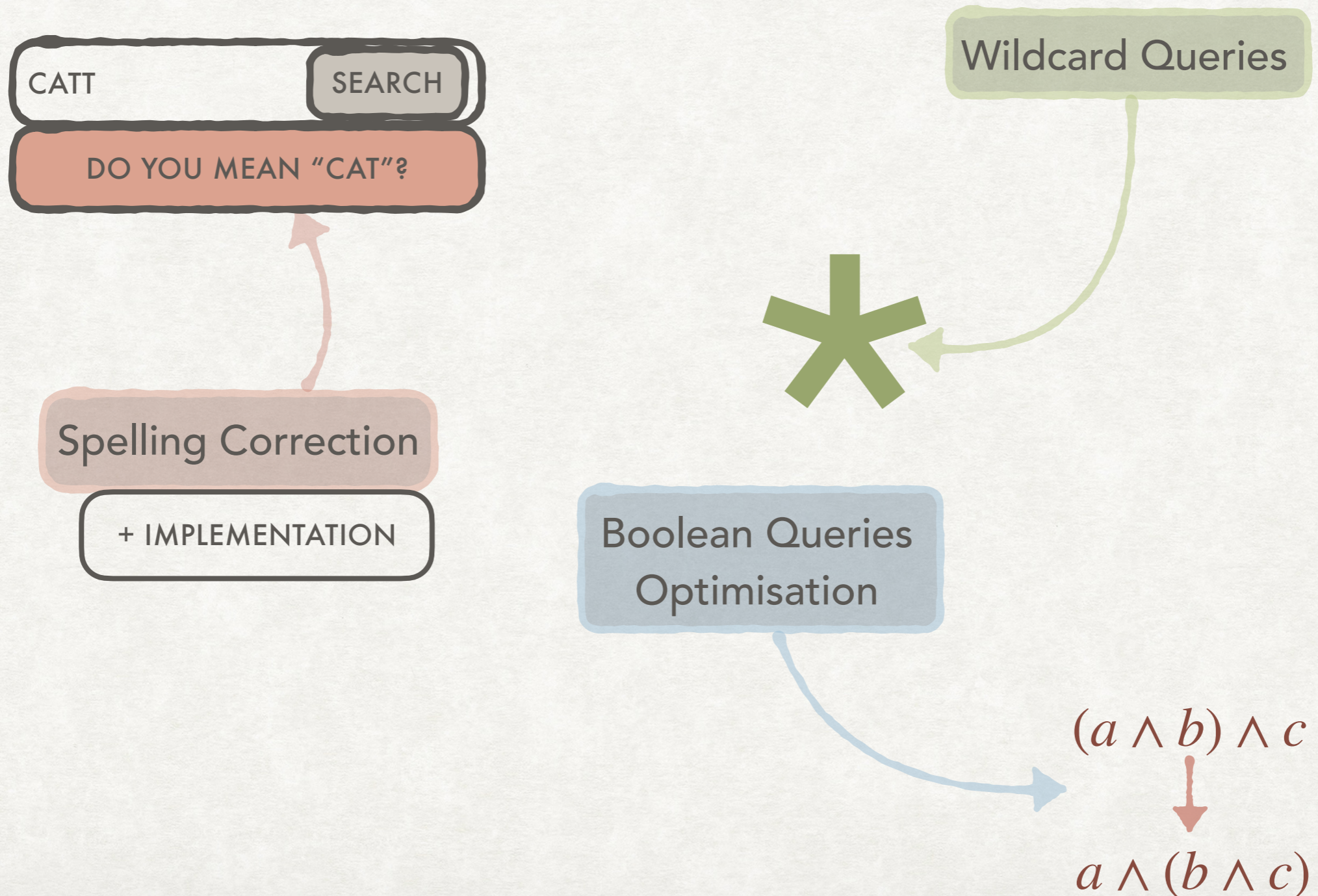
WILDCARD QUERIES

WHAT ARE WILDCARD QUERIES?
SEARCHING AN ENTIRE SET OF WORDS

- Examples of wildcard queries:

  - **Car***: captures "**car**", "**cars**", "**car**t", "**carb**on", etc.

  - ***e*a**: captures "fl**ea**", "**ea**r", "h**ea**d", "**E**v**a**", 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*
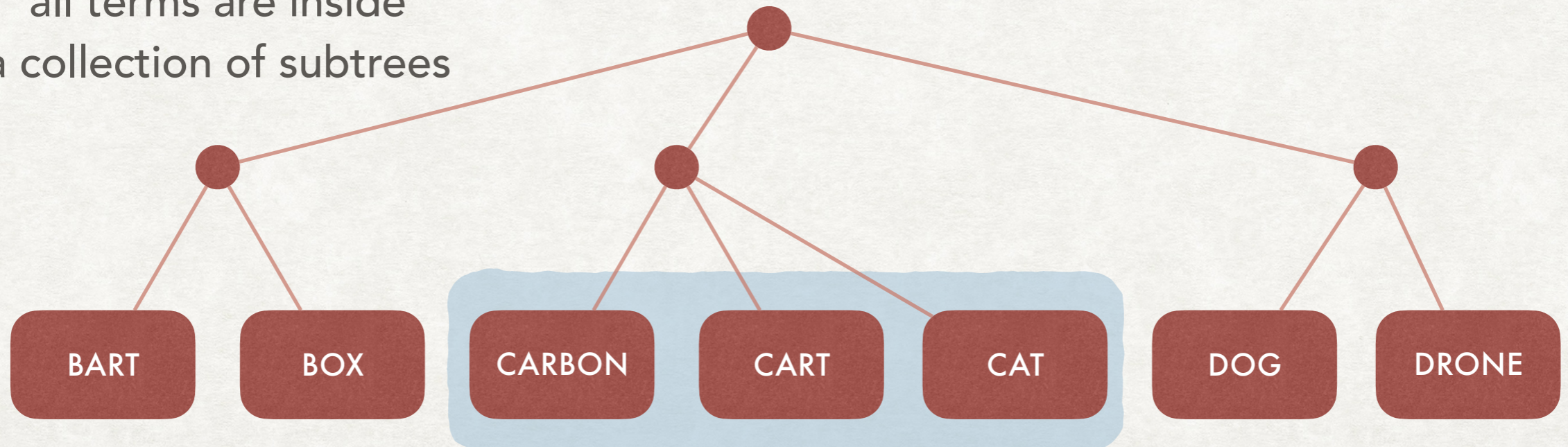
Trailing wildcard
there is only one wildcard
and it is at the end of the word

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

BART BOX CARBON CART CAT DOG DRONE

LEADING WILDCARDS
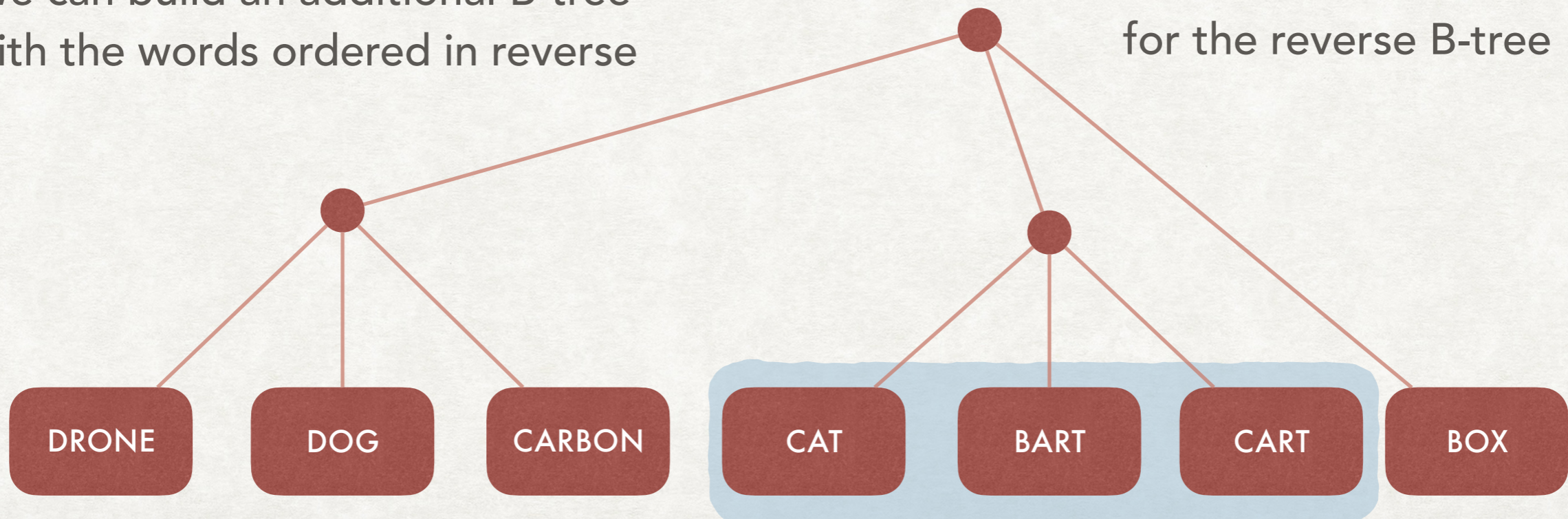## AND REVERSE (B-)TREES

*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

DRONE   DOG   CARBON   CAT   BART   CART   BOX

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

| C | A | T | $ | ← Special *"end of word"* symbol |

| A | T | $ | C |

| T | $ | C | A | Rotations of the word

| $ | C | A | T |
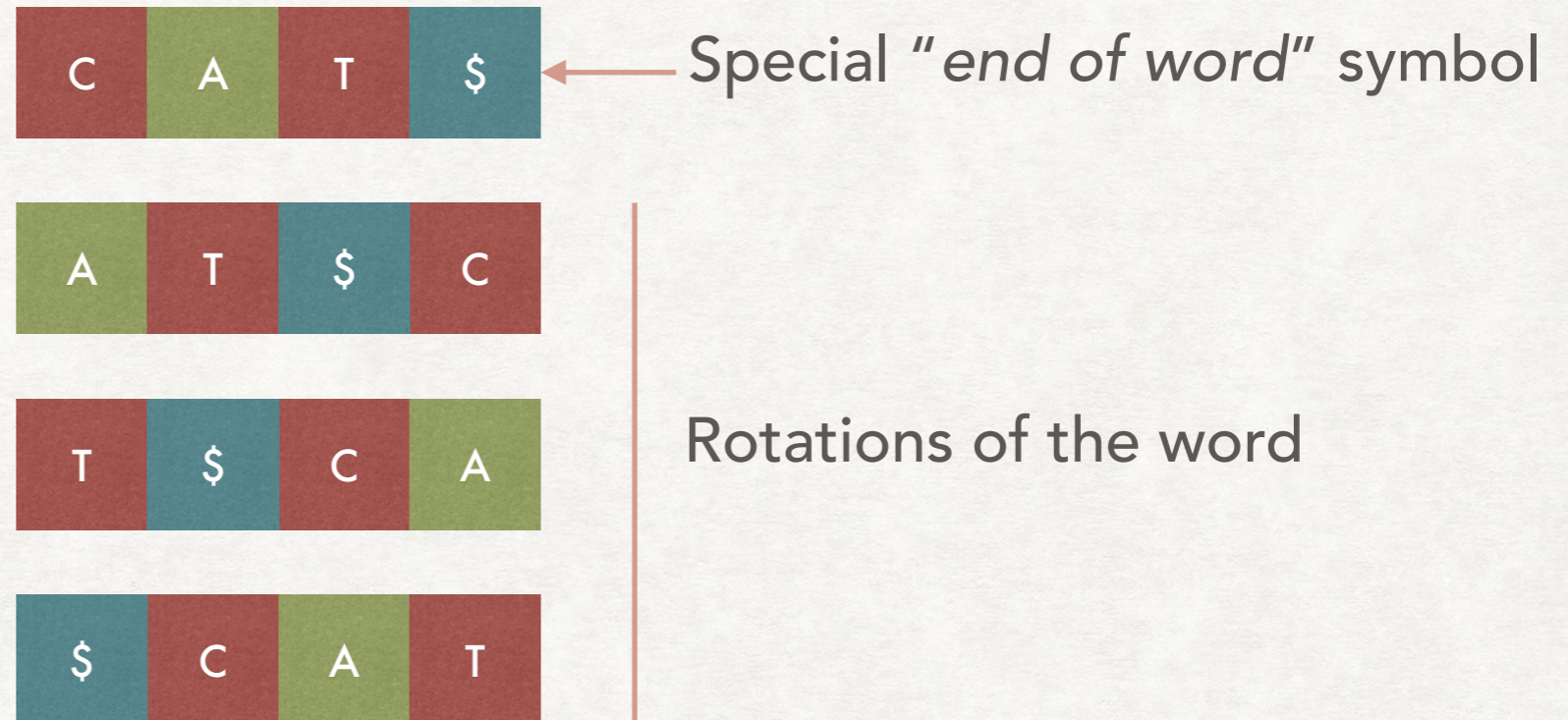
We insert all the rotations of the word (including the "end of word") in the dictionary.

All the rotations of the same word points to the **same** postings list
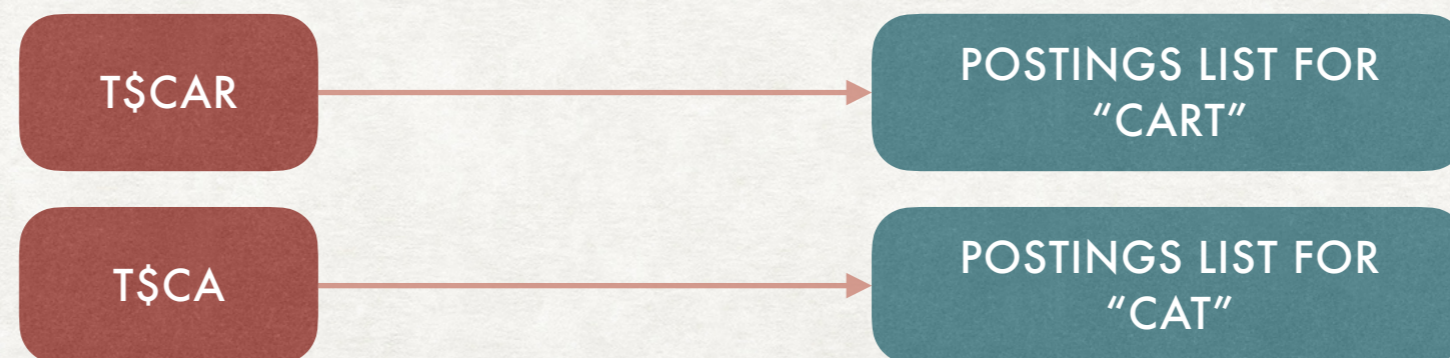
PERMUTERM INDEX
MANAGING GENERAL WILDCARD QUERIES

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

T$CAR	→	POSTINGS LIST FOR "CART"
T$CA	→	POSTINGS LIST FOR "CAT"

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
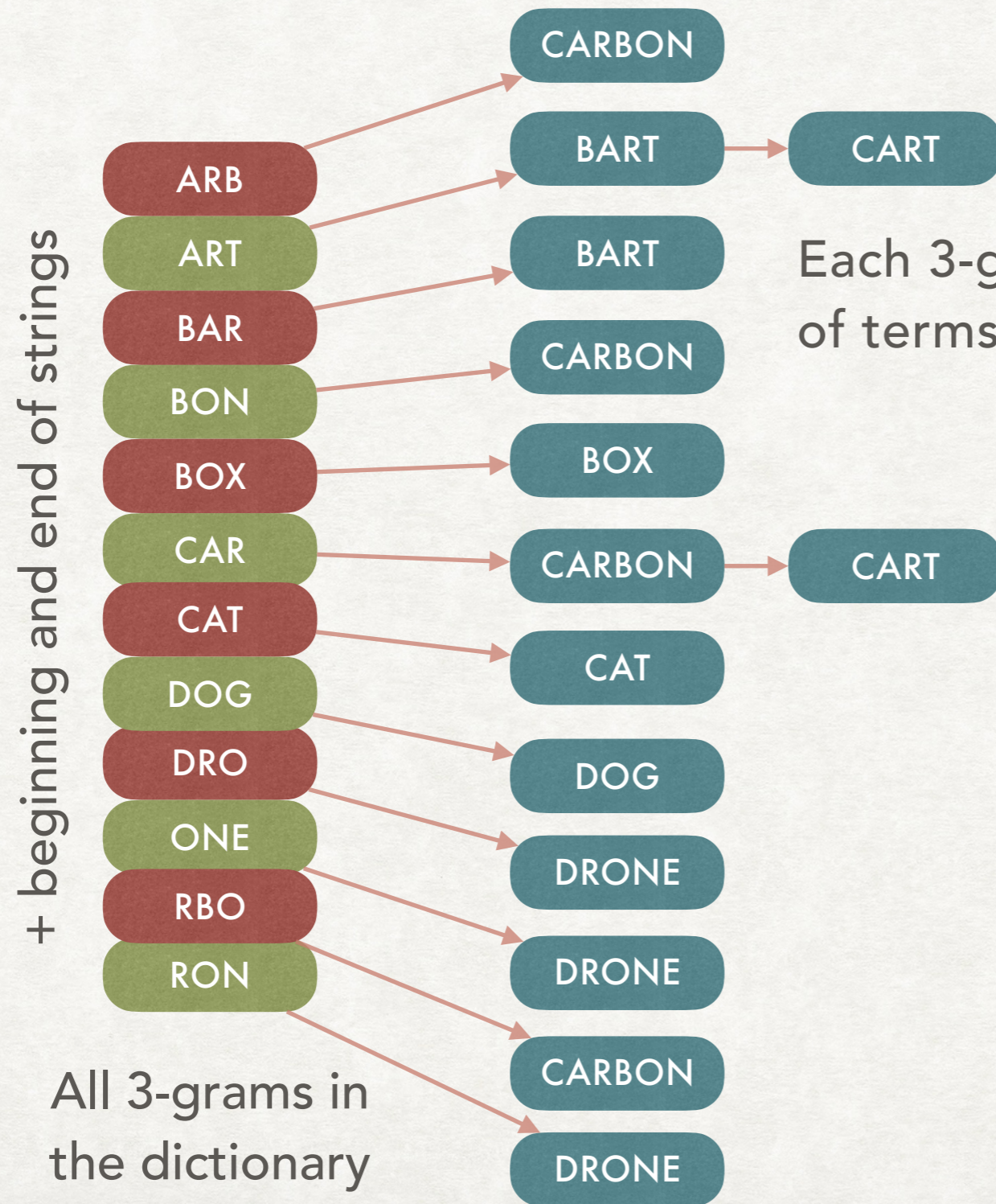
DRONE

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

+ beginning and end of strings

ARB → CARBON

ART → BART → CART

BAR → BART

BON → CARBON

BOX → BOX

CAR → CARBON → CART

CAT → CAT

DOG → DOG

DRO → DRONE

ONE → DRONE

RBO → CARBON

RON → DRONE

All 3-grams in the dictionary

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

The current structure of the system:

k-GRAMS

↓

TERMS

↓

POSTINGS
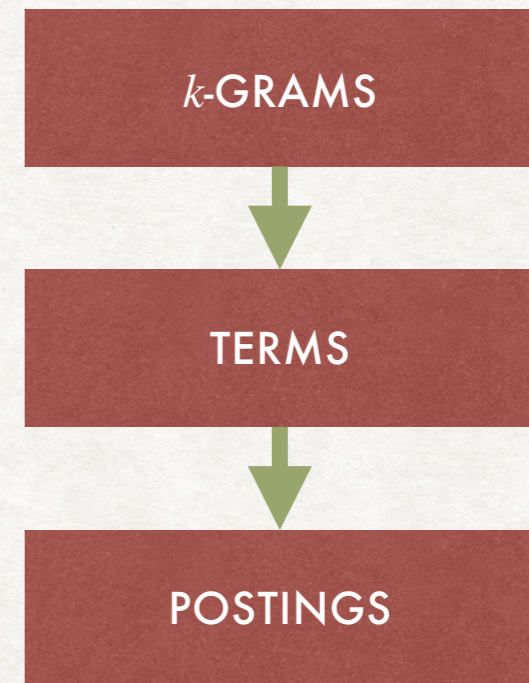
K-GRAMS INDEXES
HOW TO USE THEM TO ANSWER QUERIES

CARBON

ARB → CARBON

ART → BART → CART

BAR → BART

BON → CARBON

BOX → BOX

CAR → CARBON → CART

⋮

ON$ → CARBON

⋮

$CA → CARBON → CART → CAT

Our query: **CA*ON**

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

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

Search each one of the 3-grams

Intersect the results: CARBON
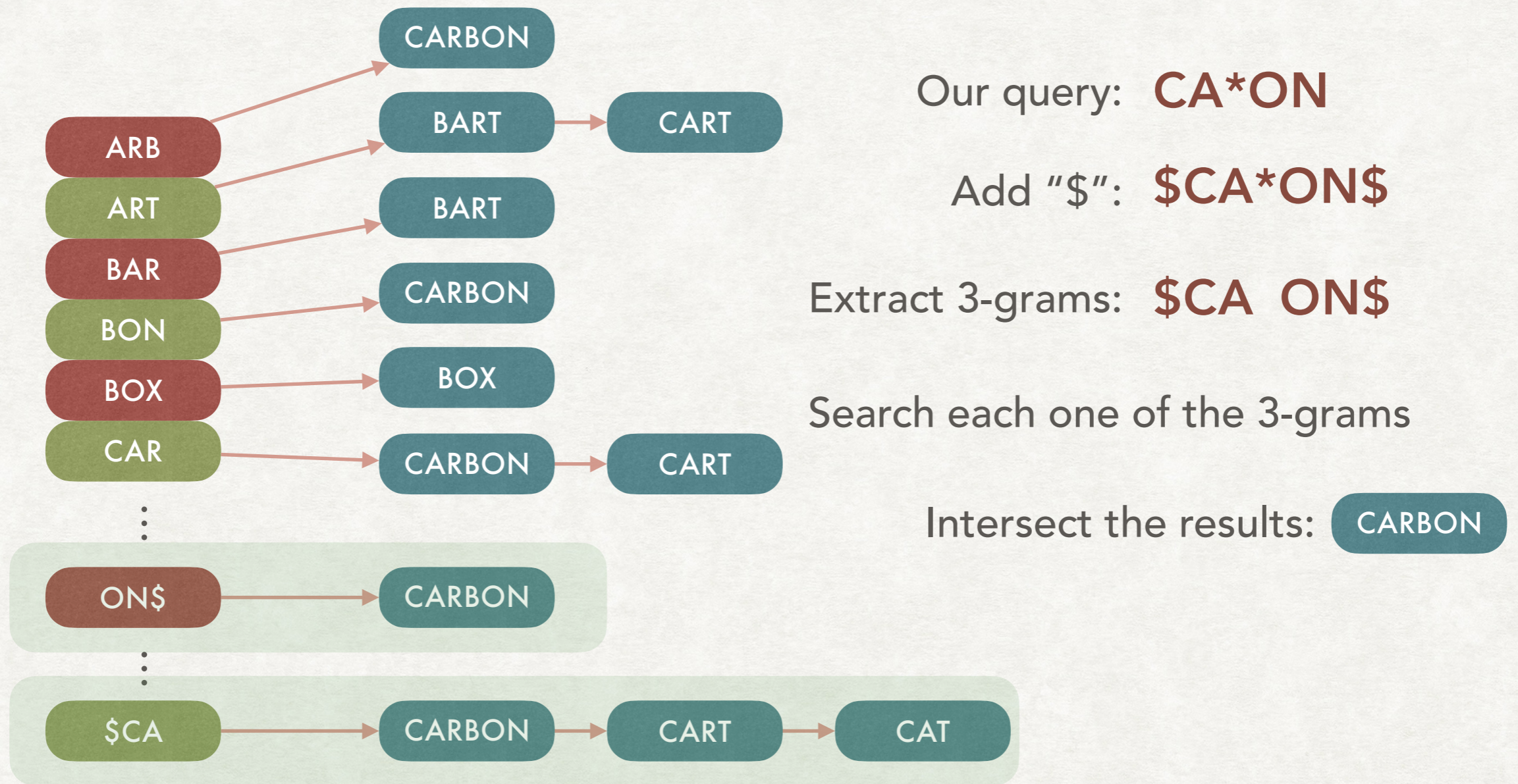
K-GRAMS
ADVANTAGES AND DISADVANTAGES

- 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
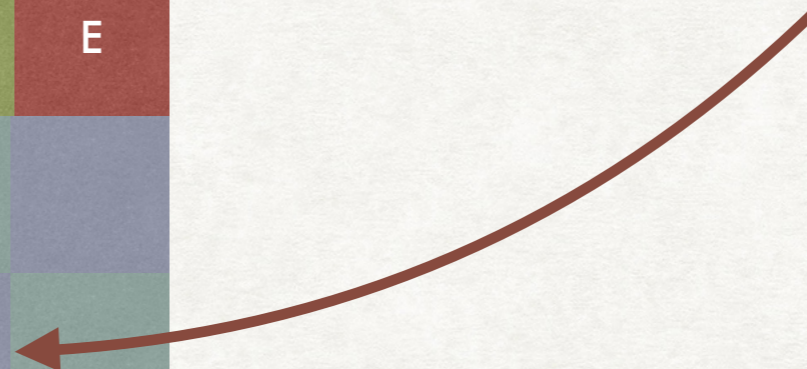## WITH DYNAMIC PROGRAMMING

- 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

	ε	H	O	M	E
ε					
H					
O					
U					
S					
E					

Distance between "HOM" and "H"

Distance between "HOUS" and "HO"

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

MORS**OM**OLL**OO**

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