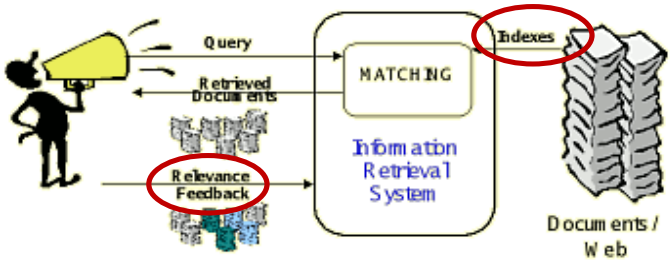


Wyszukiwanie i Przetwarzanie Informacji

Information Retrieval & Search

Irmina Masłowska

irmina.maslowska@cs.put.poznan.pl

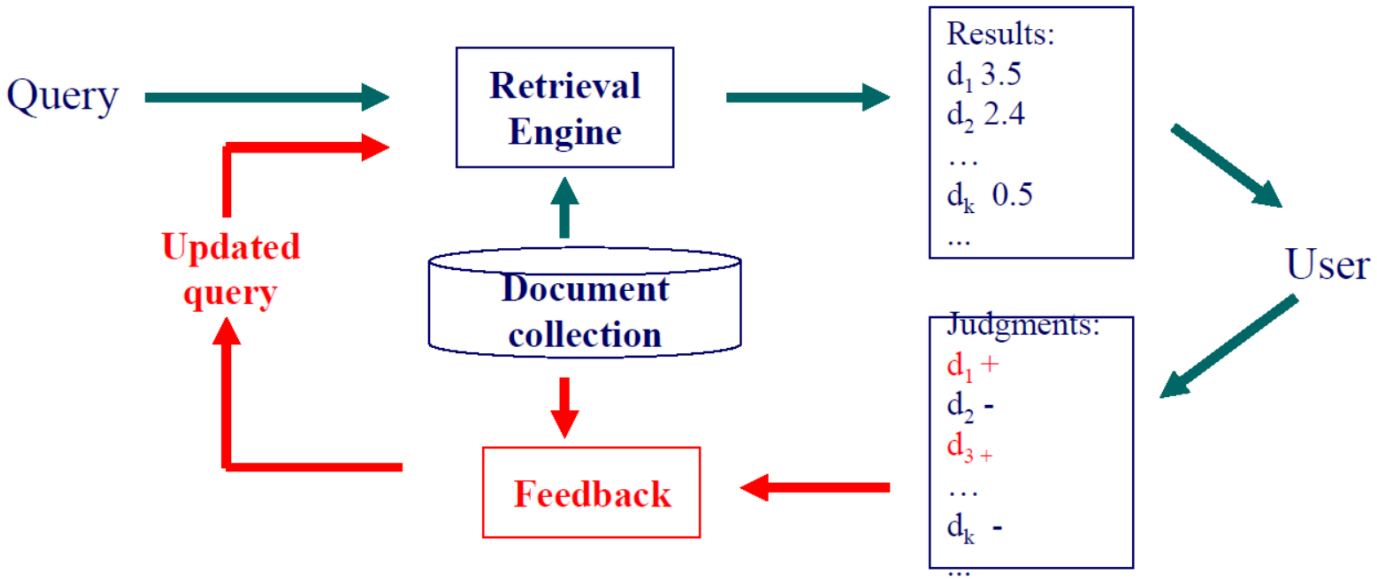
www.cs.put.poznan.pl/imaslowska/wipi/

Relevance feedback

Dokładność i kompletność wyników systemu IR dla konkretnych zapytań można poprawiać bazując na informacji zwrotnej pochodzącej od użytkowników

Informacja ta może być przekazywana przez użytkowników *explicite*, *implicite*, lub wcale ;)

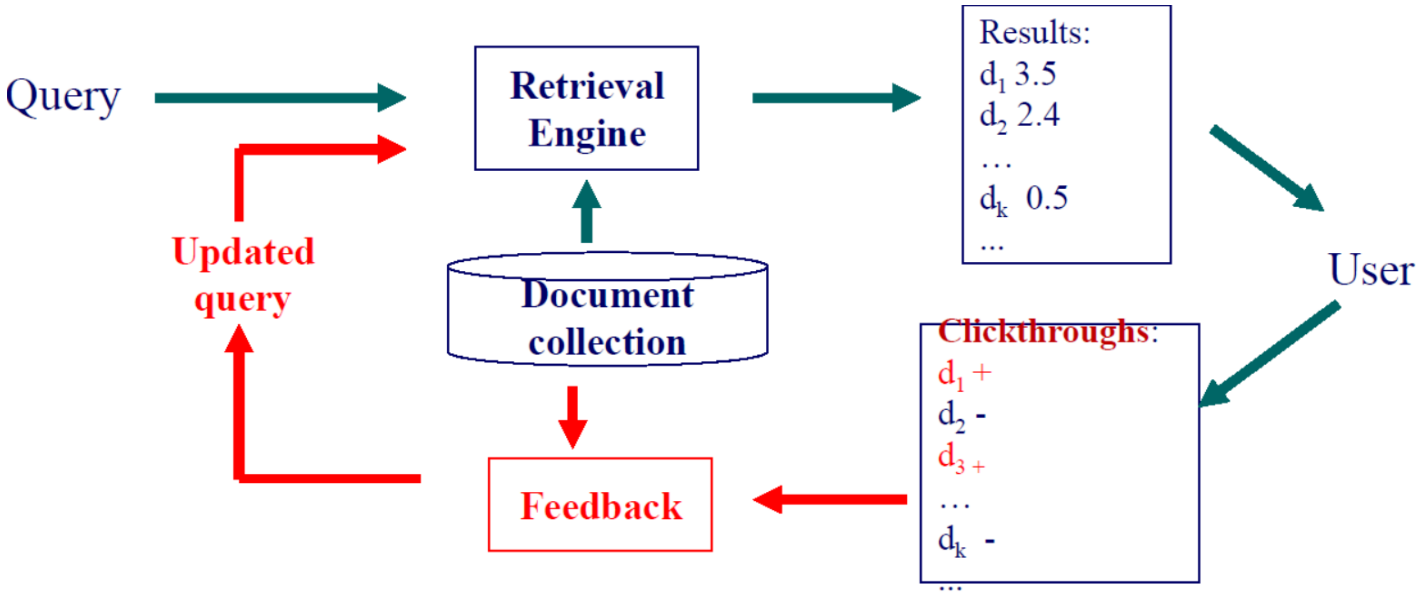
Explicit relevance feedback



- Użytkownik ocenia adekwatność kilku/-nastu dokumentów z czołówki rankingu dla danego zapytania
- Na podstawie ocen system reformułuje zapytanie
- Dobra skuteczność, ale wymaga dodatkowego wysiłku użytkownika i skomplikowania interfejsu wyszukiwarki

Grafiki z wykładu *"Text Retrieval Methods: Feedback in TR"* – ChengXiang Zhai

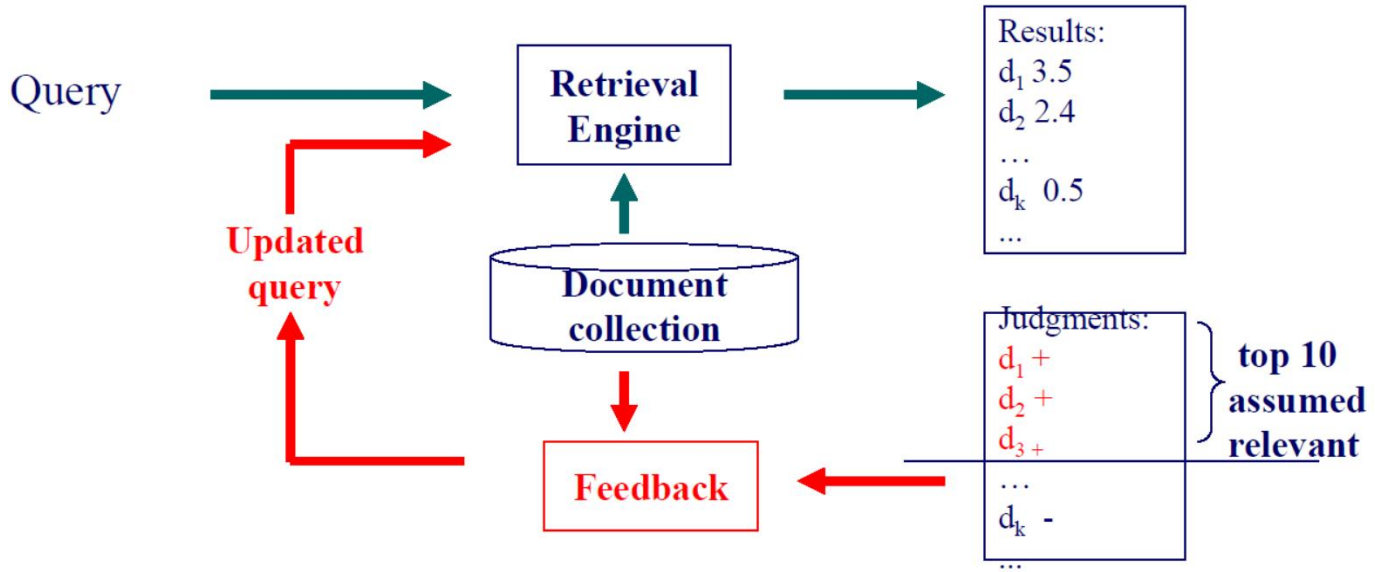
Implicit feedback



- Wyniki obejrzane (kliknięte) przez użytkownika zostają uznane za adekwatne, pozostałe za nieadekwatne
- Na tej podstawie system reformułuje zapytanie
- Takie oceny nie są całkowicie wiarygodne, ale nie wymaga się od użytkownika żadnych dodatkowych działań

Grafiki z wykładu *"Text Retrieval Methods: Feedback in TR"* – ChengXiang Zhai

Pseudo relevance feedback



- System oblicza ranking dla zapytania po czym uznaje za adekwatne pierwsze *k* dokumentów w rankingu
- System reformułuje zapytanie i dopiero zwraca ranking
- Takie oceny są niewiarygodne, ale nie wymaga się od użytkownika żadnych działań a mechanizm zostaje ukryty

Grafiki z wykładu *"Text Retrieval Methods: Feedback in TR"* – ChengXiang Zhai

Feedback in Vector Space Model

Mechanizm działania – modyfikacja zapytania:

- dostosowanie wag termów z zapytania
- dodanie do zapytania nowych termów z określoną wagą (*query expansion*)

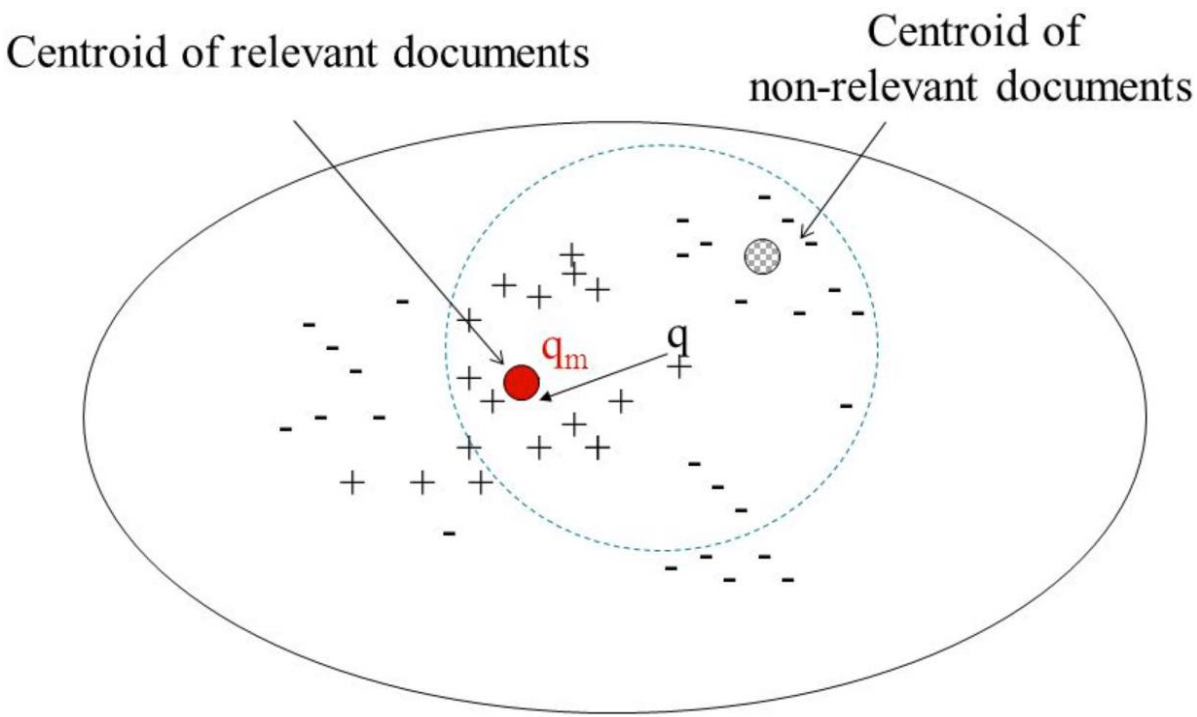
Feedback in Vector Space Model

Mechanizm działania – modyfikacja zapytania:

- dostosowanie wag termów z zapytania
- dodanie do zapytania nowych termów z określoną wagą (*query expansion*)

Najbardziej znana technika – **metoda Rocchio** modyfikuje wektor zapytania za pomocą liniowej kombinacji oryginalnego zapytania q i wektorów dj dokumentów adekwatnych (D_r) i nieadekwatnych (D_{nr})

Rocchio feedback



przesunięcie wektora zapytania w kierunku centroidu dokumentów istotnych i oddalenie od centroidu dokumentów nieistotnych

Grafika z wykładu *"Text Retrieval Methods: Feedback in TR"* – ChengXiang Zhai

Rocchio feedback

$$\vec{q}_m = \alpha\vec{q} + \beta\frac{1}{|D_r|}\sum_{\vec{d}_j \in D_r}\vec{d}_j - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j \in D_{nr}}\vec{d}_j$$

q_m , q zapytanie zmodyfikowane i oryginalne

D_r , D_{nr} dokumenty adekwatne i nieadekwatne

α, β, γ wagi

- Przy wielu ocenionych dokumentach oryginalne zapytanie traci na ważności ($\beta > \alpha$)
- Pozytywna informacja ma zwykle większą wagę niż negatywna ($\beta > \gamma$)
- Negatywna informacja może nie być w ogóle brana pod uwagę ($\gamma = 0$)
- Rozsądne wartości α =1, β=0.75 i γ=0.15

Rocchio feedback

- W praktyce wektor zapytania bywa ‘przycinany’ (pozostawia się jedynie niewielką liczbę termów o najwyższych wagach w centroidzie dokumentów adekwatnych) – lepsza wydajność
- Aby uniknąć nadmiernego dopasowania (*over-fitting*) zaleca się stosowanie większej wartości wagi α
- Metoda wykorzystywana także w *pseudo relevance feedback* – wówczas waga β powinna przyjmować mniejszą wartość niż kiedy dostępne są oceny użytkownika

Query reformulation

Modyfikacje zapytań i podpowiedzi często bazują na słownikach (*thesauri*)

- słowniki ‘ręcznie’ utrzymywane przez edytorów:

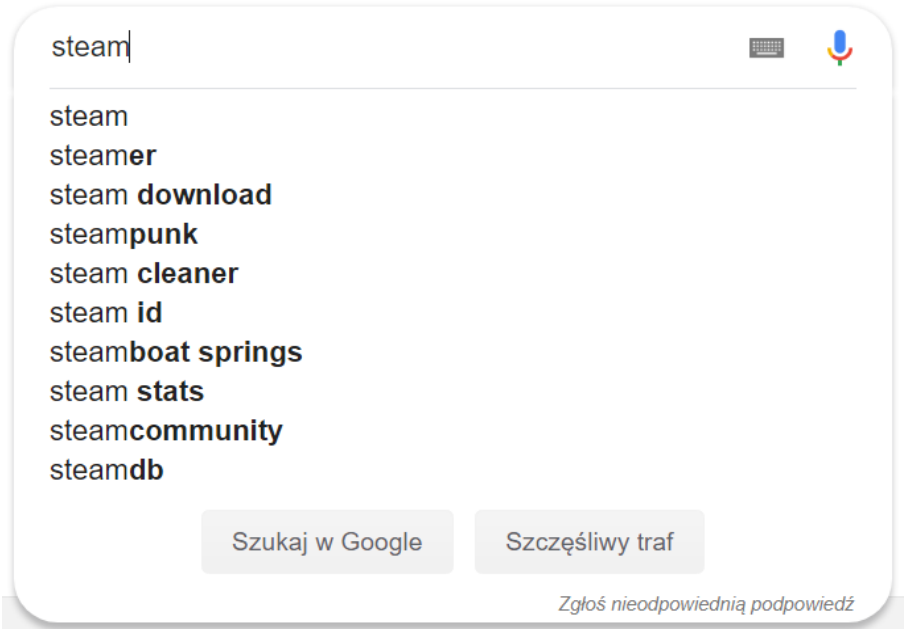
MeSH - https://www.nlm.nih.gov/pubs/factsheets/mesh.html

WordNet – http://wordnet.princeton.edu, itp.

- słowniki tworzone automatycznie (np. na podstawie analizy współwystępowania słów)
- w wyniku analizy zebranych uprzednio zapytań użytkowników (*query log mining*)

Query reformulation

Dla poprawy dokładności i kompletności w zastosowaniach internetowych wyszukiwarki wspierają użytkownika na etapie formułowania zapytania (podpowiedzi) oraz przy obsłudze zapytań



Wyszukiwania podobne do: steam

steam **download**
steam **pl**
steam **pobierz**
steam **rejestracja**
login to steam **with email**
steam **win 10 64bit**
steam **forgot password**
gmail steam

Wyszukiwania podobne do: steamboat springs

steamboat **geyser**
steamboat **gejzer**

Query reformulation

Dodatkowe termy sugerowane jako uzupełnienie zapytania mogą mieć na nie dwojaki wpływ:

- uściślać znaczenie (zawężanie zapytania)
- uogólniać znaczenie (rozszerzanie zapytania)

W obu przypadkach używa się powszechnie dość dyskusyjnego określenia *rozszerzanie zapytania* ze względu na identyczny mechanizm – dokładanie do zapytania dodatkowych termów

Query expansion

Rozszerzanie zapytań (ang. *query expansion*)

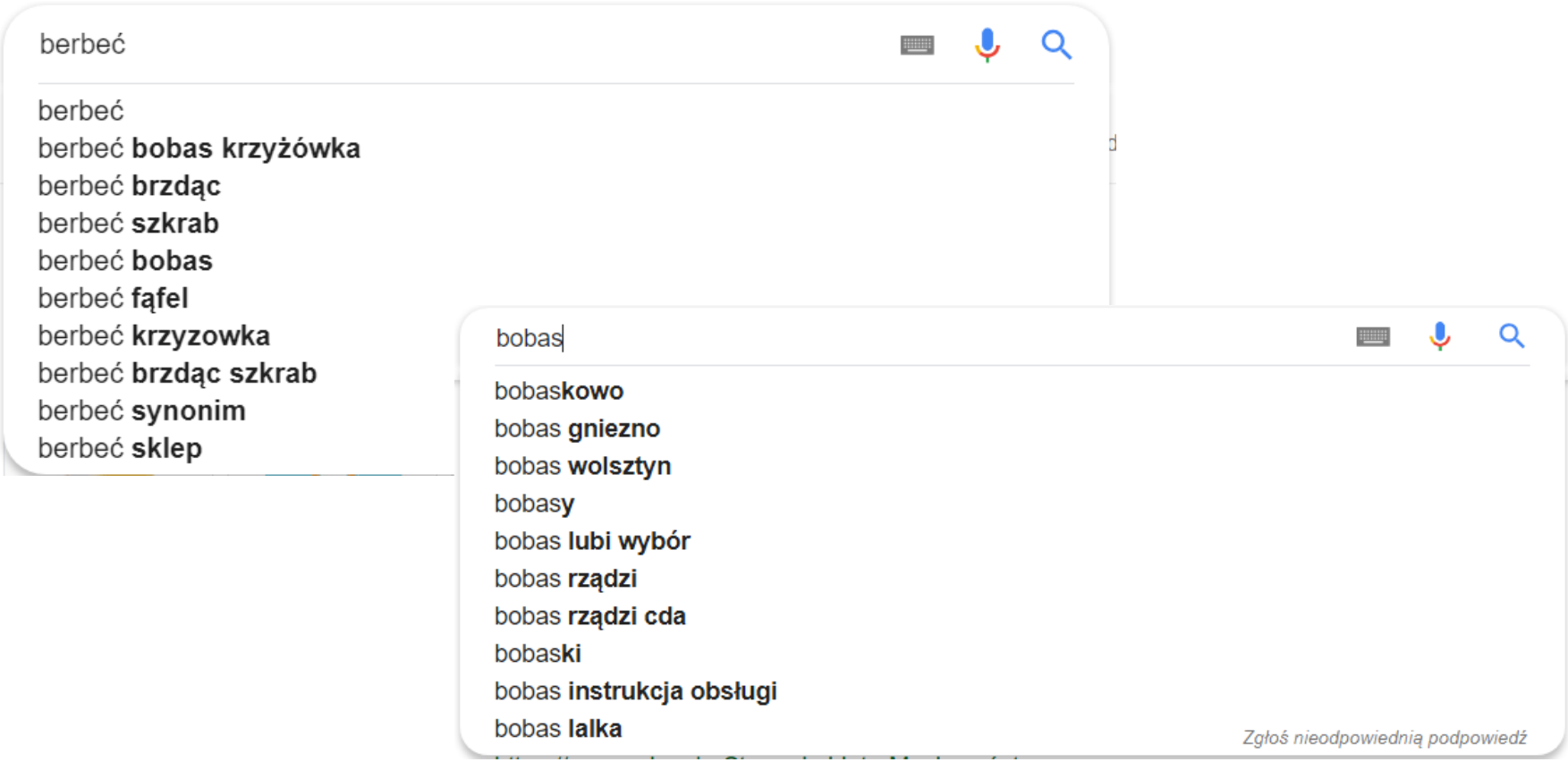
- Dodawanie do zapytania „podobnych” słów
- Celem jest poprawa kompletności (*recall*) odpowiedzi systemu IR
- Często realizowane bez udziału użytkownika
- Adresuje problem synonimów, ale nie polisemii słów

Query refinement

Zawężanie zapytań (ang. *query refinement*, *query narrowing*)

- Sugerowanie użytkownikowi (kilku) wariantów bardziej szczegółowych zapytań (podpowiadanie dodatkowych słów)
- Celem jest poprawa dokładności (*precision*) odpowiedzi systemu IR
- Użytkownik decyduje czy skorzystać z podpowiedzi i której
- Adresuje zarówno problem synomimów jak i polisemii

Query expansion, refinement?



Indeksowanie kolekcji dokumentów

Sekwencyjne przeglądanie (online) dokumentów w poszukiwaniu zapytania jest możliwe lub konieczne jedynie gdy kolekcja tekstów:

- jest stosunkowo mała
- ulega tak częstym zmianom, że czas względnej stabilności nie wystarcza na przeprowadzenie wstępnego przetwarzania
- ograniczenia zasobów uniemożliwiają wykorzystanie ich na przechowywanie indeksu

Indexing

Indeksowanie to transformacja = przekształcenie dokumentów do struktury danych umożliwiającej szybkie wyszukiwanie (wyliczając wstępnie ile się da)

Indeksowanie jest oczywiście niezbędne w zastosowaniach internetowych ze względu na nierealny czas przetwarzania online kolekcji o takich rozmiarach

- Indeks to struktura ‘nadbudowana’ na dokumentach tekstowych w celu przyspieszenia ich przeszukiwania.
- Stosowana w kolekcjach dużych i względnie statycznych (ang. *semi-static*).
- Indeks odwrotny (ang. inverted index) to dominująca metoda indeksowania wspierająca podstawowe algorytmy wyszukujące (stosowana we wszystkich wyszukiwarkach)

Why not term-document matrix

- Rozważmy N=1000000 (milion – 1M) dokumentów, każdy po około 1000 słów
- Średnio 6 bajtów/słowo (j. angielski)
  - objętość danych w dokumentach - ok. 6 GB
- Przyjmijmy, że liczba *unikalnych* termów w dokumentach wynosi M=500000 (500K)

Why not term-document matrix

- 500K x 1M - powstała macierz obejmuje pół biliona (500G) elementów
  - liczby całkowite 32-bitowe – 4 bajty
  - 500K x 1M x 4 = 2TB
- Jednakże w macierzy będzie max. 1 miliard liczb niezerowych (1000 słów x 1 milion dokumentów)
  - macierz jest bardzo rzadka
  - 1000 x 1M x 4 = 4GB
  - Co jeśli N=60 bilionów stron? (Google w 2016)
- Należy użyć reprezentacji rejestrującej tylko **wystąpienia** termów w dokumentach (z nie ich brak)

Inverted index

- Zapewnia szybki dostęp do wszystkich dokumentów zawierających dany term (plus info o częstości i pozycji w dokumencie)
- Dla każdego termu otrzymujemy listę krotek
  - (*docID*, *freq*, *pos*)
- Przy zapytaniu możemy pobrać takie listy dla wszystkich termów zapytania i pracować dalej na dokumentach związanych z zapytaniem
  - zapytania Boole'owskie: operacje na zbiorach
  - zapytania jęz. naturalnego: np. sumowanie wag termów
- Wszystkie listy muszą być posortowane – dostęp do list i krotek będzie w czasie logarytmicznym

Składniki indeksu odwrotnego

słownik (ang. *dictionary*, *vocabulary*) – zbiór wszystkich rozróżnialnych jednostek indeksujących (termów) w kolekcji

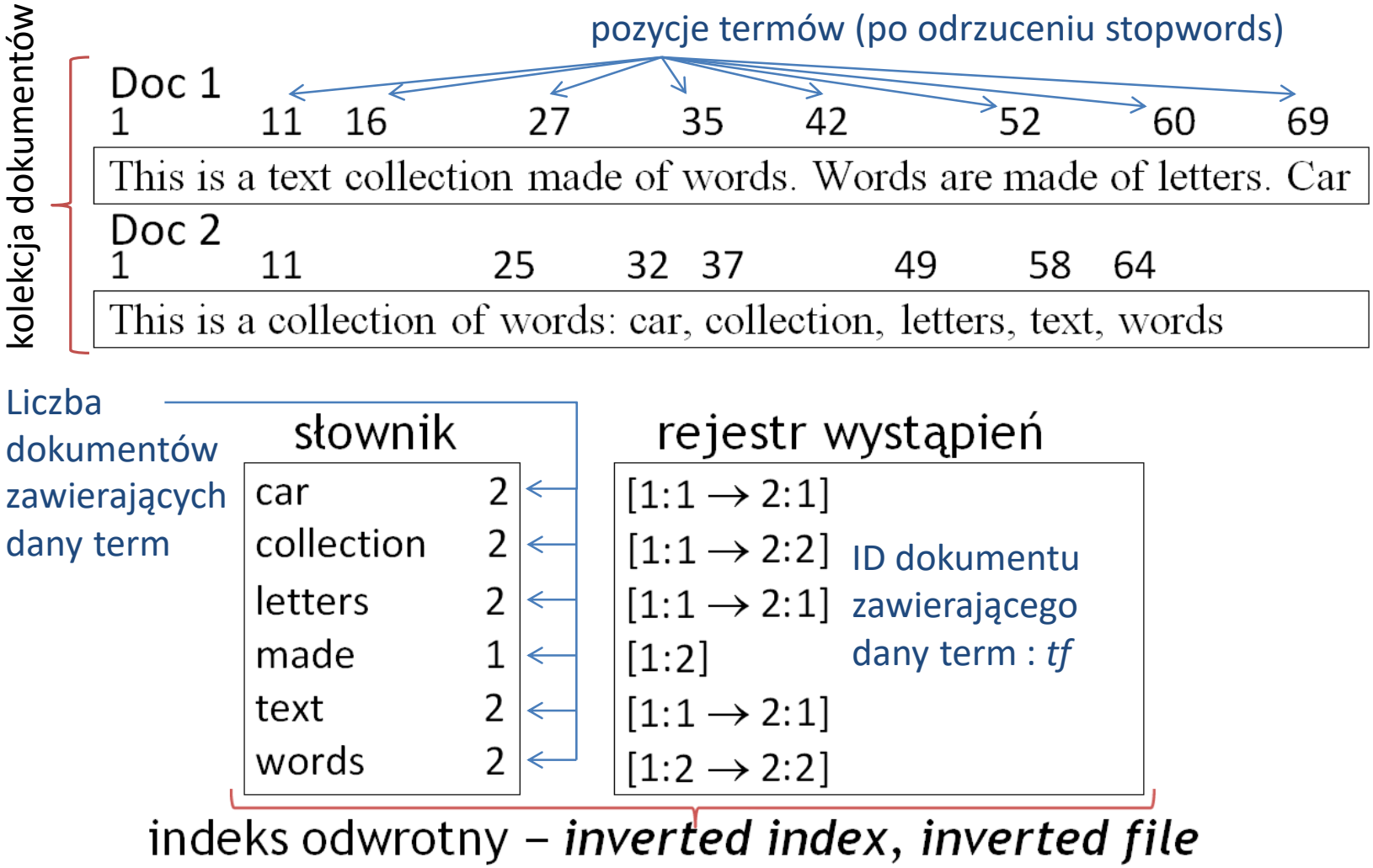
rejestr wystąpień (ang. *postings*, *occurences*) – zbiór list zawierających miejsca wystąpień odpowiadającego termu w kolekcji (z dokładnością do dokładnej pozycji termu w danym dokumencie – *full indexing*, lub do bloku tekstu (np. ID dokumentu) – *block indexing*)

warto przechowywać także info nt. liczby dokumentów zawierających dany term (*document frequency*), etc.

Inverted index

- Słownik: skromniejszy rozmiar
  - Potrzeba szybkiego dostępu do dowolnego miejsca
  - Najlepiej w pamieci operacyjnej
  - Hash table, B-tree, trie, …
- Rejest wystąpień: ogromny
  - Będzie przeszukiwany sekwencyjnie
  - Może być przechowywany na dysku
  - Może zawierać IDs dokumentów, częstości termów, pozycje termów, itd..
  - Zalecana kompresja

Inverted index example



Positional inverted index example

1	11	16	27	35	42	52	60	69
This is a text collection made of words. Words are made of letters. Car								

1	11	25	32	37	49	58	64
This is a collection of words: car, collection, letters, text, words							

słownik

car	2
collection	2
letters	2
made	1
text	2
words	2

rejestr wystąpień (w/ *positional postings*)

[1:[69] → 2:[32]]
[1:[16] → 2:[11,37]]
[1:[60] → 2:[49]]
[1:[27, 52]]
[1:[11] → 2:[58]]
[1:[35, 42] → 2:[25,64]]

IDs dokumentów
+ pozycje termów
tu: wyznaczone
względem znaków

pełny indeks odwrotny - ***full inverted index***

Rejestr wystąpień może zawierać *pozycje* termów wyznaczone względem pojedynczych znaków lub względem słów (przykład w materiałach lab.)

Positional i.e. Full inverted index

Rejestr wystąpień może zawierać pozycje wyznaczone względem termów lub pojedynczych znaków

- pozycje względem termów (*i*-ta pozycja odpowiada *i*–temu termowi) wspomagają zapytania gdzie ważna jest bliskość słów (jak np. frazy)
- pozycje względem znaków (*i*-ta pozycja odpowiada *i*–temu znakowi) ułatwiają bezpośredni dostęp do odpowiednich pozycji w tekście

Inverted index vs. Full inverted index

- Przestrzeń słownika wg. prawa Heapsa – $O(N^{\beta})$

  N – rozmiar kolekcji dokumentów

  β - stała z przedziału [0; 1] charakterystyczna dla kolekcji, w praktyce $\beta \in$ [0.4; 0.6]

  Czyli rozmiar słownika rośnie *proporcjonalnie do pierwiastka kwadratowego* rozmiaru kolekcji

Inverted index vs. Full inverted index

- Przestrzeń rejestru wystąpień – $O(N)$

  w praktyce jest to ok. 30%-40% rozmiaru kolekcji (eliminacja znaczników)

  Powyższe szacowanie dotyczy tzw. „pełnych indeksów”

  Stosując adresowanie blokowe i określanie pozycji termu poprzez *numer bloku* (np. *Id dokumentu*) możliwe jest stworzenie indeksu o rozmiarze ok. 5% rozmiaru kolekcji

Block addressing in an inverted index

- Kolekcja zostaje podzielona na bloki
- Bloki o stałym rozmiarze lub bloki ‘naturalne’ (pliki, strony internetowe, etc.) 1 *block* = 1 *retrieval unit*
- Listy wystąpień zawierają IDs bloków bez pozycji
- Oszczędność pamięci - kilku wystąpieniom danego termu w jednym bloku odpowiada tylko jedna pozycja na jego liście postings
- Granularność powoduje, że do obsługi części zapytań konieczne będzie sekwencyjne przeszukiwanie wewnątrz bloków tekstu

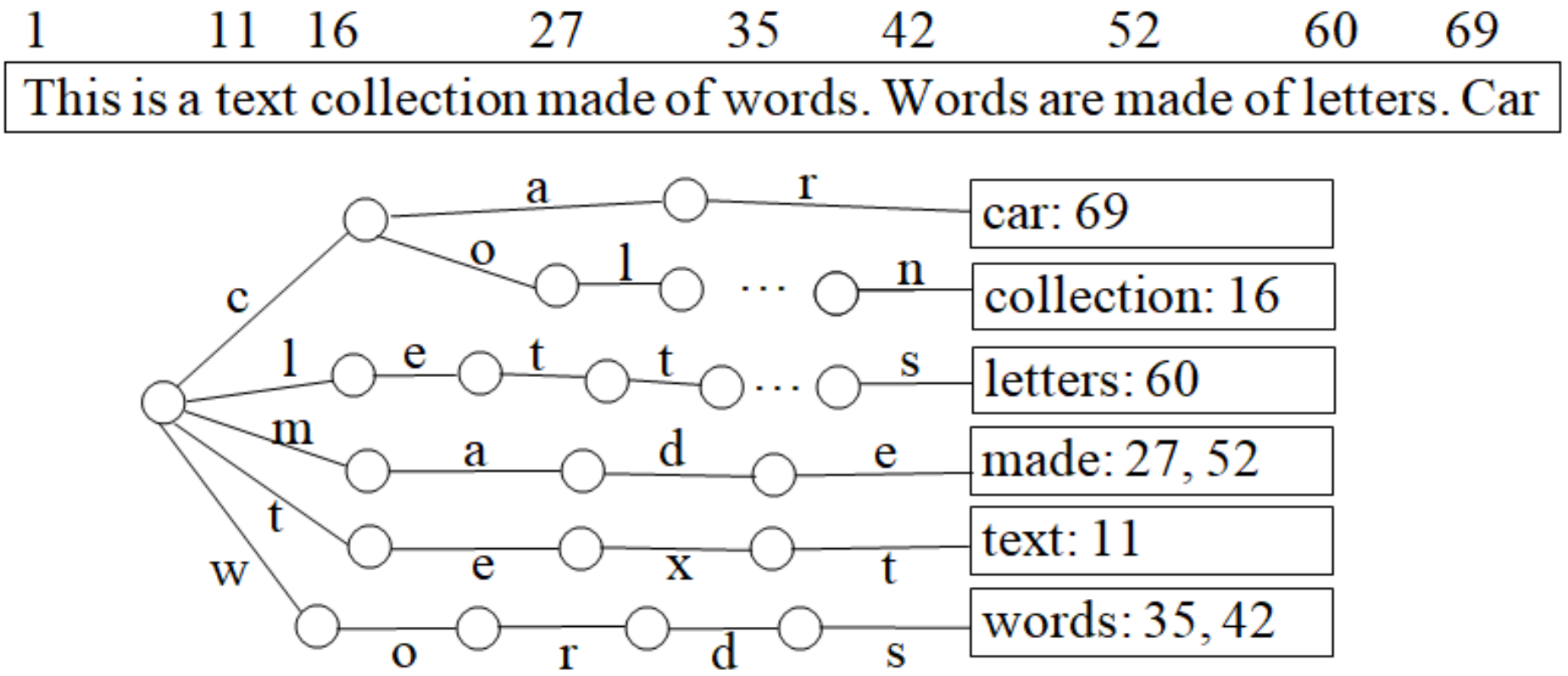
Inverted index construction

W ogólności budowa indeksu ma złożoność czasową $O(N)$ – wymaga liniowego przejścia przez wszystkie dokumenty kolekcji

Podczas konstrukcji indeksu dla mniejszych kolekcji tekstów można wykorzystać strukturę zwaną *trie* do przechowywania termów i ich wystąpień w kolekcji

Rozwiązanie bardzo efektywne, gdy *trie* mieści się w pamięci operacyjnej

Trie for inverted index construction



drzewo *trie* w budowie indeksu odwrotnego

Trie for inverted index construction

Własności drzewa trie

- Każda krawędź drzewa *trie* jest etykietowana pojedynczym znakiem
- Każda krawędź wychodząca z danego węzła musi być etykietowana innym znakiem
- Każdemu indeksowanemu ciągowi znaków (tu: termowi indeksującemu/słowu kluczowemu) odpowiada jedna ścieżka od korzenia drzewa do liścia

Inverted index construction

Dla większych kolekcji, których indeksy nie mieszczą się w pamięci operacyjnej, proces tworzenia indeksu będzie kontynuowany do wyczerpania pamięci. Następnie tak powstały indeks częściowy I_1 jest zapisywany na dysku, pamięć jest czyszczona i wykorzystana do utworzenia kolejnego indeksu częściowego I_2, itd.

Na dysku zostaje utworzonych szereg indeksów częściowych I_i , które agreguje się hierarchicznie parami (I_1 z I_2 daje $I_{1,2}$; I_3 z I_4 daje $I_{3,4}$;...;następnie $I_{1,2}$ z $I_{3,4}$ daje $I_{1,4}$; itd.) W ten sposób powstaje indeks całkowity

Inverted index construction

General steps:

1. Make a pass through *a part of* the collection assembling all *term–docID* pairs
2. Sort the pairs with the *term* as the dominant key and *docID* as the secondary key
3. Organize the *docID*s for each *term* into a postings list and compute statistics like term and document frequency
4. Store *intermediate* results (*runs*) on disk

Inverted index construction

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

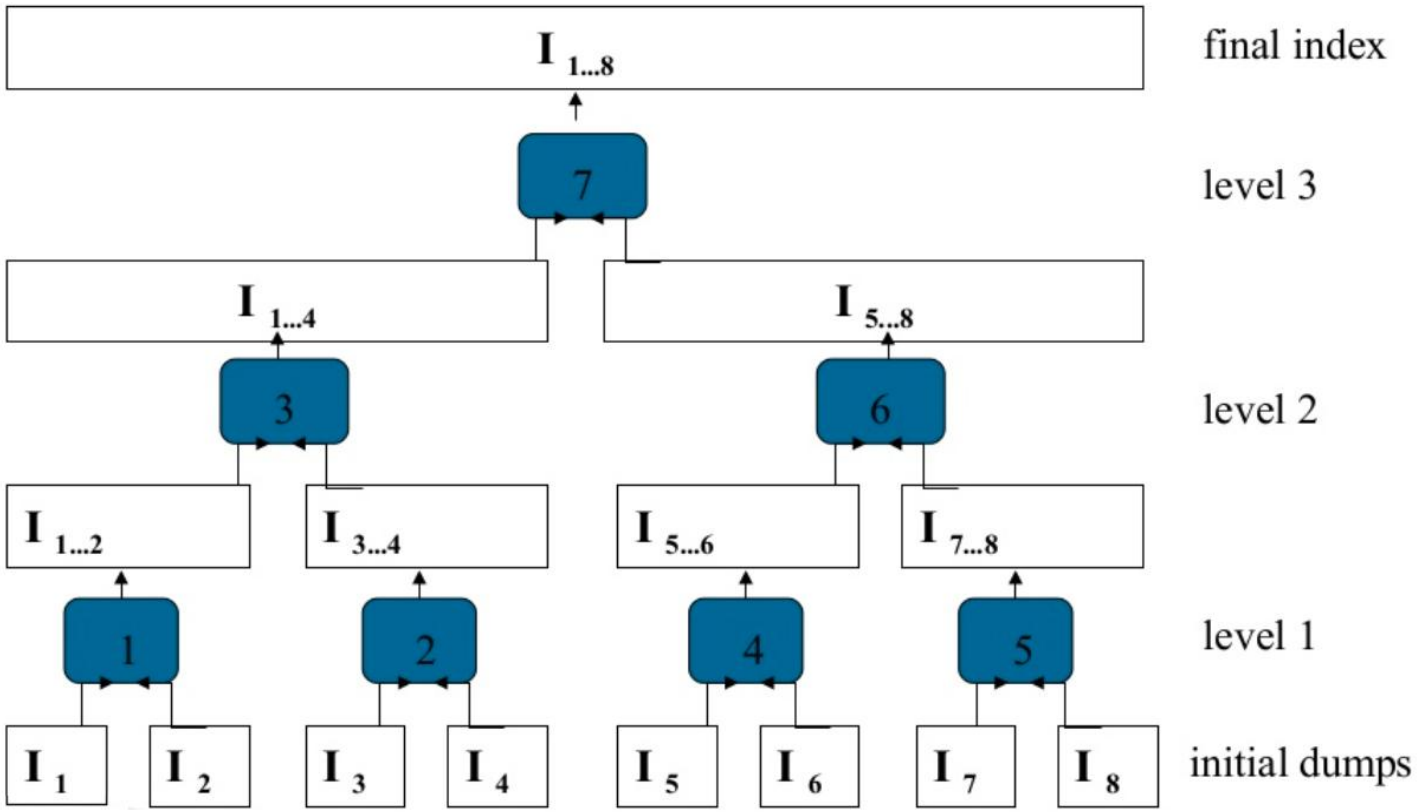
SORT by term

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

STORE

term	doc freq	postings
ambitious	1	2
be	1	2
brutus	2	1 → 2
capitol	1	1
caesar	2	1 → 2
did	1	1
enact	1	1
hath	1	2
I	1	1
i'	1	1
it	1	2
julius	1	1
killed	1	1
let	1	2
me	1	1
noble	1	2
so	1	2
the	2	1 → 2
told	1	2
you	1	2
was	2	1 → 2
with	1	2

Inverted index construction



Źródło: *https://www.slideshare.net/PRINCEOFSUNCITY/inverted-index*

Inverted index construction

Agregacja dwóch indeksów polega na sumowaniu ich posortowanych słowników, przy czym dla każdego słowa występującego w słownikach obu indeksów należy połączyć odpowiednie listy wystąpień

Zauważmy, iż proces konstrukcji gwarantuje, iż pozycje wystąpień z indeksu I_{i+1} można po prostu dołączyć na końcu listy pozycji indeksu I_i (ID dokumentów będą posortowane; tak samo pozycje termów w indeksie pełnym)

Złożoność czasowa łączenia dwóch indeksów to $O(N_1+N_2)$, gdzie N_1 i N_2 to rozmiary indeksów łączonych

Inverted index construction

Liczba indeksów częściowych jest rzędu $O(N/M)$, gdzie M to rozmiar pamięci operacyjnej

Na każdym poziomie łączy się indeksy częściowe odpowiadające całemu indeksowi a zatem koszt czasowy procesu jest rzędu $O(N)$

Aby połączyć $O(N/M)$ indeksów częściowych potrzeba $\log_2 (N/M)$, poziomów

A zatem koszt całego procesu łączenia indeksów częściowych w jeden spójny indeks całkowity jest rzędu $O(N \log (N/M))$

Inverted index construction

Dodatkowe techniki usprawniające:

- łączenie więcej niż 2 indeksów na raz (mniej poziomów hierarchii ale więcej odwołań do pamięci dyskowej)
- oszczędzanie przestrzeni wykorzystywanej podczas budowy indeksu poprzez zapisywanie rezultatu łączenia indeksów częściowych w tych samych blokach
- łączenie kolejnych indeksów na bieżąco – podczas ich powstawania (oszczędność przestrzeni dyskowej poprzez natychmiastową eliminację powtarzających się słów)

Inverted index maintenance

Konserwacja raz utworzonego indeksu jest również *stosunkowo* tania:

- Po dodaniu do kolekcji tekstu o rozmiarze N' wystarczy zbudować dla niego indeks częściowy, który jest następnie łączony z dotychczasowym indeksem w czasie $O(N + N' \log (N'/M))$
- Usuwanie tekstu z kolekcji: $O(N)$

Eksploatacja indeksu odwrotnego

szukanie za pomocą indeksu obejmuje 3 kroki:

- Przeszukanie słownika (ang. *vocabulary search*)
- Wyszukanie wystąpień (ang. *retrieval of occurences*)
- Operacje na wystąpieniach (ang. *manipulation of occurrences*)

Vocabulary search

Wyszukanie w słowniku wszystkich termów użytych w zapytaniu (frazy są rozbijane na pojedyncze termy)

Proste leksykograficzne uporządkowanie słownika umożliwia wyszukiwanie w czasie $O(\log N)$ bez dodatkowych nakładów pamięci

Warto przechowywać słownik w osobnym pliku, szczególnie jeśli jego rozmiar mieści się w pamięci operacyjnej

Vocabulary search

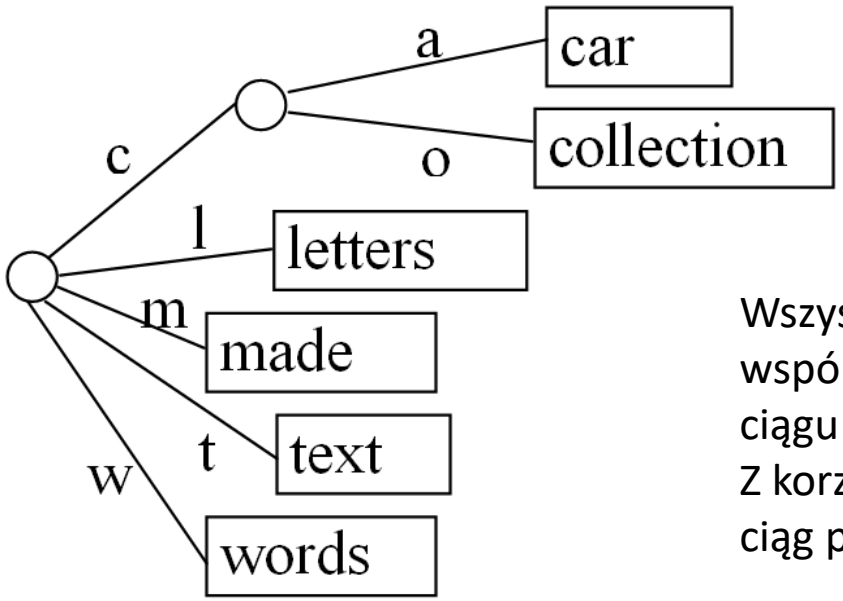
Wykorzystanie dodatkowych struktur usprawnia wyszukiwanie

Odszukanie w słowniku pojedynczego termu o długości m ma złożoność czasową rzędu $O(m)$ przy zastosowaniu dodatkowych typu ‘trie’, lub *const* przy wykorzystaniu struktur haszujących

Struktury takie stanowią oczywiście dodatkowe obciążenie zasobów pamięci

Vocabulary search with trie

11	16	27	35	42	52	60	69
This	is	a	text	collection	made	of	words. Words are made of letters. Car



Wszyscy potomkowie węzła mają wspólny prefiks indeksowanego ciągu znaków
Z korzeniem drzewa jest skojarzony ciąg pusty

słownik wykorzystujący drzewo *trie*

Vocabulary search with trie

Drzewo *trie* to drzewo poszukiwań przechowujące w węzłach fragmenty kluczy, co pozwala przyspieszyć wyszukiwanie, w stosunku do porównania całego klucza

Dostępne opercje z wykorzystaniem drzewa *trie*:

- sprawdzenie, czy słowo jest w drzewie
- znalezienie najdłuższego prefiksu słowa występującego w drzewie
- wyszukanie wszystkich słów o podanym prefiksie

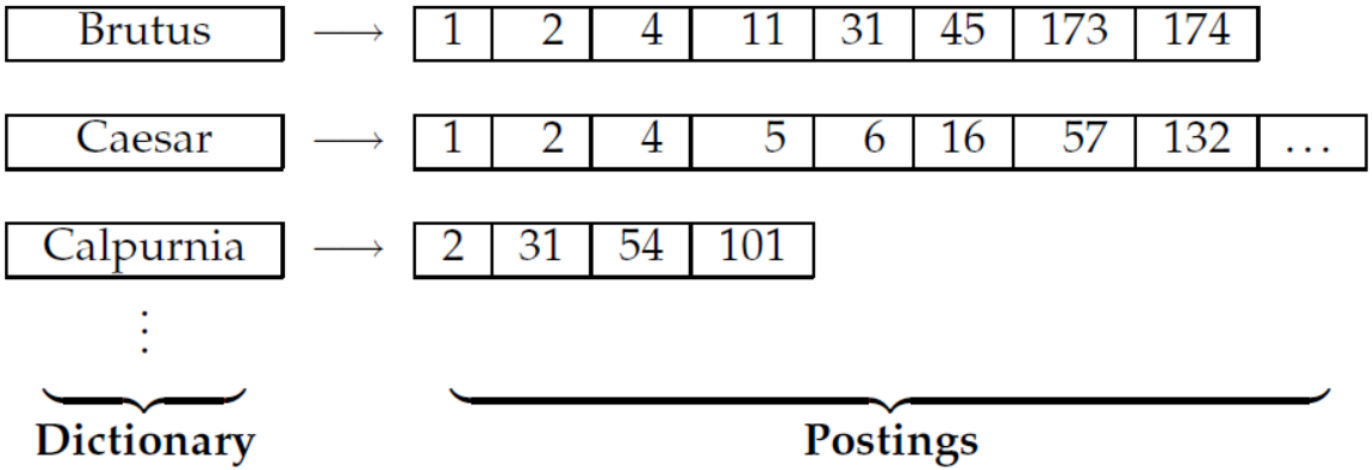
Retrieval of occurrences

Wyszukanie wystąpień dla zapytania w postaci pojedynczego termu daje pojedynczą listę wystąpień tego termu w kolekcji

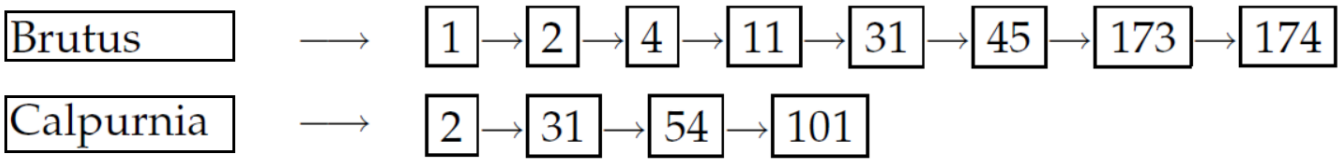
Dokumenty z tak wyłonionej listy wystąpień mogą zostać zwrócone jako odpowiedź na zapytanie (czasami bez potrzeby dodatkowego przetwarzania – jeśli listy wystąpień były uporządkowane nie wg ID dokumentów, ale np. miary TF, itp.)

Dla zapytań angażujących kilka termów wynikiem jest zbiór list wystąpień

Retrieval of occurrences



The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk



postings lists for Brutus Calpurnia

Część rysunków w tej prezentacji pochodzi z książki *http://nlp.stanford.edu/IR-book/*

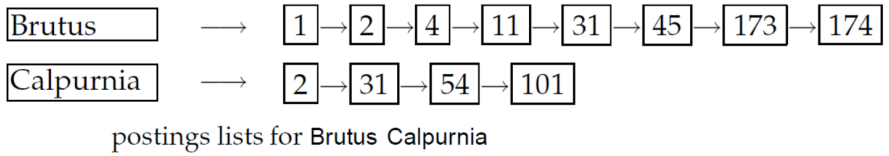
Manipulation of occurrences

Obsługując zapytanie typu **OR** wystarczy zwrócić jako odpowiedź sumę zbiorów dokumentów z wyłonionych list postings

Obsługa zapytań typu **AND** wymaga identyfikacji dokumentów, które zawierają wszystkie termy z zapytania – czyli wyznaczenia **iloczynu** zbiorów dokumentów z wyłonionych list postings

Warto wykorzystać fakt, iż niektóre listy postings będą znacznie krótsze od innych

Obsługa zapytania $term_1$ **AND** $term_2$



postings lists for Brutus Calpurnia

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

Algorithm for the intersection of two postings lists p_1 and p_2

Manipulation of occurrences

Obsługa zapytania $term_1$ **AND** $term_2$ … **AND** $term_n$

```
INTERSECT(⟨t_1, ..., t_n⟩)
1  terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
2  result ← postings(first(terms))
3  terms ← rest(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do result ← INTERSECT(result, postings(first(terms)))
6     terms ← rest(terms)
7  return result
```

Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms

Searching with inverted index

- Wykazano, że cały proces eksploatacji indeksu odwrotnego ma złożoność czasową $O(N^{\alpha})$

  N – rozmiar kolekcji

  α – stała zależna od zapytania; dla zapytań o przeciętnej selektywności przyjmująca wartości z przedziału [0.4; 0.8]

Phrase queries

W przypadku *pełnego* indeksu odwrotnego (zawierającego tzw. *positional postings*) jeżeli poszukiwana jest *fraza* – czyli uporządkowana sekwencja termów – trzeba podczas synchronicznego przeglądania wszystkich list odpowiadających termom tworzącym frazę dodatkowo porównywać informację odnośnie pozycji termów – tak aby zidentyfikować te dokumenty, gdzie wszystkie słowa występują w zadanym porządku

Tu również warto wykorzystać fakt, iż niektóre listy wystąpień są znacznie krótsze od innych

Phrase queries

Przy wyszukiwaniu fraz z użyciem indeksu odwrotnego z adresowaniem blokowym (np. zawierającego na listach wystąpień tylko IDs dokumentów) także oblicza się iloczyn zbiorów – list wystąpień poszczególnych słów w celu identyfikacji tych bloków (dokumentów), które zawierają wszystkie słowa frazy

Wewnątrz tak zidentyfikowanych bloków (dokumentów) trzeba następnie zastosować przeglądanie sekwencyjne

Biword indices

Do wyszukiwania fraz można też wykorzystać indeks odwrotny ze słownikiem opartym na *bigramach* (word 2–grams)

- indeks taki umożliwia bezpośrednią obsługę zapytań dwuwyrazowych (dość typowe zapytania)
- dłuższe zapytania mogą być obsłużone poprzez rozbicie ich na bigramy i wyłonienie tylko tych dokumentów, które zawierają wszystkie te bigramy

Biword indices

Do wyszukiwania fraz można też wykorzystać indeks odwrotny ze słownikiem opartym na *bigramach* (word 2–grams)

- indeks taki umożliwia bezpośrednią obsługę zapytań dwuwyrazowych (dość typowe zapytania)
- dłuższe zapytania mogą być obsłużone poprzez rozbicie ich na bigramy i wyłonienie tylko tych dokumentów, które zawierają wszystkie te bigramy
- ryzyko błędnych dopasowań (*false positive matches*)

„to be or not to be” → „to be” AND „be or” AND „or not” AND „not to”

Biword indices and Phrase indices

Wykorzystanie indeksu ze słownikiem wykorzystującym dłuższe ($n > 2$) n–gramy znacznie minimalizowałoby ryzyko błędnych dopasowań

Jednak wykorzystanie termów indeksujących odpowiadających wszystkim ciągom n słów występujących w indeksowanym tekście (czy nawet samym bigramom) może nie być opłacalne

Prowadziłoby do szybkiego *puchnięcia* indeksu, przy czym wiele takich termów nigdy nie wystąpiłoby w zapytaniach użytkowników

A równolegle trzeba by też utrzymywać słownik termów dla pojedynczych słów, aby efektywnie obsługiwać zapytania jednowyrazowe

Biword indices and Phrase indices

Warto natomiast dodać do słownika popularne pary (ew. niektóre dłuższe ciągi) słów, jak np. imiona i nazwiska sławnych osób, nazwy własne, tytuły, etc., które stanowią popularne zapytania użytkowników

Największe jednak korzyści może dać uwzględnienie *n*–gramów takich termów, których składowe są słowami bardzo pospolitymi (np. '*dr no*', '*the who*'), nawet jeśli nie są to bardzo popularne zapytania

Combined index

W praktyce do obsługi zapytań typu fraza dość dobrze sprawdza się index mieszany – ang. *combined index* – – bądący połączeniem indeksu odwrotnego zawierającego pojedyncze słowa kluczowe i wybrane termy złożone z większej liczby słów kluczowych oraz indeksu pozycyjnego

Szacuje się jednak, że indeksy pozycyjne (pełne) są średnio 2-4 razy (w zależności od specyfiki kolekcji i języka dokumentów) większe od ‘zwykłych’

Tablice sufiksów

Umożliwiają efektywną obsługę zapytań, które wymagają wyszukania w tekstach ciągów słów (fraz) lub ciągów znaków (gdy tekstowa baza danych jest kolekcją dokumentów NIE złożonych ze słów)

- Indeksowany tekst jest traktowany jako jeden długi ciąg znaków
- Indeksowane pozycje w tekście (mogą to być same początki słów kluczowych) odpowiadają początkom leksykograficznie różnych sufiksów tekstu

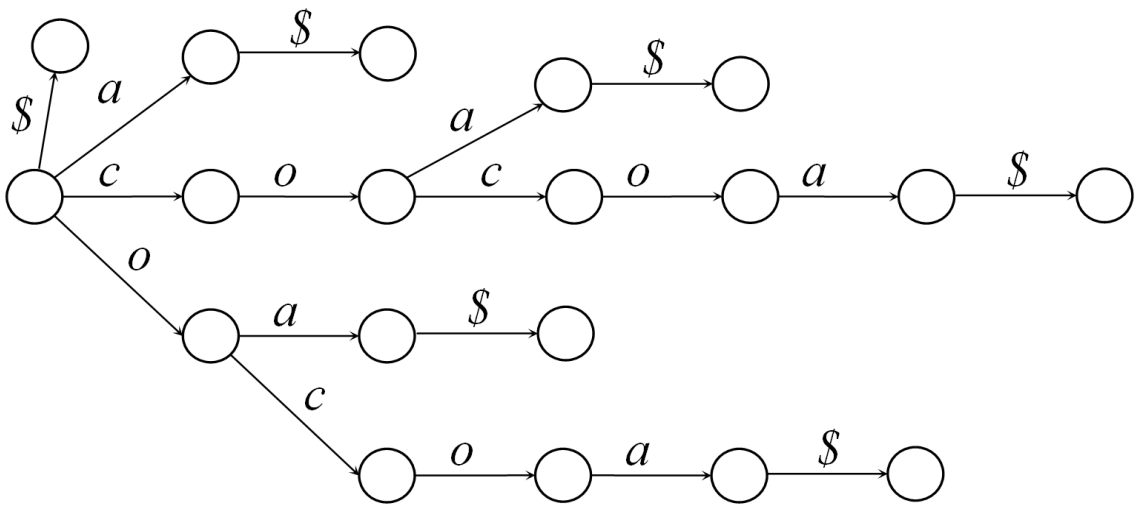
Prefixes and suffixes

Prefiks to określenie początkowych znaków ciągu; ciąg *S* jest prefiksem ciągu znaków *U* jeśli *U*=*SU*', gdzie *U*' jest ciągiem znaków

Sufiks to określenie końcowych znaków ciągu; ciąg *S* jest sufiksem ciągu znaków *U* jeśli *U*=*U*'*S*, gdzie *U*' jest ciągiem znaków

Nieskompresowane drzewo sufiksów

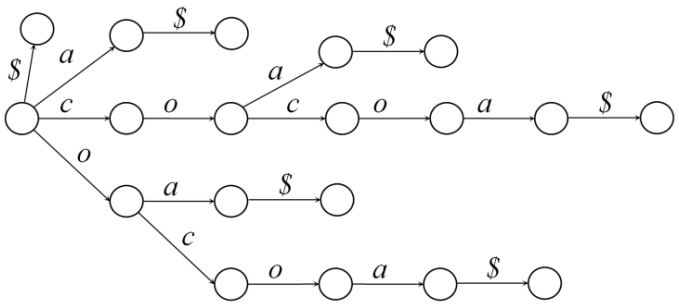
Suffix trie to struktura drzewiasta *trie* służąca do przechowywania ciągów znaków, w której każdy indeksowany sufiks ciągu można znaleźć na ścieżce od korzenia do któregoś liścia



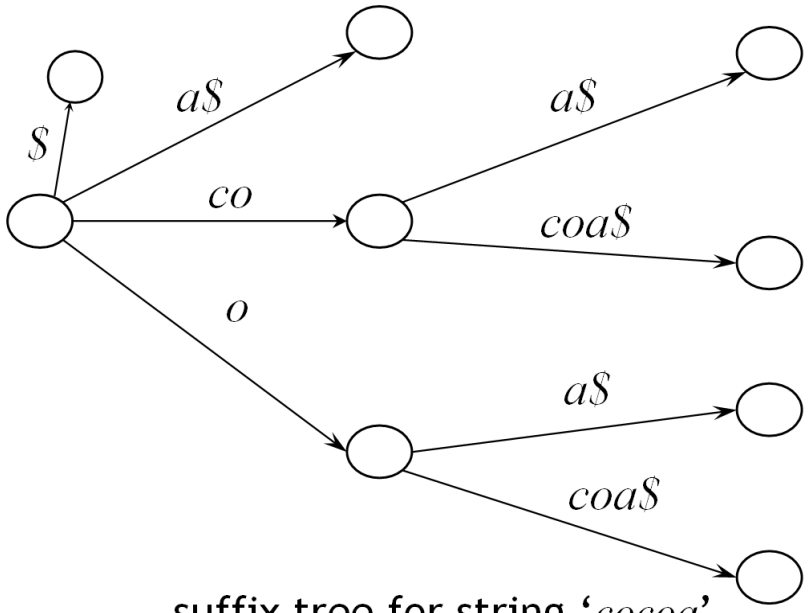
suffix trie for string '*cocoa*'

Drzewo sufiksów

Suffix tree to bardziej zwięzła forma struktury *suffix trie* eliminująca każdy węzeł o pojedynczym potomku (poprzez połączenie takiego węzła z potomkiem w jeden węzeł)



suffix trie for string '*cocoa*'



suffix tree for string '*cocoa*'

Suffix tree

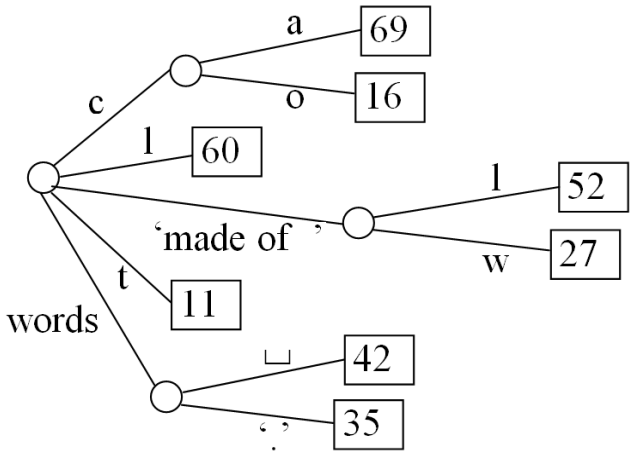
- Każdy węzeł drzewa sufiksów (za wyjątkiem korzenia) ma co najmniej dwóch potomków a każda krawędź jest etykietowana niepustym podciągiem ciągu znaków S
- Każda krawędź wychodząca z danego węzła musi mieć etykietę o innym znaku początkowym
- Etykieta węzła (lub liścia) w drzewie sufiksów powstaje z połączenia ciągów etykietujących krawędzie tworzące ścieżkę od korzenia drzewa do danego węzła (liścia)
- Gdy indeksowana jest każda pozycja tekstu drzewo sufiksów dla ciągu S złożonego z m znaków ma dokładnie m liści (+1 dla $\$$)

Searching with suffix trees

Gdy indeksowane są tylko pozycje tekstu odpowiadające początkom słów kluczowych funkcjonalność drzewa sufiksów jest podobna do funkcjonalności indeksu odwrotnego

Ale wyszukanie dowolnej frazy o długości m – w czasie $O(m)$

11	16	27	35	42	52	60	69
This is a text collection made of words. Words are made of letters. Car							



suffix tree for a sample text

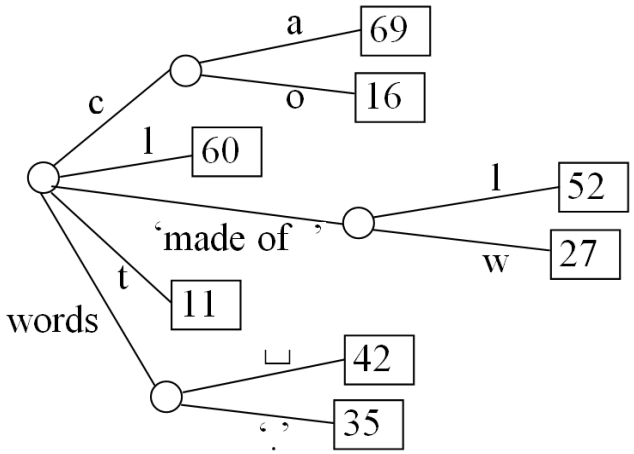
w liściach drzewa mamy wskaźniki do pozycji odpowiadających początkom indeksowanych sufiksów tekstu

Searching with suffix trees

Gdy indeksowane są tylko pozycje tekstu odpowiadające początkom słów kluczowych f
jest podobna do
Ale wyszukanie dowolnej fraz

wpisując do wyszukiwarki zapytanie ***ość*** oczekujemy informacji o ościach, a nie dokumentów zawierających słowa „*mość*”, „*ilość*”, „*złość*”, ...

11 16 27 35 42
This is a text collection made of words. Words



suffix tree for a sample text

w liściach drzewa mamy wskaźniki do pozycji odpowiadających początkom indeksowanych sufiksów tekstu

Suffix trees vs. Suffix arrays

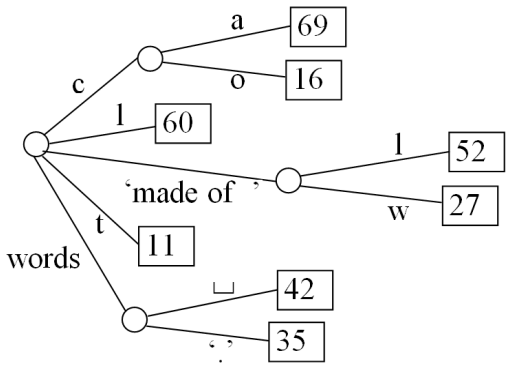
- Przestrzeń potrzebna do przechowywania drzewa sufiksów indeksującego tylko początki słów kluczowych tekstu o rozmiarze N – $O(N)$
- W praktyce jest to ok. 120%-240% rozmiaru kolekcji
- Tablice sufiksów są znacznie bardziej ekonomiczne pod względem zajętości pamięci – zajmują przestrzeń ok. 40% rozmiaru kolekcji (czyli porównywalnie z indeksem odwrotnym), zatem mogą być stosowane dla dużych kolekcji

Suffix array

Tablica sufiksów to posortowana tablica wszystkich sufiksów danego ciągu, a ściślej **tablica zawierająca wskaźniki do wszystkich indeksowanych sufiksów** tekstu uporządkowanych leksykograficznie

Kolejność odpowiada przejściu po kolejnych liściach drzewa sufiksów (tu: od górnego liścia do dolnego)

11	16	27	35	42	52	60	69
This	is a	text	collection	made of	words.	Words are	made of letters. Car



69	16	60	52	27	11	42	35

suffix array for a sample text

Phrase queries w/ suffix arrays

Sama tablica sufiksów (bez dodatkowych struktur) pozwala na wyszukiwanie dowolnej długości fraz w zindeksowanym tekście poprzez wykonanie przeszukiwań binarnych

W tablicy należy wyznaczyć przedział wskaźników do sufiksów tekstu, które rozpoczynają się szukaną frazą

Wskaźniki te sąsiadują w tabeli ze względu na leksykograficzne uporządkowanie wszystkich sufiksów tekstu

11	16	27	35	42	52	60	69
This is a text collection made of words. Words are made of letters. Car							

query: „made of letters”

69	16	60	52	27	11	42	35

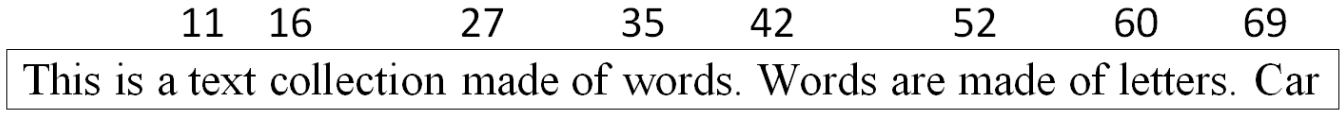
suffix array for a sample text

Phrase queries w/ suffix arrays

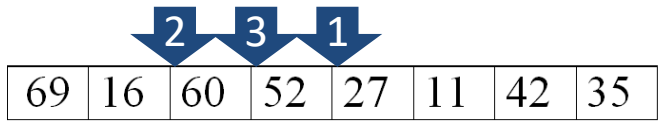
Sama tablica sufiksów (bez dodatkowych struktur) pozwala na wyszukiwanie dowolnej długości fraz w zindeksowanym tekście poprzez wykonanie przeszukiwań binarnych

W tablicy należy wyznaczyć przedział wskaźników do sufiksów tekstu, które rozpoczynają się szukaną frazą

Wskaźniki te sąsiadują w tabeli ze względu na leksykograficzne uporządkowanie wszystkich sufiksów tekstu



query: „made of letters”



suffix array for a sample text

Searching with suffix arrays

Eksploatacja indeksu opartego na tablicach sufiksów

W drzewie sufiksów proste operacje wyszukania słowa, frazy czy prefiksu o długości m można wykonać w czasie $O(m)$ – tzw. *trie search*

W tablicy sufiksów bez dodatkowych struktur te same operacje wymagają czasu $O(\log N)$ – *binary search*, ale niezbędne odwołania do pamięci dyskowej podczas binarnego przeszukiwania tablicy sufiksów podwyższają złożoność czasu wyszukiwań do $O(N \log N)$

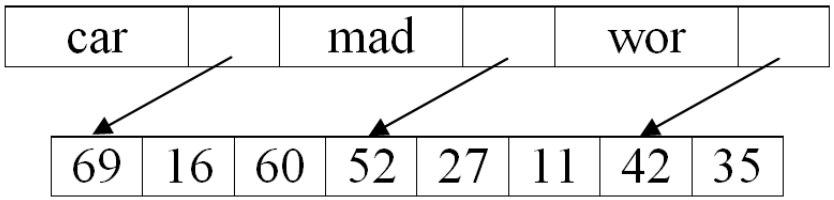
Searching with suffix arrays

Często dodatkowo przechowuje się tzw. *supra-index*

aby zminimalizować liczbę odwołań do dysku przy przeszukiwaniu binarnym zawartości wskaźników

Najprostszy supra-index powstaje gdy weźmiemy pierwszych l znaków z każdej jednej na b pozycji tablicy sufiksów

simple supra-index



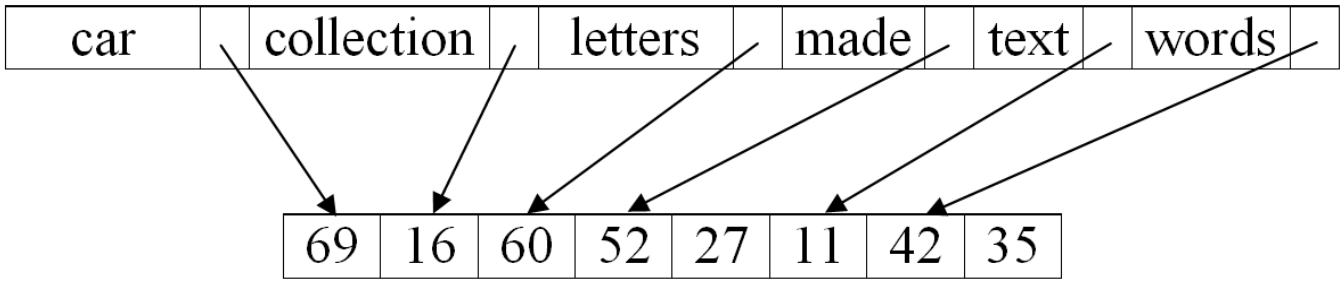
Ale wielkość przedziału próbkowania nie musi być stała

liczba znaków również

Searching with suffix arrays

Supra-index można zbudować np. z termów

Tablica sufiksów z takim supra-indeksem różni się od indeksu odwrotnego jedynie tym, że pozycje wystąpień są uporządkowane leksykograficznie względem tekstu, który występuje za danym termem

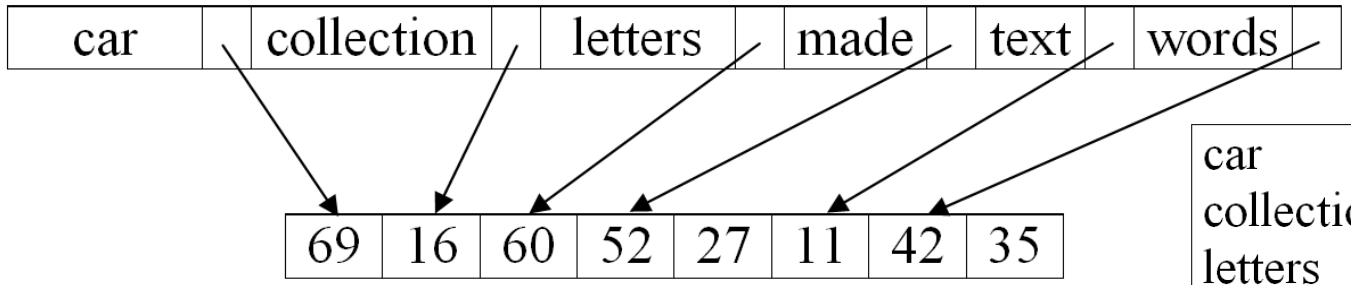


tablica sufiksów z supra-indeksem po termach

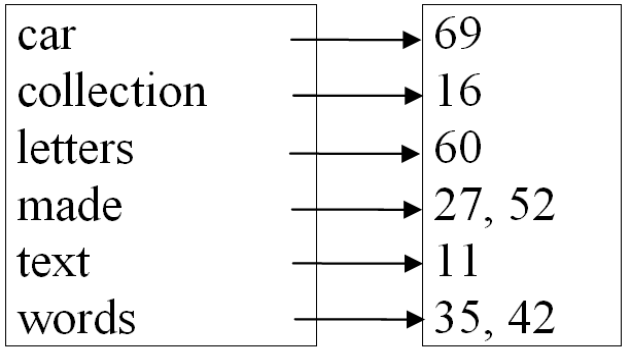
Searching with suffix arrays

Supra-index można zbudować np. z termów

Tablica sufiksów z takim supra-indeksem różni się od indeksu odwrotnego jedynie tym, że pozycje wystąpień są uporządkowane leksykograficznie względem tekstu, który występuje za danym termem



tablica sufiksów z supra-indeksem po termach



indeks odwrotny

Suffix arrays vs. Inverted index

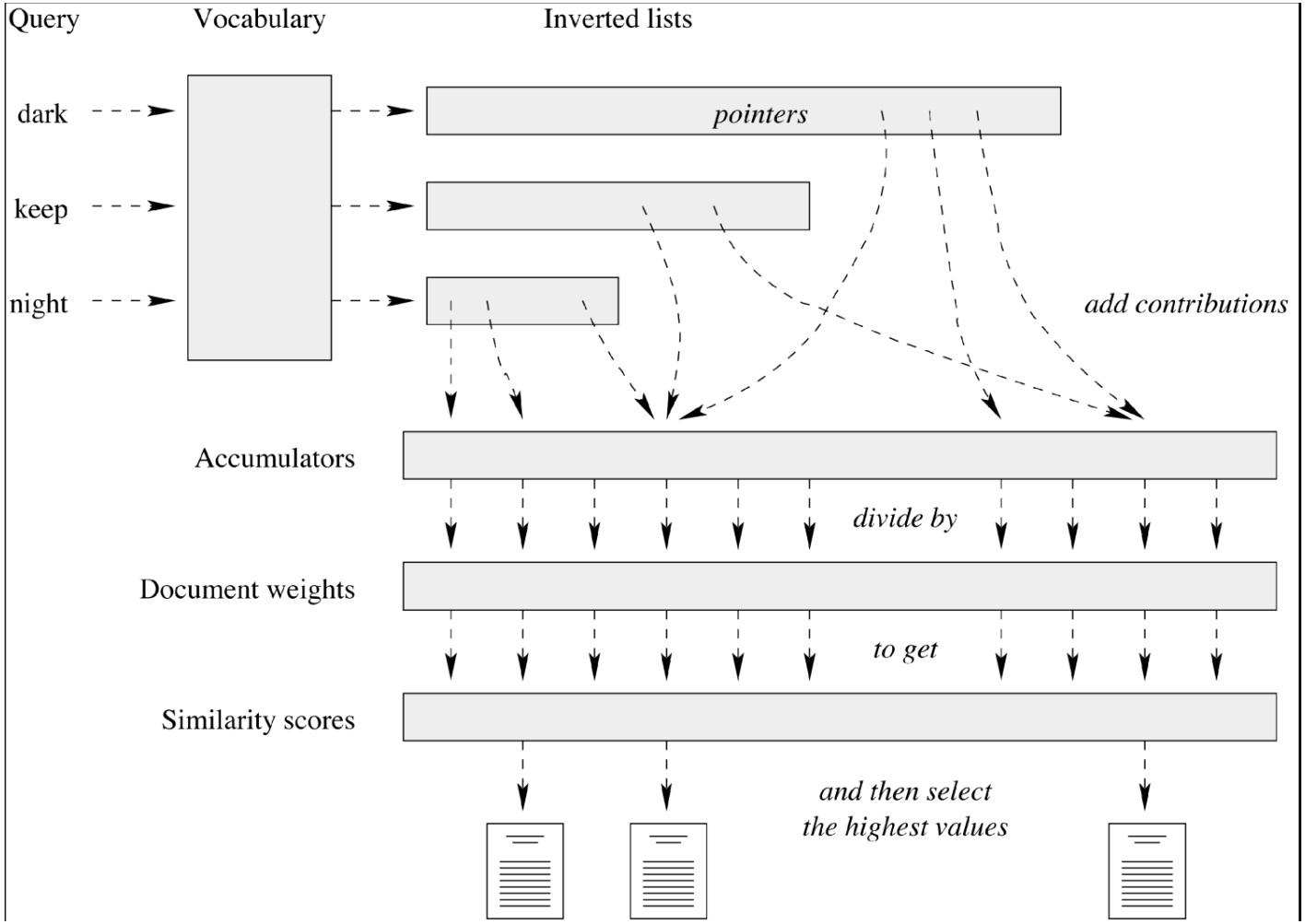
Jeżeli celem jest wyszukiwanie pojedynczych słów lub krótkich zapytań, to *SA* będą gorszym rozwiązaniem od zwykłego indeksu odwrotnego (kosztowniejsza budowa, pielęgnacja, kompresja, ...)

Częste wyszukiwanie długich fraz sankcjonuje wykorzystanie *SA* – dzięki nim czas wyszukiwania nie ulega pogorszeniu (w indeksie odwrotnym pogorszenie jest znaczne)

Free text queries

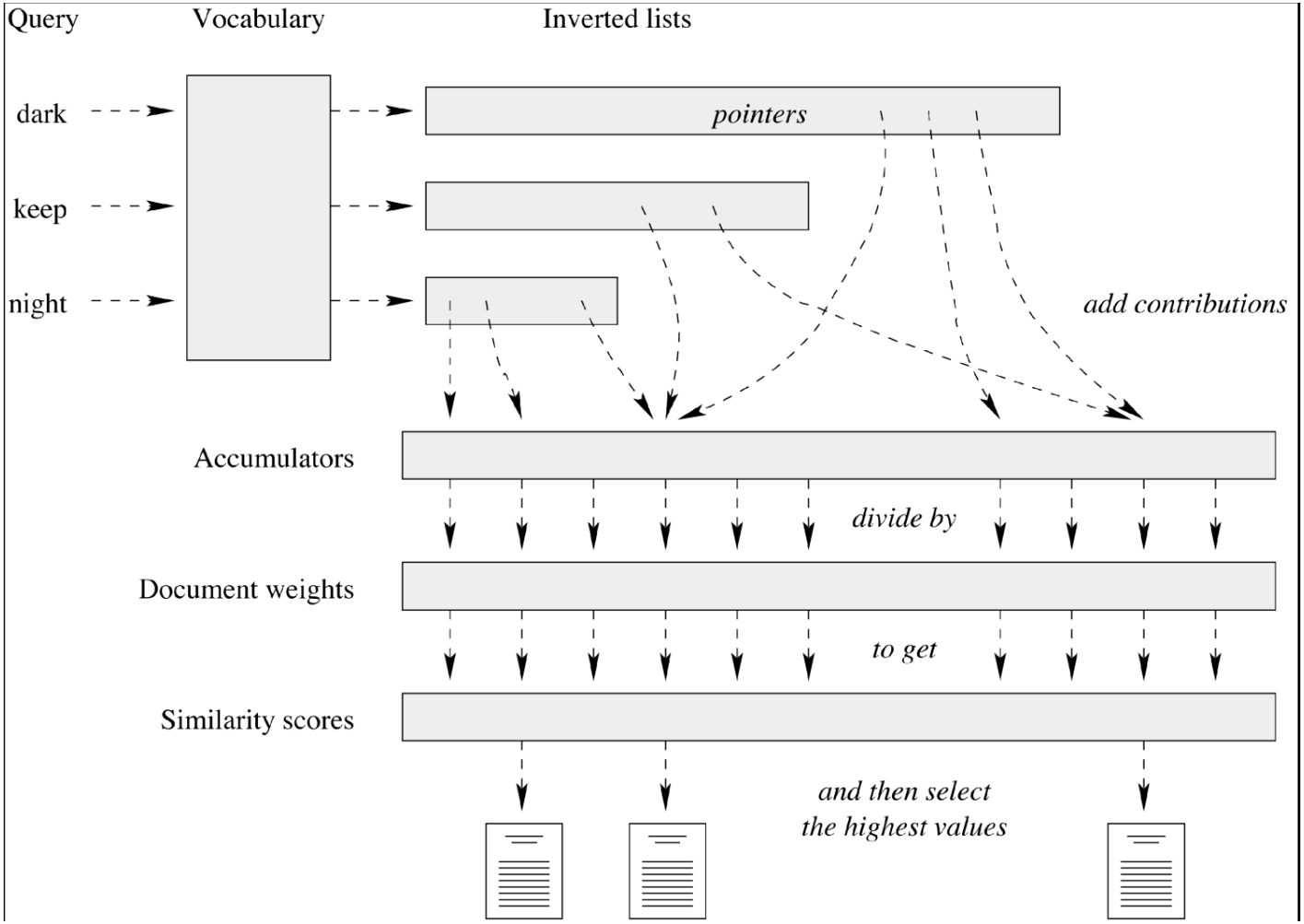
- Naturalne dla użytkowników wyszukiwarek interetowych
- Podobne do zapytań typu OR
- Oczekiwanym wynikiem jest lista rankingowa dokumentów – *ranked list* (SERP – *search engine results page*)
- Odpowiedź wymaga wykonania szybkiej agregacji wag termów z zapytania po listach *postings*

Ranked retrieval



J. Zobel, A. Moffat „*Inverted Files for Text Search Engines*" ACM Comput. Surveys, Vol. 38, No. 2

Ranked retrieval



$$a_{ij} = tf_{ij} \cdot idf_i$$

$$sim(d_j, q) = \frac{\vec{d}_j \bullet \vec{q}}{|\vec{d}_j| \cdot |\vec{q}|}$$

J. Zobel, A. Moffat „*Inverted Files for Text Search Engines*" ACM Comput. Surveys, Vol. 38, No. 2

Obsługa zapytania: $term_1\ term_2 \ldots term_n$

```
CosineScore(q)
 1  float Scores[N] = 0
 2  Initialize Length[N]
 3  for each query term t
 4  do calculate w_{t,q} and fetch postings list for t
 5      for each pair(d, tf_{t,d}) in postings list
 6      do Scores[d] += wf_{t,d} × w_{t,q}
 7  Read the array Length[d]
 8  for each d
 9  do Scores[d] = Scores[d] / Length[d]
10  return Top K components of Scores[]
```

accumulators

may employ *idfs*

for retrieved docs

The basic algorithm for computing vector space scores

Ranked retrieval

query: keep dark night
N = 80

log(80/4)
log(80/3)
log(80/1)

dark: 4 docs → (d1:3) (d2:4) (d3:1) (d4:5)
keep: 3 docs → (d2:3) (d4:1) (d5:3)
night: 1 doc → (d6:3)

accumulators		d1	d2	d3	d4	d5	d6
		0	0	0	0	0	0
(d1:3)	⇒	3	0	0	0	0	0
(d2:4)	⇒	3	4	0	0	0	0
(d3:1)	⇒	3	4	1	0	0	0
(d4:5)	⇒	3	4	1	5	0	0
(d2:3)	⇒	3	7	1	5	0	0
(d4:1)	⇒	3	7	1	6	0	0
(d5:3)	⇒	3	7	1	6	3	0
(d6:3)	⇒	3	7	1	6	3	3

Dla prostoty przykładu akumulatory tylko sumują *tf* (całkowie wartości)
Jednak dostępna informacja pozwala na uwzględnienie wartości *idf* termów

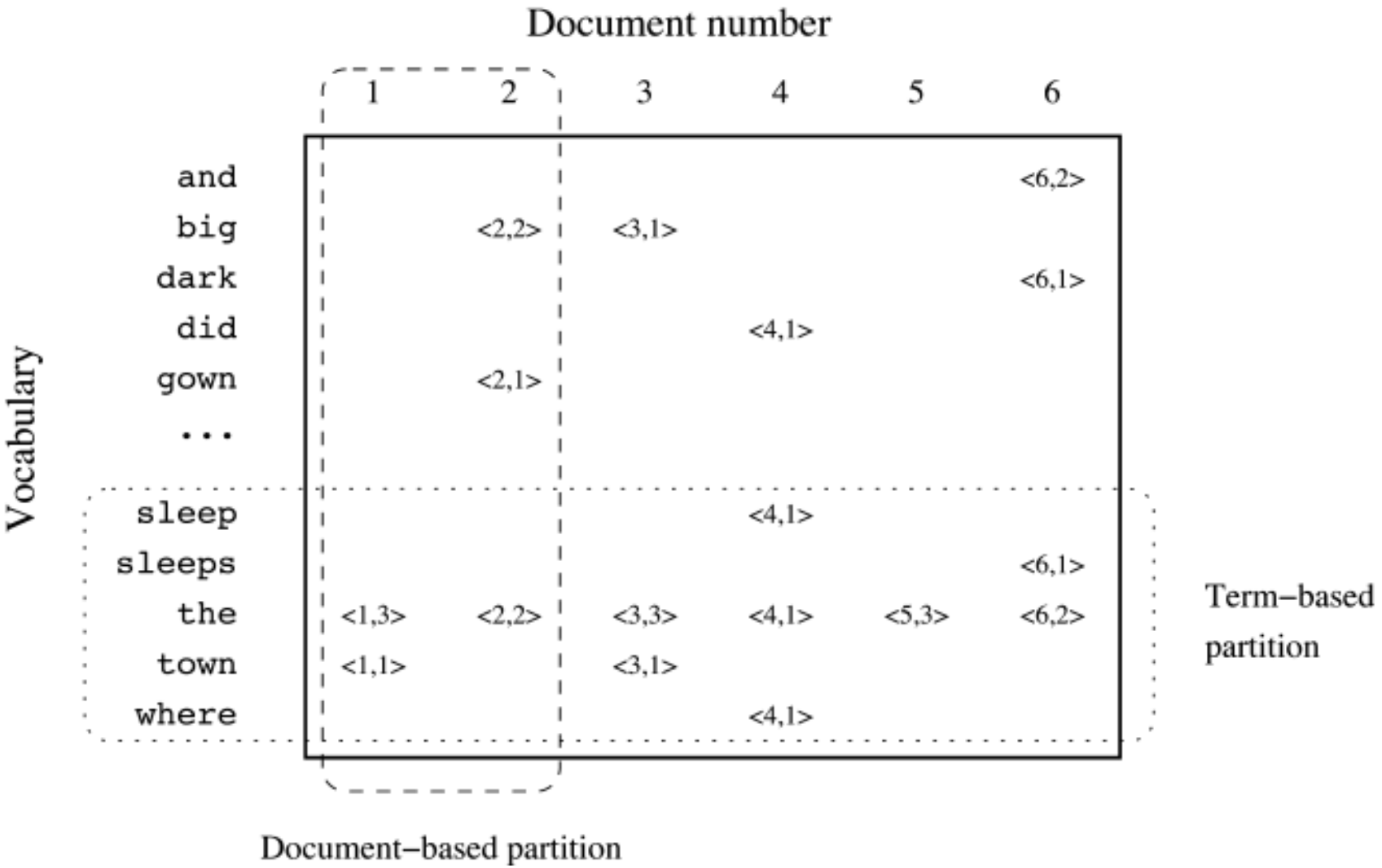
Ranked retrieval - efficiency

- Caching (np. gotowych list rankingowych dla popularnych zapytań, list *postings* wybranych termów)
- Ograniczenie obliczeń tylko do najbardziej obiecujących dokumentów (*fastest growing accumulators*)
- Przetwarzanie równoległe

Distributed indexing

- **Rozmiary** rzeczywistych kolekcji (*internet*) powodują, że proces budowy i eksploatacji indeksu nie może być efektywnie zrealizowany na pojedynczej maszynie
- Kosztowne jest także zapewnienie *fault tolerance* systemu: taniej jest wykorzystać wiele tanich jednostek niż pojedynczą ‘bezawaryjną’ jednostkę
- Do budowy rozsądnej wielkości indeksu w zastosowaniach webowych potrzeba dużych klastrów składających się z tysięcy maszyn
- Proces konstrukcji indeksu ma docelowo prowadzić do budowy **indeksu rozproszonego** (*distributed index*), który jest podzielony na wiele maszyn – podział ze względu na *termy*, albo na *dokumenty*

Distribution by terms / documents



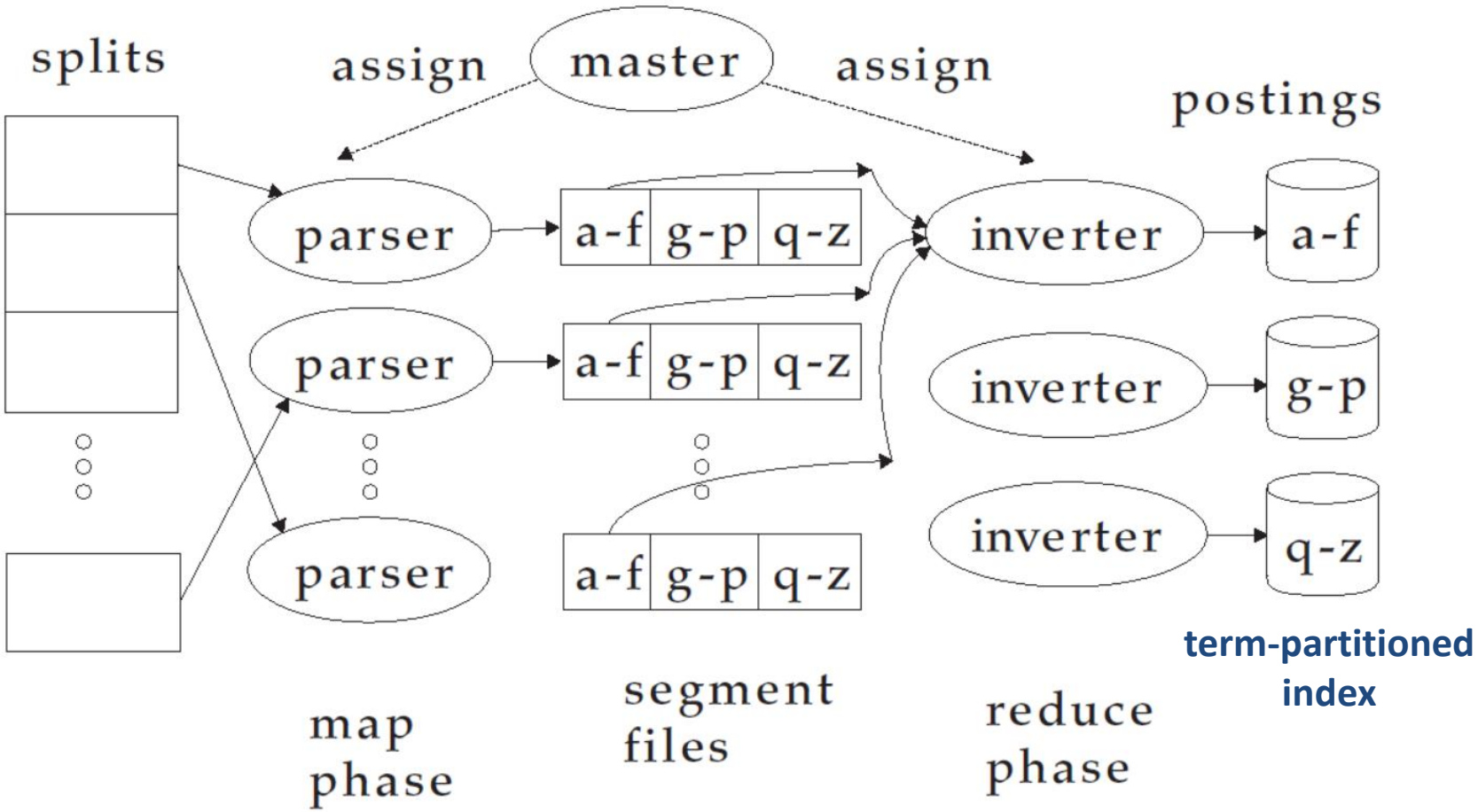
Nonpositional index construction

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
       then postings_list = ADDTODICTIONARY(dictionary, term(token))
       else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
       then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

Inversion of a block in single-pass in-memory indexing

yields document-partitioned index

Distributed indexing



An example of distributed indexing with MapReduce

Term-partitioned index

Rozproszenie względem termów

- ☹ Trudniejsze w ekspolatacji i silnie obciążające dla maszyny koordynującej
- ☹ Obsługa wielowyrazowych zapytań wymaga przesyłu długich list postings pomiędzy węzłami obliczeniowymi – koszt może przewyższyć zyski ze współbieżności
- ☹ *Load-balancing* jest uzależniony od rozkladu współwystępowania termów w dokumentach (można optymalizować na etapie konstrukcji), ale także od współwystępowania termów w zapytaniach (trudniej optymalizować, ze względu na chwilowe nagłe zmiany lub nieprzewidziane długofalowe trendy)
- ☹ Awaria pojedynczej maszyny uniemożliwia poprawną obsługę zapytań, zawierających dane termy – natychmiast zauważalne dla użytkowników
- ☺ Pozwala jednak redukować dostępy do pamięci dyskowej i transfery, gdyż lista postings każdego termu może być przechowywana w sposób ciągły (kolejne bloki) na dysku pojedynczej maszyny
- ☺ Każda maszyna ma pełną informację odnośnie podzbioru termów, więc obsługa pojedynczego zapytania angażuje tylko kilka konkretnych maszyn

Document-partitioned index

Rozproszenie względem dokumentów

- ☺ Popularniejsze rozwiązanie – wykorzystywane przez wyszukiwarki
- ☺ Jest bardziej naturalne i prostsze w implementacji (patrz. budowa indeksu) oraz pielęgnacji (dodawanie/usuwanie dokumentów)
- ☺ Awaria czy niedostępność pojedynczej maszyny nie blokuje obsługi zapytań (w wynikach będzie brakowało jedynie jakiejś części dokumentów) i pozostaje w zasadzie niezauważalna dla użytkowników
- ☺ Pozwala na podział kolekcji względem jakości/popularności – dokumenty, które będą często wysoko w wynikach zapytań trafiają do szybszych/wydajniejszych indeksów. Mniej wydajne indeksy będą przeszukiwane jedynie, gdy tamte nie dadzą wystarczająco wyników
- ☹ Każde zapytanie jest wysyłane do wszystkich węzłów obliczeniowych, których odpowiedzi muszą być scalone do prezentacji użytkownikowi
- ☹ W rezultacie wymaganych jest więcej lokalnych operacji na dyskach
- ☹ Globalne statystyki – takie jak np. *idf* – trzeba obliczać procesami w tle i odświeżać okresowo, gdyż poszczególne maszyny ‘widzą’ tylko swój fragment kolekcji

Distributed indexing

Rozproszenie odnosi się do podziału kolekcji dokumentów i ich indeksów na wiele maszyn – obsłużenie zapytania wymaga dokonania agregacji odpowiedzi z wielu indeksów

Replikacja (*replication*, *mirroring*) oznacza przechowywanie tylu identycznych kopii systemu, żeby lokalna obsługa zapytań była efektywnie realizowana

Distributed indexing

Implementacja Google'a jest oparta na indeksie rozproszonym względem dokumentów wykorzystującym masywną replikację i redundancję na każdym poziomie: maszyny, klastra obliczeniowego, site'u

Google claim: „*we keep each piece of data stored on at least two servers*" (dotyczy to nie tylko wyszukiwarki, ale gmaila, usług chmurowych, itd.) ... „*we store another copy of the most important data on digital tape*"

Google container data center tour

www.youtube.com/watch?v=zRwPSFpLX8I (at 3:33)

www.youtube.com/watch?v=avP5d16wEp0

Explore a Google data center with Street View:

www.google.com/about/datacenters/inside/streetview/

Index compression

- Efektywne przeszukiwanie skompresowanych tekstów i indeksów ma kluczowe znaczenie przy obecnych rozmiarach kolekcji
- W ogólności kompresja nie musi pogarszać czasów wyszukiwań – czas potrzebny na dekompresję kompensowany jest rzadszymi odwołaniami do pamięci dyskowej
- Kompresja tablic sufiksów jest trudniejsza niż indeksów odwrotnych, gdyż zawierają one niemal losowe permutacje wystąpień, ale istnieją metody budowy *SA* dla skompresowanych tekstów, które zachowują oryginalne związki leksykograficzne

Jak skompresować listę postings?

- Obserwacje
  - Lista postings jest posortowana (po *docID*, ew. *termFreq*)
  - Mniejsze liczby będą występować częściej (*freq*, *pos*) - dlaczego?
  - Dla *docID* można użyć reprezentacji „*d-gap*” (zapisywanie różnic)
    - zamiast: ID1, ID2, ID3, … (np. 1000, 1001, 1100, 1102, 1103)
    - na liście postings: ID1, ID2-ID1, ID3-ID2, … (1000, 1, 99, 2, 1)
- Sposoby kodowania liczb całkowitych:
  - Binary coding, unary coding, γ-coding, δ-coding
  - Wykorzystajmy fakt częstszego występowania małych liczb. Ich zapis zrealizujmy na mniejszej liczbie bitów, kosztem większej liczby bitów dla dużych liczb

Integer encoding

Binary coding:

- zapis każdej liczby ma stałą liczbę bitów
- zaleta: „standardowe” kodowanie – proste odwzorowanie w procesorze
- wady: należy przyjąć jakiś rozmiar danych – większe liczby się nie zmieszczą, małe liczby będą „marnowały” bity

Unary coding:

- Liczba $x \geq 1$ jest kodowana jako x bitów 0 oraz bit „stopu” – 1
- Przykład 3 => 0001; 5 => 000001; 11 => 000000000001
- Zaleta: małe liczby – krótki zapis
- Wada: długość zapisu rośnie szybko z wielkością liczby
- Złożoność pamięciowa – liniowa (dokładnie $x+1$ bitów)

γ-coding (Peter Elias, 1975):

- Liczbę $\boldsymbol{x} \geq 1$ przedstawiamy jako $2^N + k$, gdzie $N = \lfloor \log x \rfloor$
- Zapisujemy $\boldsymbol{N}$ metodą unary coding, następnie $\boldsymbol{k}$ metodą binary coding na $\boldsymbol{N}$ bitach
- Przykład: $2 = 2^1 + 0 \rightarrow 010$; $3 = 2^1 + 1 \rightarrow 011$; $5 \rightarrow 00101$
- Złożoność pamięciowa – logarytmiczna; do zapisu $\boldsymbol{x}$ potrzeba $\lfloor \log x \rfloor + 1 + \lfloor \log x \rfloor$ bitów

δ-coding (Peter Elias, 1975):

- Liczbę $\boldsymbol{x} \geq 1$ przedstawiamy jako $2^N + k$, gdzie $N = \lfloor \log x \rfloor$
- Zapisujemy wykładnik (konkretnie $\boldsymbol{N+1}$) jako γ-code, następnie $\boldsymbol{k}$ jako binary code na $\boldsymbol{N}$ bitach
- Przykład: $3 = 2^1 + 1 \rightarrow 0101$; $5 = 2^2 + 1 \rightarrow 01101$
- Złożoność pamięciowa – logarytmiczna; do zapisu $\boldsymbol{x}$ potrzeba: $2\lfloor \log (\lfloor \log x \rfloor + 1) \rfloor + 1 + \lfloor \log x \rfloor$ bitów

Integer encoding

Algorytm dekodowania zapisu *gamma encoding*:

1. Read and count 0s from the stream until you reach the first 1. Call this count of zeroes N
2. Considering the 1 that was reached to be the first digit of the integer, with a value of 2^N, read the remaining N digits of the integer

Przykład: 0001010 $\rightarrow$ 000 1 010 $\rightarrow$ N=3 $\rightarrow$ $2^3 + 2$

Wynik: 10

Integer encoding

Algorytm dekodowania zapisu *delta encoding*:

1. Read and count 0s from the stream until you reach the first 1. Call this count of 0s L
2. Considering the 1 that was reached to be the first digit of an integer, with a value of 2^L, read the remaining L digits of the integer. Call this integer N+1, and subtract 1 to get N
3. Put a 1 in the first place of our final output, representing the value 2^N
4. Read and append the following N digits

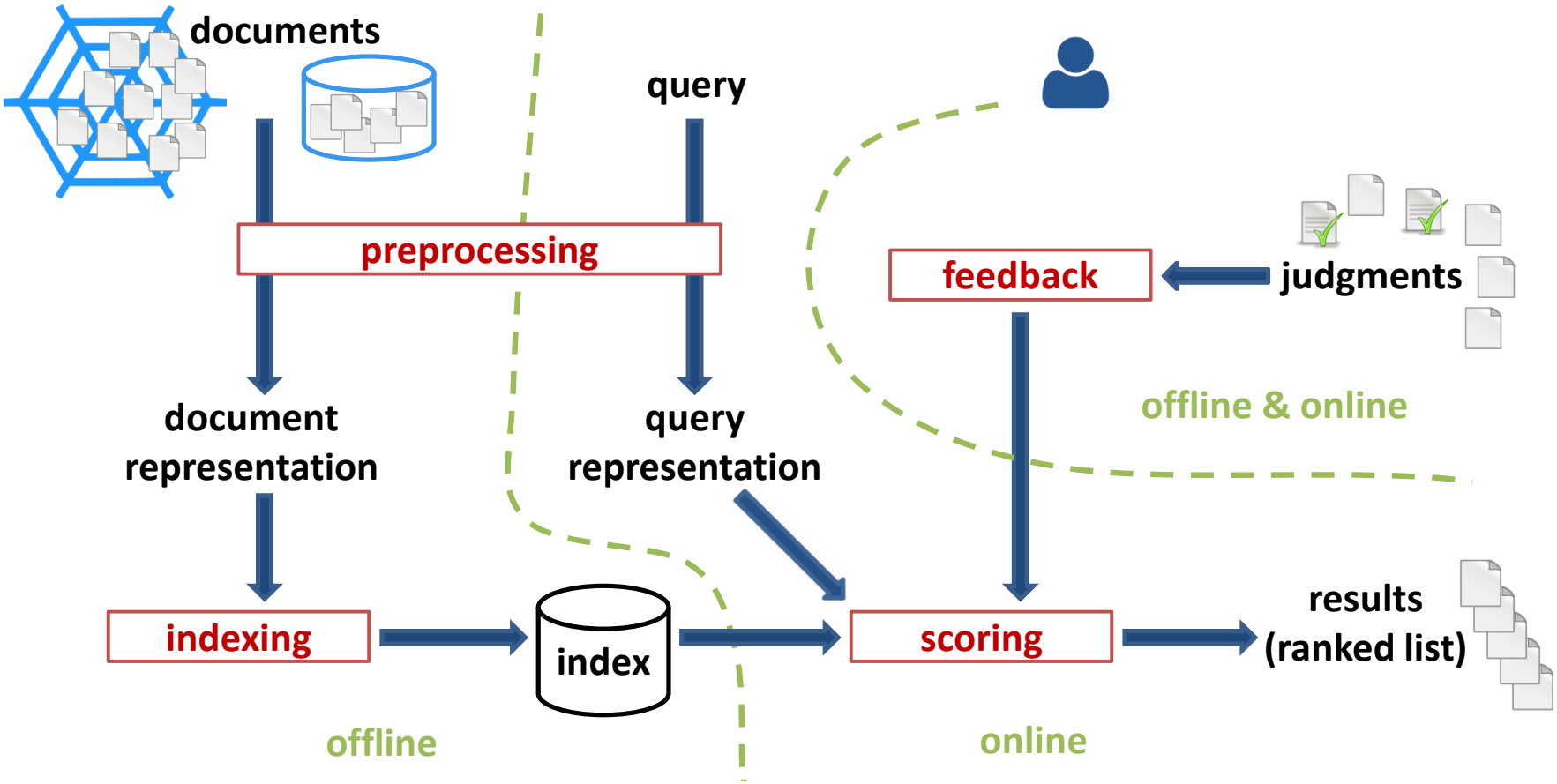
Przykład: 00100010 $\rightarrow$ 00 1 00 *010* $\rightarrow$ L=2 $\rightarrow$ N+1=2^2+0=4 $\rightarrow$ N=3 $\rightarrow$ $2^3 + 2$

Wynik: 10

Integer encoding

Number	N	N+1	Binary	γ Encoding	# bits	δ Encoding	# bits
$1 = 2^0 + 0$	0	1	00000001	1	1	1	1
$2 = 2^1 + 0$	1	2	0000001 0	0 1 0	3	0 1 0 *0*	4
$3 = 2^1 + 1$	1	2	0000001 1	0 1 1	3	0 1 0 *1*	4
$4 = 2^2 + 0$	2	3	000001 00	00 1 00	5	0 1 1 *00*	5
$5 = 2^2 + 1$	2	3	000001 01	00 1 01	5	0 1 1 *01*	5
$6 = 2^2 + 2$	2	3	000001 10	00 1 10	5	0 1 1 *10*	5
$7 = 2^2 + 3$	2	3	000001 11	00 1 11	5	0 1 1 *11*	5
$8 = 2^3 + 0$	3	4	00001 000	000 1 000	7	00 1 00 *000*	8
$9 = 2^3 + 1$	3	4	00001 001	000 1 001	7	00 1 00 *001*	8
$10 = 2^3 + 2$	3	4	00001 010	000 1 010	7	00 1 00 *010*	8
$11 = 2^3 + 3$	3	4	00001 011	000 1 011	7	00 1 00 *011*	8
$12 = 2^3 + 4$	3	4	00001 100	000 1 100	7	00 1 00 *100*	8
$13 = 2^3 + 5$	3	4	00001 101	000 1 101	7	00 1 00 *101*	8
$14 = 2^3 + 6$	3	4	00001 110	000 1 110	7	00 1 00 *110*	8
$15 = 2^3 + 7$	3	4	00001 111	000 1 111	7	00 1 00 *111*	8
$16 = 2^4 + 0$	4	5	0001 0000	0000 1 0000	9	00 1 01 *0000*	9
$17 = 2^4 + 1$	4	5	0001 0001	0000 1 0001	9	00 1 01 *0001*	9
$67 = 2^6 + 3$	6	7	0100 0011	000000 1 000011	13	00 1 11 *000011*	11
$256 = 2^8 + 0$	8	9	xxxx xxxx	00000000 1 00000000	17	000 1 001 *00000000*	15

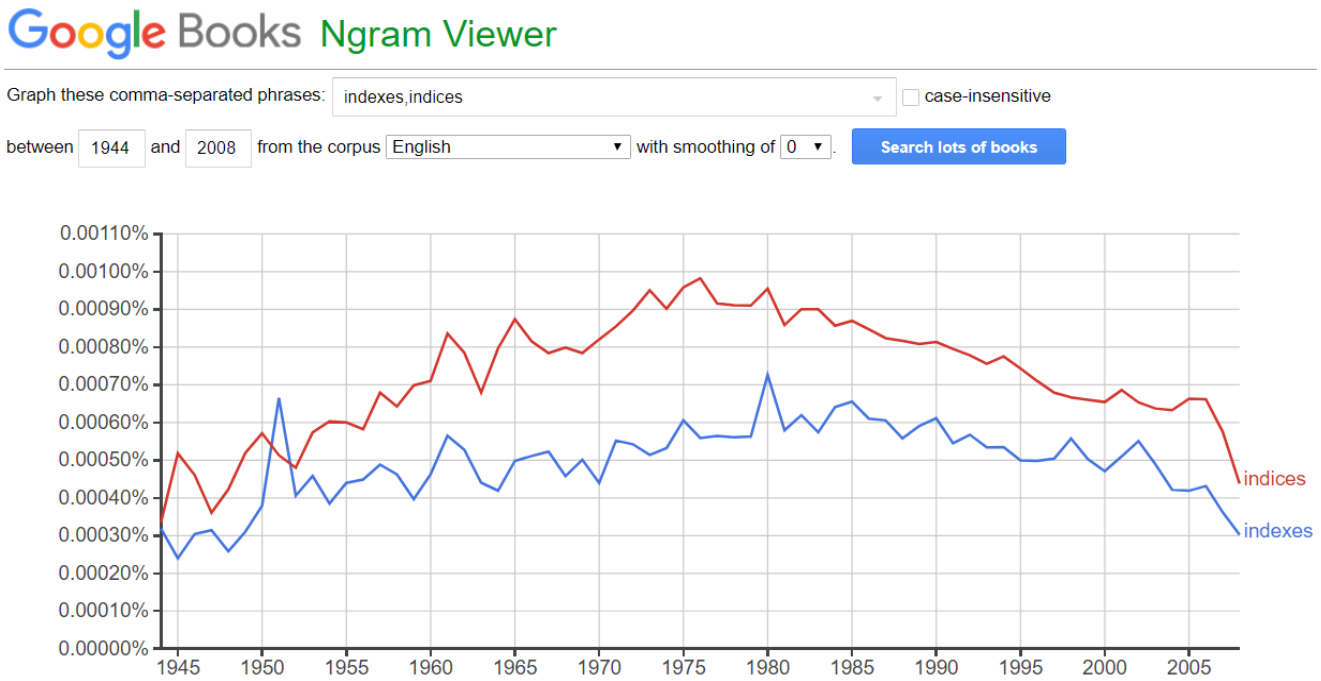
Summary – IR system architecture



Na podstawie *"Text Retrieval Methods"* – ChengXiang Zhai

Links to check out

- *http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf*
  (artykuł S. Brina i L. Page'a z 1998 r. – m.in. szczegóły nt. oryginalnego indeksu Google'a)
- *https://morsecode.scphillips.com/translator.html*
  (tłumacz tekstów na alfabet Morse'a – zapis graficzny, dźwiękowy, świetlny)
- *https://books.google.com/ngrams/info*



Exercises

I. Dla zapytania *q*: (0, 1, 0, 1, 1) użytkownik ocenił 2 dokumenty *d1*: (1, 4, 0, 4, 0) i *d3*: (1, 2, 0, 4, 0) jako adekwatne, a *d2*: (0, 1, 0, 1, 5) jako nieadekwatny. Wyznacz postać zapytania q_m zmodyfikowanego przy użyciu metody *Rocchio relevance feedback* dla podanych wag: α=1, β=0.5, γ=0.1

II. Ile liści będzie miało drzewo sufiksów dla ciągu znaków „*kuskus*”? Narysuj jego nieskompresowaną postać.

III. Wyznacz tablicę sufiksów dla powyższego ciągu.

IV. a) Zapisz liczbę 15 w kodowaniu δ, b) podaj zapis dziesiętny liczby 00101 zapisanej za pomocą kodowania γ, c) jakie liczby zapisano z wykorzystaniem kodowania γ w następującym ciągu: 001110100001110