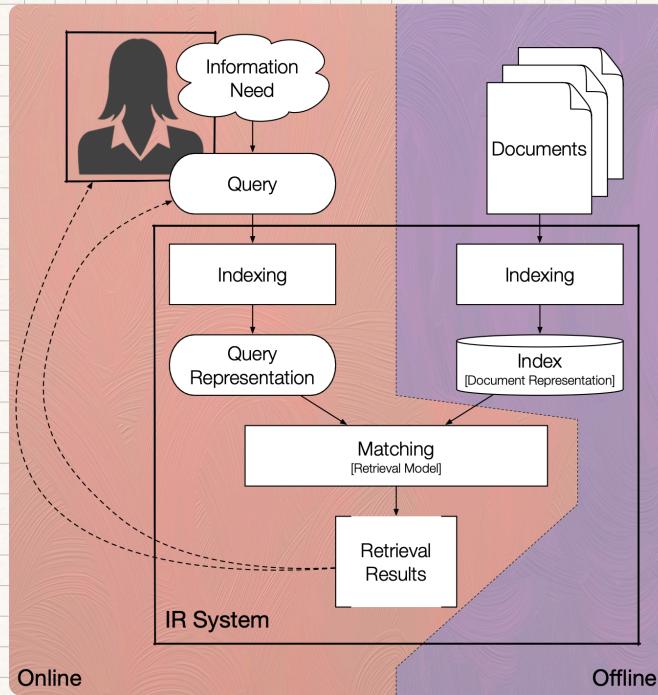


IL DIAGRAMMA Y



PARTE ONLINE

Ovvero quella parte che viene applicata da quando l'utente fa la query a quando riceve i risultati. Dopo aver scritto la query:

- Indexing Online

Trasforma la query in qualcosa di migliore per essere usato per ricavare i documenti necessari. Tipicamente si divide la query in token per poi usare i soli token più rappresentativi, ovvero la **co**representazione della query.

PARTE OFFLINE

Modellare i dati per poter ricavare informazione in modo più performante

- Indexing

È simile all'indice dei libri, dove cerchiamo di usare valori testuali che fanno riferimento univocamente al documento (ID, Nome, Titolo).

DALL'INFORMAZIONE NECESSARIA ALLA QUERY

Un utente fa una ricerca se gli manca un'info che gli serve. Ogni utente crea le query in modo soggettivo e più utenti creano query diverse per la stessa necessità. Perciò non esiste una definizione rigorosa e matematica di Document Relevant.

TEXT PROCESSING

ENCODING

1) ASCII: 128 caratteri

2) UNICODE STANDARD: tutti (quasi) i caratteri di tutti i linguaggi del mondo. [Ø, 10FFFF].

Definiamo **code point** il numero collegato al carattere.

Questo code point inizia con "U+" seguito da 4/8 cifre esadecimale. **NON È UN SCHEMA ENCODING.**

3) UTF - 8 / 16 / 32

Il code point Øx0C36 rappresenta un carattere ma non rappresenta la sequenza Øx0C e Øx36. Per questo si usa la codifica UTF-8, dove i caratteri sono codificati ovunque fra 1 e 6 bytes. Quindi con un numero di byte variabili rispetto al carattere.

L'encoding ASCII è mappata in UTF-8 con i 7 bit meno significativi, e il bit più significativo è Ø.

Tutti i caratteri che in UNICODE hanno codifica superiore a "U+ØØFF" sono codificati come sequenza di due o più bytes. Il primo byte di una sequenza code sempre nel range ØxC0 e ØxFD e i bit più significativi settati a 1 indicano di quanti byte è composta la sequenza:

Øs: 110xxxxx dice che la sequenza ha 2 bytes.

110xxxxx 10xxxxxx

Range UTF-8

Number of Bytes	Number of bits in Code Point	Range
1	7	00000000 - 0000007F
2	11	00000080 - 000007FF
3	16	00000800 - 0000FFFF
4	21	00001000 - 001FFFFF
5	26	00200000 - 03FFFFFF
6	31	04000000 - FFFFFFFF

PASSI PER CONVERTIRE UN UNICODI IN UTF-8

- 1) Sceglie il giusto range della tabella così da copiare quanti byte servono
- 2) Partire dal bit meno significativo, e copiare i bit del code point da destra verso sinistra nel byte meno significativo
- 3) Arrivati a usare tutti gli 8 bit del byte corrente si va verso il nuovo byte continuando con la cifra del code point
- 4) Continuare finché non si esauriscono i bit del code point. Riempire di zeri per finire il byte.

Esempio:

Convertire U+0557

Consultando la tabella servono 2 byte, quindi dobbiamo riempire 110XXXXX 10XXXXXX.

Convertiamo in binario il numero esadecimale

0000 0101 1110 0111

Riempiamo $10xxxxxx$ con il code point:

$10 \text{ } \underline{100111}$

Continuiamo con $110xxxxx$

$110 \text{ } \underline{10111}$

Da cui ricaviamo il risultato: $11010111 \text{ } 10100111$

TOKENISATION

1) CORPUS

È una collezione di documenti che un computer è in grado di leggere e usare. In pratica qualsiasi cosa abbia del testo dentro (PDF, Pagine WEB...).

Ogni elemento di un corpus è un **documento**.

2) TOKEN

È un'istanza di una sequenza di caratteri in qualche documento particolare, che sono raggruppati in una unità semanticamente utile.

3) TYPE

È la classe che ogni token composta da una certa sequenza di caratteri.

4) VOCABULARY

È il set di tutti i tipi presenti in un corpus.

5) TERM

Un elemento del vocabolario

Esempio:

"They lay back on the San Francisco grass
and looked at the stars and their"

La frase è composta da

15 TOKEN

13 TERMS ("and" e "the" sono ripetuti e quindi vengono contati una sola volta)

HOPPS LAW

Dato N numero di token e $|V|$ la cardinalità del vocabolario, abbiamo che

$$|V| = \kappa N^\beta \text{ con } 0.67 < \beta < 0.75 \\ 30 \leq \kappa \leq 200$$

Quindi si può dire che $|V|$ cresca più velocemente della radice quadrata di N (\sqrt{N} perché $\beta \in [0.67; 0.75[$)

TOKENIZATION

È il compito di dividere i documenti in token (parole)

Approccio space-based

Possiamo dividere le parole, presupponendo che siano separate da spazi.

C'è però il problema che non tutte le lingue umane dividono le parole con gli spazi (Cinese, Giapponese, ...)

Problemi della tokenizzazione

- 1) A volte se togliamo la punteggiatura compromettiamo il token (occhiali, prezzi, date, email, URLs, ...)
- 2) Una parola non sta insieme da sola (we're, don't, ...)
- 3) Esistono espressioni e parole che hanno significato solo insieme (New York, rock'n'roll, ...)

BIT-PAIR ENCODING

Tutti gli algoritmi per fare la tokenizzazione sono divisi in due parti: **Token Reader** che prende un corpus grezzo e crea il vocabolario, e **Token segmenter** che prende una frase grezza e la divide in token usando il vocabolario creato prima.

ALGORITMO BPS:

Preso come vocabolario l'insieme di tutti i caratteri individuali:

$$\{ A, B, C, D, \dots, a, b, c, d, \dots \}$$

loop ($i \leq k$) // k iterazioni.

{

1) Scegli la coppia di caratteri con più alto supporto nel corpus

2) Aggiungi il nuovo simbolo dato dall'unione dei due simboli più presenti, nel vocabolario

3) Rimpiazza ognuno dei simboli adiacenti A, B trovati in 1) con AB.

K++

}

La maggior parte degli algoritmi presupone che le parole siano separate da spazi nel corpus, perciò si aggiunge un simbolo di fine stringa a ogni parola, solitamente "-".

Esempio:

- Original corpus

low low low low lowest lowest newer newer
newer newer newer wider wider wider new new

- End-of-word tokens

low_ low_ low_ low_ lowest_ lowest_ newer_ newer_
newer_ newer_ newer_ wider_ wider_ wider_ new_ new_

- Starting vocabulary

_, d, e, i, l, n, o, r, s, t, w

Corpus	Vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

→ Numeri di token "low"
nel corpus

- Merger er to er : totale di "er"

Corpus	Vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	Fase di Merge
6 n e w e r _	
3 w i d e r _	
2 n e w _	

Corpus

5 l o w _
2 l o w e s t _
6 n e w er_
3 w i d er_
2 n e w _

Vocabulary

_, d, e, i, l, n, o, r, s, t, w, er

- Merger **er** to **er_**: Totale ↗ "er_"

Corpus

5 l o w _
2 l o w e s t _
6 n e w er_
3 w i d er_
2 n e w _

Vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er

Corpus

5 l o w _
2 l o w e s t _
6 n e w er_
3 w i d er_
2 n e w _

Vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er

- Merger **n e** to **ne** ↗ 8

Corpus

5 l o w _
2 l o w e s t _
6 ne w er_
3 w i d er_
2 ne w _

Vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er, ne

- The next merges are

Merge	Vocabulary
(ne, w)	_, ..., w, er, er_, ne, new
(l, o)	_, ..., w, er, er_, ne, new, lo
(lo, w)	_, ..., w, er, er_, ne, new, lo, low
(new, er_)	_, ..., w, er, er_, ne, new, lo, low, newer_
(low, _)	_, ..., w, er, er_, ne, new, lo, low, newer_, low_

NORMALIZZAZIONI

Quello il compito di trasformare i token in un formato standard (U.S.A. vs USA, Gabriele vs gabriele, ...)

- Case folding**: ridurre tutto in minuscole
- Lemmatization**: prendere solo il lemma di una parola
δs: Streaming \Rightarrow Stream
- Morfema**: l'unità più piccola ma significativa di una parola:
 - STEMS
 - AFFIXES

POSTER STEMMER ALGORITHM

Algoritmo per fare stemming in inglese. È basato su 5 regole:

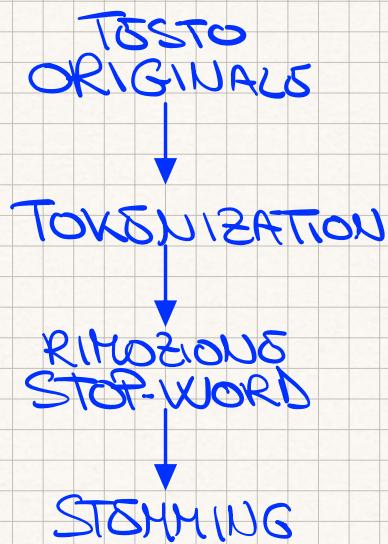
- SSES \rightarrow SS (caresses \rightarrow caress)
- IES \rightarrow I (ponies \rightarrow pony)
- SS \rightarrow SS (caress \rightarrow caress)
- S \rightarrow (cats \rightarrow cat)

Questo processo è **irreversibile**.

STOP WORDS

Sono tutte quelle parole estremamente comuni che hanno piccoli compiti non indispensabili (Articoli, preposizioni, congiuntioni, ...)

Non bisogna usarle per indicizzare.



Note: Non si può cambiare ordine di questo processo.
poiché se faccio stemming prima di rimuovere le
stop words potrei eliminare parole importanti. (Andes)

INDICES

Struttura dati utile a velocizzare la ricerca nei documenti di termini chiave.

Prima di copiare come sono fatti oggi gli indici dei motori di ricerca, vediamo come costruire una struttura dati semplice ma difficile da gestire:

MATRICE DI INCISURA TERM-DOCUMENT

Token \ Documenti	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Token						
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

È una matrice dove le righe indicano i termini del vocabolario del corpus, mentre le colonne fanno riferimento ai documenti. Il valore (i, j) può essere 0 o 1 o \emptyset e indica se il termine i è presente nel documento j . È una matrice estremamente sparsa.

Consideriamo 10^6 documenti, ognuno con 10^3 parole.

Se ogni parola in media occupa 6 byte, allora ogni doc è grande 6 GB.

Considerando che i termini unici per ogni doc siamo
per metà: $M = 0,5 \cdot 10^6$

In queste circostanze la matrice di incidenza sarà
grande:

$$M \cdot N = 0,5 \cdot 10^6 \cdot 10^6 = \frac{10^{12}}{2} \text{ N } 500 \text{ GB.}$$

Innanzitutto è estremamente **sparso**.

UTILITÀ INDICI:

- 1) DURATA LA RICERCA UNDAR
- 2) VELOCITÀ
- 3) SUPPORTO PER CONTINUI AGGIORNAMENTI

Come valutare un indice?

- 1) Indexing Time
- 2) Indexing Space (Spazio usato in fase di creazione)
- 3) Index Storage
- 4) Query Latency
- 5) Query Throughput

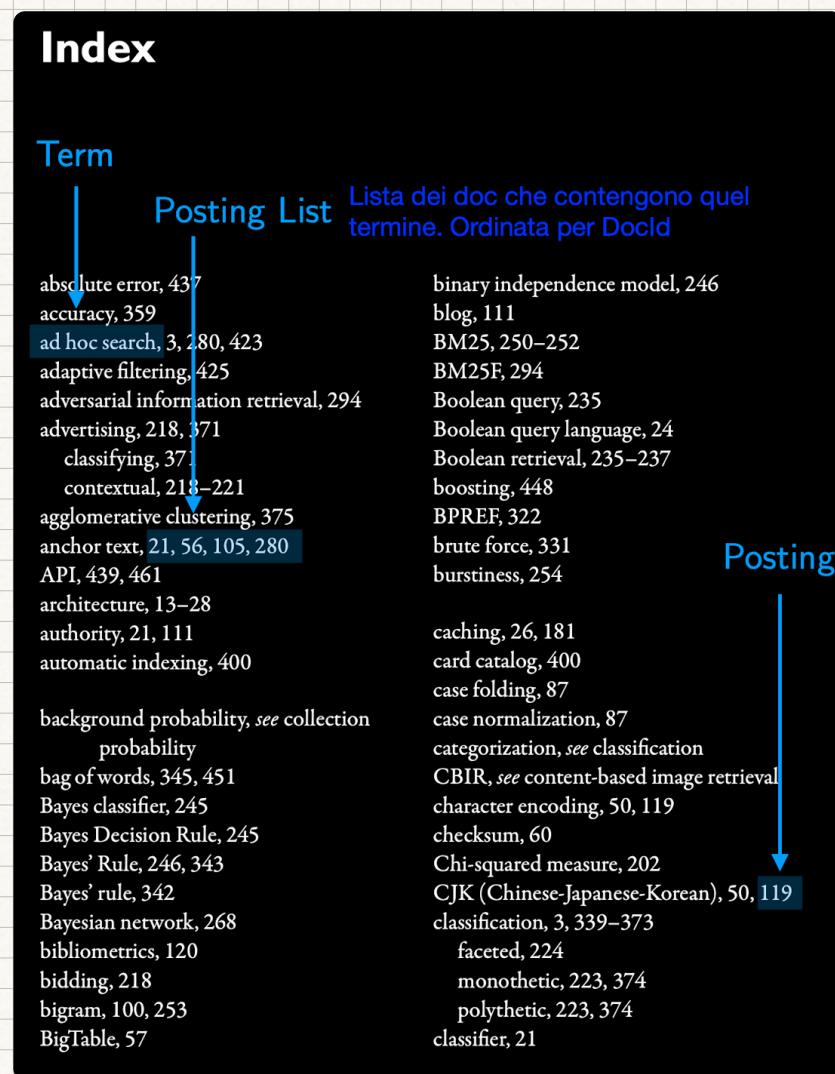
In generale per creare un indice bisogna fare un trade-off fra la sua **grandezza** e la sua **velocità**.
Più l'indice è grande più è veloce. Più è compresso,
più ci vuole del tempo per decomprimere prima di
usarlo.

INVERTED INDEX

È la struttura dati che, nei sistemi moderni, fa le veci della matrice di indicizzazione. È detto "inverted" perché sono i documenti a essere associati ai termini e non il contrario.

Componenti:

- DocID
- Posting List : Ordinata per docid.
- Posting



POSTING LIST

Di base è composta da una lista di docid ordinata ma ci possiamo aggiungere ulteriori informazioni:

1) **Term Frequencies**: il supporto, anche se chiamarlo frequenza è matematicamente scorretto

2) **Position**: posizione del termine nel documento.

STRUTTURE DATI DI SUPPORTO

1) Lexicon o Vocabulary

Contiene statistiche associate ai veri termini, come la frequenza o numero di doc in cui il termine appare.

Contiene uno **Lookup Tab** ^(tab di ricerca) che, portando dai termini dell'indice restituisce l'offset di una lista che di fatto contiene tutte le posting list dell'indice invertito. È un modo veloce per ritrovare la posting list di un dato termine.

2) Documenti Index

Struttura dati che contiene info sul singolo doc, come l'URL, il titolo, la **Lunghezza**, ecc ...

3) Collection Statistics

Salvata in un file diverso per le statistiche generali della collezione (**numero di doc, numero di termini, numero di posting**) ...)

INDEXING

INDICIZZARE LE FRASI

È utile a volte indicizzare più parole insieme (New York City). Per farlo dobbiamo indicizzare tutti gli **N-gram** di una collezione:

per: "A B C D" indicizziamo tutti i 2-grami:

<AB AC AD BC BD CD>

Poiché è evidente che indicizzare tutti gli **N-gram** porta ad avere molte più posting list ma meno popolate

ALGORITMO IN-MEMORY INDEX

1 BUILD-INDEX (D)

```
2   I ← HashTable() // Lista delle posting list
3   docId = 0
4   for (forall document d ∈ D)
5     docId ++
6     tokens ← Parse(d)
7     Rimuovere i duplicati da tokens.
8     for (t in tokens)
9       if (It ∉ I) // It = posting list di t
10        It ← crea Array()
11        It.append(docId)
12    }
13  }
14 return I
```

Questo algoritmo però è totalmente inefficiente per D molto grande, anche perché l'accesso alla lista delle posting list va sincronizzato per un uso in parallelo. Ci serve qualcosa che scali!

Per renderlo scalabile, mentre creiamo l'indice andiamo un documento alla volta, così da avere le posting list incomplete fino alla fine. Considerando 8 bytes per la coppia (termId, docId) è evidente come, per grandi collezioni, non può funzionare. Non starà mai in memoria centrale.

Come facciamo?

Possiamo pensare di salvare i risultati intermedi nel disco secondario. Usare il disco per fare il merge di posting list è un incubo.

BLOCKED SORT-BASED INDEXING

- 1) Record: coppia (TermId, docId)
- 2) Block: 10^7 records. $\approx 80\text{Mb}$.

L'idea dell'algoritmo è quella di **accumulare posting** per ogni block, ordinarle e poi scriverle sul disco. Poi fare il **merge dei blocchi**.

```
BSBIndexConstruction ()  
{  
    docId =  $\emptyset$   
    while (ci sono doc da processare)  
    {  
        docId ++
```

//Accumula record da vari doc finché riempio un
block ← ParseNextBlock() //block

// Ordino i record per poi unire quei record
// con stesso termID, creando la posting List
BSBI-Index(Block)

// Scrivo sul disco il risultato

WriteBlockToDisk(Block, fload)

Merge Blocks ($f_1 \dots f_{\text{last}}; f_{\text{merged}}$)

Esempio: Merge di due indici:

Index A	aardvark	2	3	4	5	apple	2	4
---------	----------	---	---	---	---	-------	---	---

Index B aardvark 6 9 actor 15 42 68

Index A	aardvark	2	3	4	5		apple	2	4
---------	----------	---	---	---	---	--	-------	---	---

Index B	aardvark	6	9	actor	15	42	68
---------	----------	---	---	-------	----	----	----

Combined index | aardvark | 2 | 3 | 4 | 5 | 6 | 9 | actor | 15 | 42 | 68 | apple | 2 | 4

È più efficiente fare un multi-way merge, leggendo tutti i blocchi contemporaneamente. Questo si fa tenendo due buffer: ~~read~~ e ~~write~~ per leggere i blocchi e scrivere il risultato.

Per ogni iterazione, prendere il termine con minor
tokensId non ancora processato e fare il merge di tutte
le posting list di quel termine.

SINGLES-PASS IN-MEMORY INDEXING

L'algoritmo sopra descritto presuppone di riuscire a far entrare il vocabolario in memoria. Ma non è quasi mai possibile. Abbiamo bisogno del vocabulary per mappare termini con termId.

- 1) Creare un vocabulary per ogni block.
- 2) Non fare il sort, ma semplicemente accumulare le varie posting nelle posting list quando occorre.

SPIMI - Indexing (token-stream)

```
1 output-file = NewFile()
2 dictionary = NewHash()
3 while (C'è spazio in memoria)
4 {
5     token ← next(token-stream)
6     if (term(token) &lt; dictionary) Non è nel dizionario
7         posting-list = AddToDictionary(dictionary, term(token))
8     else
9         posting-list = GetPostingList(dictionary, term(token))
10        if (posting-list è piena)
11            posting-list = DoublePostingList(dictionary, term(token))
12            AddToPostingList(posting-list, docId(token))
13    }
14    sort_terms ← SortTerms(dictionary)
15    WriteBlockToDisk(sort_terms, dictionary, output-file)
16 }
```

Quando facciamo il merge delle posting-list di due blocchi, può capitare che lo stesso termine abbia due termId diversi. Basta sceglierne uno dei due e usarlo come termId della posting list globale. Finché c'è spazio **SPILL** può indicizzare.

POSTING

Dentro una singola posting possiamo trovare:

- 1) Solo il docId
- 2) docId e frequenza nel doc del termine
- 3) docId e impact score
- 4) docId, frequenza nel doc del termine e lista posizioni

STRUTTURE DATI NECESSARIE

- 1) Inverted Index
- 2) Lexicon con info sui ogni termine (Dove inizia la posting list, quanti doc lo contengono, ...)
- 3) Document Index contenente per ogni doc, URL, Bright, pageRank, ...

Alla fine della fiera tutte queste strutture sono file salvati nel disco.

La struttura più grande è l'inverted index (lista di liste)

Lexicon Layout

- Contains **one element for each distinct term**
- Dictionary, hash table, succinct data structure or **disk-based data structure**
Complessità di accesso di una hash Tab. O(1)
- **Lookup based on term** (the term is key) At ~~ogni~~ time
- Stores **start of** corresponding **posting list** in index
 - A **file offset** for a disk-based index or a pointer
- Also stores **length of list**, maybe other items
 - Can get large in some cases

Document Index Layout

- Contains **one element for each indexed document**
- Keeps the **mapping between documents and docids**
- Given docid, we need to be able to **look up URL**
- Maybe store **document size** and **pagerank**
- Simplest approach: **records ordered by docid**
- More **space-efficient**:
 - Store URLs in alphabetic order, maybe compressed
 - Replace URL in above table by pointer or offset
 - Also allows lookup of docID by URL

Inverted Index Layout

- The inverted index is usually stored **on disk**, and **in memory**
- Usually **multiple inverted indexes**, with a constant number of documents per index
- In **compressed form**, even if in memory (**never ever** fully decompressed)
- De-facto posting format is **(docid, frequency)**
- Do not store docids and frequencies in **interleaved form**
 - d,d,d,d,d f,f,f,f,f YES Per via della compressione.
 - d,f,d,f,d,f,d,f,d,f NO

QUERY PROCESSING

Esistono due modi per processare le query:

1) CONJUNCTIVE

Ovvio ritrovando i documenti che contengono TUTTI i termini di cui è composta la query

2) DISJUNCTIVE

Ovvio ritrovando i documenti che contengono almeno uno dei termini di cui è composta la query

BOOLEAN RETRIEVAL

Il risultato sono TUTTI i documenti che soddisfano la query. Questo approccio è adatto a piccole o medie collezioni di documenti. È spesso usato in contesti specifici, dove le query sono tutte relative al medesimo argomento, e chi fa le query è uno specialista nel campo (Studi medici, Studi legali, ...)

In altre parole le collezioni di documenti sono specializzate nell'argomento in questione.

Le query sono fatte da specialisti, e possono essere lunghe anche pagine e pagine.

RANKED RETRIVAL

Qui l'utente non è specializzato e fa una query che è spesso approssimativa (Utenti che cercano vari motivi di ricerca). Ottimo per grandi collezioni di documenti, dove per scegliere quali documenti ritrovare (Non si possono ovviamente ritrovare tutti) si crea un sistema di rank, che usa la seguente funzione di score:

$$S_q(q, d) = \sum_{t \in q} S_t(q, d)$$

Ovvero la somma degli score di ogni termine della query. È una somma, quindi può solo crescere.

Si ritrovano i top K documenti, in termini di score function.

Vedremo meglio come costruire la score function, ma spesso si basa sul concetto di Lexical matching, ovvero vedere i documenti che contengono i termini della query. Questa funzione di scoring è definita:

$$s: Q \times D \rightarrow \mathbb{R}$$

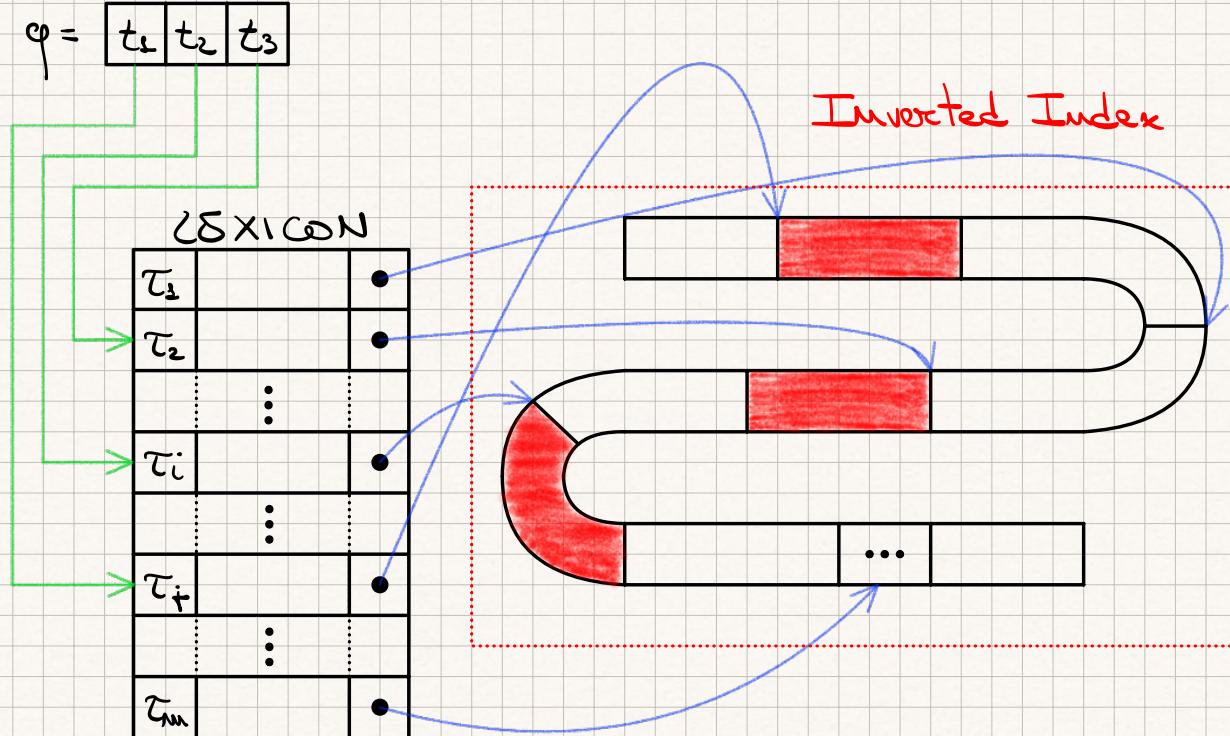
Ovvero assegna un numero reale ad ogni coppia termine-documento.

TERM AT A TIME

Attraversiamo la posting list termine della query per termine della query.

Quindi partiamo dai termini della query:

$$q = [t_1 \ t_2 \ t_3]$$



Analizziamo la posting list di ogni termine (quelle in rosso nell'esempio). Per esempio assumiamo che la posting list di t_1 sia

$$t_1 \rightarrow [4 \ 11 \ 7 \ 1 \ 10 \ 5]$$

Dove viene rappresentato il docID e la frequenza del termine nel documento. Usiamo la frequenza come score function. Dopo aver analizzato la prima posting list avremo la seguente classifica:

RANK	DOCID	SCORE
1	4	11
2	10	5
3	7	1
4	/	/

Supponiamo ora che il secondo termine abbia la seguente Posting list

$$t_2 \rightarrow \boxed{7} \boxed{13} \boxed{9} \boxed{3}$$

Dopo averla analizzata avremo la classifica:

RANK	DOCID	SCORE
1	7	14
2	4	11
3	10	5
4	9	3

Con lo score del documento 7 che è la somma degli score dei termini della query presenti nel doc 7 presi singolarmente.

PRO TART

- 1) Facile
- 2) Cache-friendly: per via del principio di località spaziale della cache, dato che scores ha Posting list in sequenza.

CONTRO TAAT

1) Richiede troppo memoria per contenere gli score parziali di ogni documento.

2) Molto complicato gestire query booleane o frasali perché analizzando un termine per volta è difficile trovare tutti i documenti che hanno al loro interno 2 termini contemporaneamente.

```
procedure TERMATATIMERETRIEVAL( $Q, I, f, g, k$ )
     $A \leftarrow \text{HashTable}()$ 
     $L \leftarrow \text{Array}()$ 
     $R \leftarrow \text{PriorityQueue}(k)$ 
    for all terms  $w_i$  in  $Q$  do
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
         $L.add(l_i)$ 
    end for
    for all lists  $l_i \in L$  do
        while  $l_i$  is not finished do
             $d \leftarrow l_i.\text{getCurrentDocument}()$ 
             $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
             $l_i.\text{moveToNextDocument}()$ 
        end while
    end for
    for all accumulators  $A_d$  in  $A$  do
         $s_d \leftarrow A_d$ 
         $R.add(s_d, d)$ 
    end for
    return the top  $k$  results from  $R$ 
end procedure
```

Recupera la posting list dei termini della query

Per ogni termine scatta la PL per calcolare lo score dei documenti

Ordina i documenti per score decrescente

Dizionario

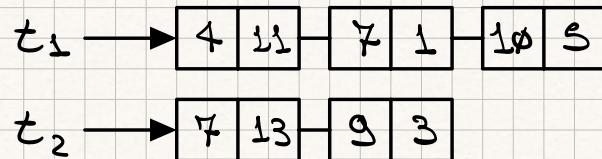
Inverted I.

Score f

TAAT non è più in uso dal 1995.

DOCUMENT AT A TIME

Venne usata in ranked disjunctive retrieval. Qui le varie posting list vengono processate in parallelo così da esaminare un intero documento per volta. Assumiamo di avere le seguenti posting list:



Nel primo passaggio analizziamo il doc 4 presente solo nella PL di t_1 .

RANK	DOCID	SCORE
1	4	11
2	-	-
3	-	-
4	-	-

Nel secondo passaggio esaminiamo il doc 7 presente sia nella PL di t_1 che in quella di t_2 . Vanno analizzate in parallelo, e lo score sarà la somma delle frequenze (1+13):

RANK	DOCID	SCORE
1	7	14
2	4	11
3	-	-
4	-	-

```

procedure DOCUMENTATATIMERETRIEVAL( $Q, I, f, g, k$ )
     $L \leftarrow \text{Array}()$ 
     $R \leftarrow \text{PriorityQueue}(k)$ 
    for all terms  $w_i$  in  $Q$  do
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
         $L.\text{add}( l_i )$ 
    end for
    for all documents  $d \in I$  do
         $s_d \leftarrow 0$ 
        for all inverted lists  $l_i$  in  $L$  do
            if  $l_i.\text{getCurrentDocument}() = d$  then
                 $s_d \leftarrow s_d + g_i(Q)f_i(l_i)$ 
            end if
             $l_i.\text{movePastDocument}( d )$ 
        end for
         $R.\text{add}( s_d, d )$ 
    end for
    return the top  $k$  results from  $R$ 
end procedure

```

Recoreso la
posting list
dei termini
della query

Per ogni doc
va a prendere
la score in
ogni PL del
doc corrente
e calcola la
score

PRO DAAT

- 1) Usa meno memoria di TART perché non bisogna salvare gli scores parziali. Calcola lo score finale del documento.
- 2) Supporta query booleane e frasali, perché cerca per documento e ha tutte le posting del doc.

CONTRO DAAT

- 1) Molto meno cache-friendly di TART, perché legge le varie PL non in sequenza.

POSTING LIST ITERATOR

È spesso conveniente vedere la posting list come un iterator sulle sue posting.

Fra le varie API abbiamo:

- 1) $p.\text{getDocId}()$
- 2) $p.\text{getScore}()$
- 3) $p.\text{next}()$: ritorna la posting successiva
- 4) $p.\text{nextGtQ}(d)$: ritorna la posting con $\text{docId} > d$.
Il parametro d è detto skipping

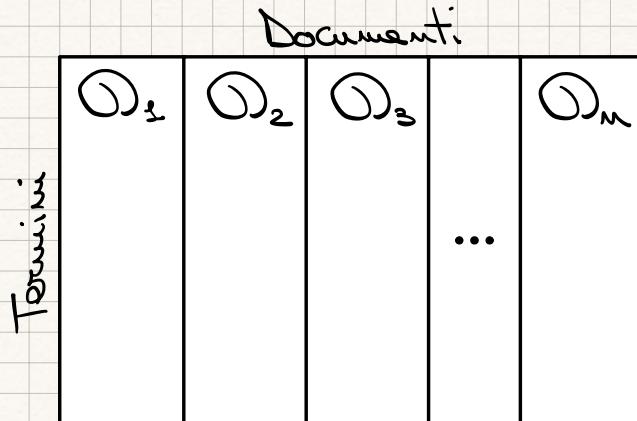
DISTRIBUTING QUERY PROCESSING

Tutto il processo, dai termini della query all' indice è distribuito in più macchine. Questo è forzato dal fatto che l'inverted index è troppo grande per una macchina sola. Abbiamo dei nodi nel pool di server usati:

- 1) **MANAGER MACHINE**: Riceve tutte le query poi invia i termini alle macchine giuste che contengono l'indice. Riceverà poi i risultati, li organizzerà e li ritorna all'utente.
- 2) **INDEX SERVER**: sono le macchine che contengono un pezzo di indice. Ognuna di queste macchine performa una porzione del query-process.

DOCUMENT DISTRIBUTION

È la tecnica più usata per partitionare l'indexer index fra più macchine, ovvero dividendo i documenti:



Ogni server contiene una piccola porzione dei documenti della collezione. Il manager invia una copia della query a ogni index server, che restituirà i suoi top k documenti. Bisogna poi fare il merge dei vari risultati dentro una singola **ranked list**, di questo si occupa il manager.

Problemi con questo approccio:

Se voglio il numero di documenti in cui un termine appare (Ricerca per riga della matrice) devo chiedere a tutti gli index-server e attendere il più lento.

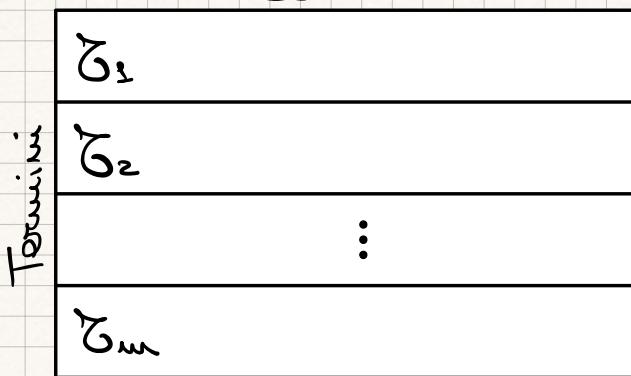
Penso provare a bilanciare il carico di dati che ogni server possiede ma c'è un problema NP-Hard.

Inoltre per fare il rank finale, ho bisogno di sapere le statistiche dell'intera collezione di docs.

TERM DISTRIBUTION

Dividiamo la lista per termini, e le diamo ai vari server

Documenti.



Quindi ogni index server avrà intere posting list salvate. In questo caso viene scelto un solo index server, solitamente il server che ha la posting list più grande, per processare la query. Gli altri server inviano info necessarie al server che sta processando. Il risultato viene inviato al manager.

Problemi con questo approccio:

Analizzare termini in server diversi è difficile.

QUERY LOG

Impronta delle varie query passate a capire quali termini accompagnano query simili e li salvo insieme. C'è uno studio detto, su quali termini salvare sulla stessa macchina.

CACHING

Si può usare un meccanismo di caching per query molto popolari nel web. Possiamo mettere in cache:

- 1) Query popolari : (Facebook, Maps, ...)
- 2) Inverted list comuni

Ovviamente la cache va aggiornata se si vuole prevenire il problema dei dati obsoleti.