UNIVERSIDAD
COMPLUTENSE
MADRID

Índices

Sistemas de Gestión de Datos y de la Información

Enrique Martín - emartinm@ucm.es

Máster en Ingeniería Informática

Fac. Informática

Índices

- Un **índice** es la **estructura de datos** principal que se utiliza para la recuperación de información.
- El objetivo de un índice es **almacenar** la **información importante** de los documentos para acceder de manera **rápida** a la hora de resolver consultas.

Índices

- ¿Cuál es la información importante de los documentos? Principalmente la aparición de **términos clave** en ellos. La información almacenada puede ser más o menos detallada:
  - Si el término clave aparece o no
  - El número de apariciones del término clave
  - Las posiciones en las que aparece

Índices

- El principal tipo de índices utilizado en recuperación de información se conoce como **índices invertidos**.
- La razón es que en la colección, la relación directa es que “los documentos contienen los términos clave”. En el índice invertido la relación es al revés, relacionamos “términos clave con los documentos que los contienen”

Índices: ejemplo

Vocabulario	#doc	d1	d2	d3
el	3	5	1	2
amigo	1	-	-	2
hola	3	4	3	1
verde	3	5	1	4
pez	1	-	2	-
coral	1	6	-	-
agua	3	3	3	2
mar	2	3	-	1
abuela	1	-	4	-

- Consideramos 3 documentos: d1, d2 y d3
- #doc almacena el número de documentos donde aparece
- Para cada documento, almacena el número de apariciones

Índices: ejemplo

Vocabulario			
el	(1,5)	(2,1)	(3,2)
amigo	(2,2)		
hola	(1,4)	(2,3)	(3,1)
verde	(1,5)	(2,1)	(3,4)
pez	(2,2)		
coral	(1,6)		
agua	(1,3)	(2,3)	(3,2)
mar	(1,3)	(3,1)	
abuela	(2,4)		

- Es un sistema real, hay miles o millones de documentos.
- Para la mayoría de los términos, la fila estará llena de ceros.
- Por ello lo más normal es almacenar las ocurrencias como **listas ordenadas**.

Índices invertidos completos

- Los índices invertidos vistos hasta ahora únicamente almacenan los **documentos** en los que aparece un término clave, y el **número de ocurrencias**.
- Si queremos poder responder a consultas sobre **frases** o que tengan en cuenta la **proximidad** de los términos clave, deberíamos extender la información que almacenan.

Índices invertidos completos

- Para cada término clave un índice invertido completo almacenará:
  - El número de documentos que lo contienen.
  - Una lista completa de apariciones, es decir, una lista de tuplas
    (#doc, n, [pos1, …, posn])
    donde:
    - *#doc* es un número de documento
    - *pos1,…,posn* son posiciones dentro del documento.

Índices invertidos completos

- La información almacenada como *posición* puede ser de más o menos detallada:
  - Carácter en el que empieza
    Obtención de extractos del documento, búsqueda de frase, búsqueda de cercanía
  - Número de palabra en el documento
    Búsqueda de frase, búsqueda de cercanía
  - Número de bloque/párrafo en el que aparece
    Búsqueda de cercanía

Índices invertidos completos: ejemplo

Vocabulario	***ni***	**Lista de apariciones**
el	3	[(1,5,[1,3,5,7,9]), (2,1,[3]), (3,2,[17,29])]
amigo	1	[(2,2,[4,86])]
hola	3	[(1,4,[22,24,56,71]), (2,3,[1,10,20]),(3,1,[1])
verde	3	[(1,5,[19,25,32,40,59]),(2,1,[5]),(3,4,[76,80,90,111])]
pez	1	[(2,2,[45,123])]
coral	1	[(1,6,[101,134,145,204,250,278])]
agua	3	[(1,3,[67,78,106]), (2,3,[56,78,99,301]), (3,2,[1,34])]
mar	2	[(1,3,[500,509,700]), (3,1,[72])]
abuela	1	[(2,4,[303,581,600,708])]

Índices invertidos vectoriales

- Un índice invertido completo contiene toda la información posible.
- A partir de esa información se podrían calcular sobre la marcha los **pesos** (w_{ij}) necesarios para calcular la similitud en consultas vectoriales.
- Para consultas vectoriales es mucho más eficiente calcular todos los pesos y almacenar esa información directamente en el índice:
  - En lugar de listas de posiciones → un número decimal representando el peso.

Índices invertidos vectoriales: ejemplo

Vocabulario	***ni***	**Pesos**
abrigo	3	[(1, 0.0), (2, 0.0), (3, 0.0)]
gol	1	[(1, 4.097)]
abrazo	3	[(1, 0.0), (2, 0.0), (3, 0.0)]
pie	2	[(1, 0.585), (2, 0.585)]
paella	1	[(1, 1.585)]

Ejemplo de índice invertido vectorial para una colección de N=3 documentos y un vocabulario V={abrigo, gol, abrazo, pie, paella}

Búsquedas con índices

Búsqueda con índices

- Hasta ahora hemos visto dos modelos de recuperación de información, que nos definen de manera **formal** qué significa que un documento encaje con una consulta.
- Sin embargo, no hemos abordado cómo calcular este encaje de manera **eficiente** en un sistema real.

Búsqueda con índices

- Una aproximación muy simple sería recorrer todos los documentos *di*, y para cada uno de ellos calcular su valor de encaje con la consulta *q*:

```
Buscar( c:Colección, q:Consulta):Colección
 res := Ø;
 para cada documento ‘d’ en ‘c’:
     si R(d,q) >= UMBRAL entonces
      res := res U {d};
 devolver res;
```

Búsqueda con índices

- Esta solución es **sencilla**, pero **muy poco eficiente** porque tiene que recorrer todos los documentos en cada consulta.
- En general, tenemos un número muy alto de documentos pero solo queremos recuperar un número bajo de resultados relevantes.
- Para conseguir eficiencia necesitaremos sacar provecho de los **índices**.

Modelo booleano

- Para realizar una consulta t1 AND t2 tendremos que:
  - Recuperar la lista de apariciones de t1y t2 del índice (p1 y p2 respectivamente).
  - Cruzar las listas y quedarse con los documentos comunes → INTERSECT(p1,p2).
- Todo esto hay que realizarlo de manera **eficiente**.

Modelo booleano

- Obtener la lista de apariciones de un término es rápido: solo debe consultarse el índice.
  - index[t1] → p1
  - index[t2] → p2
- Esta lista estará ordenada de manera **ascendente** por identificador de documento.
- Si **pi** es una lista de apariciones, **docID(pi)** nos devuelve el identificador de documento de su primera entrada.

Modelo booleano

```
INTERSECT(p1,p2):
  answer = []
  while p1 != [] and p2 != []:
    if docID(p1) == docID(p2):
      add(answer, docID(p1))
      p1 = next(p1)
      p2 = next(p2)
    elif docID(p1) < docID(p2):
      p1 = next(p1)
    else:
      p2 = next(p2)
  return answer
```

Ejemplo INTERSECT

index			
el	(1,5)	(2,1)	(3,2)
amigo	(2,2)		
hola	(1,4)	(2,3)	(3,1)
verde	(1,5)	(2,1)	(3,4)
pez	(2,2)		
coral	(1,6)		
agua	(1,3)	(2,3)	(3,2)
mar	(1,3)	(3,1)	
abuela	(2,4)		

Consulta “amigo AND verde”

- p1 = index[“amigo”] = [(2,2)]
- p1 = index[“verde”] = [(1,5), (2,1), (3,4)]

Invocaremos a:

```
INTERSECT(
  [(2,2)],
  [(1,5), (2,1), (3,4)]
)
```

Ejemplo INTERSECT

- 1ª iteración:

```
docID(p1) = docID([(2,2)]) = 2
docID(p2) = docID([(1,5), (2,1), (3,4)]) = 1
```

La primera entrada no apunta al mismo identificador de documento (2 != 1) así que se avanza la lista p2 por tener el identificador menor:

```
p1 = [(2,2)]
p2 = [(1,5), (2,1), (3,4)] → [(2,1), (3,4)]
answer = []
```

Ejemplo INTERSECT

- 2ª iteración:

```
docID(p1) = docID([(2,2)]) = 2
docID(p2) = docID([(2,1), (3,4)]) = 2
```

Ambas entradas apuntan al documento 2, así que este documento encaja con la consulta. Se añade el documento 2 al conjunto de soluciones y se avanzan ambas listas:

```
p1 = [(2,2)] → []
p2 = [(2,1), (3,4)] → [(3,4)]
answer → [2]
```

Ejemplo INTERSECT

- 3ª iteración:

```
p1 = []
p2 = [(3,4)]
```

Como p1 ha se ha vaciado se termina el bucle y se devuelve el conjunto de resultados **`answer = [2]`**

Modelo booleano

- La función INTERSECT(p1,p2) saca provecho de que las listas están **ordenadas**: cada lista se recorre como mucho una vez.
- Si p1 tiene longitud **x** y p2 longitud **y**, INTERSECT tiene un coste en **O(x+y)**. Además, el resultado de INTERSECT tendrá un tamaño máximo de **min(x,y) elementos**.
- El resultado de INTERSECT es una **lista de identificadores de documento ordenados**.
- Para consultas t1 AND t2 AND … AND tn se puede extender INTERSECT usando la heurística de *“intersecar primero las listas más cortas”*.

Modelo booleano

```
INTERSECT(<p1,...,pn>):
  terms = sortByLength(<p1,...,pn>)
  answer = head(terms)
  terms = tail(terms)
  while terms != [] and answer != []:
    e = head(terms)
    answer = INTERSECT(answer,e)
    terms = tail(terms)
  return answer
```

Modelo booleano

- La intuición de la heurística es *minimizar el crecimiento de la lista answer*. Por eso se intersecan las listas obtenidas del índice de menor a mayor tamaño.
- `head(lista)` devuelve el primer elemento de una lista: `head([1,2,3]) = 1`
- `tail(lista)` devuelve todos los elementos de la lista salvo el primero:
  `tail([1,2,3]) = [2,3]`

Modelo booleano

- Notad que `INTERSECT(<p1,...,pn>)` invoca a `INTERSECT(answer,e)`, donde:
  - `answer` es una lista de identificadores de documento: `[4,18,22]`
  - `e` es una lista obtenida del índice, así que contiene identificadores de documento y número de repeticiones: `[(4,2), (18,5), (22,7)]`
- Manejar esta diferencia de representación es sencillo a través de la función `docID()`

Modelo booleano

- Para consultas OR se necesitaría una función **UNION(p1,p2)** para unir listas de apariciones de manera eficiente. También se hace uso de que están **ordenadas**.
- El operador NOT puede ser poco eficiente si se considera aislado, pero se pueden crear funciones específicas para su uso más común:
  - x AND (NOT y) → **INTERSECT_NOT(p1,p2)**

Consultas de frase

- Las **consultas de frase** requieren encontrar términos contiguos en documentos, p. ej.
  - *“Stanford University”*
  - *“European Union”,*
  - *“President of the United States of America”*
  - *“Universidad Complutense de Madrid”*
- Para este tipo de consultas se utilizan los **índices completos**

Índices completos

- Almacenan las **posiciones** (*offset* o número de palabra) en las que aparece cada término en cada documento.
- Para resolver una consulta de frase, utilizan las listas de apariciones de todas las palabras involucradas.
- Se usa una variante de INTERSECT que en caso de coincidencia revisa si las posiciones relativas son las esperadas.

Índices completos

- INTERSECT_PHRASE(<p1,...,pn>):

```
  answer = []
  while allNotEmpty(<p1,...,pn>):
    if sameDocID(<p1,...,pn>):
      if consecutive(<p1,...,pn>):
        add(answer, docID(p1))

    <p1,...,pn> = advanceMin(<p1,...,pn>)
  return answer
```

Índices completos

- `sameDocID()` → bool: las cabezas de p1,...,pn almacenan el mismo docID.
- `consecutive()` → bool: las cabezas de p1,...,pn son consecutivas.
- `advanceMin()` : avanza las listas con menor docID en su cabeza (cada lista está ordenadas por docID). Puede avanzar todas las listas
- `allNotEmpty()` → bool: ninguna lista p1,...,pn está vacía

Ejemplo con índice completo

Índice		
el	**3**	**[(1,5,[1,3,5,7,9]), (2,1,[3]), (3,2,[17,29])]**
amigo	**1**	**[(2,2,[4,86])]**
hola	3	[(1,4,[22,24,56,71]), (2,3,[1,10,20]),(3,1,[1])
verde	**3**	**[(1,5,[19,25,32,40,59]),(2,1,[5]),(3,4,[76,80,90,111])]**
pez	1	[(2,2,[45,123])]
coral	1	[(1,6,[101,134,145,204,250,278])]
agua	3	[(1,3,[67,78,106]), (2,3,[56,78,99,301]), (3,2,[1,34])]
mar	2	[(1,3,[500,509,700]), (3,1,[72])]
abuela	1	[(2,4,[303,581,600,708])]

Queremos resolver la consulta de frase exacta **“el amigo verde”**

Ejemplo con índice completo

- Se obtendrán las listas del índice asociadas a los términos clave “el”, “amigo” y “verde”:
  - `p1 = index[“el”] = [(1,5,[1,3,5,7,9]), (2,1,[3]), (3,2,[17,29])]`
  - `p2 = index[“amigo”] = [(2,2,[4,86])]`
  - `p3 = index[“verde”] = [(1,5,[19,25,32,40,59]),(2,1,[5]), (3,4,[76,80,90,111])]`
- Se invocará a `INTERSECT_PHRASE(<p1,p2,p3>)` con dichos valores.

Ejemplo con índice completo

- **1ª iteración:**

```
p1 = [(1,5,[1,3,5,7,9]),      (2,1,[3]), (3,2,[17,29])]
p2 = [(2,2,[4,86])]
p3 = [(1,5,[19,25,32,40,59]), (2,1,[5]), (3,4,[76,80,90,111])]
```

Las cabezas de las listas no apuntan todas al mismo documento: `docID(p1) = 1`, `docID(p2) = 2`, `docID(p3) = 1`. Por lo tanto, se avanzan las lista de cabeza más pequeña (en este caso **p1** y **p3**).

```
p1 → [(2,1,[3]), (3,2,[17,29])]
p2 = [(2,2,[4,86])]
p3 → [(2,1,[5]), (3,4,[76,80,90,111])]
answer = []
```

Ejemplo con índice completo

- **2ª iteración:**

```
p1 = [(2,1,[3]),    (3,2,[17,29])]
p2 = [(2,2,[4,86])]
p3 = [(2,1,[5]),    (3,4,[76,80,90,111])]
```

Las cabezas de las listas apuntan todas al mismo documento 2, así que se comprueba si en las listas de posiciones existe una cadena consecutiva invocando a **consecutive(<p1, p2, p3>)**. Esta cadena existe en las posiciones 3-4-5, así que “el amigo verde” aparece en el documento 2.

Se añade el documento 2 al resultado y se avanzan las listas (en este caso todas contienen el mínimo documento en la cabeza)

```
p1 → [(3,2,[17,29])]
p2 → []
p3 → [(3,4,[76,80,90,111])]
answer → [2]
```

Ejemplo con índice completo

- **3ª iteración:**

  `AllNotEmpty(<p1, p2, p3>)` devuelve `False` porque `p2=[]`, así que el bucle termina.

  Se devuelve el resultado **`answer=[2]`**

Modelo vectorial

- Buscar los documentos que encajan con una consulta utilizando el **modelo vectorial** y el índice invertido es un poco más elaborado.
- Recordemos la función de **relevancia**:

$$R(dj,q) = \frac{dj \cdot q}{|dj||q|} = \frac{\sum_{i=1}^{t} w_{ij} w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2} \sqrt{\sum_{i=1}^{t} w_{iq}^2}}$$

Modelo vectorial

- Según parece, vamos a necesitar todos los pesos w_{ij} del documento y w_{iq} de la consulta para poder calcular la similitud.
- Si tengo que acceder a todos los datos de cada documento, ¿para qué me sirve el índice invertido?

$$R(dj,q) = \frac{dj \cdot q}{|dj||q|} = \frac{\sum_{i=1}^{t} w_{ij} w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2} \sqrt{\sum_{i=1}^{t} w_{iq}^2}}$$

Modelo vectorial

- Miremos primero el producto escalar:

$$dj \cdot q = \sum_{i=1}^{t} w_{ij} w_{iq}$$

- **PREGUNTA:** ¿Para una consulta dada, de verdad necesitamos conocer todos los w_{ij} ?

Modelo vectorial

- Miremos primero el producto escalar:

$$dj \cdot q = \sum_{i=1}^{t} w_{ij} w_{iq}$$

- **PREGUNTA:** ¿Para una consulta dada, de verdad necesitamos conocer todos los w_{ij} ?
- **RESPUESTA:** NO, únicamente para aquellos términos clave *ki* en los que w_{iq} es mayor que 0. Para los términos clave *kl* tales $w_{lq} = 0$ entonces $\boldsymbol{w_{lj} \cdot w_{lq} = w_{lj} \cdot 0 = 0}$.

Modelo vectorial

- Como *q* es un vector con 1 en los términos buscados y 0 en los no buscados, el producto escalar *dj* • *q* se simplifica:

$$d_j \cdot q = \sum_{i=1}^{t} w_{ij} w_{iq} \rightarrow \sum_{i \in \text{terms}(q)} w_{ij}$$

para los términos *i* que aparecen en *q*.

- *(Los términos i que no aparecen en q no deben considerarse, pues no contribuyen al sumatorio debido a que* $w_{iq} = 0$*)*

Modelo vectorial

- Por último, la norma |q| es un factor positivo constante en todas las similitudes, por lo que podemos omitirlo y el **orden relativo** de los documentos no cambia:

$$R(dj,q) > R(dk,q) \iff$$

$$\frac{dj \cdot q}{|dj||q|} > \frac{dk \cdot q}{|dj||q|} \iff$$

$$\frac{1}{|q|}\frac{dj \cdot q}{|dj|} > \frac{1}{|q|}\frac{dk \cdot q}{|dj|} \iff$$

$$\frac{dj \cdot q}{|dj|} > \frac{dk \cdot q}{|dj|}$$

Modelo vectorial

- Para el cálculo de la norma de |*dj*| necesitamos **todos sus pesos**, incluso para los términos clave que no aparecen en la consulta.

$$|dj| = \sqrt{\sum_{i=1}^{t} w_{ij}^2}$$

- **Pero no depende de la consulta → precalcular y almacenar**
- Supondremos que esta información está almacenada en una estructura `Length[N]` donde `Length[j]` = |*dj*|

Modelo vectorial: pseudocódigo

```
FASTCOSINESCORE(q, index, k):
1 Scores[N] = 0 # Diccionario de resultados
2 for t in q:
3   p = index[t] # Lista de pesos 'w' del término t en cada
                 # documento 'd'
4   for (d,w) in p:
5     Scores[d] += w # Evitamos el  producto
6 for d in Scores:     # Dividimos por |d|
7   Scores[d] = Scores[d]/Length[d]
8 return best(k,Scores)
```

Modelo vectorial: pseudocódigo

- 1) Inicializamos el producto escalar parcial asociado a cada documento d (como máximo habrá N)
- 3) Obtenemos la lista [(doc,peso),...(doc,peso)] asociada al término t
- 4-5) Sumamos el peso en cada producto escalar parcial
- 6-7) Dividimos entre la norma |d|
- 8) Obtenemos los k documentos con mayor relevancia

Ejemplo modelo vectorial

Índice		
...		
amigo	**1**	**[(2, 4.25)]**
hola	3	[(1, 2.75), (2, 5.25), (3, 1.15)
verde	**3**	**[(1, 3.45), (2, 7.02), (3, 6.65)]**
...		

Queremos resolver la consulta **amigo verde**, considerando que hemos precalculado la norma de los documentos en `Length`:

```
Length[1] = 22.75
Length[2] = 88.95
Length[3] = 14.55
...
```

Ejemplo modelo vectorial

- Antes de procesar las entradas del índice, **inicializamos** `Scores`.
- Puede ser una **tabla completa** para cada documento donde cada entrada se inicializa a 0: acceso constante, alto uso de memoria.
- Puede ser un **diccionario vacío**, dado que una consulta no involucrará un número elevado de documentos: acceso logarítmico o prácticamente constante, uso menor de memoria.

Resultado:

```
Scores[1] = 0
Scores[2] = 0
Scores[3] = 0
...
```

Ejemplo modelo vectorial

- Procesamos la lista de pesos del primer término clave “amigo”:
  `index[“amigo”] = [(2, 4.25)]`
- Por cada entrada se incrementa la entrada Scores del documento asociado:

  Resultado:

  ```
  Scores[1] = 0
  Scores[2] = 0+4.25 = 4.25
  Scores[3] = 0
  ...
  ```

Ejemplo modelo vectorial

- Procesamos la lista de pesos del segundo término clave “verde”:

  `index[“verde”] = [(1,3.45), (2,7.02), (3,6.65)]`
- Por cada entrada se incrementa la entrada Scores del documento asociado:

  Resultado:

```
Scores[1] = 0+3.45     = 3.45
Scores[2] = 4.25+7.02  = 11.27
Scores[3] = 0+6.65     = 6.65
...
```

Ejemplo modelo vectorial

- Una vez hemos terminado de procesa todos los términos clave, dividimos el valor de `Scores` por la norma del documento precalculada en `Length`:

  Resultado:

  ```
  Scores[1] = 3.45 /22.75 = 0.15
  Scores[2] = 11.27/88.95 = 0.13
  Scores[3] = 6.65 /14.55 = 0.46
  ...
  ```

- Los 3 documentos más relevantes (en orden) para la consulta **amigo verde** serán: [3, 1, 2].
- El resto de documentos que no contenían ni “amigo” ni “verde” tendrán relevancia 0 en `Scores`.

Bibliografía

Bibliografía

- **Information Retrieval: Implementing and Evaluating Search Engines.** Stefan Büttcher, Charles L. A. Clarke, Gordon V. Cormack. MIT Press, 2010.
- **Introduction to Information Retrieval**. *Christopher D. Manning, Prabhakar Raghavan y Hinrich Schütze*. Cambridge University Press. 2008. http://www-nlp.stanford.edu/IR-book/

Bibliografía

- **Modern Information Retrieval, the concepts and technology behind search**, second edition. *Ricardo Baeza-Yates y Berthier Ribeiro-Neto*. Pearson Education Limited (2011).
- **Search Engines: Information Retrieval in Practice** (International Edition). *W. Bruce Croft, Donald Metzler y Trevor Strohman*. Person Education. 2010.