Tópicos em Recuperação de Informação[1]

Nivio Ziviani

[1]Conjunto de transparências elaborado por Nivio Ziviani e Patrícia Correia

Recuperação de Informação

- Atividade de recuperar itens de informação armazenados em um meio que possa ser recuperado por computador.
- Lida com a representação, armazenamento, organização e acesso a itens de informação.
- Computador produziu uma sociedade que se alimenta da informação.
  - Produção de dados é espantosa: transações financeiras, procedimentos legais, atividades de governo, indústria editorial, etc.
- Como tratar um grande número de documentos (*gigabytes* de dados)?

Desafios

1. Armazenar os dados eficientemente:

   Uso de compressão.

2. Permitir acesso rápido:

   Uso de indexação (eletrônica).

Métodos tradicionais de compressão e busca têm que ser adaptados para responder aos desafios acima. Ex.:

- índice + texto (ambos comprimidos) ocupam menos da metade do arquivo original.
- o tempo para responder a uma consulta é muito menor do que se a base de dados não tivesse sido comprimida!

Bases de documentos

- Armazenamento e recuperação eficientes.
- Compressão $+$ indexação.

  $\Downarrow$

  Soluções isoladas existem há muitos anos, mas o uso combinado das duas técnicas não possível até muito recentemente.
- Imagens fazem parte de documentos.
  - Diagramas e fotografias.
  - Ocupam várias vezes mais espaço do que texto.

Tecnologia

- Muitas técnicas que vamos utilizar no curso foram inventadas e testadas recentemente e somente agora estão sendo colocadas em prática.

- Nenhuma máquina de busca comercial usa todas as técnicas que vamos estudar (ainda).
  - elas seguem o "*hard way*" - *hardware way* - com muito disco e muita memória principal.

Indexação

Índice

Mecanismo utilizado para localizar um dado termo em um texto (custo $<$ linear).

Tipos mais comuns de índices:

- Arquivos Invertidos (*inverted files* ou *postings files*).
- Arquivos de Assinaturas (*signature files*).
- Mapas de Bits (*bitmaps*).

Arquivos Invertidos

Tipicamente composto de:

1. arranjo contendo todas as palavras distintas do texto (chamado de vocabulário).
2. para cada palavra do vocabulário, uma lista de todos os documentos (identificados por números de documentos) nos quais aquela palavra ocorre.

A parte 2 pode ser mais geral: para cada palavra do vocabulário, uma lista de apontadores para todas as ocorrências daquela palavra no texto.

Exemplo de Arquivo Invertido

Documento	Texto
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

Exemplo de Arquivo Invertido

Número	Termo	Documento
1	cold	(2;1,4)
2	days	(2;3,6)
3	hot	(2;1,4)
4	in	(2;2,5)
5	it	(2;4,5)
6	like	(2;4,5)
7	nine	(2;3,6)
8	old	(2;3,6)
9	pease	(2;1,2)
10	porridge	(2;1,2)
11	pot	(2;2,5)
12	some	(2;4,5)
13	the	(2;2,5)

Consultas em Arquivos Invertidos

Termo único

Busca no vocabulário e recupera a lista de ocorrências (em documentos).

Conjunção: "palavra AND ... AND palavra"

Intersecção das listas de ocorrências.

Disjunção: "palavra OR ... OR palavra"

União das listas de ocorrências.

Negação: "NOT ..."

Complemento do resultado.

Exemplo: "some AND hot"

$< 4, 5 >$ AND $< 1, 4 > \;\Rightarrow\; < 4 >$

Granularidade do Índice

Corresponde a precisão com que uma palavra é localizada pelo índice:

1. Grossa ("*coarse-grained* ") : identifica apenas um bloco de texto.

2. Moderada ("*moderate-grained* ") : identifica apenas um documento.

3. Fina ("*fine-grained* ") : identifica cada palavra (outras vezes, cada byte).

Granularidade do Índice

Grossa:

- Índice requer menos memória.
- Consulta exige mais processamento em cada bloco.
- Consulta de frases leva a maior número de *"false matches"* (cada palavra aparece em algum lugar do bloco mas não em posições contíguas).

Fina (Indexação por palavra):

- Busca por frases é eficiente.
- Proximidade pode ser tratada sem ir ao texto.
- Busca aproximada é eficiente.
- Índice requer mais memória (uma ordem de grandeza).

Indexação por Palavra

Número	Termo	(Documento;Palavras)
1	cold	[2;(1;6),(4;8)]
2	days	[2;(3;2),(6;2)]
3	hot	[2;(1;3),(4;4)]
4	in	[2;(2;3),(5;4)]
5	it	[2;(4;3,7),(5;3)]
6	like	[2;(4;2,6),(5;2)]
7	nine	[2;(3;1),(6;1)]
8	old	[2;(3;3),(6;3)]
9	pease	[2;(1;1,4),(2;1)]
10	porridge	[2;(1;2,5),(2;2)]
11	pot	[2;(2;5),(5;6)]
12	some	[2;(4;1,5),(5;1)]
13	the	[2;(2;4),(5;5)]

Índice cresce:

- Cada apontador: número do documento e a posição da palavra no documento.
- Uma palavra pode ocorrer várias vezes em um mesmo documento.

Espaço Ocupado pelo Índice

Texto: N documentos

Índice: f apontadores

Espaço: $f.\lceil \log N \rceil$ bits

Métodos para Construção de Índice

Existem vários métodos para a construção de índices invertidos:

1^o - Matriz de freqüência (*frequency matrix*)

- Cada linha corresponde a um documento e cada coluna corresponde a um termo do vocabulário.
- Para criar o índice, deve-se gerar a transposta da matriz de freqüência.

Construção do Índice

Matriz de freqüência

Documento	Texto
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

	Termo												
	cold	days	hot	in	it	like	nine	old	pease	porridge	pot	some	the
1	1	–	1	–	–	–	–	–	2	2	–	–	–
2	–	–	–	1	–	–	–	–	1	1	1	–	1
3	–	1	–	–	–	–	1	1	–	–	–	–	–
4	1	–	1	–	2	2	–	–	–	–	–	2	–
5	–	–	–	1	1	1	–	–	–	–	1	1	1
6	–	1	–	–	–	–	1	1	–	–	–	–	–

Construção do Índice

Matriz de freqüência

Número	Termo	Documento					
		1	2	3	4	5	6
1	cold	1	–	–	1	–	–
2	days	–	–	1	–	–	1
3	hot	1	–	–	1	–	–
4	in	–	1	–	–	1	–
5	it	–	–	–	2	1	–
6	like	–	–	–	2	1	–
7	nine	–	–	1	–	–	1
8	old	–	–	1	–	–	1
9	pease	2	1	–	–	–	–
10	porridge	2	1	–	–	–	–
11	pot	–	1	–	–	1	–
12	some	–	–	–	2	1	–
13	the	–	1	–	–	1	–

Construção do Índice

Matriz de freqüência

- Para construir o índice: ler cada documento da coleção, ao final escrever a matriz, linha por linha no disco.
- Construção é bastante simples.
- Entretanto solução é bastante cara!

  Ex.: Bíblia contém 8.965 termos e 31.101 documentos.

  Tamanho da matriz: 8.965 x 31.101 x 4 bytes $\simeq$ 1 Gigabyte de memória principal.
- Soluções mais econômicas devem ser consideradas.

Construção do Índice

Coleção de 5 Gigabytes e 5 milhões de documentos.

Parâmetro	Símb.	Valores assumidos
Tamanho total do texto	B	5×10^9 bytes
Número de documentos	N	5×10^6
Número de palavras distintas	n	1×10^6
Número total de palavras	F	800×10^6
Número de ponteiros de índice	f	400×10^6
Tamanho final do arquivo invertido comprimido	I	400×10^6 bytes
Tamanho da estrutura de vocabulário dinâmico	L	30×10^6 bytes
Tempo de *seek* em disco	t_s	10×10^{-3} seg
Tempo de transferência em disco, por byte	t_r	$0,5 \times 10^{-6}$ seg
Tempo de codificação do arquivo invertido, por byte	t_d	5×10^{-6} seg
Tempo de comparação e troca de um registro de 10 b	t_c	10^{-6} seg
Tempo de *parsing*, *stemming* e procura por termo	t_p	20×10^{-6} seg
Total de memória principal disponível	M	40×10^6 bytes

Construção do Índice

Custo:

$$T = B.t_r + F.t_p + I.(t_d + t_r)$$

$B.t_r$: tempo de leitura
$F.t_p$: *parsing*
$I.(t_d + t_r)$: escreve arquivo invertido comprimido

- $B = 5 \times 10^9$ bytes (texto)
- $t_r = 0,5 \times 10^{-6}$ seg (tempo de transferência em disco)
- $F = 800 \times 10^6$ (número de palavras ou termos)
- $t_p = 20 \times 10^{-6}$ seg (*parsing* + *stemming* + *look up*)
- $I = 400 \times 10^6$ bytes (tamanho índice comprimido)
- $t_d = 5 \times 10^{-6}$ seg (tempo codificação)
- $T \cong 6$ horas (Bíblia < 30 segs)

Construção do Índice

Se fôssemos usar disco:

- 1^a fase:
  ler todos os documentos e gravar sequencialmente no disco: 30 minutos a mais
- 2^a fase:
  agora cada $(d, f_{d,t})$ correspondente a um termo t está em um lugar diferente no disco!
  4 milhões de segundos $\cong 6$ semanas!

Construção do Índice

Método	Seção	Figura	Memória (Mbytes)	Disco (Mbytes)	Tempo (Horas)
Linked lists (memory)	5.1	5.1	4000	0	6
Linked lists (disk)	5.1	5.1	30	4000	1100
Sort-based	5.2	5.3	40	8000	20
Sort-based compressed	5.3	–	40	680	26
Sort-based multiway merge	5.3	–	40	540	11
Sort-based multiway in-place	5.3	5.8; 5.9	40	150	11
In-memory compressed	5.4	5.12	420	1	12
Lexicon-based, no extra disk	5.4	–	40	0	79
Lexicon-based, extra disk	5.4	–	40	4000	12
Text-based partition	5.4	–	40	35	15

Construção do Índice

2^o - Inversão em Memória (*linked lists*)

- O índice é todo construído em memória principal.
- Para armazenar o vocabulário: tabela *hash* é geralmente a escolha mais econômica em termos de espaço e velocidade.
- Listas encadeadas em memória para armazenar as listas invertidas dos termos.
- O método leva cerca de 6 horas para indexar uma coleção de 5GB e consome 4GB de memória principal e nenhum espaço extra em disco.

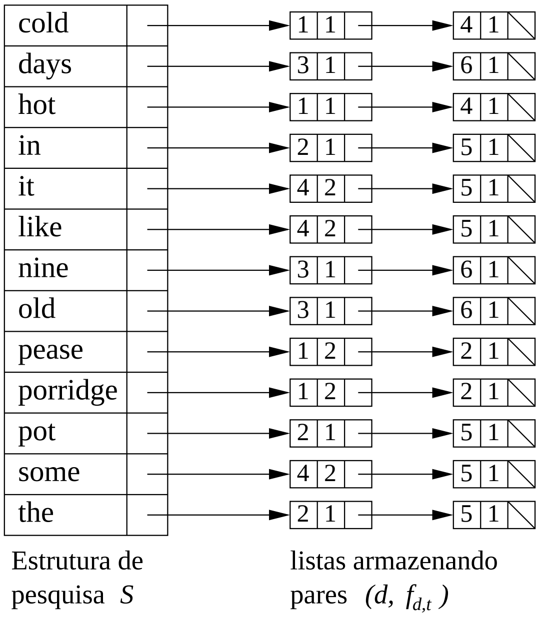
Construção do Índice

Inversão em Memória

1. /* Inicialização */
   Crie uma estrutura de dicionário vazia S.
2. /* Fase um: coleta das características dos termos */
   Para cada documento D_d na coleção, $1 \leq d \leq N$,
   (a) Leia D_d, realizando o *parser* para obter termos indexáveis.
   (b) **Para cada** termo indexável $t \in D_d$,
   i. **Faça** $f_{d,t}$ receber a frequência do termo t em D_d.
   ii. Busque por t em S.
   iii. **Se** t não estiver em S, insira-o.
   iv. Adicione um nó armazenando $(d, f_{d,t})$ na lista correspondente ao termo t.
3. /* Fase dois: saída do arquivo invertido */
   Para cada termo $1 \leq t \leq n$,
   (a) Inicialize uma nova entrada do arquivo invertido.
   (b) **Para cada** $(d, f_{d,t})$ na lista correspondente a t,
   Adicione $(d, f_{d,t})$ a essa entrada do arquivo invertido.
   (c) **Se** requerido, comprima a entrada do arquivo invertido.
   (d) Adicione essa entrada do arquivo invertido ao arquivo invertido.

Construção do Índice

Inversão em Memória



Construção do Índice

3^o - Inversão baseada em Ordenação (*sort-based*)

- O principal problema dos métodos descritos anteriormente é o alto consumo de memória principal.
- O uso de disco é inevitável quando grandes quantidades de texto precisam ser indexadas.
- Triplas $<$termo, doc, freq$>$ são armazenadas em arquivos temporários.
- O índice é obtido ordenando-se estas triplas em ordem ascendente do termo e depois do documento.

1. /* Inicialização */
   Crie uma estrutura de dicionário vazia S.
   Crie um arquivo temporário vazio em disco.
2. /* Processe o texto e escreva arquivo temporário */
   Para cada documento D_d na coleção, $1 \leq d \leq N$,
   (a) Leia D_d, realizando o *parser* para obter termos indexáveis.
   (b) **Para cada** termo indexável $t \in D_d$,
      i. **Faça** $f_{d,t}$ receber a frequência do termo t em D_d.
      ii. Busque por t em S.
      iii. **Se** t não estiver em S, insira-o.
      iv. Escreva um registro $(t, d, f_{d,t})$ no arquivo temporário, onde t é representado por seu número de termo em S.
3. /* Ordenação interna para gerar *runs* */
   Faça k receber o número de registros que podem ser carregados em memória.
   (a) Leia k registros da arquivo temporário.
   (b) Ordene-os em ordem crescente de t e, para valores iguais a t, em ordem crescente de d.
   (c) Escreva a *run* ordenada novamente no arquivo temporário.
   (d) Repita até não haver mais *run* a ser ordenada.
4. /* *Merging* */
   Intercale as *runs* em pares no arquivo temporário até existir somente uma *run* ordenada.
5. /* Geração do arquivo invertido */
   Para cada termo $1 \leq t \leq n$,
   (a) Inicialize uma nova entrada do arquivo invertido.
   (b) Leia todas as triplas $(t, d, f_{d,t})$ do arquivo temporário e gere a emtrada do arquivo invertido para o termo t.
   (c) Se requerido, comprima a entrada do arquivo invertido.
   (d) Adicione essa entrada do arquivo invertido ao arquivo invertido.

Construção do Índice

Inversão baseada em Ordenação

- Um método de ordenação, como o *Quicksort*, pode ser usado para esta etapa.
- O resultado da ordenação é gravado em disco e chamado de *run* ordenada.
- A próxima fase é a intercalação de todas *runs* ordenadas para gerar o índice final.
- Se existirem R *runs*, depois de $\lceil \log R \rceil$ passos o processo de intercalação (*external sort*) é finalizado, restando um único arquivo temporário em disco.

Construção do Índice

Inversão baseada em Ordenação

- O passo final na indexação é ler o arquivo temporário ordenado, gerando uma saída comprimida, se necessário.

- A inversão para a coleção de 5GB leva cerca de 20 horas usando 40MB de memória principal e 8GB de espaço extra em disco.

- Devido a grande quantidade de espaço em disco consumida durante a inversão, este método é considerado o melhor para coleções de tamanho moderado (10 a 100 MB).

Reduzindo Custos na Construção do Índice

- Utilizar técnicas de compressão e estratégias mais sofisticadas de intercalação
  - Compressão dos arquivos temporários.
  - Intercalação de múltiplos caminhos (*multiway*).
  - Intercalação *in-place*.

Intercalação de Múltiplos Caminhos

- Uso de intercalação de múltiplos caminhos reduz o tempo de construção do índice e a quantidade de espaço em disco necessária para a geração do índice final.

- Como: fila de prioridade (*heap*).

- Custo: $\lceil \log R \rceil$ comparações.

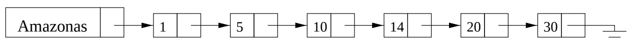
- Esta estratégia reduz o tempo de construção do arquivo invertido para a coleção de 5GB para cerca de 11 horas usando 40MB de memória principal e 540MB de espaço temporário em disco.

- Estes custos podem ainda ser reduzidos utilizando uma intercalação *in-place*.

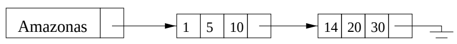
Compressão de Índice em Memória

Comprimir listas na memória equivale a aumentar a memória disponível.

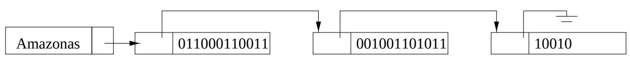
Representação de uma lista de ocorrências:



Utilizando blocos de memória:



Comprimindo dados na memória:



Custo de CPU: $O(u + \frac{u}{\sqrt{M}} \log \frac{u}{M})$

u: Tamanho do texto

M: Tamanho da memória principal

Tipos de Consultas

Existem dois tipos de consultas:

- Lógica (*boolean*)
  - lista de termos que são combinada através dos operadores lógicos AND, OR e NOT.
- Ordenada (*ranked*)
  - envolve uma heurística que é aplicada para medir a similaridade de cada documento com a consulta.

Consulta Lógica

- A resposta à consulta é formada pelos documentos que satisfazem a condição estipulada.

  Ex.: *text* AND *compression* AND *retrieval*

- Busca ampla ⇒ aumenta revocação (*recall*).

- Maioria do documentos são relevantes ⇒ aumenta precisão (*precision*).

  Ex.: (*text* OR *data* OR *image*) AND
  (*compression* OR *compaction* OR *decompression*) AND
  (*archiving* OR *retrieval* OR *storage* OR *indexing*)

  ( ) indicam a ordem das operações.

Consulta Ordenada

- Esforço para obter:
  - medidas de similaridade.
  - estratégias de ordenação.
- Com o objetivo de manter:
  - revocação alta.
  - precisão alta.

1. Técnica simples:
   - contagem do número de termos da consulta que aparecem no documento.
2. Técnica sofisticada (tamanho do documento):
   - atribui um peso numérico a cada termo.
   - medida do cosseno.
   - modelo vetorial.

Consulta Ordenada

3. Muitas outras técnicas...
4. Coleção TREC:
   - 50 consultas.
   - medida do cosseno consegue precisão acima de 40% na média (difícil de conseguir com modelo booleano).

Processamento de Consultas Lógicas

1. Vocabulário é pesquisado para cada termo.
2. Cada entrada do arquivo invertido é recuperada (e decodificada).
3. As entradas são intercaladas (interseção, união ou complemento).
4. Documentos são recuperados de acordo com a lista final.

Tipicamente, consultas com 5 a 10 termos levam cerca de 1 segundo para o processamento das listas.

- Visualização depende do tamanho da resposta.

Tipos de Consultas Lógicas

Conjuntivas:

text AND *compression* AND *retrieval*

- Termos são ordenadas pela freqüência do termo na coleção f_t.
- Lista de documentos candidatos é inicializada a partir do termo t de menor f_t: $C \leftarrow I_t$

Algoritmo para Consultas Conjuntivas

1. **Para cada** termo t da consulta **faça**
   (a) Obtenha t.
   (b) Pesquise no vocabulário.
   (c) Armazene f_t e o endereço de I_t, a entrada do arquivo invertido para t.
2. Identifique o termo t da consulta com o menor f_t.
3. Leia a entrada I_t correspondente do arquivo invertido. Faça $C \leftarrow I_t$, sendo C a lista de $candidatos$.
4. **Para cada** termo t restante **faça**
   (a) Leia a entrada I_t do arquivo invertido.
   (b) **Para cada** $d \in C$ **faça**
   Se $d \notin I_t$ **então**
   $C \leftarrow C - \{d\}$.
   (c) **Se** $|C| = 0$ **então**
   Retorne *não existem respostas*.
5. **Para cada** $d \in C$ **faça**
   (a) Procure o endereço do documento d.
   (b) Recupere o documento d e o apresente ao usuário.

Ordem de Processamento dos Termos

Razões para iniciar C com termo de menor f_t:

1. Minimizar espaço temporário de memória
   - Tamanho de C é não crescente.
   - Maior tamanho é na inicialização.
   - C pode ser reduzido rapidamente, algumas vezes a 0.

Ordem de Processamento dos Termos

Razões para iniciar C com termo de menor f_t:

2. Minimizar tempo de resposta

- Processamento envolve consulta a membros da lista e não intercalação.
  - Intercalação envolvendo $\mid C \mid$ candidatos e f_t apontadores: $\mid C \mid + f_t$ passos.
  - Se $\mid C \mid = 60$ e $f_t = 60000 \Rightarrow 60060$ passos.
  - Como as listas estão ordenadas: busca binária $\Rightarrow$ $\mid C \mid \log 60060$.
  - Para j>2 tamanho da menor lista é O(1).
  - Para j=2 tamanho da menor lista é O$(n^{\beta(1-2\beta)} \log n)$.

Tipos de Consultas Lógicas

Não-Conjuntivas:

(*text* OR *data* OR *image*) AND
(*compression* OR *compaction*) AND
(*retrieval* OR *indexing* OR *archiving*)

- Cada termo entre parêntesis pode ser processado simultaneamente:
  - candidatos ficam no conjunto se aparecem em qualquer membro do grupo OR.
- C deve ser inicializado com o menor conjunto resultante da união dos membros.
- Aproximação para tamanho de cada conjunto:
  - soma dos f_t de cada termo constituintes ignorando possibilidade de repetição no OR (aproximação pessimista).

Como Armazenar e Acessar o Vocabulário

O vocabulário de um arquivo invertido armazena os termos que serão usados para buscar na coleção e informações auxiliares para permitir que consultas sejam processadas.

Método	**Armazenamento**
Fixed Length Strings	28 Mbytes
Terminated Strings	20 Mbytes
Four-entry blocking	18 Mbytes
Front coding	15,5 Mbytes
Hashing perfeito	13 Mbytes

Para um vocabulário de 1.000.000 de termos

Estruturas

Fixed Length Strings

- Para um vocabulário de 1.000.000 de palavras:
  - 20 bytes: palavra (assumindo que palavras ocupam no máximo 20 bytes).
  - 4 bytes: endereço do arquivo invertido.
  - 4 bytes: f_t (número de docs que contêm o termo t).
  - Vocabulário requer 28 Megabytes.
- Esta alternativa é cara. Uma redução pode ser conseguida se todas as palavras forem concatenadas como uma palavra longa.

Estruturas

Fixed Length Strings

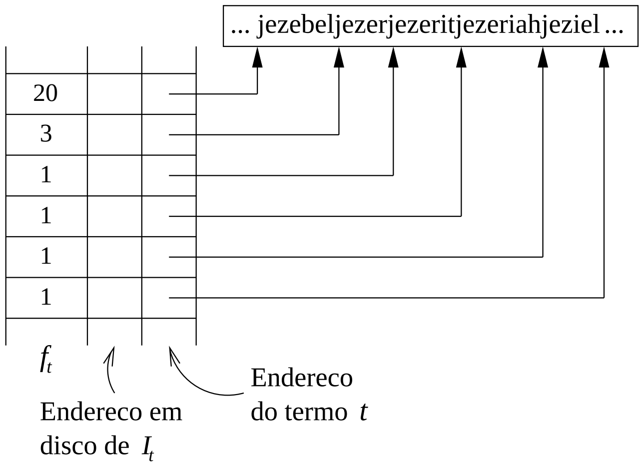
Estruturas

Terminated Strings

- Cada termo ocupa exatamente o seu tamanho.
- Para um vocabulário de 1.000.000 de palavras:
  - O vocabulário requer 20 Megabytes.
- O espaço requerido pode ainda ser reduzido eliminando muitos dos ponteiros de strings.

Estruturas

Terminated Strings



Estruturas

Four-entry Blocking

- 1 byte antes de cada palavra para indicar o seu tamanho.

$\Downarrow$

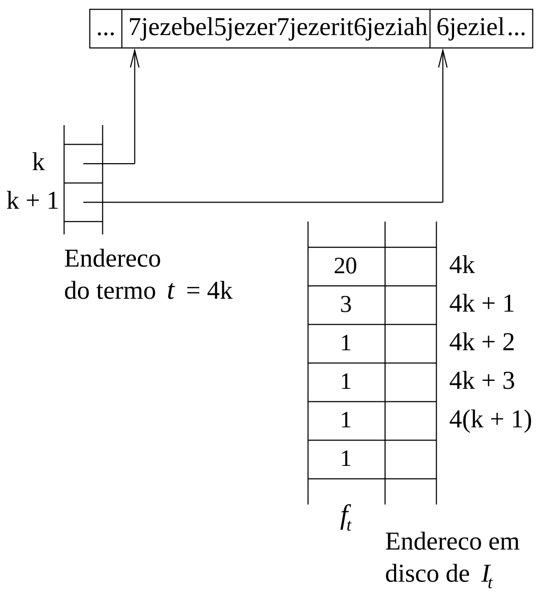
- Para cada grupo de 4 palavras, 12 bytes são ganhos.

$\Downarrow$

- Ganho de 2 Megabytes. O espaço total necessário é de 18 Megabytes.
- Para blocos de 8 palavras, o ganho é de + 0.5 Megabytes.
- Para blocos de 16 palavras, o ganho é de + 0.25 Megabytes.

Estruturas

Four-entry Blocking



Estruturas

Front Coding

- O ganho depende do vocabulário. Na média, 3 a 5 caracteres casam com custo adicional de 1 byte.

$$\Downarrow$$

- Ganho médio é de 2.5 bytes por palavra.

- Regra do dedão:
  - *Front coding* salva aproximadamente 40% do espaço para vocabulário da lingua inglesa.

- Uma estratégia 3-em-4 *front-coding*: salva 4 bytes em cada 3 palavras com custo extra de 2 bytes.

$$\Downarrow$$

- Total de espaço necessário: 15.5 Megabytes.

Estruturas

Front Coding

Palavra	Front coding completo	Front coding "3-in-4" parcial
7, jezebel	3, 4, ebel	, 7, jezebel
5, jezer	4, 1, r	4, 1, r
7, jezerit	5, 2, it	5, 2, it
6, jeziah	3, 3, iah	3, , iah
6, jeziel	4, 2, el	, 6, jeziel
7, jezliah	3, 4, liah	3, 4, liah
6, jezoar	3, 3, oar	3, 3, oar
9, jezrahiah	3, 6, rahiah	3, , rahiah
7, jezreel	4, 3, eel	, 7, jezreel
11, jezreelites	7, 4, ites	7, 4, ites
6, jibsam	1, 5, ibsam	1, 5, ibsam
7, jidlaph	2, 5, dlaph	2, , dlaph

A palavra antes de *jezebel* era *jezaniah*.

Estruturas

Front Coding

- Cada entrada do vocabulário pode ser mais reduzida ainda:

$$f_t \Rightarrow \lceil \log N \rceil \text{ bits, no exemplo anterior.}$$

$$\Downarrow$$

$$f_t + d \approx 28 \text{ bits cada.}$$

$$\Downarrow$$

- Ganho de mais 1 Megabyte.

- É possível fazer melhor?

  Resp.: Sim, não armazenando o vocabulário!

Referências

[WMB99] Witten I., Moffat A., Bell C. Managing Gigabytes Compressing and Indexing Documents and Images. 2nd Ed., 1999, pag. 161-169.

Hashing Perfeito

Uma função de transformação (*hashing function*) transforma uma conjunto de n chaves x_j em um conjunto de valores inteiros no intervalo:

$$1 \leq h(x_j) \leq m$$

com duplicações (ou colisões) permitidas.

Exemplo de função:

$$h(x) = x \bmod m$$

onde,

x: inteiro $\quad m > \frac{n}{\alpha}, \quad \alpha = \frac{\#\text{registros}}{\text{espaços disponíveis}}$

Ex.: $n = 100 \; chaves \; inteiras$

$h(x) = x \bmod 143$

$\alpha = 0.7$

$m = 143$

Hashing Perfeito

- Quando $h(x_i) = h(x_j)$ sse $i = j$

$$\Downarrow$$

Função Hashing Perfeita (PHF)

  – Neste caso não há colisões.

- Quando $m = n \quad (\alpha = 1.0)$

$$\Downarrow$$

Função Hashing Perfeita Mínima (MPHF)

  – 1 acesso.

  – Não há lugares vazios na tabela.

Hashing Perfeito

- Quando $x_i < x_j$ e $h(x_i) < h(x_j)$ a ordem é preservada.

$$\Downarrow$$

MPHF com Ordem Preservada (OPMPHF)

  – Chaves são localizadas em $O(1)$.

  – Não há espaço vazio.

  – Processamento na ordem lexicográfica.

Hashing Perfeito

Qual a vantagem da OPMPHF h ?

- Não há necessidade de armazenar os termos t do vocabulário.

- Para qualquer termo t basta armazenar na posição da tabela hash $h(t)$:
  - endereço do arquivo invertido.
  - f_t (# de documentos onde o termo t aparece.)

Desvantagem:

- Espaço ocupado para descrever a função h.

Entretanto, é possível obter um método com $m \cong 1.25n$.

Hashing Perfeito

OPMPHF:

$$h_j(t) = (\sum_{i=1}^{|t|} t[i] \times w_j[i]\ ) \bmod m,\ j = 1, 2$$

onde:

$t[i]$: i-ésimo caractere do termo t na base

$$36 = \begin{cases} 0 - 9 & \text{digitos 0 - 9} \\ 10 - 35 & \text{caracteres a - z} \end{cases}$$

$|\ t\ |$: número de caracteres do termo t

Um arranjo especial g mapeia $1 \ldots m$ para o intervalo $1 \ldots n$

Hashing Perfeito

Para obter a OPMPHF h(t):

$$h(t) = g(h_1(t)) +_n g(h_2(t))$$

onde:

$+_n$: adição $\bmod\ n$

Ex.: $4 +_5 7 = 1$

Resumo:

- Avalia duas funções não perfeitas.
- Converte os resultados usando o arranjo g.
- Adiciona resultados usando $\bmod\ n$.

Hashing Perfeito

Como obter o arranjo g:

Grafo com m-vértices e arcos $(h_1(t), h_2(t))$ para cada t.

- Cada termo corresponde a 1 arco.

- $h_1(t)$ e $h_2(t)$ definem quais vértices devem ser ligados.

- Cada arco é rotulado com o valor $h(t)$, o valor desejado da função hash para o termo t.

Encontrar mapeamento tal que para cada arco $(h_1(t), h_2(t))$ o mapeamento dá o rótulo do arco $g(h_1(t)) +_n g(h_2(t)) = h(t)$.

Hashing Perfeito

Termo t	$H_1(t)$	$H_2(t)$	$H_3(t)$
jezebel	5	9	0
jezer	5	7	1
jezerit	10	12	2
jeziah	6	10	3
jeziel	13	7	4
jezliah	13	11	5
jezoar	4	2	6
jezrahiah	0	3	7
jezreel	6	3	8
jezreelites	8	4	9
jibsam	9	14	10
jidlaph	3	1	11

x	$g(x)$
0	0
1	4
2	0
3	7
4	6
5	0
6	1
7	1
8	3
9	0
10	2
11	2
12	0
13	3
14	10

Hashing Perfeito

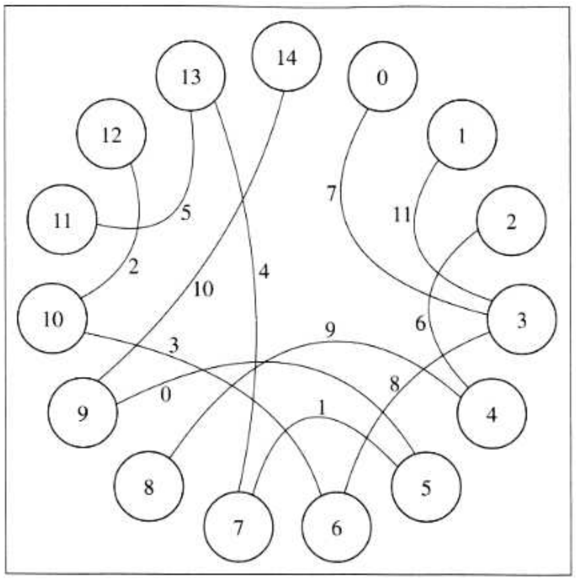


Figure 4.4 Graph corresponding to hash function of Table 4.3.

Hashing Perfeito

Como obter o arranjo g:

- Para um grafo qualquer, achar tal mapeamento, se existir, é difícil.
- Entretanto, para grafos acíclicos o problema é fácil (vide exemplo).
- Grafo com ciclo acaba rotulando de novo um vértice já processado com um rótulo diferente!
- Logo, tem-se que procurar um grafo sem ciclo, onde o processo de rotular é sempre possível.

Hashing Perfeito

Para rotular um grafo acíclico:

1. **Para cada** $v \in V$ **faça**
   $g[v] \leftarrow desconhecido$.
2. **Para cada** $v \in V$ **faça**
   Se $g[v] = desconhecido$ **então**
   $Rotule(v, 0)$.

Onde a função $Rotule(v, c)$ é definida por:

1. **Se** $g[v] \neq desconhecido$ **então**
   Se $g[v] \neq c$ **então**
   Retorne *Falha: o grafo não é acíclico.*
   Senão
   Retorne *Este vértice já foi visitado.*
2. $g[v] \leftarrow c$.
3. **Para cada** $u \in Adjacente(v)$ **faça**
   $Rotule(u, h((v, u)) - g[v])$.

Adjacente(v): vértices que compartilham um arco com v.

$h((v, u))$: rótulo associado com o arco (v, u).

Hashing Perfeito

Geração da OPMPHF:

1. Escolha um valor para m.

2. Escolha os pesos $w_1[i]$ e $w_2[i]$ para $1 \ll i \ll max_t \mid t \mid$.

3. Gerar o grafo $G = (V, E)$ onde:
   $V = \{1, \ldots, m\}$ e
   $E = \{(h_1(t), h_2(t)) \mid t \in L\}$

4. Tente obter o mapeamento g.

5. Se houve falha, retornar ao passo 2.

6. Retornar $w_1[i]$, $w_2[i]$ e g.

Teoria de Grafos Randômicos

- Para $m \ll 2m$
  - Prob. de gerar grafo acíclico $\rightarrow$ 0, quando n cresce.

- Para $m > 2m$
  - Prob. de um grafo randômico de m vértices e n arestas ser acíclico é aproximadamente:

$$\sqrt{\frac{m - 2m}{m}}$$

- O número esperado de grafos gerados até encontrar o primeiro acíclico é:

$$\sqrt{\frac{m}{m - 2m}}$$

  Ex.: para $m = 3n$
  $\sqrt{3} \cong 1.7$

- Problema: espaço $m = 3n$ é muito!

Hashing Perfeito

Solução para reduzir m:

- Ao invés de usar um 2-grafo, onde cada arco conecta 2 vértices, usar uma terceira função hash h_3 randômica, formando um 3-grafo sobre os m vértices:
  - cada arco é uma tripla $h_1(t), h_2(t), h_3(t)$.
  - função hash: $h(t) = g(h_1(t)) +_n g(h_2(t)) +_n g(h_3(t))$.

- Fica mais complexo, mas a lógica é a mesma.

- Neste caso, para $m > 1.23n$ o número de tentativas para para gerar o grafo é O(1).

- TREC com mais de 500.000: $<$ 1 minuto.

Paradoxo do Aniversário

Quantas pessoas têm que ser colocadas juntas para que a probabilidade de que 2 pessoas façam aniversário no mesmo dia seja maior que 50%?

A probabilidade se inserir n itens consecutivos sem colisão é:

$$\frac{m-1}{m} \times \frac{m-2}{m} \times \ldots \times \frac{m-n+1}{m} = \prod_{i=1}^{n} \frac{m-i+1}{m} = \frac{m!}{(m-n)!m^n}$$

n	p
10	0.883
22	0.524
23	0.493
30	0.303

$p \approx \frac{n(n-1))}{730}$ para n pequeno

$n = 10 \Rightarrow p \approx 87.7\%$

Referências

[CHM92] Czech, Z.J., Havas, G., Majewski, B.S. An Optimal Algorithm for Generating Minimal Perfect Hashing functions. Information Processing Letters, 43, 1992, 257-264.

[HMWC93] Havas, G., Majewski, B.S., Wormald, N.C., Czech, Z.J. Graphs, Hipergraphs and Hashing. Proceedings 19th International Workshops on Graph - Theoretic Concepts in Computer Science. Springer-Velasq LNCS, 1993.

Precisão e Revocação

- Revocação: mede a proporção dos documentos relevantes presentes no banco de dados que foi recuperada.

$$R_r = \frac{\text{número relevante que é recuperado}}{\text{número total relevante}}$$

- Precisão: mede a proporção dos documentos recuperados em uma consulta que são relevantes.

$$P_r = \frac{\text{número recuperado que é relevante}}{\text{número total recuperado}}$$

- Documentos relevantes: são aqueles semanticamente relacionadas coma questão pesquisadas.

Precisão e Revocação

r		R_r	P_r
1	R	10%	100%
2	–	10%	50%
3	–	10%	33%
4	R	20%	50%
5	R	30%	60%
6	–	30%	50%
7	R	40%	57%
8	–	40%	50%
9	–	40%	44%
10	–	40%	40%
11	–	40%	36%
12	R	50%	42%
13	R	60%	46%
14	R	70%	50%
15	–	70%	47%
16	R	80%	50%
17	–	80%	47%
18	–	80%	44%
19	R	90%	47%
20	–	90%	45%
21	–	90%	43%
22	R	100%	45%
23	–	100%	43%
24	–	100%	42%
25	–	100%	40%

Revocação	Precisão	Precisão interpolada
0%	–	100%
10%	100%	100%
20%	50%	60%
30%	60%	60%
40%	57%	57%
50%	42%	50%
60%	46%	50%
70%	50%	50%
80%	50%	50%
90%	47%	47%
100%	45%	45%
média 3-pontos		53%
média 11-pontos		61%

Precisão e Revocação

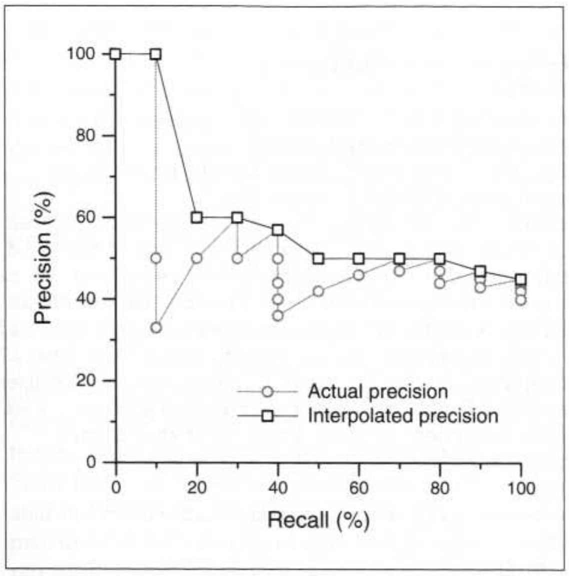


Figure 4.10 Recall-precision curve for ranking of Table 4.9.

Consulta Ordenada

Coordinate matching:

Técnica simples: conta o número de termos da consulta que aparece em cada documento.

- Quanto maior o número de termos da consulta em um documento, mais provável que ele seja relevante.
- Consulta se torna híbrida, intermediária entre a consulta conjuntiva AND e a consulta disjuntiva OR:
  - documento que contenha qualquer dos termos é resposta potencial.
  - preferência é para documentos que contenham todos ou a maioria dos termos.
- Toda informação necessária está no arquivo invertido.

Consulta Ordenada

d	**Documento** D_d
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	In the pot cold, in the pot hot,
5	Pease porridge, pease porridge,
6	Eat the lot.

Coleção com 6 documentos e 10 termos.

$Q = \text{``}eat\text{''} \Rightarrow D_6$

$Q = \text{``}hot\ porridge\text{''}$

Conjuntiva: D_1

Coordinate matching: $D_1 > D_2 = D_4 = D_5 > D_3 = D_6$

Similaridade do Produto Interno

Produto interno entre vetor da consulta e vetores de docs.

(a)	d	**Vetores de documentos** ($w_{d,t}$)									
		col	*day*	*eat*	*hot*	*lot*	*nin*	*old*	*pea*	*por*	*pot*
	1	1	0	0	1	0	0	0	1	1	0
	2	0	0	0	0	0	0	0	1	1	1
	3	0	1	0	0	0	1	1	0	0	0
	4	1	0	0	1	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	1	1	0
	6	0	0	1	0	1	0	0	0	0	0
(b)	*eat*	0	0	1	0	0	0	0	0	0	0
	hot porridge	0	0	0	1	0	0	0	0	1	0

Similaridade entre consulta Q com documento D_d:

$$M(Q, D_d) = Q.D_d$$

onde,

$$X.Y = \sum_{i=1}^{n} x_i.y_i$$

Ex.: $M(\text{``}hot,\ porridge\text{''},\ D_1) = 2$

Problemas

1. Não considera freqüência do termo.

   Ex.: $porridge$ aparece 2 vezes em D_1 e 1 vez em D_2 $Q = \text{``}porridge\text{''}$: os 2 documentos são ordenados igualmente.

2. A fórmula não considera a raridade do termo.

   Ex.: eat aparece somente em D_6 e $porridge$ aparece em D_1, D_2 e D_5.

3. Longos documentos são favorecidos (talvez indevidamente pela diversidade do vocabulário, devido ao tamanho).

Soluções

1^o **Problema:**

Substituir bit por inteiro indicando o número de vezes que o termo aparece no documento.

Ex.: $M(\text{“}hot,\ porridge'',\ D_1)$

(0,0,0,1,0,0,0,0,1,0).(1,0,0,1,0,0,0,2,2,0) = 3

Esta idéia pode ser generalizada:

$w_{d,t}$: document-term weight e $w_{q,t}$: query-term weight

$$M(Q, D_d) = Q.D_d = \sum_{i=1}^{n} w_{q,t}.w_{d,t}$$

Quando t não ocorre em Q, $w_{q,t} = 0$

Logo,

$$M(Q, D_d) = \sum_{t \in Q} w_{q,t}.w_{d,t}$$

Soluções

2^o **Problema:**

Dar peso aos termos de acordo com o inverso da freqüência em documentos - idf (*inverse document frequency*).

$$w_t = \log_e \left(1 + \frac{N}{f_t}\right)$$

onde,

N: número de documentos na coleção.

f_t: número de documentos que contém temo t.

$$w_{d,t} = (1 + \log_e f_{d,t}).w_t$$

Conhecida como regra TF x IDF (*term frequency x inverse document frequency*)

Lei de Zipf:

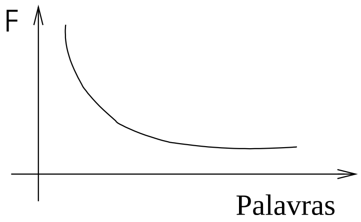
Freqüência do termo é inversamente proporcional ao seu *“rank”*.

Soluções

Zipf, 1949

Human Behavior and the Principle of Least Effort

Frequência da i-ésima palavra mais frequente é $1/i^{\Theta}$ vezes a da palavra mais frequente, $\Theta \geq 1$



Soluções

w_t: **Peso do termo** t

$w_t = \frac{1}{f_t}$

$w_t = log_e(1 + \frac{N}{f_t})$

$w_t = log_2 \frac{N}{f_t}$ Persin, Zobel & Sacks_Davis: JASIS, 1996

- Logaritmo não deixa um termo com $f_t = 1$ ser duas vezes mais importante que um temo com $f_t = 2$

Soluções

$r_{d,t}$: ***Relative term frequency***

- Existem muitas maneiras para calcular $r_{d,t}$

$r_{d,t} = 1 + log_e f_{d,t}$

$r_{d,t} = 1$

$r_{d,t} = f_{d,t}$

$r_{d,t} = log_2 f_{d,t}$ Persin, Zobel & Sacks_Davis: JASIS, 1996

- Crescimento menor quando $f_{d,t}$ cresce com o uso do logaritmo (crescimento monotônico)

Valores de documentos podem ser calculados como:

$W_{d,t} = r_{d,t}$ ou

$W_{d,t} = r_{d,t} \cdot w_t$ (TF × IDF)

$r_{d,t}$: crescimento monotônico
w_t: decrescimento monotônico

Soluções

- A direção de dois vetores (documento e consulta), ou mais precisamente a *diferença da direção de cada vetor*, é o que importa.
- Ângulo Θ entre dois vetores satisfaz:

$$X.Y = |X||Y| \cos\Theta$$

$X.Y$: produto interno

$|X| = \sqrt{\sum_{i=1}^{n} x_i^2}$: distância Euclidiana do vetor X

X: vetor n-dimensional $< x_i >$

$$\cos\Theta = \frac{X.Y}{|X||Y|}$$

Soluções

3^o **Problema:**

Documentos longos são favorecidos porque contêm mais termos e o valor do produto interno cresce.

Introduzir a normalização:

$$M(Q, D_d) = \frac{\sum_{t \in Q} w_{q,t}.w_{d,t}}{\mid D_d \mid}$$

onde,

$\mid D_d \mid = \sum_i f_{d,t}$ (comprimento do documento D_d)

Soluções

Implicações:

1. Normalização descrita anteriormente é a distribuição Euclidiana do documento – i.e., comprimento no espaço n-dimensional dos pesos documento-termo que descrevem o documento

2. Fórmula dá uma visualização clara do que a regra de ranking significa:
   - Documentos são pontos na região positiva do espaço n-dimensional, onde pequenos documentos estão mais próximos da origem e *consultas são raios a partir da origem*

Modelo de Espaço Vetorial

Documentos e consultas são representados por vetores em um espaço n-dimensional.

Da álgebra vetorial:

$$\cos\Theta = \frac{X.Y}{\mid X \mid . \mid Y \mid} = \frac{\sum_{i=1}^{n} x_i . y_i}{(\sqrt{\sum_{i=1}^{n} x_i^2})(\sqrt{\sum_{i=1}^{n} y_i^2})}$$

Normalização: distância Euclidiana do documento (comprimento no espaço n-dimensional do conjunto de pesos doc-termo).

Similaridade expressa pelo $\cos\Theta$: quanto maior o cosseno maior a similaridade.

$\cos\Theta = 1$ para $\Theta = 0$

$\cos\Theta = 0$ para $\Theta = 90^o$

Modelo de Espaço Vetorial

$$\cos(Q, D_d) = \frac{Q.D_d}{\mid Q \mid \mid D_d \mid} = \frac{\sum_{i=1}^{n} w_{q,t}.w_{d,t}}{W_q \, W_d}$$

onde,

$W_d = \sqrt{\sum_{i=1}^{n} w_{d,t}^2}$ e $W_q = \sqrt{\sum_{i=1}^{n} w_{q,t}^2}$

- Pode-se usar qualquer método *"term-weighting"*.
- Um bom método (simples): TF x IDF

$$\cos(Q, D_d) = \frac{1}{W_d \, W_q} \sum_{t \in Q \bigcap D_d} (1 + \log_e f_{d,t}). \log_e (1 + \frac{N}{f_t})$$

- Como W_q é constante para uma dada consulta ele pode ser desprezado sem alterar o *"ranking"*.

Modelo de Espaço Vetorial

d	Vetores de documentos ($w_{d,t}$)										$\mathbf{W_d}$
	col	*day*	*eat*	*hot*	*lot*	*nin*	*old*	*pea*	*por*	*pot*	
1	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.7	1.7	0.0	2.78
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	1.73
3	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.73
4	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.7	2.21
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.7	1.7	0.0	2.39
6	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.41
f_t	2	1	1	2	1	1	1	3	3	2	
w_t	1.39	1.95	1.95	1.39	1.95	1.95	1.95	1.10	1.10	1.39	

d	Consulta			
	eat	*porridge*	*hot porridge*	*eat nine day old porridge*
	$W_q = 1.95$	$W_q = 1.10$	$W_q = 1.77$	$W_q = 3.55$
1	0.00	0.61	0.66	0.19
2	0.00	0.58	0.36	0.18
3	0.00	0.00	0.00	0.63
4	0.00	0.00	0.36	0.00
5	0.00	0.71	0.44	0.22
6	0.71	0.00	0.00	0.39
Top	6	5	1	3

Medida do Cosseno

- Exige mais informação do que o processamento de consultas booleanas
- Decisões importantes sobre como estruturar a informação para fazer o *processamento do ranking eficiente*:

1. Precisamos da $f_{d,t}$ no índice
2. Como avaliar a medida do cosseno
   - Para grandes coleções é potencialmente caro
     - tempo CPU
     - largura de banda de disco
     - demanda por memória
   - Soluções
     - limitar o número de candidatos ao ranking
     - ordenar por $f_{d,t}$
     - representação eficiente de w_d

Implementação da Medida do Cosseno

Freqüência de t em d ($f_{d,t}$):

- $f_{d,t}$ tem que ser adicionado ao índice.
- $f_{d,t}$ é necessário para a maioria dos métodos de atribuição de peso aos termos (*term weight assignment*).
- Entrada para *"porridge"* no arquivo invertido:
  <3;[1,2,5]> passa a ser <3;[(1,2)(2,1),5,2)]>

  onde,
  $f_{porridge} = 3$ (*porridge* ocorre em 3 documentos: 1,2,5)
  $f_{1,porridge} = 2$ (*porridge* ocorre 2 vezes no documento 1)

$$\Downarrow$$

$$< f_t; [(d_1, f_{1,t}), (d_2, f_{2,t}), ..., (d_{f_t}, f_{d_{f_t},t})] >$$

Implementação da Medida do Cosseno

Como resolver problema de espaço extra necessário para armazenar $f_{d,t}$?

Código unário:

1	0
2	10
3	110

Logo, a entrada para o temo t no arquivo invertido cresce de um número de bits igual ao número total de ocorrências de t na coleção.

$$F_t = \sum_{d=1}^{N} f_{d,t} \quad \text{(total de ocorrências de } t\text{)}$$

$$F = \sum_{i=1}^{n} F_t \quad \text{(total de palavras na coleção)}$$

Logo, o arquivo invertido cresce de F bits.

Implementação da Medida do Cosseno

Fato:

É possível indexar toda palavra do texto usando menos de 1 byte por apontador (mesmo usando $f_{d,t}$)

Elias-γ:

- γ nunca é muito pior que unário, mas pode ser muito melhor!
  - Para documentos pequenos ocupa 1 bit a mais do que o unário
  - Muito melhor quando existem muitos termos com $f_{d,t}$ alto

Implementação da Medida do Cosseno

Método	**Bits por apontador**			
	Bíblia	*GNUbib*	*Comact*	*TREC*
Unário	1.27	1.16	1.74	2.49
γ	1.38	1.23	1.88	2.13
δ	1.53	1.32	2.15	2.41
Observed frequency	1.27	1.15	1.72	2.02
Interpolative	0.76	0.45	1.41	1.77

- Coleções *Bible, GNUbib, Comact*: código unário é perto do ótimo. Isto ocorre porque a distribuição das probabilidades é tipo: $Pr[x] = (\frac{1}{2})^x$ (probabilidades são múltiplos de 1/2).
- Usando compressão, mesmo para índice aumentando de $f_{d,t}$, para TREC o índice ocupa 6.3% do texto.
- Elias-γ: melhor opção para coleções grandes em linguagem natural. Consome $1 + 2\log x$ bits (menor do que o unário para termos com $f_{d,t}$ alto).

Cálculo do Valor do Cosseno

$$\cos(Q, D_d) = \frac{1}{W_d\, W_q} \sum_{t \in Q \bigcap D_d} (1 + \log_e f_{d,t}).\log_e(1 + \frac{N}{f_t})$$

$W_q =$ constante

$W_d = \sqrt{\sum_{i=1}^{n} w_{d,t}^2}$

$w_{d,t} = (1 + \log_e f_{d,t}).w_t$

$w_t = \log_e(1 + \frac{N}{f_t})$

Estruturas de dados:

1. Arranjo para W_d (pode ser pré-computado).

2. Acumuladores indexados por d para acumular $\cos(Q, D_d)$ (qualquer doc d contendo algum termo de Q faz $A_d > 0$).

Cálculo do Valor do Cosseno

Para recuperar r documentos usando a medida do cosseno

1. $A \leftarrow \{\}$. A é o conjunto de acumuladores.
2. **Para cada** termo na consulta $t \in Q$ **faça**
   (a) Obtenha t.
   (b) Pesquise no vocabulário.
   (c) Armazene f_t e o endereço de I_t, a entrada do arquivo invertido para t.
   (d) $w_t \leftarrow 1 + \log_e(N/f_t)$.
   (e) Leia a entrada do arquivo invertido I_t.
   (f) **Para cada** par $(d, f_{d,t})$ em I_t **faça**
      i. **Se** $A_d \notin A$ **então**
         $A_d \leftarrow 0$,
         $A \leftarrow A + \{A_d\}$.
      ii. $A_d \leftarrow A_d + \log_e(1 + f_{d,t}) * w_t$
3. **Para cada** $A_d \in A$ **faça**
   $A_d \leftarrow A_d/W_d$.
   A_d agora é proporcional ao valor $cosseno(Q, D_d)$.
4. **Para** $1 \leq i \leq r$ **faça**
   (a) Escolha d tal que $A_d = max\{A\}$.
   (b) Procure o endereço do documento d.
   (c) Recupere o documento d e o apresente para o usuário.
   (d) $A \leftarrow A - \{A_d\}$.

Cálculo do Valor do Cosseno

Pontos importantes:

1. W_q e $f_{q,t}$ são ignorados
   W_q é constante e não afeta o *ranking*.
   $f_{q,t} = 1$ para palavras presentes.
   $f_{q,t} = 0$ para palavras ausentes.

2. Muita memória é necessária
   W_d: 4 bytes por documento.
   A_d: 4 bytes por documento.

3. $r \ll N$
   Passo 4 não precisa ordenar completamente A_d, basta extrair os r maiores valores.

Cálculo do Valor do Cosseno

Memória para W_d:

TREC 2: 740.000 documentos

1. Guardar W em disco e acessar seqüencialmente.

2. Guardar $f_{d,t}/W_d$ ao invés de $f_{d,t}$
   - Problema: se o arquivo invertido é mantido comprimido esta opção é muito cara!

Logo, passa-se menos de 2 bits por $f_{d,t}$ para 16 ou 32 bits.

Cálculo do Valor do Cosseno

Memória para W_d:

3. Compromisso entre tudo na memória e tudo no disco

   - Armazenar pesos aproximados na memória que possam dirigir o acesso a disco para os valores exatos.

   - 6 a 8 bits de memória principal são suficientes para manter o número de acessos um pouco maior que r.

   - Suponha que b bits serão usados para codificar uma aproximação de x.

   - Suponha que limites inferiores e superiores para que x sejam conhecidos: $L \leq x \leq U$.

   - Usar um conjunto de *"buckets"* que crescem como uma seqüência geométrica:

$$B = \left(\frac{U+\epsilon}{L} \right)^{2^{-b}}$$

Cálculo do Valor do Cosseno

Memória para W_d:

3. (Continuação)

   - Código c para x:

$$f(x) = \lfloor \log_B(x/L) \rfloor = \left\lfloor \frac{\log x/L}{\log B} \right\rfloor$$

   - O intervalo de valores de x que correspondem a um código c:

$$g(c) \leq x \leq g(c+1)$$

onde,

$$g(c) = L.B^c$$

Cálculo do Valor do Cosseno

Memória para W_d:

3. (Continuação)

Ex.: para aproximar valores no intervalo $10 \leq x \leq 18$ por b = 2 bits
$\epsilon = 0.1$
$B = \left(\frac{18.1}{10} \right)^{2^{-2}} = (1.81)^{1/4} = 1.160$

Logo, o código para 15.3 seria:

$$c = f(15.3) = \left\lfloor \log_{1.16} \frac{15.3}{10} \right\rfloor = \left\lfloor \frac{\log 15.3}{1.16} \right\rfloor = \lfloor 2.865 \rfloor = 2$$

Logo 10, ou $c = 2$, é o código que é expandido para o intervalo $13.456 \leq x < 15.610$.

Cálculo do Valor do Cosseno

Memória para W_d ($L = 10.0$, $U = 18.0$ e $b = 2$):

Código c	Valor do código	Intervalo corresp. para x	$x = g(c+0.5)$
00	0	[10.00, 11.60)	10.77
01	1	[11.60, 13.46)	12.49
10	2	[13.46, 15.61)	14.49
11	3	[15.61, 18.01)	16.81

Cálculo do Valor do Cosseno

Como usar no *ranking*:

1. $\hat{W}_d$ para normalizar:

   $\hat{W}_d = g(c_d)$
   $c_d = f(W_d)$ (código de b bits usado para representar W_d)

   Esta aproximação pode alterar a ordem um pouco.

2. Usar pesos aproximados para recuperar W_d do disco e calcular o *ranking* exato:

   Dado c_d de um documento d e $g(c_d)$ é um limite inferior para W_d e fornece limite superior para o cosseno a ser calculado.

$\Downarrow$

Este limite pode ser usado para eliminar documentos!

Cálculo do Valor do Cosseno

Para determinar os r documentos de topo usando pesos aproximados

1. **Para** $1 \leq d \leq N$ **faça**
   $A \leftarrow A_d/g(c_d)$.
   A_d é o valor do $cosseno$ aproximado; $A_d \geq cosseno(Q, D_d)$.
2. $H \leftarrow r$ valores de topo de A_d.
3. **Para cada** $d \in H$ **faça**
   (a) Leia W_d do disco.
   (b) Armazene o endereço do documento d.
   (c) $A_d \leftarrow A_d * g(c_d)/W_d$.
   Documentos em H têm os valores do $cosseno$ exatos.
4. **Para** $1 \leq d \leq N$ **faça**
   (a) **Se** $A_d \notin H$ **e** $A_d > min\{H\}$ **então**
   i. Leia W_d do disco.
   ii. $A_d \leftarrow A_d * g(c_d)/W_d$.
   iii. **Se** $A_d > min\{H\}$ **então**
   $H \leftarrow H - \{min\{H\}\} + \{A_d\}$,
   Armazene o endereço do documento d.

   H agora contém os r valores do $cosseno$ exatos do topo.

Cálculo do Valor do Cosseno

A_d: valor para documento d

c_d: código aproximado para W_d

$g(c)$: função inversa para a aproximação

H:

- contém as r respostas
- inicializado com r valores aproximados do cosseno usando limite inferior em W_d
- ao final contém r maiores valores exatos

Cálculo do Valor do Cosseno

Passo 2: Set H $\leftarrow$ top r values of A_d

- Verifica se qualquer dos documentos que não estejam nos top r poderiam pertencer ao conjunto:
  - Compara o menor de H com o limite superior de A_d calculado previamente para d.
  - Se $A_d > min\{H\}$ então lê do disco valor exato (que é menor que valor aproximado).
  - Se valor exato ainda é maior do que $min\{H\}$ considere o documento d.

Ex.: TREC com arranjo de pesos aproximados por 8-bit, os top 25 podem ser identificados recuperando apenas 30 valores exatos.

Tratamento dos Acumuladores A_d

Heurísticas baseadas no fato de que entradas do arquivo invertido são processadas na ordem crescente de f_t (ou decrescente de w_t), como nas consultas booleanas.

Heurística: *ranking* será uma aproximação da regra do cosseno.

Regra simples:

- Limite "a priori" no número de acumuladores > 0 permitidos, parando de adicionar novos valores quando limite é atingido.
- Termos de peso alto são considerados e termos de peso baixo são desconsiderados.

Tratamento dos Acumuladores A_d

Regra simples:

E os termos restantes?

1. *Quit*
   Ignora contribuição dos termos não processados (mesmo se eles são para documentos que já tenham acumuladores processados).

2. *Continue*
   Continua o processamento de entradas do arquivo invertido:
   - documentos ainda sem acumuladores são ignorados.
   - documentos para os quais acumuladores tenham sido criados continuam a ter contribuições para o seu cosseno acrescidas.

Tratamento dos Acumuladores A_d

Regra simples:

1. *Quit*
   - Processamento rápido: entradas não processadas são as mais longas.
   - Resultados pobres de *ranking*.

2. *Continue*
   - *Retrieval performance* comparável ao método exato para o cosseno.

3. Experimentos TREC com conjunto de consultas padrão:
   - 5000 acumuladores são suficientes para extrair os 1000 top documentos.

Ordenação

- Como evitar ordenação?

  $N = 1.000.000$

  $N \log N = 20$ milhões

  $r \ll N$

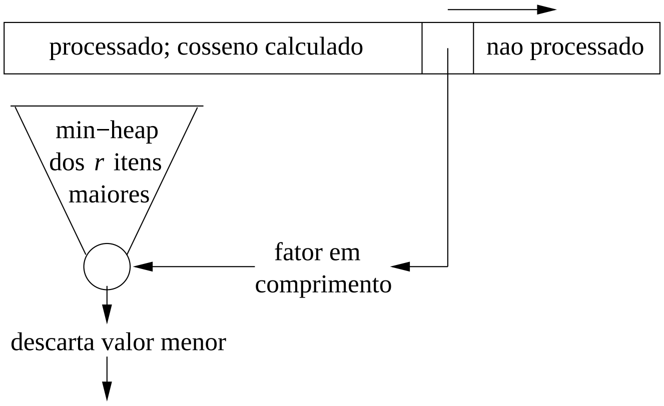
  $\Downarrow$

  Uso de um *heap*

- Construção $\approx 2N$
  - r extrações $\approx 2r \log N$

- Qual o problema de usar um *heap* com N elementos?
  - espaço para armazenar N endereços de documentos.

Ordenação



Solução: armazenar no *heap* r valores (*min-heap*).

Algoritmo:
- inicia com os primeiros r acumuladores (com endereço dos documentos).
  Custo: $2r$

- lê novo acumulador: se maior que o menor valor do *heap* $\Rightarrow$ substituir.

Ordenação

Para selecionar os r valores do cosseno de topo

1. $H \leftarrow \{\}$. H é o *heap*.
2. **Para** $1 \leq d \leq r$ **faça**
   (a) $A_d \leftarrow A_d/W_d$.
   (b) Armazene o endereço do documento d.
   (c) $H \leftarrow H + \{A_d\}$.
3. Construa H no *heap*.
4. **Para** $r+1 \leq d \leq N$ **faça**
   (a) $A_d \leftarrow A_d/W_d$.
   (b) **Se** $A_d > min\{H\}$ **então**
      i. $H \leftarrow H - min\{H\} + \{A_d\}$.
      ii. "Peneire" H.
      iii. Armazene o endereço do documento d.
   H agora contém os r valores do $cosseno$ exatos do topo.
5. **Para** $1 \leq i \leq r$ **faça**
   (a) Selecione d tal que $A_d = max\{H\}$.
   (b) Recupere o documento d e o apresente para o usuário.
   (c) $H \leftarrow H - \{A_d\}$.

Processamento de Consultas

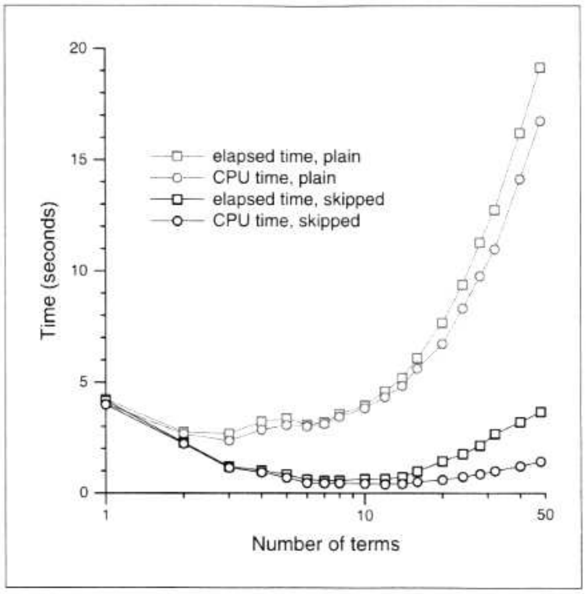


Figure 9.10 Time required by mg for Boolean queries.

Processamento de Consultas

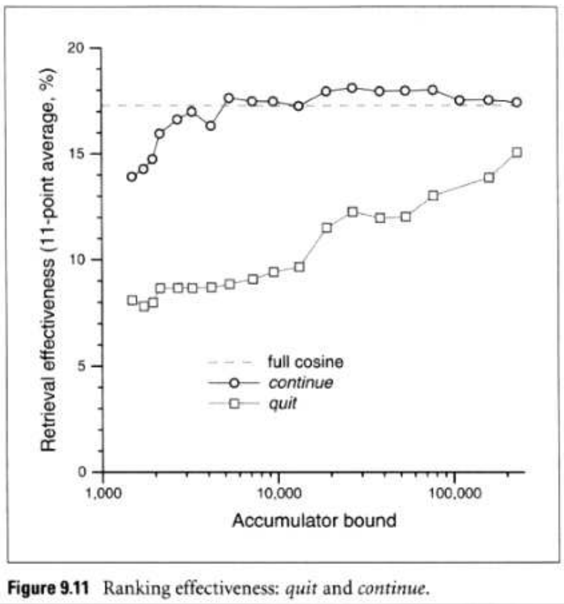


Figure 9.11 Ranking effectiveness: *quit* and *continue*.

Processamento de Consultas

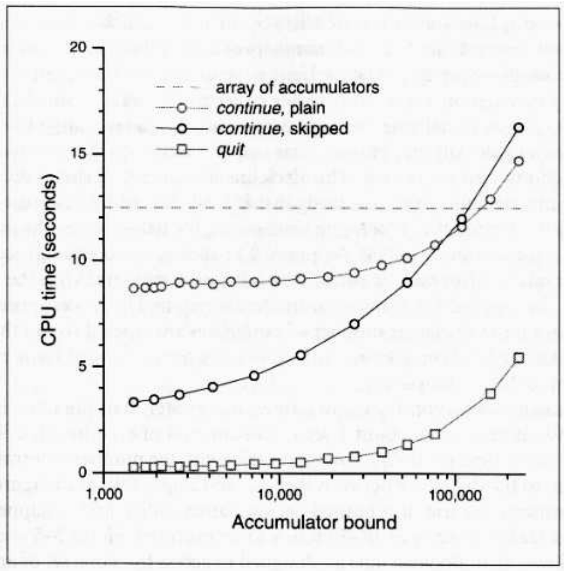


Figure 9.12 Time required by ranked queries.

Medida do Cosseno

Algoritmo Básico:

1. Crie e inicialize uma estrutura de acumuladores (A).
2. **Para cada** termo t na consulta **faça**.
   (a) Recupere lista invertida do termo t do disco.
   (b) **Para cada** par $[d, f_{d,t}]$ na lista invertida t **faça**:
   i. **Se** $A_d \notin A$ **então**
   $A_d = 0$,
   $A = A + \{A_d\}$
   ii. $A_d = A_d + w_{q,t} * w_{d,t}$ $(sim_{q,d,t})$
3. Divida cada acumulador A_d por W_d.
4. Recupere os r maiores valores para os acumuladores e retorne os documentos correspondestes.

Medida do Cosseno

$C_{q,d} = \frac{\sum_t sim_{q,d,t}}{W_d}$

$sim_{q,d,t} = w_{q,t} \times w_{d,t}$
similaridade parcial entre q e d em relação ao termo t

$w_{x,t}$: peso de t na consulta ou no documento

$W_d = \sqrt{\sum_t w_{d,t}^2}$
W_d: comprimento do documento d (pré-computado)

$w_{d,t} = \log_2 f_{d,t} \times \log_2 \frac{N}{f_t}$
Persin, Zobel & Sacks_Davis: JASIS, 1996

$w_{d,t} = (1 + \log_e f_{d,t}) \times \log_e (1 + \frac{N}{f_t})$
WMB, 1999

Medida do Cosseno

Estruturas de Dados para A_d:

- Aloca acumulador para para documento d no início
- Adiciona dinamicamente um acumulador para cada documento d quando $sim_{q,d,t} > 0$

Medida do Cosseno

Custo de avaliar a medida do cosseno pode ser muito alto:

- consultas podem conter muitos termos
- uma medida de similaridade é atribuida a cada documento contendo um dos termos da consulta

Solução:

- técnicas para filtrar documentos durante a ordenação
- acumulador é incrementado somente se a combinação $tf \times idf$ é grande o suficiente para causar alteração na ordem final

Medida do Cosseno

Duas heurísticas adicionais para limitar o número de acumuladores:

1. Estratégia de corte que processa listas invertidas na ordem w_t decrescente, permitindo criar acumuladores enquanto $w_t > max_{t \in Q} \frac{w_t}{3}$
   - Avaliação do restante dos termos não cria mais acumuladores, mas podem contribuir quando já houver A_d criado
2. Processa uma lista invertida apenas se w_t do termo mais todos os outros termos restantes é tal que o documento ranqueado na posição $r+1$ poderia entrar nos r documentos do topo
   - Garante os top r documentos mas não necessariamente na mesma ordem da medida exata do cosseno

Medida do Cosseno

Como avaliar:

1. Pontencialmente caro para grandes coleções
   - tempo CPU
   - largura de banda de disco
   - demanda por memória
2. Soluções
   - limitar o número de documentos candidatos ao ranking
   - ordenar por $f_{d,t}$
   - representação eficiente dos W_d

Objetivo:

- procurar não usar mais memória e tempo de CPU necessários ao processamento da consulta booleana

Medida do Cosseno - Persin

Custos Principais:

- Memória
  - armazena as medidas de similaridade
  - A_d: um acumulador por documento
- Acesso a disco
  - transferência das listas invertidas do disco para a memória
- Tempo CPU
  - para processar informação do índice

Medida do Cosseno - Persin

Limitação do número de candidatos usando Persin, Zobel & Sacks_Davis:

- Lista invertida ordenada por $f_{d,t}$

  $< 5; (1,2), (2,2), (3,5), (4,1), (5,2) >$

  $< 5; (3,5), (1,2), (2,2), (5,2), (4,1) >$

  outra representação:

  $< 5; (5, 1:3), (2, 3:1, 2, 5), (1, 1:4) >$

  permite usar diferenças nos números dos documentos.

Medida do Cosseno - Persin

Como usar lista invertida ordenada por $f_{d,t}$:

- Normalmente idf é previamente calculado e usado para ordenar os termos da consulta de forma que termos mais raros são processados primeiro
- Entretanto, $f_{d,t}$ também é importante. Logo, uma boa solução é processar todas as listas em paralelo, um bloco de cada vez:
  - primeiro bloco da lista invertida de cada termo da consulta é lido e o produto $(1 + \log_e f_{d,t}) \times (\log_e 1 + \frac{N}{f_t})$ é calculado para cada bloco
  - todos os documentos de cada bloco recebem a mesma contribuição no acumulador (mesmo $f_{d,t}$)
  - processar na ordem decrescente de produto (maiores contribuições primeiro)

Medida do Cosseno - Persin

Como usar lista invertida ordenada por $f_{d,t}$:

Vantagens:

1. Mais precisa
   - cada bloco é processado no momento certo
   - se limite no número de acumuladores é usado, processamento pode usar *quit* ou *continue* quando limite é atingido, e agora sabemos que contribuições desprezadas são ainda menores
2. Menos processamento
   - um item de qualquer lista com $f_{d,t} = 1$ dificilmente será processado
3. Menos transferência de disco
   - listas longas raramente são lidas na sua totalidade
4. Ranking melhor do que *quit* e *continue* com lista invertida ordenada pelo número do documento

Medida do Cosseno - Persin

Como usar lista invertida ordenada por $f_{d,t}$:

Desvantagens:

1. Difícil processar consultas booleanas

Medida do Cosseno - Persin

Como filtar documentos:

Transição gradual da inclusão para a omissão de documentos

1. termos da consulta são ordenados por idf decrescente (como no algoritmo básico)
2. antes de processar cada termo t calcula dois parâmetros: S_{ins} e S_{add}, $S_{add} \leq S_{ins}$
   - S_{add} permite ignorar itens da lista invertida
   - S_{ins} permite ignorar alguns documentos

- Propostas anteriores decidiam processar ou descartar listas invertidas inteiras
- Persin permite decidir processar ou rejeitar documentos individuais

Medida do Cosseno - Persin

Algoritmo:

1. Crie e inicialize uma estrutura de acumuladores (A).
2. Ordene termos da consulta por idf decrescente.
3. $S_{max} = 0$
4. **Para cada** termo t na consulta **faça**.
   (a) Compute os valores para os filtros f_{ins} e f_{add}.
   (b) Recupera lista invertida do disco.
   (c) **Para cada** par $[d, f_{d,t}]$ na lista invertida **faça**.
   i. **Se** $fd,t > f_{ins}$ **então**
   Crie A_d se necessário
   $A_d = A_d + w_{q,t}.w_{d,t}$ $(sim_{q,d,t})$
   ii. **Se não**, **Se** $fd,t > f_{add}$ e A_d existe **então**
   $A_d = A_d + w_{q,t}.w_{d,t}$ $(sim_{q,d,t})$
   iii. **Se não**, Desvie para o próximo termo da consulta
   iv. $S_{max} = max(S_{max}, A_d)$.
5. Divida cada acumulador A_d por W_d.
6. Recupere os r maiores valores para os acumuladores e retorne os documentos correspondestes.

Medida do Cosseno - Persin

Como obter f_{ins} e f_{add}

- São função da similaridade parcial acumulada do documento mais relevante no momento (S_{max})

  $S_{ins} = C_{ins} \times S_{max}$, $\qquad S_{add} = C_{add} \times S_{max}$
  onde, $0 \leq C_{add} \leq C_{ins}$, não constantes

- $sim_{q,d,t} \leq w_{q,t} \times w_{d,t} = f_{d,t} \times idf \times f_{q,t} \times idf$

  ou,
  $$\frac{sim_{q,d,t}}{f_{q,t} \times idf^2} \leq f_{d,t}$$

  (processa elemento como condição sobre $f_{d,t}$)

  logo, $f_{ins} = \dfrac{C_{ins} \times S_{max}}{f_{q,t} \times idf^2}$, $\quad f_{add} = \dfrac{C_{add} \times S_{max}}{f_{q,t} \times idf^2}$

  (Valores de f_{ins} e f_{add} são constantes durante o processamento de uma lista invertida)

Medida do Cosseno - Persin

Observações sobre f_{ins} e f_{add}

- permitem transição suave da aceitação para rejeição de elementos nas listas invertidas
- fica progressivamente mais difícil adicionar ou atualizar os acumuladores
- para primeiros termos o valor de S_{max} é pequeno e o valor do idf é alto
- na medida que S_{max} cresce e idf diminui f_{ins} e f_{add} crescem até que, no limite, todos $f_{d,t} < f_{add}$ (e processar a lista a partir deste ponto não promove mudanças significativas nos acumuladores)

Medida do Cosseno - Persin

Nova versão:

1. Cálculo de f_{ins} e f_{add}, na versão original, ocorre uma vez para cada termo t e permanece igual durante o processamento da lista invertida do termo t
2. Na nova versão, o cálculo de f_{ins} e f_{add} ocorre toda vez que S_{max} é alterado.

Medida do Cosseno - Persin

Algoritmo nova versão:

1. Crie e inicialize uma estrutura de acumuladores (A).
2. $S_{max} = 0$
3. **Para cada** termo t na consulta **faça**.
   (a) Compute os valores para os filtros f_{ins} e f_{add}.
   (b) Recupere primeiro bloco de cada lista do disco.
   (c) Calcule $sim_{q,d_s,t}$ e Insere $sim_{q,d_s,t}$ em H.
4. **Enquanto** houver blocos em H **faça**.
   (a) Extrai bloco de H e insere próximo bloco da lista.
   (b) Calcule $sim_{q,d_s,t}$.
   i. **Se** $fd, t > f_{ins}$ **então**
      Crie A_d se necessário
      $A_d = A_d + w_{q,t}.w_{d,t}$ $(sim_{q,d,t})$
   ii. **Se não**, **Se** $fd, t > f_{add}$ e A_d existe **então**
      $A_d = A_d + w_{q,t}.w_{d,t}$ $(sim_{q,d,t})$
   iii. **Se** $A_d > S_{max}$ **então**
      $S_{max} = max(S_{max}, A_d)$.
      Compute os valores para os filtros f_{ins} e f_{add}.
5. Divida cada acumulador A_d por W_d.
6. Recupere os r maiores valores para os acumuladores e retorne os documentos correspondentes.

Medida do Cosseno - Persin

Exemplo:

$$\begin{array}{lll} t_1 & \rightarrow & < 4; (4, 2 : 1, 3), (2, 1 : 2), (1, 1 : 5) > \\ t_2 & \rightarrow & < 1; (2 : 1, 5) > \\ t_3 & \rightarrow & < 2; (8, 1 : 7), (4, 1 : 6) > \end{array}$$

$$w_{d,t} = tf \times idf = \log_2 f_{d,t} \times \log_2 \frac{N}{f_t}$$

$N = 8$

$idf_{t_1} = \log_2 \frac{8}{4} = 1$
$idf_{t_2} = \log_2 \frac{8}{1} = 3$
$idf_{t_3} = \log_2 \frac{8}{2} = 2$

$tf \times idf$:

$Bloco_{1,t_1} = \log_2 4 \times 1 = 2$ $\quad$ $Bloco_{2,t_1} = \log_2 2 \times 1 = 1$
$Bloco_{1,t_2} = \log_2 2 \times 3 = 3$ $\quad$ $Bloco_{2,t_3} = \log_2 4 \times 2 = 4$
$Bloco_{1,t_3} = \log_2 8 \times 2 = 6$ $\quad$ $Bloco_{3,t_1} = \log_2 1 \times 2 = 0$

Medida do Cosseno - Persin

Exemplo:

Ordem de processamento dos blocos:

1. Persin Original: (t_2, t_3 e t_1)
   (2,1:5), (8,1:7), (4,1:6), (4,2:1,3), (2,1:2), (1,1:5)
2. Nova versão:
   (8,1:7), (4,1:6), (2,1:5), (4,2:1,3), (2,1:5)

Compressão de Texto

- Compressão de texto significa encontrar formas para representar o texto em poucos *bits* ou *bytes*.
- Ponto importante em um ambiente de RI.
- Escolha atrativa, se não obrigatória.
- Ganho: menos espaço de armazenamento e menos tempo para ser transmitido.
- O preço a pagar: tempo para codificar e decodificar o texto.

Compressão de Texto e Sistemas de RI

- O maior obstáculo para armazenar texto na forma comprimida é a necessidade dos sistemas de RI em acessar o texto randômicamente .
- Necessário decodificar o texto inteiro do início até que a palavra desejada seja alcançada.
- Tópicos em métodos de compressão de texto que são desejáveis para serem usados em uma ambiente de recuperação de informação:
  - acesso randômico barato para palavras dentro do texto comprimido
  - permitir descompressão em pontos intermediários no texto comprimido.
  - buscar diretamente sobre o texto comprimido comprimindo o padrão.

Métodos de Compressão

Características importantes:

- Economia de espaço (taxa de compressão acima de 25%).
- Velocidade de compressão e descompressão. Em algumas situações a velocidade de descompressão é mais importante que a velocidade de compressão.
- Capacidade de casamento de padrão comprimido:
  - definido como a tarefa de realizar casamento de padrão em um texto comprimido sem descomprimi-lo.
  - busca seqüencial pode ser acelerada comprimindo-se a chave de busca ao invés de decodificar o texto comprimido.
  - como uma conseqüência disso, muito menos texto tem que ser examinado.

Fases do Processo de Compressão

Modelagem

A probabilidade é atribuída para cada próximo símbolo.

Codificação

Próximo símbolo é representado pelo seu código relacionado à distribuição de probabilidade obtida pela fase de modelagem.

Definições

1. Um *símbolo* aqui é usualmente um caractere, uma palavra ou um número fixo de caracteres.

2. O conjunto de todos os possíveis símbolos no texto é chamado *alfabeto*.

3. *Taxa de compressão* é o tamanho do arquivo comprimido como uma porcentagem do arquivo descomprimido.

4. Um *método de compressão de ordem zero* opera símbolo por símbolo com nenhum contexto usado para predizer o próximo símbolo, e assim cada símbolo de entrada é tratado como um símbolo independente.

Abordagens para Compressão de Texto

Métodos Estatísticos

Uma probabilidade é obtida para cada símbolo e o código obtido é baseado na probabilidade.

Métodos de Dicionário

Substitui uma seqüência de símbolos por um ponteiro que aponta para uma ocorrência anterior da seqüência.

Métodos Estatísticos

- Contam com a geração de boas estimativas de probabilidade para cada símbolo.

- Quanto mais precisas as estimativas são, melhor é a compressão obtida.

- A tarefa de estimar a probabilidade sobre cada próximo símbolo é chamada de *modelagem*.

- Uma vez que estas probabilidades estão disponíveis, os símbolos são convertidos em dígitos binários, um processo chamado de *codificação*.

Métodos Estatísticos

- Na prática, ambos o codificador e decodificador usam o mesmo modelo.

- O decodificador interpreta a saída do codificador (com referência ao mesmo modelo) para encontrar o símbolo original.

Modelagem e codificação são atividades distintas

Métodos Estatísticos

- Tarefa de modelagem: a probabilidade é estimada para cada símbolo.
- Tarefa de codificação: baseada nesta probabilidade, um código é atribuído para cada símbolo por vez.
- Códigos menores atribuídos para os símbolos mais prováveis.

Uso do Modelo na Prática

Codificador

Gera códigos usando o modelo.

Decodificador

Interpreta os códigos com referência ao modelo.

Codificador e decodificador usam o mesmo modelo

Uso do Modelo na Prática

- Codificação: representação de um símbolo com a probabilidade p em aproximadamente $\log \frac{1}{p}$ bits.
- Codificador: produz um conjunto de bits ou bytes do conjunto de probabilidades que governa a escolha do próximo símbolo.
- Decodificador: decodifica o símbolo usando o mesmo conjunto de probabilidades usado pelo codificador.

Métodos de Dicionário

- Usa representação de ponteiros para codificar referências para entradas no dicionário.
- Dicionário é uma lista de símbolos (geralmente chamado frases) que são esperados ocorrerem freqüentemente.
- Ponteiros para entradas de dicionários necessitam menos espaço que a frase que eles substituem, obtendo assim a compressão.

  Ex.: jan, feb,$\cdots$, dec são codificados como 1, 2, $\cdots$, 12
- A distinção entre modelagem e codificação não existe em métodos de dicionários.
- Nenhuma probabilidade explícita associada as frases.
- Métodos de dicionários mais conhecidos: família Ziv-Lempel.

Métodos de Dicionário

- Alcança compressão trocando grupos de símbolos consecutivos (ou frases) por um ponteiro para uma entrada no dicionário.
- Decisão central: seleção das entradas no dicionário.
- A escolha de frases pode ser feita através de esquemas estáticos, semi-adaptativos ou adaptativos.
- Família Ziv-Lempel de esquema de dicionário:
  - troca strings de caracteres com uma referência para uma ocorrência anterior da string.
  - se um ponteiro para uma ocorrência de uma string é armazenado em menos bits que a string que ele substituiu então a compressão é alcançada.

Métodos de Dicionário

Desvantagens sobre os Métodos Estatísticos

- Não é possível iniciar a decodificação no meio do arquivo comprimido.
- Acesso direto para uma posição no texto comprimido não é possível.

$$\Downarrow$$

Métodos de dicionário não são adequados para sistemas de RI

Relacionamento entre Probabilidades e Códigos

- Claude Shannon (1948) em seu teorema:
  Em um esquema de codificação ótimo, um símbolo que é esperado para ocorrer com probabilidade p deve ser atribuído um código de tamanho $\log_2 \frac{1}{p}$ bits.
- O número de bits no qual o símbolo é melhor codificado representa o conteúdo informacional do símbolo.
- A quantidade média de informação por símbolo sobre o alfabeto todo é chamado de *entropia* da distribuição de probabilidade, que é dada por:

$$E = \sum p_i \log_2 \frac{1}{p_i}$$

  Medida em bits por símbolo e representa um limite que nunca pode ser batido em um dado modelo.

Entropia

- Quantidade de ordem (ou redundância) em uma mensagem.
  - pequena: alta ordem.
  - grande: alta desordem.
- Idealmente, o tamanho da codificação deve ser igual a entropia da mensagem .

Modelos de Compressão

- *Estático*: mesmo modelo para todos os textos de entrada.
- *Semi-estático*: diferentes modelos para cada texto.
  - aprender distribuição de prob. no primeiro passo.
  - comprimir texto no segundo passo usando código fixo derivado da distribuição aprendida no primeiro passo.
  - enviar modelo para o decodificador antes da mensagem codificada.
- *Adaptativo*: inicia com nenhuma informação sobre o texto e aprende sobre sua distribuição de probabilidade.
  - uma passagem sobre o texto.
  - não necessita enviar o modelo em separado para o decodificador.
  - código usado para um símbolo particular é baseado no texto já codificado.
  - maior desvantagem: descompressão tem que iniciar do começo e nenhum acesso randômico é possível.

Modelo Baseado em Palavra

- Considera palavras ao invés de caracteres como símbolos.
  Strings de caracteres no conjunto {A..Z, a..z} separadas por caracteres que não estão no conjunto {A..Z, a..z}.
- Melhores taxas de compressão são alcançadas (palavras carregam muito do significado em linguagem natural).
- Palavras são os átomos sobre os quais a maioria dos sistemas de recuperação de informação são construídos.
- Freqüências de palavras também são úteis para responder consultas envolvendo combinação de palavras.

Escolha do Método de Codificação

- Método de codificação mais conhecido: Huffman (1952).
  - usualmente usado com modelagem estática ou semi-estática.
  - pode ser usado com modelagem adaptativa.

Escolha do Método de Codificação

Codificação Aritmética:

- Computa o código incrementalmente, um símbolo por vez.
- Natureza incremental não permite iniciar a decodificação no meio do arquivo comprimido.
- Esta característica faz este método inadequado para ser usado em um ambiente de recuperação de informação.

Algoritmo da Árvore de Huffman

1. Liste todos os possíveis símbolos com suas probabilidades.

2. Considere os dois símbolos com as menores probabilidades.

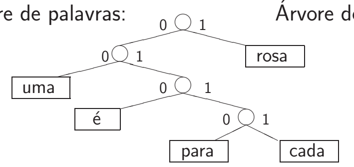
3. Substitua estes dois por um único conjunto, cuja probabilidade é a soma das probabilidades individuais.

4. Repita até que a lista contenha somente um membro.

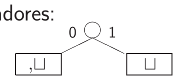
Esquema de Compressão

Alfabetos Separados:

Árvore de palavras:



Árvore de separadores:



Texto original: para cada rosa, uma rosa é uma rosa

Texto comprimido: 0110 1 0111 1 1 0 00 1 1 1 010 1 00 1 1

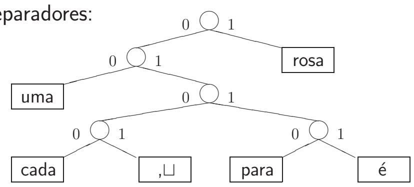
- Texto pode ser reconstruído de qualquer posição.

- Taxa de compressão: 25% a 30%.

Esquema de Compressão

Alfabeto Único:

Árvore de palavras + separadores:



Texto original: para cada rosa, uma rosa é uma rosa

Texto comprimido: 0110 0100 1 0101 00 1 0111 00 1

Arquivos	Palavras (p)	Espaços simples(s)	s/p
AP	38.977.670	30.020.803	0,77
DOE	28.505.125	22.001.366	0,77
FR	34.455.982	25.506.763	0,74
WSJ	42.710.250	33.909.610	0,79
ZIFF	39.675.248	27.225.864	0,69

Árvore de Huffman Canônica

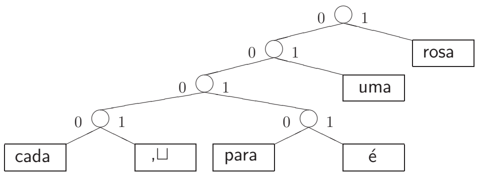
- O número de árvores de Huffman para uma dada distribuição de probabilidade é grande.

- Trocando sub-árvores da esquerda e da direita de qualquer nodo interno resulta em uma árvore diferente na estrutura, mas tamanho do código médio não é afetado.

- Ao invés de usar qualquer tipo de árvore é adotado uma *árvore canônica*, a qual impõe uma ordem particular para a codificação de bits.

- Uma árvore de Huffman é canônica quando a altura da sub-árvore esquerda de qualquer nodo nunca é menor que a sub-árvore direita, e todas as folhas estão em ordem crescente de probabilidades da esquerda para a direita.

Árvore de Huffman Canônica



Símbolo	Prob.	Código Antigo	Código Can.
cada	$1/9$	0100	0000
,␣	$1/9$	0101	0001
para	$1/9$	0110	0010
é	$1/9$	0111	0011
uma	$2/9$	00	01
rosa	$3/9$	1	1

Árvore de Huffman Canônica

- Folha mais profunda na posição mais a esquerda (elemento com menor probabilidade) conterá somente zeros.
- Os códigos seguintes estarão em ordem crescente dentro de cada nível.
- A cada mudança de nível desloca-se um bit para a esquerda no contador.

Código de Huffman Canônico

- Seqüência ordenada S de pares (x_i, y_i), $1 \leq i \leq \ell$, onde x_i representa o número de elementos no nível i e y_i representa o valor numérico do primeiro código no nível i.
- Para o exemplo, a seqüência ordenada é: $S = \langle(1,1),(1,1),(0,\infty),(4,0)\rangle$
- O par $(4,0)$ em S corresponde ao quarto nível e indica que existem quatro nodos neste nível e que o nodo mais a esquerda é atribuído código com valor 0 (código 0000 para o quarto nível).
- Conjuntos de códigos tendo o mesmo tamanho são representações binárias de inteiros consecutivos.
- Interpretados como inteiros, os códigos de 4 bits são 0, 1, 2 e 3, código de 2 bits é 1 e código de 1 bit também é 1.

Código de Huffman Canônico

Como decodificar:

- Se o primeiro caractere é 1, um código foi identificado e o símbolo correspondente pode ser decodificado.
- Se este valor é 0, um segundo bit é adicionado e os dois bits são novamente interpretados como um inteiro e usado para indexar a tabela e identificar o símbolo correspondente.
- Uma vez que nós lemos "00" nós sabemos que o código tem quatro bits e portanto nós podemos ler mais dois e usá-los como um índice dentro da tabela.
- Este fato pode ser explorado para possibilitar eficiente codificação e decodificação com pequeno *overhead*. Além disso, muito menos memória é requerida, o que é especialmente importante para grandes vocabulários.

Referências

- Ziviani, N. and Ribeiro-Neto, B. "Text Operations". In: Baeza-Yates, R. and Ribeiro-Neto, B. *Modern Information Retrieval*, Chapter 7, Addison-Wesley, 1999, 28 pages.

- Moura, E. S., Navarro, G., Ziviani, N. and Baeza-Yates, R. "Fast Searching on Compressed Text Allowing Errors", *21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Australia, August 1998.

Compressão de Índices

- Arquivos invertidos são amplamente usados para indexar grandes arquivos de texto.

- O tamanho de um arquivo invertido pode ser reduzido usando compressão nas listas invertidas.

- Números dos documentos em ordem ascendente: seqüência de *d-gaps* (a diferença entre d_{k+1} e d_k) entre os números de documentos.

- Processamento é seqüencial desde início da lista: números originais dos documentos podem ser obtidos através do cálculo da soma dos *d-gaps*.

  Ex.: Suponha que um termo ocorra em 8 documentos:

  $<$8; 3, 5, 20, 21, 23, 76, 77, 78$>$

  Calculando-se os *gaps* entre os documentos:

  $<$8; 3, 2, 15, 1, 2, 53, 1, 1$>$

Compressão de Índices

- Ambos os métodos de representação das listas necessitam de $\lceil \log N \rceil$ bits por apontador, onde N é o número de documentos da coleção.

- Entretanto, *d-gaps* podem ser codificados em muito menos do que $\lceil \log N \rceil$ bits.

- Os *gaps* são pequenos para palavras freqüentes e grandes para palavras infreqüentes, a compressão pode ser obtida codificando-se pequenos valores com códigos menores
  - Por exemplo, o código unário, no qual um inteiro x é codificado como $x - 1$ bits 1 e seguido de um bit 0.

    Ex.: O código unário para o inteiro 3 é 110.

Métodos para Compressão de Índices

Métodos locais

Modelo de compressão para cada lista associada a um termo é ajustado de acordo com algum parâmetro local. Por exemplo, a freqüência do termo.

Métodos globais

Listas são comprimidas usando um modelo comum de compressão

- Não-parametrizados: Unário, Elias-δ e Elias-γ.

- Parametrizados: Golomb
  - envolve algum parâmetro que pode ser ajustado à distribuição dos *d-gaps*
  - exige duas passadas no texto.

Métodos para Compressão de Índices

Método	Referência
Métodos Globais	
Não parametrizados	
Unário	
Binário	
γ	Elias (1975); Bentley and Yao (1976)
δ	Elias (1975); Bentley and Yao (1976)
Parametrizados	
Bernoulli	Golomb(1966); Gallager and Van Voorhis (1975)
Observed frequency	
Métodos Locais	
Bernoulli	Witten, Bell, and Nevill (1992); Bookstein, Klein, and Raita (1992)
Skewed Bernoulli	Teuhola (1978); Moffat and Zobel (1992)
Hyperbolic	Schuegraf (1976)
Observed frequency	
Batched frequency	Moffat and Zobel (1992)
Interpolative	Moffat and Zobel (1996)

Codificação Unária

Para a codificação de um inteiro x usando o método unário repete-se $(x-1)$ bits com valor 1, em seguida acrescenta-se um bit com o valor 0.

1	0
2	10
3	110
4	1110
5	11110
6	111110
7	1111110
8	11111110
9	111111110
10	1111111110

Codificação de Elias-γ

O número é dividido em duas partes:

- Primeira parte:
  - código unário de $1 + \lfloor \log x \rfloor$ bits.
- Segunda parte:
  - tamanho do código em bits: $\lfloor \log x \rfloor$ bits.
  - codificando o número $x - 2^{\lfloor \log x \rfloor}$ em binário.

Ex.: $x = 5$

$1 + \lfloor \log 5 \rfloor = 3$ (110 em unário)

$5 - 2^{\lfloor \log 5 \rfloor} = 5 - 2^2 = 1$ (01 em binário de tamanho 2)

portanto o código de 5 é 11001.

Codificação de Elias-δ

O número é dividido em duas partes:

- Primeira parte:
  - código Elias-γ de $1 + \lfloor \log x \rfloor$ bits.
- Segunda parte:
  - tamanho do código em bits: $\lfloor \log x \rfloor$ bits.
  - codificando o número $x - 2^{\lfloor \log x \rfloor}$ em binário.

Ex.: $x = 5$

$1 + \lfloor \log 5 \rfloor = 3$ (101 em Elias-γ)

$5 - 2^{\lfloor \log 5 \rfloor} = 5 - 2^2 = 1$ (01 em binário de tamanho 2)

portanto o código de 5 é 10101.

Codificação de Golomb

Solomon Golomb, USC (1966):

- Definir o valor do parâmetro b.
- $(q+1)$ em unário, onde $q = \lfloor (x-1)/b \rfloor$.
- $r = (x-1) - q \times b$ codificado em binário com $\lfloor \log b \rfloor$ ou $\lceil \log b \rceil$ bits.
- Valores de r com $\lfloor \log b \rfloor$ bits começam com 0.
- Valores de r com $\lceil \log b \rceil$ bits começam com 1.

Ex.: $b = 3$ e $x = 7$

$q = \lfloor (7-1)/3 \rfloor = 2$. Então $(q+1)$ em unário é 110

$r = (7-1) - 2 \times 3 = 0$

representação de 7 é 1100.

Exemplos de Códigos para Inteiros

Gap x	Unary	Elias-γ	Elias-δ	Golomb ($b=3$)
1	0	0	0	00
2	10	100	1000	010
3	110	101	1001	011
4	1110	11000	10100	100
5	11110	11001	10101	1010
6	111110	11010	10110	1011
7	1111110	11011	10111	1100
8	11111110	1110000	11000000	11010
9	111111110	1110001	11000001	11011
10	1111111110	1110010	11000010	11100

- O código Elias-δ para um inteiro arbitrário x requer $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits.
- Para pequenos valores de x os códigos de Elias-γ são menores que os de Elias-δ. Entretanto, no limite, como x torna-se grande, a situação é revertida.