

## **Managing Gigabytes Overheads**

Title Page

Introduction

IR Concepts

Text Compression

Indexing

Querying

Index Construction

Image Compression

Textual Images

Mixed Text and Images

Implementation

The Information Explosion

Guide to the mg System

Guide to the NZDL

# Managing Gigabytes

Ian H. Witten  
Alistair Moffat  
Timothy C. Bell

# Introduction

## **What's it all about?**

Managing Information

Providing Information Services

## **How much information?**

Gigabytes...Terabytes...Petabytes

## **What type of information?**

All types...

## **How?**

Compression and Decompression

Indexing and Retrieval

Internet Services

## **What has changed from the 60's?**

Storage capacities

Workstations and their computational power

Internetworking

Freely available information servers

# Giga/Tera/Peta

Million	Mega
Billion	Giga
Trillion	Tera
Quadrillion	Peta

If 1 megabyte = 1 book, Then:

Gigabyte = 1000 megabytes = 1000 books

Terabyte = 1000 gigabytes = 1 million books

Petabyte = 1000 terabytes = 1 billion books

*Note: the 1000's above are really 1024*

How much does a Terabyte cost today?

The mg book has 1.4m characters

*350kb if compressed to two bits per character*

*525kb if compressed to three bits per character*

*700kb if compressed to four bits per character*

*875kb if compressed to five bits per character*

*1.05 mb if compressed to six bits per character*

Images?

# IR Concepts

# IR Concepts

## Concordances

*Inversion by a Compiler (human)*  
*(Compiler & Computer/Computer)*

*Constructed on demand*

*British National Corpus ([info.ox.ac.uk/bnc/](http://info.ox.ac.uk/bnc/))*

*contains over 4000 texts and 100 million words*

*Project Gutenberg (<http://promo.net/pg/>)*

*contains over 5000 texts*

## Full-text retrieval (FTR)

*Inverted Index*

## Catalogs

*Topics and Maps*

## Bush's Memex

*As We May Think*

*Trails of association? FTR? (comments p.6)*

## Compression techniques

*Problem one: space required to store documents*

*Speed of transmission*

*Lossy and loss-less methods*

## Indexing techniques

*Problem two: time required to search/retrieve documents*

*Stop lists (Index: 30% uncompressed, 4% compressed)*

*Granularity*

*Boolean and ranked queries*

*"Context" and "Associated" (comments p.11)*

## Images

*Diagrams and photographs*

*Document images*

*Textual images and optical character recognition (OCR)*

*Audio and Video images*

# IR Concepts (cont)

## Comparisons

*Disk access time is critical factor*

## Document databases/collections (DL)

*What Color was George Washington's White Horse?*

## Database versus IR

mg system

## Where is the context?

*In the text, in the concept map, or in the head?*

## Plagiarism

*A Case of Academic Plagiarism*

*The case of C.V. Papadopoulos*

# Text Compression

# Concepts

## General

*Text compression is lossless*

*Compression occurs if the coded bit-stream is shorter than the input bit stream*

*Logic: processor speed is increasing faster than disk speed and capacity ==> compression is economical...*

*Modeling: estimating probabilities of symbols in the input text*

*Coding: converting the symbol into the codeword*

*Decoding: converting the codeword into the symbol*

*Coding is well-understood, modeling is more of an art*

*Better modeling ==> higher compression*

*<Example of the general idea>*

## Static models versus Adaptive models

### *Static models*

*Huffman coding (early 50's)*

*fast, require moderate memory*

*English text: 5 bits/character*

### *Adaptive models*

*Ziv-Lempel compression (late 70's)*

*very fast, does not have large memory requirements*

*English text: <4 bits/character*

### *Arithmetic coding*

*Prediction by partial matching (early 80's)*

*slower, requires larger memory*

*English text: just over 2 bits/character*

### *Block-sorting (1994)*

*Burrows-Wheeler transform*

*Adaptive only within block*

*Move-to-front coder*

*English text: just over 2 bits/character*

# Concepts (cont)

## Symbolwise versus Dictionary methods

*Symbolwise methods (statistical methods)*

*Estimate probabilities of symbols*

*Code one symbol at a time*

*Codewords shorter for more likely symbols, longer for less likely symbols*

*Huffman, arithmetic coding and Block-sorting*

*Dictionary methods*

*Words and "phrases" replaced with index to "dictionary"*

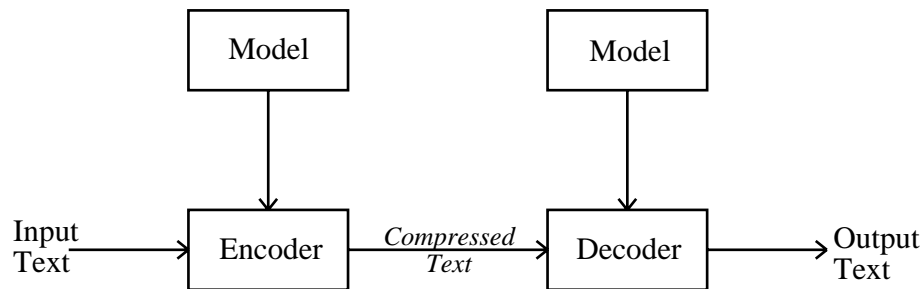
*Ziv-Lempel*

*replace strings of characters with reference to previous occurrence  
compression is achieved if #ref bits < string it replaces*

# Models

Alphabet: set of all possible symbols

Models provide the PD of symbols in the alphabet



## Information content

*The number of bits in which a symbol,  $s$ , should be encoded*

$$I(s) = -\log \Pr[s] \quad \text{Remember: } -\log(x/y) = \log(y/x)$$

## Entropy (of the PD over the alphabet)

*The average amount of information content per symbol*

*(average number of bits per symbol)*

$$H = \sum_s \Pr[s] * I(s) = \sum_s -\Pr[s] * \log \Pr[s]$$

*Shannon's source coding theorem:*

*$H$  provides a lower bound, measured in bits/symbol, that can be achieved by any coding method (assuming independent symbols and the PD is correct)*

## What's important?

*Good predictions of the probability for each symbol*

*Poor predictions give low probability ==> high entropy*

*Good predictions give high probability ==> low entropy*

*Note: A symbol with zero probability can not be coded, so all symbols must be given a non-zero probability*

*Note: The encoder can not use look-ahead ... the encoder can see the look-ahead symbol but the decoder can not...*

# Models (cont)

## Finite-context models of order $m$

*Models that use the previous  $m$  symbols to help predict the next symbol*

## Other modeling approaches

*FSA in which each state has a PD for the alphabet*

*FSA's must remain synchronized*

*Grammar models in which production rules are used to predict PD*

## Types of modeling

### *Static modeling*

*Use same PD for all input texts*

### *Semi-static modeling*

*Generate a new PD for each input text*

*Adv: better than static*

*Disadv: requires two passes (generate PD, encode)  
must transmit model to decoder for each input text  
impractical for some applications*

### *Adaptive modeling*

*Start with "estimated" PD and modify it during encoding based upon symbol frequencies in the input text*

### *Zero-frequency problem*

*Symbols with probability of zero can not be coded*

*Solutions:*

*Add one to total count and divide evenly among symbols that have not appeared*

*Start with a count of one for each symbol in the alphabet*

*ZFP is most acute:*

*At beginning of text*

*For short files*

*When many contexts are used*

*Adv: good for general purpose compression utilities*

*Disadv: not suitable for random access to files (FTR)*

# Models (cont)

## Higher order models

*Zero-order model*

*Each symbol is independent*

*No context*

*First-order model*

*Uses previous symbol to predict probability of current symbol*

*Uses "context" to "condition" PD*

*m-order model*

*Uses the context of the previous m symbols to condition PD*

## Adapting the model's structure

*Add more detail to an area of the model that is heavily used*

*Finite-context models*

*Add higher order contexts*

*FSA models*

*Add more states and transitions*

# Coding

Determining the output representation of a symbol based upon a PD supplied by a model

*Short codewords for likely symbols*

*Longer codewords for unlikely symbols*

*Goal: low entropy*

*Goal: fast coder*

*Tradeoff: speed versus compression performance*

## Huffman Coding

*Symbolwise coder*

*Example:*

Symbol	Codeword (by tree)	Frequency	Codeword (by hand)
a		5	
b		5	
c		10	
d		20	
e		30	
f		20	
g		10	

*Build codewords by hand*

*Build codewords by tree*

*Code a string*

*Bitwise decoding*

*Prefix code (prefix-free code)*

*No codeword is the prefix of another codeword*

*(If it were you would have ambiguity)*

*General algorithm for HC*

*See figure 2.7 p.34*

# Huffman Coding (cont)

HC is fast if PD is static

Adaptive HC is possible...

*by dynamically modifying the tree but arithmetic coding is usually preferable (less memory, equivalent speed, better compression)*

HC shines for FTR

*Good compression, fast, ease of random access, word-based*

## Canonical Huffman Coding

Same codeword lengths as HC

Imposes a particular choice on the code bits

Decodes very efficiently

See example in Table 2.2, p. 35

*Alphabet consists of words in the input text*

*Compression based upon the CHC of normal output of*

*Full-Text indexer: Word, Freq, {ref, ref, ...}*

*A lexical sort has been done on Word and then Codeword*

Important feature of CHC

*When codewords are sorted in lexical order, they are also in order from longest to shortest codeword*

Notes:

*Codewords are not stored!*

*Eventually, the symbols get compressed out of existence!*

# Canonical Huffman Coding (cont)

## Encoding

### *Inputs:*

*Length of codeword for each symbol*

*How many entries each symbol is away from the first symbol of that length*

*Codeword for first symbol of that length*

### *Output:*

*Codeword = Codeword for first symbol of that length  
+ distance to this codeword*

## Decoding

### *Inputs:*

*List of symbols ordered by lexical order of codewords*

*Array storing first codeword of each length and its index into the list of symbols*

### *Output:*

*Determine length by comparing first  $n$  bits*

*Subtract first  $n$  bit codeword value from input  $n$  bit value*

*Use result to index list of symbols for this length*

# Canonical Huffman Coding (cont)

## Assigning a CHC

*First:*

*For each symbol in the alphabet:*

*Use Huffman's algorithm to calculate the desired length of the corresponding codeword (fig 2.13, p.46)*

*Second:*

*Count the number of codewords of each length (step 1, fig. 2.9)*

*Set the first codeword to be generated for each length*

*(note: a code of length  $n$  uses some bits in the code of length  $n-1$  ... firstcode starts the code for  $n-1$  after this) (step 2, fig. 2.9)*

*Set the codewords for each symbol (steps 3&4, fig. 2.9)*

## Encoding a CHC

*Data structures:*

<u>Symbol</u>	<u>Length</u>	<u>Delta</u>	<u>First Codeword</u>
---------------	---------------	--------------	-----------------------

*Find Symbol, use corresponding Length to index First Codeword, add Delta to First Codeword, output rightmost Length bits*

## Decoding a CHC

*Data structures:*

<u>Symbol</u>	<u>First Codeword</u>	<u>Index into Symbol</u>
---------------	-----------------------	--------------------------

Sorted by  
Codeword  
Length  
(Lexical order)

*Determine length by comparing first  $l$  bits to firstcode[ $l$ ]*

*Subtract first  $l$  bit codeword value from input  $l$  bit value*

*Use result to index list of symbols for this length*

(Figure 2.10, p. 40)

# Arithmetic Coding

## Advantages versus Huffman

*Can code arbitrarily close to entropy  
(based upon arithmetic precision)  
Advantages are most apparent when one symbol  
has a high probability (images)  
Less primary memory is required  
Particularly suited to adaptive models where:  
High probabilities are occurring  
Many different PDs are used*

## Disadvantages versus Huffman

*Slower (especially in static or semi-static applications)  
Difficult to start decoding in the middle of a  
compressed stream  
Not appropriate for FTR due to the above two disadvantages*

## For large collections of text and images

*Text: Huffman coding (fast, random access)  
Images: Arithmetic coding*

# Arithmetic Coding (cont)

## How arithmetic coding works

*<Simple example based upon bit string and binary search>*

### *Concepts*

*Output is treated as a fractional binary number between 0&1*

*Stores two numbers:*

*Low: bottom of interval, initially 0*

*High: top of interval, initially 1*

*Range between low and high is divided based upon the PD (at the time of encoding/decoding)*

*Encoding step is simply narrowing the interval based upon the next symbol (fig. 2.19, p. 55)*

*Decoding step uses the same range narrowing algorithm and simply observes in which interval the encoded value falls (fig. 2.20, p. 55)*

### *Example (p. 54)*

*Symbolwise, adaptive, zero-order model*

*All symbol counts initialized to 1 for ZFP*

*Ternary alphabet {a,b,c}*

*Encoding/decoding string "bccb"*

# Arithmetic Coding (cont)

## Implementation

*Cumulative probabilities or frequency counts*

*Static or semi-static coding*

*Store cumulative probabilities or frequencies in vector*

*Encoder: indexes array by symbol number*

*Decoder: binary searches vector to get index (symbol #)*

*Adaptive coding*

*Cumulative probabilities have to be adjusted on the fly  
(probably use frequency counts)*

*Incremental coding*

*Reduces precision requirements*

*As soon as interval has the same prefix for high and low  
it can be output and removed*

## Compression results

*Number of output bits is proportional to the negative log  
of size of interval*

*Size of the final interval is a product of the probabilities  
encoded (so log of this interval size is same as sum of  
logs of each probability)*

*So, an  $s$  of  $Pr[s]$  contributes  $-\log Pr[s]$  bits to the output  
which is  $I(s) \Rightarrow$  entropy (except for precision  
problems, etc.)*

*Scaling can be used to work with integer numbers for speed  
with some loss of precision*

# Symbolwise Models

## Concepts

*Goal is to make the best symbol predictions possible  
Can be combined with Huffman or Arithmetic methods  
Adaptively generate PD  
Four main approaches  
PPM - predictions based on previous characters  
Block Sorting - transforming the text  
DMC - FSA  
WORD - words as symbols*

## Prediction by Partial Matching (PPM)

*Based on finite-context models of different sizes  
Starts with large (3 or 4 characters) context and steps down  
until it can match a prefix string  
Sends special "escape" character if prefix has been seen  
but not before the symbol being encoded (this is a  
ZFP in this context)*

*Method A (PPMA) - Assign escape character a count of 1*

*Method C (PPMC) - Assign escape character a probability of  $r/(n+r)$  where  $r$ =total distinct characters in this context and  $n$ =total characters in this context*

*Note: As the number of distinct characters rises ( $r$  increases), the probability rises that you will see a "new distinct" character that will have to be escaped by sending a special escape character*

*Note: As the percentage of distinct characters falls ( $n$  increases), so does the probability that you will see a "new distinct" character*

*Method D (PPMD) - Assign escape character a probability of  $r/(n+r)$*

*Method X - Hapax Legomena, etc.*

*PPM is very effective and is often combined with arithmetic coding since it provides high probabilities  
SAKDC is based upon PPM*

# Symbolwise Models (cont)

## Block-sorting Compression

*Transform text, encode — decode, inverse transform*

*Burrows-Wheeler transform*

*Sort each character in the text using its context as the sort key  
working from right to left (fig. 2.25, p. 66)*

*Transformed text is the characters in order of their sorted contexts*

*Coding is done with a move-to-front coder*

*Inverse transform requires "sorting" the permuted text which gives  
the last character of the context, the permuted text and its sorted  
list are used to produce the original text (fig. 2.26, p. 67)*

*See code p.68 and fig. 2.27*

## Dynamic Markov Compression (DMC)

*Based on FSA*

*Adaptive bitwise compression*

*Probabilities and fsa structure adapt as coding proceeds*

*ZFP avoided by setting each transition to a 1*

*Frequencies of transitions are recorded and used for PD  
(fig. 2.28, p. 70)*

*Structure is adapted through "cloning"*

*Heavy use of transition causes new state to be generated*

*Allows new PD for new state (hopefully better estimates)*

*(fig. 2.30, p. 71)*

## Word-based compression (WORD)

*Input text is broken into word symbols and non-word symbols*

*Particularly suited to FTR*

*Words and their frequency are also very valuable for  
query processing as well as coding*

*Potential problems*

*Numbers (e.g. in financial tables)*

*Page numbers*

*In the static or semi-static case a CHC is ideal*

# Dictionary Models

## Concepts

*Replacing substrings in the input text with codewords that identify the substring in a "dictionary"*

*Fixed codewords rather than PD*

*Static approaches*

*Digram coding*

*128 Ascii codes + 128 common digrams*

*8 bit output code*

*Compression?*

*Phrase coding*

*"the" "and" "tion" etc.*

*Semi-static*

*Build a new codebook for each text that is compressed*

*Overhead for transmission of codebook, etc.*

*Adaptive approaches*

*Based on Ziv-Lempel*

*A substring is replaced by a pointer to a previous occurrence in the input text*

*Codebook is all prior text, codewords are pointers*

*Prior text makes a good dictionary and is implicitly transmitted as part of the process*

# Dictionary Models (cont)

## LZ77

*Particularly suitable when resources required for decoding must be minimized*

*Encoder output is set of triples:*

*How far back to look in the text*

*Length of phrase*

*Next character from the input*

*Recursive references are allowed (fig. 2.32, p. 76)*

*Restrictions are placed on:*

*Pointer: say 13 bits*

*Length: say 4 bits*

*Improvements*

*Variable length code for pointers and lengths (e.g. use CHC)*

*Include third element (char from input) only when necessary*

*Data structures for finding longest matching phrase*

*Trie, hash, binary search tree*

*Decoding is fast (one array lookup)*

*Decoding program is simple and can be included with the compressed data in the same file*

*GZIP variant of LZ77*

*Next three characters to encode are hashed to give head of LL*

*Length of LL is restricted for speed*

*Huffman codes used for length and offset*

*Raw characters are only sent when no match occurs*

*Length is sent before offset and raw characters are Huffman encoded with length for efficiency (so you know whether the current symbol is a raw character or a length)*

*Options: If compression is more important than speed, gzip uses a look-ahead for better matching*

*Huffman codes are generated semi-statically based on 64kb blocks (so gzip is not single pass)*

*LZRW1 is faster but at a price of compression performance (only one phrase is kept for each three character hash, so searching is faster)*

# Dictionary Models (cont)

## LZ78

*Places restrictions on which substrings can be referenced*  
*No limit on previous window*  
*Only one coded representation for the same string*  
*Characters to be encoded are represented by the number of the longest parsed substring that matches, followed by a raw character (fig. 2.35, p. 80)*  
*Parsing can be efficiently done with a trie (fig. 2.36, p. 81) (hashing with current node and input char may be faster)*  
*Problems:*  
*Data structure (trie, etc.) grows throughout coding and must be "pruned" when memory becomes a problem*  
*Decoding is slower than LZ77 due to data structure*

### *LZW variant of LZ78*

*Encodes only phrase numbers*  
*Does not have raw characters in output (list of phrases is initialized to include input alphabet)*  
*New phrase is constructed from a coded one by appending the first character of the next phrase to it (fig. 2.37, p. 82)*  
*Unix compress utility*  
*Number of bits for phrase numbers are gradually increased as coding proceeds*  
*Limits number of phrases*  
*When dictionary is full, adaptation ceases*  
*If compression performance degrades, the dictionary is cleared and rebuilt*

## Other Ziv-Lempel variants

*Can be classified based on how they:*  
*Parse input text*  
*Represent pointers and lengths*  
*Prevent dictionary from using too much memory*

# Synchronization

## FTR

*Requires random access, but the best compression methods:  
use variable length codes and their models are adaptive  
(starting from the beginning of the file)  
So, for FTR it is usually preferable to use a static model*

## Multiple documents in one compressed file

*When does one document end and another begin?*

*Bit offsets, byte offsets, etc.*

*Considering that your document pointer size is likely to be fixed  
(say, 32 bits), there is a trade-off wrt its granularity:*

*smaller granularity ==> less documents*

*larger granularity ==> more documents*

*For granularity greater than bit level and variable length codes,  
there is the question of which bit is the last bit of the document*

## Self-synchronizing codes

*Not particularly useful for FTR*

*Originally designed for crypto work*

*Most variable length codes will self-synchronize*

*(table 2.5, p. 88)*

*Fixed-length codes can not be self-synchronizing*

*Adaptive compression does not allow self-synchronization*

*since the decoder will be out of phase with the encoder*

# Performance Comparisons

## General comments

*Compression versus speed trade-off*

*Encoding versus decoding speed trade-off*

*Speed versus memory trade-off*

*Random access versus compression trade-off*

## Results on the Canterbury corpus

*See tables 2.6, 2.7 and figure 2.42 on p. 90-93*

*See figure 2.43, p. 95*

*See table 2.8, p. 96*

*See figure 2.44, p. 98*

Should you use Unix compress or GZIP?

# Indexing

# Concepts

## General

*Issue: How to organize information so that queries can be resolved efficiently and the relevant data extracted quickly*  
*Accurate and comprehensive indexing is a necessity*  
*Since documents are stored in a compressed format, we want to retrieve and decode only those portions of the collection that are of interest*

*A **document collection** is made up of **documents** which are described by a set of **terms***

*Queries use these terms to identify documents of interest, these documents (or their proxies) are returned*

***False matches:** documents that satisfy the query according to the index but in fact are not answers*

### **Document granularity**

*Size of unit that is returned in response to queries*

*For example: paper, section, paragraph, sentence*

*Depends on unit of storage that makes sense*

*Smaller granularity ==> larger index ==> more space*

### **Index granularity**

*The resolution to which term locations are recorded within each document*

*For document granularity of "paper", additional information on section or paragraph or sentence would make sense*

*Depends on type of queries that will be allowed and desired speed for example: proximity and phrase-based queries*

*Smaller granularity ==> larger index ==> more space*

*Since indexes themselves can become quite large, we want to devise methods for index compression*

*In a document collection of 1 million documents, a document-level index pointer would require 20 bits if uncompressed ... this can be reduced to about 6 bits for typical document collections*

### **Term transformations**

*Reduce the size of the index*

*Case folding, Stemming, Stop words, Thesaural substitution (synonyms)*

# Sample Document Collections

See table 3.1 and figures 3.1-3.4, p. 107-110

N: the number of documents in the collection

F: the total number of terms in the collection

$n$ : the number of stemmed terms  
(distinct terms)

$f$ : the number of pointers in a document-level  
index (size of the index)

## Inverted File Indexing

Aka "postings files" or "concordances"

For each term in the lexicon:

*A list of pointers to all occurrences of that term stored in  
ascending (or descending) sequence*

*The pointer could be a document pointer or a document  
pointer plus additional information depending on the  
granularity of the index (see table 3.3 versus table 3.4)*

### Queries on inverted indexes

*Single term: locate entry, retrieve documents from list*

*Conjunction of terms: intersection of lists*

*Disjunction of terms: union of lists*

*Negation of terms: complement of lists*

Uncompressed inverted files can require  
50-100% of the space of the text they index

*Total space is  $f \lceil \log N \rceil$  bits for document-level index*

# Compressing Inverted Files

$$\langle f_t; d_1, d_2, \dots, d_{f_t} \rangle$$

Since  $d_k < d_{k+1}$  for all  $k$ ,  $d_{k+1}$  can be stored as a d-gap from  $d_k$

For example:

$\langle 8; 3, 5, 20, 21, 23, 76, 77, 78 \rangle$

$\langle 8; 3, 2, 15, 1, 2, 53, 1, 1 \rangle$

Note that the sum of the gap sizes equals  $d_{f_t}$

For compression, we need a *model* and a *coding method*

## Models

*Describe the PD of gap sizes*

*Goal: higher probability gaps get coded in fewer bits, etc.*

*Global - every inverted file entry is compressed with the same model*

*Local - each inverted file entry uses its own model*

*(usually based upon a parameter such as  $f_t$ )*

*Local models outperform global models but are more complex to implement*

# Nonparameterized Models

## Flat binary

$\lceil \log N \rceil$  bits per pointer (fixed-length representation)

*Implicit probability model: each gap size is equally likely (uniformly random in 1 to N)*

## Unary code

$X \geq 1$  is encoded as  $X-1$  one bits followed by a zero bit  
Each entry requires  $d_f$  bits (sum of gap sizes =  $d_f$  and each gap of size  $X$  is encoded in  $X$  bits)

*Inverted file might require  $nN$  bits*

*Implicit probability model:  $\Pr[x] = 2^{-x}$  for gaps of length  $x$  (binary exponential decay) (favors small gaps, large gaps are coded in too many bits)*

## Gamma code ( $\gamma$ )

*Prefix code followed by suffix code*

*Unary code followed by binary code*

*Unary code*

*Specifies how many bits are required to code  $x$*

$$1 + \lfloor \log x \rfloor$$

*Binary code*

*Codes  $x - 2^{\lfloor \log x \rfloor}$  in  $\lfloor \log x \rfloor$  bits*

*Example:*

*Encoding  $x = 10$*

*Unary code:  $1 + \lfloor \log 10 \rfloor = 1 + 3 = 4 = 1110$*

*Binary code:  $10 - 2^{\lfloor \log 10 \rfloor} = 10 - 8 = 2 = 010$*

*Gamma code for  $x=10$  is 1110010*

*Decoding*

*Extract unary code ( $c_u$ ); Extract binary code ( $c_b$ )*

$$x = 2^{c_u - 1} + c_b$$

# Nonparameterized Models (cont)

## Gamma code ( $\gamma$ ) (cont)

*x is represented in  $\approx 1+2\log x$  bits (one  $\log x$  for power of 2 and one  $\log x$  for the remainder)*

*Implicit probability model:  $Pr[x] \approx 2^{-(1+2\log x)} = \frac{1}{2x^2}$*

*(remember  $x^2 = 2^{2\log x}$ ) (inverse square)*

$$\begin{aligned} x^2 &= 2^{2\log x} \\ \log x^2 &= 2 \log x \\ 2 \log x &= 2 \log x \end{aligned}$$

*A more general view of the gamma code*

*Unary code represents an index,  $k+1$ , into a vector  $V$  such that*

$$\sum_{i=1}^k v_i < x \leq \sum_{i=1}^{k+1} v_i$$

*Binary code represents a residual value in  $\lceil \log V_k \rceil$  bits*

$$r = x - \sum_{i=1}^k v_i - 1$$

$$V_{\text{gamma}} = \langle 1, 2, 4, 8, 16, \dots \rangle$$

*Different vectors would give different encodings*

## Delta code ( $\delta$ )

*Prefix code followed by suffix code*

*Same as gamma code except the prefix code is gamma code*

*Gamma code followed by binary code*

*x is represented in  $\approx 1+2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$*

*Implicit probability model:*

$$Pr[x] \approx 2^{-(1+2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor)} \approx \frac{1}{2x(\log x)^2}$$

# Parameterized Models

## Global Bernoulli model

*Parameterize based on density of pointers*

*Assume:*

*The terms are distributed uniformly across the documents*

*The  $f$  pointers in the inverted file are independent*

*The  $f$  pointers are randomly selected from the  $nN$  possible term-document pairs*

*The probability that any randomly selected document contains any randomly selected term is then  $p = \frac{f}{nN}$*

*The chance of a gap size of  $x$  is the probability of having  $x-1$  non-occurrences of that term followed by one occurrence of the term or  $Pr[x] = (1-p)^{x-1} p$  (geometric distribution)*

*Golomb code*

*For some parameter  $b$  any number  $x > 0$  is encoded in two parts:*

*$q = \lfloor (x-1)/b \rfloor$   $q+1$  is encoded in unary*

*$r = x - qb - 1$   $r$  is encoded in prefix free binary in  $\lfloor \log b \rfloor$  or  $\lceil \log b \rceil$  bits*

*If  $b$  is chosen carefully then the golomb code generates an optimal prefix-free code (see eq. 3.1 and 3.2, p. 120)*

*Note:  $b$  gives a bucket size and  $q+r$  are based upon this bucket size*

*$V_{\text{golomb}} = \langle b, b, b, b, \dots \rangle$*

## Global observed frequency model

*Build exact distribution based upon observed frequencies of gap sizes and code with an arithmetic or huffman coder*

*Note: only slightly better than gamma or delta codes*

*Note: simple local frequency codes outperform global frequency codes*

# Parameterized Models (cont)

## Local Bernoulli model

*If  $f_t$  is stored for each term then a Bernoulli model can be used on each inverted file entry and a golomb code may be used for encoding*

*$b$  will vary with  $f_t$  for each inverted file entry*

*Frequent terms are coded with small values of  $b$  while less frequent terms are coded with larger values of  $b$*

*Could use a gamma code to compress  $f_t$*

## Skewed-Bernoulli model

*In reality, terms are not scattered randomly*

*Terms tend to cluster (e.g. in chronologically ordered documents)*

*One possible vector is:  $V_T = \langle b, 2b, 4b, 8b, \dots \rangle$*

*with  $b$  chosen as median gap size in each inverted file entry (half of the gaps will fall into the first bucket)*

*$b$  can not be calculated from  $f_p$ , so the authors suggest a gamma encoded representation of  $N/b$  be added to the inverted file entries*

## Local hyperbolic model

*(not covered)*

# Parameterized Models (cont)

## Local observed frequency model

*Based upon actual observed frequencies in each inverted file entry*

*To reduce the number of models, combine frequencies under one model*

*Batched frequencies*

*Binary logarithmic batching*

*Batch by  $\lfloor \log f_t \rfloor$  then encode with a huffman code*

*Encode model selector ( $f_t$  or  $\lfloor \log f_t \rfloor$ ) with a gamma code*

## Context sensitive compression

*Based on the context of gaps actually occurring*

*Interpolative code*

*Recursively calculate ranges and code in minimal number of bits*

*Assumes you have access to the complete inverted file entry (IFE)*

*See figure 3.6, p. 127*

## Performance of index compression

*See table 3.8, p. 129*

*Total size of index =  $f$  \* bits per pointer from table 3.8*

*Local better than global*

*Bernoulli's better than frequencies since they do not*

*require the storage of parameters for the various models*

*For the majority of practical purposes,*

*Local Bernoulli using golomb code wins!*

# Signature Files

## Concepts

*Each document has a signature (descriptor) (bit vector)*

*Each indexed term is used to generate several hash values*

*Bits of the signature corresponding to those hash values  
are set to 1*

*Collisions may result in fewer bits set than hash functions*

*See table 3.9, p. 130 and table 3.10, p. 131*

*Testing a query term*

*The term is hashed*

*If a document signature has all hash bits set*

*then the term "probably" occurs in the document and the  
document must be {fetched, decoded, stemmed, scanned}  
to determine if the term "actually" occurs*

*else the term does not occur*

*False matches can be kept arbitrarily low by computing more  
hash functions and extending the document signature*

*Effective when the number of query terms is high and ineffective  
when 1 or 2 query terms are used*

*Conjunctive terms*

*Returns set of No's and Maybe's*

*Negative terms*

*Returns set of Yes's and Maybe's*

*More complex queries (including disjunction)*

*Require subexpressions to be evaluated and then combined in  
a three-valued logic (see fig. 3.7 and table 3.11, p. 133)*

*Yes's can be accepted, No's can be rejected, and Maybe's  
must be fetched, decoded, stemmed, scanned*

*Fast query processing requires Maybe set to be small*

# Signature Files (cont)

## Bitslicing

*Transposing the matrix of document signatures  
Storing the bits associated with one hash function  
together so they can be obtained with one disk access  
Only those bit slices corresponding to hash values need  
to be read when processing a query  
See table 3.12, p. 134*

## Size of signature file

*Characteristics of documents and likely queries must be known*

*Parameters:*

*b - the number of bitslices to be accessed for each query  
typical values are 6 to 12  
tradeoff between bitslice versus false match processing  
q - minimum number of terms expected to be present in  
each query  
q=1 is conservative but saves you if the number of terms  
is less than q for q>1  
z - desired bound on the expected number of false matches  
per query  
small z ==> large index (increased signature width)  
large z ==> increased query time (more false matches)  
typical value for z is 1*

*For the TREC collection*

*If b=8, q=1, z=1 then  
W=7134 (signature width)  
1456 (independent hashings for an average document)  
631 Mbytes (size of signature file)*

*For other collections*

*See tables 3.13 and 3.14, p. 136 for bits/pointer analysis  
Calculation of signature width (W)  
See p. 138-139 (not covered)*

# Signature Files (cont)

## Comments

*Which query is most likely?*

*For single term or few term queries, compressed inverted indexes are better*

*For  $b \gg 1$  term queries, bitsliced signature files can be better*

*Text is not random, so models that assume uniform random distributions should be viewed with suspicion*

*Hash functions must yield uniformly random values*

*For FTR*

*Size of signature file is roughly equivalent to uncompressed inverted index (30% to 70% of original text)*

*Variable length documents*

*Signature files are inefficient when documents vary in size since the same number of bits are assigned to all documents*

*Longer documents will have more bits set and will thus be more likely to be returned as false matches*

# Bitmaps

## Concepts

*Each term in the lexicon has an associated bit vector*

*The bit vector has a bit for each document*

*A bit is set to 1 if the term appears anywhere in the document*

*Efficient for boolean queries*

*Extravagant memory resources*

# Compression of Signature Files

## Comments

*A signature file is already in "compressed" form since it was created probabilistically based upon the original text... although the compression is lossy*

*Not much point ...*

# Compression of Bitmaps

## Comments

*In general, raw bitmaps are not useful for large scale information retrieval*

*Although, bit vector compression can be useful if random access to inverted file entries is important*

*If one regards the inverted file entry as a bit vector, hierarchical bitvector compression (with slight modification) gives random access to the pointers ... see figure 3.8, p.142*

# Comparison of Indexing Methods

## Bitmaps

*Consume an order of magnitude more secondary storage than either signature files or inverted files ... impractical*

## Signature Files

*False matches cause unnecessary access to main text*

*Some minimum number of bitslices must always be retrieved*

*Manipulations become complex if disjunction and negation are allowed!*

*Can not be used to support ranked queries!*

*Disastrous when document lengths are highly variable!*

*Two to three times larger than compressed inverted file indexes!*

## Compressed Inverted Files

*Lexicon should be held in main memory*

**Compressed inverted files are the most useful method of indexing a large collection of variable-length text documents**

# Case Folding

## Concepts

*User doesn't want to worry about the case  
Case-insensitive vs case-sensitive queries*

# Stemming

## Concepts

*Stripping suffixes to reduce a word to root form  
(see figure 3.9, p.146)*

*Note: final representation of root word doesn't matter as long as it is unique and repeatable*

*Stemming is not appropriate for all parts of a collection  
(for example, the author field of bibliographic records)*

*Hundreds of rules and exceptions in a FSA*

## Effect on Index Size

*Results in a substantial reduction in the inverted index file*

*Fewer and denser inverted lists*

*Fewer pointers in total*

*Cost of storing a stemmed lexicon for querying in addition to the unstemmed lexicon used during compression*

# Stop Words

## Concepts

*Don't index frequently occurring words (they don't give much discrimination anyway...low information content)*

*For TREC ... see figure 3.10, p. 148*

*In an uncompressed index, a quarter of the inverted file space would be saved*

*In a compressed index, the savings do not amount to much*

*Who is to say which words should not be indexed?*

# Querying

# Concepts

Formulating queries is an art

Two types of query

*Boolean*

*Terms combined with the connectives: and, or, not*

*Synonym expansion can be helpful in identifying terms*

*Exact*

*Ranked*

*A list of terms that characterize documents of interest*

*A heuristic is used to gauge the similarity of the documents to the query*

*The r most closely matching documents are returned*

*Inexact*

*More complex information must be kept and the computational costs are higher*

*Coordinate matching: simple counting of term occurrence*

*Vector space methods: term and document weights; cosine measure; length of documents*

## Recall versus Precision

*Recall*

$$\frac{\text{Relevant documents retrieved}}{\text{Total number of relevant documents in the collection}}$$

*Broad query gives higher recall (in general)*

*Precision*

$$\frac{\text{Relevant documents retrieved}}{\text{Total number of retrieved documents}}$$

*Narrow query gives higher precision (in general)*

*Goal (in general)*

*Maximizing both recall and precision*

*Each application (or user) has to make the choice*

# Accessing the lexicon

## Concepts

*Locating query terms in the collection's lexicon to identify potential candidates (possibly relevant documents)*

## Lexicon

*List of terms + auxiliary information for query processing*

*Minimal information is  $\{t, f_p, I_t\}$  or {term, frequency of this term in the collection (number of documents the term appears in), pointer to inverted file entry}*

# Storing the lexicon

See table 4.1, p. 156

## A simple array

### *Fixed-length term strings*

*{Fixed-length string, Integer counter, Integer pointer}*

*Assuming 20 character strings, 4 byte integers and*

*1 million terms, this would take about 28 megabytes*

*Is this a problem? In the future? (fig. 4.1, p. 157)*

### *Variable-length term strings*

*Terms stored in one continuous string*

*Array entry is a 4 byte character pointer into this string*

*Note: average length of words in an English lexicon is about 8*

*1 million terms would take about 20 megabytes (fig. 4.2, p. 158)*

### *Blocking and storing string length*

*Terms stored in one continuous string with 1 byte length field*

*Array entry is a 4 byte block pointer, where four strings reside*

*1 million terms would take about 18 megabytes (fig. 4.3, p. 159)*

# Storing the lexicon (cont)

## Front coding ("fronting")

*Consecutive words in a sorted list are likely to share a common prefix*

*{Prefix count, Suffix count, Characters}*

*Prefix - how many characters are the same as the previous entry*

*Suffix - how many suffix characters follow*

*Characters - the suffix characters*

*How much space is saved?*

*Typically an average of 3 to 5 characters will match*

*General rule of thumb is 40% for English*

*Gap argument - p.159/160*

*"3-in-4" front coding*

*Complete front coding loses binary search capability*

*Every fourth word is stored without front coding*

*1 million terms would take about 15.5 megabytes (tab. 4.2, p. 160)*

## Coding the integers

*Each of the three values (frequency, inverted file pointer, and string pointer) could be minimally encoded to save further space (another megabyte or two)*

*Further reductions would occur if we could only get rid of those pesky little lexicon term strings....*

*For **static** collections, we can get rid of the terms by using order preserving minimal perfect hash functions*

# Minimal perfect hashing

## Concepts

### Hashing

A hash function  $h$  maps a set of  $n$  keys  $x_j$  into a set of integer values  $h(x_j)$  in the range  $0 \leq h(x_j) \leq m-1$  with duplicates allowed

A typical hash function is  $h(x_j) = x_j \bmod m$  for:

$n$  integer keys

$\alpha$  loading factor (ratio of records to available addresses)

$m \geq n/\alpha$  and prime

### Collisions

What is the probability of inserting  $n$  consecutive items without collision into  $m$  slots?

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

Remember:

$$m(m-1)(m-2)\dots(m-n+1) \text{ is } \frac{m!}{(m-n)!}$$

### Birthday paradox

$m=365$ ,  $n=23$  then  $p=.493$

When two or more keys hash to the same value we have a collision

What can we do?

Reduce  $\alpha$  (increase  $m$ )

Rehashing and various data structure solutions

Avoid collisions?

### Perfect hashing

A hash function  $h$  is perfect if it has the additional property that:

for  $x_i$  and  $x_j$ ,  $h(x_i) = h(x_j)$  IFF  $i = j$  (no collisions)

### Minimal Perfect hashing (MPHF)

A hash function  $h$  is minimal perfect if it is perfect and  $m=n$  ( $\alpha=1$ )

Each of the  $n$  keys maps into a unique integer between 1 and  $n$

Keys are located in constant time with no space overhead

### Order Preserving Minimal Perfect hashing (OPMPHF)

A hash function  $h$  is order preserving minimal perfect if it is minimal perfect and it has the property that if

$x_i < x_j$  then  $h(x_i) < h(x_j)$

Can be processed in sorted order (if necessary)

Returns the sequence number of the key directly

MPHF and OPMPHF are useful for "static" collections

# OPMPHF

## Construction (conceptual)

### *Materials*

*Two normal hash functions  $h_1(t)$  and  $h_2(t)$  that map strings into integers in the range of 0 to  $m-1$  for  $m \geq n$*

$$h_j(t) = \left( \sum_{i=1}^{|t|} t[i] * w_j[i] \right) \text{ mod } m$$

*where:  $t[i]$  is the radix-36 value of the  $i$ th character of term  $t$*

*Note: you probably want  $t[i]+1$*

*$|t|$  is the length of the string*

*$w_j$  is a different vector of weights for each hash function*

*Note: use random number generator to provide seeds*

*A special array  $g$  that maps numbers 1 to  $m$  into the range 1 to  $n$   
(tab. 4.3, p. 163)*

### *Procedure*

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

*where:  $h(t)$  returns the ordinal number of string  $t$*

### *Space requirements for an OPMPHF*

*Array  $g$  is  $m$  items long and requires  $m \log n$  bits where  
 $m > 1.25n$*

*1 million terms would take about 13 megabytes*

# OPMPHF (cont)

## Construction (in general)

### *Step one*

*Generate random integers into the different vectors of weights  $w_i$  for each hash function  $h_1(t)$  and  $h_2(t)$*

*This will randomize the mappings of  $h_1$  and  $h_2$*

### *Step two*

*Build a graph  $G$  with  $m$  vertices and  $n$  edges such that:*

*The vertices are labelled 1 to  $m$  (or 0 to  $m-1$ )*

*The edges are defined by  $(h_1(t), h_2(t))$*

*The edges are labelled with the desired value of  $h(t)$  (1 to  $n$  or 0 to  $n-1$ )*

### *Step three*

*Search for mapping  $g$  for each edge  $(h_1(t), h_2(t))$  such that*

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

*If the graph is acyclic, then  $g$  is easily derived from a traversal and labeling of the connected components*

*Choose any unprocessed vertex, label it with 0*

*Trace all connected components, label vertices with the difference between the edge label and the label of the source vertex*

*If the graph is cyclic, then iterate the whole process*

*See example from table 4.3, p. 163 and figure 4.4, p. 166*

## Construction (more specific)

*See figures 4.5 and 4.6, p. 167*

*Uses adjacency lists for graph storage*

# OPMPHF (cont)

How likely is it that  $G$  is acyclic?

*The larger  $m$ , the more likely that  $G$  is acyclic*

*From theory of random graphs:*

*If  $m \leq 2n$ , the probability of  $G$  acyclic tends toward 0 as  $n$  grows*

*If  $m > 2n$ , the probability of a random graph of  $m$  vertices  
and  $n$  edges being acyclic is approximately:*

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

*So, the expected number of graphs that would have to be generated  
until the first acyclic graph is found is:*

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

*Of course, large  $m$  is uninteresting...*

*Algorithms with 3-graphs (instead of 2-graphs) are used thus  
reducing the required  $m$  to  $m > 1.23n$*

*For TREC*

*Less than 1 minute of processor time to build an  
OPMPHF for 535,346 terms*

# Storing the lexicon (cont)

## Disk-based lexicon storage

*Write the lexicon to disk in blocks*

*Keep in-memory index of disk blocks*

*B-tree or other dynamic index structure*

*Binary search in-memory index, retrieve block, search*

*Advantage: minimal amount of main memory*

*Disadvantage: many times slower than in-memory*

*Acceptable for most queries since time is dominated*

*by decompressing the inverted file indexes and*

*accessing and decompressing responses*

*Index construction is a special case*

# Partially specified query terms

## Concepts

*Suppose that query terms contain a wildcard character (\*)*

*that matches any sequence of characters*

*Incompatible with OPMPHF since we require access to*

*the lexicon's terms*

## Brute force string matching

*Can still use "3-in-4" front coding*

*Any initial characters can be used to narrow the search*

*Exhaustive search with pattern-matching algorithms*

## n-gram indexing

*Build n-gram inverted file index to the lexicon*

*Split query terms into n-grams*

*Search inverted file for n-grams and merge entries*

*Can compress inverted file entries as before*

*Can block terms to reduce index*

*False matches must be checked with a pattern matcher*

*Tradeoff of blocking for false match checking*

*See table 4.4, p. 171*

# Partially specified query terms (cont)

## Rotated lexicons

*Index every character of every term in the lexicon*  
*Sort the character pointers based on rotation of terms*  
*Any query term with a single wildcard can be found by*  
*binary search (see table 4.5, p. 173):*

*Rotate query term until wildcard is at end*

*Binary search:*

*Probe*

*Pull out lexicon term pointer (first number)*

*Rotate lexicon term (based on second number)*

*Correct prefix?*

*No, continue binary search*

*Yes, extract all entries that have prefix match*

*A query term with multiple wildcards must be processed*  
*indirectly by establishing a set of candidates and then*  
*narrowing the field*

*Establish candidates based on longest rotated sequence of*  
*characters, etc.*

*Expensive in terms of memory but fast*

# Boolean query processing

## Conjunctive queries

### *Processing steps*

*Stem terms and locate in lexicon*

*Sort terms by increasing  $f_t$*

*Read IFE for the least frequent term (candidates)*

*Process other IFEs in order of increasing  $f_t$  by looking up current set of candidates in the IFE*

*See figure 4.8, p. 175*

### *Term processing order*

*Result can never be bigger than IFE with smallest  $f_t$*

*Increasing order of  $f_t$  saves space and time*

### *Compression*

*If IFEs are compressed they must be decompressed*

*Compression does not allow binary search, so either a merge has to be done or an index built that allows binary search*

*Compression saves space in IFEs but costs time in query processing*

### *Random access and fast lookup??*

*We need random access into the compressed IFEs to support faster searching (synchronized codes)*

*Let's index the IFEs!*

*Every  $b_t$ th document pointer in the IFE will be indexed with a {document number, bit pointer} entry in an auxiliary index (the purpose of indexing here is to save decompression time)*

### *Several issues*

*Storage mechanism for the index:*

- these are gaps let's compress them with a delta or gamma code!*
- of course, the index must be fully decoded before searching*
- the index can be stored as skips interleaved with the IFE and these can be read with one disk access (see fig. 4.9, p. 177)*

*Value of  $b_t$ : see contorted analysis on p. 177*

*TREC collection results seem promising...for  $k=100$  (candidates), conjunctive boolean queries of 5 to 10 terms run about 5 times faster while the index grows by only 5%*

# Boolean query processing (cont)

## Conjunctive queries (cont)

### *Blocked Inverted Files*

*Skipped Inverted Files generate variable length blocks*

*(based on fixed number of document pointers)*

*Blocked Inverted Files have fixed length blocks*

*Blocks can be accessed by pointer arithmetic*

*A few bits will be wasted at the end of the block*

*The first document number in each block is stored uncompressed*

*(Critical Value)*

*Binary search is carried out on critical values in blocks*

*When the search narrows to a block, processing proceeds within*

*the block using a Golomb (or other) code*

### *Blocking based on Interpolative code*

*Critical value is the middle value of the middle block*

*Next level critical values are coded relative to the higher*

*level critical value*

*Blocks are stored in sequence of preorder traversal of the*

*binary search tree*

*Sequential processing is no longer available*

*Construction of the IFE is more complex*

## Nonconjunctive queries

### *Transformations*

*Transformed into a conjunction of disjunctions*

# Ranking and IR

## Concepts

*Boolean conjunctions of disjunctions can become too complex*

*Boolean queries are appropriate for exact queries*

*Ranked queries:*

*are appropriate for inexact queries*

*a query is a list of terms that give an indication of relevance*

*the system ranks the collection wrt the query based upon  
a similarity measure*

*the top N documents (or their proxies) are returned*

## Similarity measures

*Coordinate matching*

*Count the number of query terms that appear in each document*

*If the documents and queries are represented with bit vectors,*

*an inner product of the query vector with each document vector  
gives the similarity measure*

*Three main problems:*

*it does not take into account term frequencies within a document*

*it does not take into account term scarcity in the document collection*

*long documents will automatically be favored*

*Tackling the problems:*

*Term frequencies:*

*Term  $t$  can be assigned a document-term weight  $w_{d,t}$*

*Term  $t$  can be assigned a query-term weight  $w_{q,t}$*

*$w_{q,t} = 0$  if  $t$  does not appear in  $Q$*

*The similarity measure is the inner product of these two:*

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

# Ranking and IR (cont)

## Similarity measures

*Coordinate matching*

*Tackling the problems:*

*Term scarcity:*

*Reduce document-term weights for terms that appear in many documents by scaling document term frequencies by their inverse document frequency (term weight)*

*The TF\*IDF rule*

*Term weight*

$$w_t = \log_e(1 + N/f_t)$$

*Relative term frequency*

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

*Document vectors are calculated as:*

$$w_{d,t} = r_{d,t} \quad \text{or} \quad w_{d,t} = r_{d,t} * w_t$$

*Query vectors are calculated as:*

$$r_{q,t} = 1 \quad \text{and} \quad w_{q,t} = r_{q,t} * w_t$$

*There are many heuristics for TF and IDF (p. 184)*

*Long documents:*

*A normalization factor is included to discount long documents*

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

*where  $|D_d| = \sum_i f_{d,i}$  is the document length obtained by counting the number of indexed terms*

# Ranking and IR (cont)

## Vector space models

*Since we are using  $n$ -dimensional vectors to represent documents and queries, it is natural to consider Euclidean distance as a measure of similarity*

$$M(Q, D_d) = \sqrt{\sum_{t=1}^n |w_{q,t} - w_{d,t}|^2}$$

*Actually this would give a measure of dissimilarity and it discriminates against long documents*

*What we really want is a measure of the difference in direction of the two vectors*

*From geometry we know this is given by the angle  $\theta$  between the two vectors*

*From vector algebra we know that if  $X$  and  $Y$  are two  $n$ -dimensional vectors, the angle  $\theta$  between them satisfies:*

$$X \bullet Y = |X| |Y| \cos \theta$$

*where  $\bullet$  is inner product and  $|X|$  is the Euclidean length  $\sqrt{\sum_{i=1}^n x_i^2}$*

*The angle  $\theta$  can then be calculated from:*

$$\cos \theta = \frac{X \bullet Y}{|X| |Y|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

# Ranking and IR (cont)

## Cosine rule

*Since  $\cos \theta = 1$  when  $\theta = 0$  and  $\cos \theta = 0$  when the vectors are orthogonal, the similarity measure can be taken as the cosine of the angle between the document and query vector*

$$\text{Cosine}(Q, D_d) = \frac{Q \cdot D_d}{|Q| |D_d|} = \frac{\sum_{t=1}^n w_{q,t} * w_{d,t}}{W_q W_d}$$

$$\text{Where } W_q = \sqrt{\sum_{t=1}^n w_{q,t}^2} \quad \text{and} \quad W_d = \sqrt{\sum_{t=1}^n w_{d,t}^2}$$

*The cosine rule can be used with any term weighting heuristic  
For example (from 4.3, p. 187),*

$$\text{Cosine}(Q, D_d) = \frac{\sum_{t \in Q \cap D_d} (1 + \log_e f_{d,t}) \cdot \log_e (1 + N/f_t)}{W_q W_d}$$

*Also, since  $W_q$  would be constant for any given query,  
we could just leave it out*

*See table 4.8, p. 188 ( $W_q$  was used in this table)*

# Evaluating retrieval effectiveness

## Concepts

*We need some way to quantify ranking rule performance*

*The performance should be based upon the total ranking*

*the ranking rule imposes on the collection wrt a query*

*Problem: trying to represent multidimensional behavior*

*with a single representative value*

*Problem: we want measures that include the relevance of*

*retrieved documents but only a particular human can*

*make this judgement and only after the fact*

*Two important measures of effectiveness: recall, precision*

## Recall and Precision

### *Recall*

*How many of the relevant documents in the collection have been retrieved?*

$$R_r = \frac{\text{Relevant documents retrieved}}{\text{Total number of relevant documents in the collection}}$$

*r is some cutoff point (the top r ranked documents)*

*Recall measures how exhaustive the search has been*

*Broad query gives higher recall (in general)*

### *Precision*

*How early in the ranking were the relevant documents listed?*

$$P_r = \frac{\text{Relevant documents retrieved}}{\text{Total number of retrieved documents}}$$

*r is some cutoff point (the top r ranked documents)*

*Precision measures how accurate the search has been*

*Narrow query gives higher precision (in general)*

### *Reporting recall and precision values*

*See table 4.9b, p. 190*

*Standardized recall-precision values*

*Interpolated precision*

*3-point retrieval effectiveness value (P at R of 20%,50%,80%)*

*11-point retrieval effectiveness value*

# Evaluating ret. effectiveness (cont)

## Recall and Precision (cont)

### *Recall-Precision curves*

*Plotting precision as a function of recall*

*The curve generally decreases since precision is high at low recall levels (few documents returned with most relevant) and low at high recall levels (many documents returned with many irrelevant)*

*Could attempt to compare ranking algorithms by comparing their recall-precision curves, but usually not straightforward*

## TREC project

*Prior to TREC there were no large test data sets!*

*"In pulling one hundred documents out of 740,000, the cosine rule gives a two-in-five chance that any document retrieved within the top one hundred is relevant"*

*"In the collection, "document" is an information-carrying word, but in the queries it is not and should be added to the stoplist"*

## Other measures of effectiveness

### *R-value*

*Assuming you know the total number of relevant documents in the collection ( $trd$ ), the  $r$ -value is set to  $P_{trd}$  (that is, the precision after the top  $trd$  documents are returned)*

### *ROC curve (Relative Operating Characteristics)*

*Probability of detection:*

$$P_d = \frac{\text{Number of relevant documents retrieved}}{\text{Total number of relevant documents in the collection}}$$

*Probability of false alarms:*

$$P_{fa} = \frac{\text{Number of irrelevant documents retrieved}}{\text{Total number of irrelevant documents in the collection}}$$

*Plot  $P_d$  as a function of  $P_{fa}$*

*Good ranking rule will give low false alarms at high detection*

# WWW Searching

## Spiders

*Spoof the spider*  
*Index spamming*  
*Antispam filters*

## Data harvesting

## Common queries

*Most common query is empty!*  
*20% are "adult and XXX"*  
*Precomputed query results*  
*Advertising?*

## Click throughs

*Query is executed twice*  
*Index of web pages*  
*Index of Advertisements*

# Implementing the cosine rule

## Concepts

*Making the ranking process efficient in terms of time and space*

*Two main issues:*

*How to store the within-document frequencies*

*How to evaluate the cosine formula*

## Within-document frequencies

*Each inverted file entry must be augmented by including with each document pointer the number of times the term appears in that document*

*That is,  $f_{d,t}$  must be stored in the inverted file entry along with the document number  $d_t$*

*Most  $f_{d,t}$  values are small and are frequently either 1 or 2*

*How should we code them? See table 4.10, p. 200*

*Unary code works well but Gamma code should be chosen if we use a simple code*

*Interpolative coding should be used if reducing the size of the index is most important*

*Note: it is possible to index every term in a large text using less than 1 byte per pointer, even when the index file contains term frequencies*

*TREC: an interpolative coded augmented inverted file takes 112 Mbytes, 5.4% of the data it indexes*

# Implementing the cosine rule (cont)

## Calculating the cosine value

*Evaluating the cosine measure using the TF\*IDF rule*

*For example (from 4.3, p. 187),*

$$\text{Cosine}(Q, D_d) = \frac{\sum_{t \in Q \cap D_d} (1 + \log_e f_{d,t}) \cdot \log_e (1 + N/f_t)}{W_q W_d}$$

*Notes:*

*$f_t$  is in the lexicon*

*$f_{d,t}$  is now included in the IFE*

*$W_q$  is a constant for each query and will be disregarded*

*$W_d$  must be computed and stored*

*We need a set of accumulators to accrue the document*

*cosine values (since we will process query terms one by one  
and each query term will cause us to process an IFE)*

*Assume  $W_d$  is precomputed and see figure 4.14, p. 202*

# Implementing the cosine rule (cont)

## Calculating the cosine value (cont)

*What problems do we have with fig. 4.14?*

*Problem:  $W_q$  is not taken into account ...*

*OK, we understand that...*

*Problem: We only present the top  $r \ll N$  documents, so we should not pay the price of a full sort...OK, we will use selection (heap)*

*Problem: Large amounts of memory are used for the  $W_d$  and the cosine accumulators...OK, see below...*

### *Memory for document weights*

*We are only talking about 2 to 3 megabytes...does this matter?*

*Solutions:*

*Store the weights on disk and sequentially read the file for each query...there will be disk activity anyway*

*Store  $f_{d,t}/W_d$  instead of  $f_{d,t}$ ...too expensive if the IFEs are compressed*

*Store approximate weights and then either use these directly or use them to guide access to the exact weights on disk*

*- no need to sequentially read the whole  $W_d$  file*

*- the approximate weights can order the top  $r$  documents*

### *Memory for accumulators*

*We are only talking about 2 to 3 megabytes...does this matter?*

*Solutions:*

*If few accumulators are needed, just hash*

*Only  $r$  documents will be returned, so limit the number of accumulators to  $r$  or slightly larger*

*- process IFEs in increasing  $f_t$  order until you have created the number of accumulators*

*- then either quit (fast query processing) or continue (slower query processing but good performance; also allows use of skipping in IFEs)*

# Implementing the cosine rule (cont)

## Calculating the cosine value (cont)

*Frequency-sorted indexes*

*IFE* <5;(1,2),(2,2),(3,5),(4,1),(5,2)> *Can use d-gaps*

*F-S IFE* <5;(3,5),(1,2),(2,2),(5,2),(4,1)> *Loses d-gaps*

*Chunking based on  $f_{d,t}$  ... Retains d-gaps within chunk*

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

*Savings on  $f_{d,t}$  duplicates results in a few % reduction in IFE*

*Processing for cosine evaluation*

*Process IFEs in parallel, one chunk at a time*

*Choose chunk with greatest accumulator contribution*

*Advantages*

*Faster and more accurate than quit or continue strategies*

*Saves disk reads (do not read the entire IFE)*

*Gives good retrieval effectiveness*

*Disadvantages*

*More complex boolean query processing*

# Interactive retrieval

## Concept

*Engage the user in refining the query*

*Re-evaluate the ranking based upon feedback from the user*

## Relevance feedback

*Process of modifying the query to improve retrieval effectiveness, based upon partial relevance judgements*

*Dec Hi strategy:*

$$Q_{i+1} = Q_i + \sum_{d \in R} D_d - D_n$$

*where  $R$  is the set of relevant documents and  $D_n$  is the highest ranked irrelevant document*

*That is, allow one document to negate query terms and all relevant documents to add terms*

*Notes:*

*Conventionally, no query terms have negative weights*

*Only a subset of the  $D_n$  terms are used for negation*

*General feedback strategy:*

$$Q_{i+1} = \pi Q_0 + \omega Q_i + \alpha \sum_{d \in R} D_d + \beta \sum_{d \in I} D_d$$

*where  $\pi$ ,  $\omega$ ,  $\alpha$ , and  $\beta$  are weighting constants ( $\beta \leq 0$ ),  $R$  is the set of relevant documents and  $I$  is the set of irrelevant documents*

*Notes:*

*Conventionally, documents already designated as relevant or irrelevant do not participate in the measure of retrieval effectiveness...this can make the performance look bad for iterated queries*

*Experimentally, one iteration brings substantial improvement but this diminishes rapidly for further iteration*

*There are no clear-cut guidelines*

*The retrieval system could also present the user with a sorted list of "relevant" terms and ask the user to prune the list*

*On-line thesaurii could possibly be used to benefit*

# Interactive retrieval (cont)

## Probabilistic models

### *Concepts*

*The appearance of a term in a document is interpreted as evidence that the document is relevant or irrelevant*

*Conditional probabilities of "relevant or irrelevant to the query given that the term appears" are estimated based upon some known relevance judgements (some training set)*

*Given:  $N$  documents,  $R$  relevant,  $R_t$  contain term  $t$ , and  $f_t$  for some training set, table 4.12, p. 216 estimates the conditional probabilities*

### *Computing $W_t$ using Baye's Theorem*

$$w_t = \frac{R_t / (R - R_t)}{(f_t - R_t) / (N - f_t - (R - R_t))}$$

*where the conditional probabilities come from table 4.12*

*Values  $> 1$  indicate document is probably relevant*

*Values  $< 1$  indicate document is probably irrelevant*

*Values  $\cong 1$  gives no indication of relevance*

*Assuming the occurrence of terms in documents is independent, the weight of the document can be estimated from:*

$$\text{weight}(D_d) = \prod_{t \in D_d} w_t$$

*Documents with high weights are returned as answers to the query*

*Since only document ordering is of interest not the weights, it is conventional to express  $\text{weight}(D_d)$  as a sum of logarithms:*

$$\text{weight}(D_d) = \sum_{t \in D_d} \log w_t = \sum_{t \in D_d} \log \frac{R_t / (R - R_t)}{(f_t - R_t) / (N - f_t - (R - R_t))}$$

# Distributed retrieval

## Concept

*No single host has access to the whole collection  
Several associated indexing sites*

## Distributed querying

*Should the collection appear to be monolithic?  
Can users specify list of hostnames/collection names?  
Query engine should run locally accessing data from  
remote machines*

### *Boolean queries*

*Query is transmitted to remote collections  
Results are combined and presented to user*

### *Ranked queries*

*More difficult since we want the top  $r$  from the combined collection*

#### *General algorithm*

*Receptionist receives query and passes it to librarians  
at remote collections*

*Librarians consult individual lexicons and return local  
term weights*

*Receptionist then calculates and returns global term weights  
to librarians*

*Librarians use global term weights to rank and return  $r$   
document proxies*

*Receptionist then produces final ranking of  $r$  documents*

### *Implementations*

*Full central index*

*Coarse granularity central index*

*Central lexicon*

*No central information*

# Index Construction

# Concepts

Constructing the index is one of the most challenging tasks for gigabyte collections  
The process of building an index is known as the *inversion* of the text

Conceptually:

*Build a frequency matrix where rows are documents and columns are terms and each entry is  $f_{d,t}$*

*Write out matrix in column (term) order (see tables 5.1, 5.2, 5.3)*

*Problem: size of the frequency matrix (1.4 terabytes for TREC)*

*Maybe VM is a solution? (2 months for TREC)*

We need economical methods for constructing and inverting a frequency matrix

*The final method presented created an augmented inverted index file for the TREC collection in under 2 hours on a personal computer*

Hypothetical collection

*5 gigabytes and 5 million documents (table 5.4, p. 227)*

Predicted resource requirements to invert 5Gb hypothetical collection

*See table 5.5, p. 227*

*See main memory requirements p. 227*

# Methods for index construction

## Memory-based inversion

*Concept and Algorithm*

*Binary search tree or hash table as head of linked list*

*Time&Space*

*6 hours and 4 gigabytes main memory*

*Move linked lists to disk?*

*Assuming nodes in linked list are interleaved on disk, 6 weeks*

*Appropriate for small collections (10 megabytes)*

## Sort-based inversion

*Concept*

*Use of disk is inescapable for the size of collection*

*Sequential access is the only efficient processing mode for*

*large disk files (xfer rates high, random seeks low)*

*Algorithm*

*Parse text into triples  $\langle t, d, f_{d,t} \rangle$  and write temporary file*

*Mergesort the temporary file into non-descending  $t, d$  order*

*Read the temporary file and write the inverted file*

*Time&Space*

*20 hours, 40 megabytes of main memory, 8 gigabytes of disk*

*(assuming that you really need two copies for merging)*

*Is 8 gigabytes of disk space too much?*

*Appropriate for moderate collections (100 megabytes)*

# Methods for index const. (cont)

## Compress the temporary files

### *Concept*

*Three values must be compressed  $\langle t, d, f_{d,t} \rangle$*

*$\langle d, f_{d,t} \rangle$  can be compressed as before...nonparameterized codes  
avoid two passes over the text...a delta+unary code would suffice*

*The  $t$ 's are sorted in the runs so they can be stored as  $t$ -gaps and  
coded by a unary+1 code requiring  $t\text{-gap}+1$  bits*

### *Algorithm*

*Same as sort-based inversion except...*

*Since  $t$ -gaps only work if the temporary file is sorted, we must  
interleave the parsing and internal sorting stages and this  
reduces the main memory available for each initial run*

### *Time&Space*

*Extra computation time is required for the compression but instead  
of reading and writing 4 gigabytes per pass at one hour each,  
we read and write 540 megabytes per pass at 9 minutes each  
(7 passes, 60 minutes, separate internal sorting pass versus  
9 passes, 100 minutes, no separate internal sorting pass)  
26 hours, 40 megabytes of main memory, 680 megabytes of  
temporary disk space*

## Multiway merging

*Concept: R-way merge with heap*

*Time&Space: 11 hours, 40 megabytes of main memory, 540  
megabytes of temporary disk space*

## In-place multiway merging

*Concept:*

*Block the temporary file and use a paged memory scheme*

*Algorithm:*

*Requires the blocks to be permuted at the end followed by one  
more pass to recode the  $t$ -gaps (may require some slack)*

*Time&Space*

*11 hours, 40 megabytes of main memory, 150 megabytes of  
temporary disk space*

# Methods for index const. (cont)

## Large in-memory inversion

### *Concept*

*Replace linked list with compressed entries for  $\langle d, f_{d,t} \rangle$   
in exact number of bits needed*

### *Algorithm*

*Make a preliminary pass over the text computing  $N, f_t, m_t$   
where  $m_t$  is the maximum within-document frequency*

*$f_t$  gives the number of  $\langle d, f_{d,t} \rangle$  pairs*

*$N$  gives size in bits for  $d$  and  $m_t$  gives size in bits for  $f_{d,t}$*

*Since two passes will be done, a minimal perfect hash function  
can be designed for the set of terms*

*In addition, we can compress the IFEs (and we can count the  
exact number of bits needed for this in pass one)*

*See figure 5.10, p. 246*

### *Time&Space*

*Assuming a "two-pass Golomb-coded in-memory" approach:*

*12 hours, 420 megabytes of main memory, 1 megabyte of  
temporary disk space*

*Appropriate for moderate collections (100 megabytes)*

## Lexicon-based partitioning

### *Concept and Algorithm*

*To reduce space, make multiple pass two's each generating a  
portion of the inverted file*

### *Time&Space*

*79 hours, 40 megabytes of main memory, no temporary  
disk space*

### *Use of extra disk space?*

*12 hours, 40 megabytes of main memory, 4 gigabytes of  
temporary disk space*

# Methods for index const. (cont)

## Text-based partitioning

### *Concept and Algorithm*

*Pass one calculates exact size of IF on disk and in-memory*

*Pass two builds in-memory IFs for chunks of text and merges them to disk (see figure 5.13, p. 252)*

### *Time&Space*

*15 hours, 40 megabytes of main memory, 35 megabytes of temporary disk space*

### *Accomplishments*

*Using text-based partitioning and in-memory compression, a 5 gigabyte collection can be full-text indexed in just the memory space required for the lexicon of the collection, using less than 10% more disk space than the final compressed inverted file, and at a rate of 300 megabytes of text per hour*

## Comparison of index construction methods

### *Winners*

*Sort-based inversion with multiway in-place merging*

*Text-based partitioning with in-memory compression*

*See table 5.5, p. 227 one more time*

<p>Study the table a bit.. I like sort-based multiway merged</p>
--

# Constructing signature files

## Concepts and Algorithm

*Partition memory in k-bit slices*

$k = \lfloor 8M/W \rfloor$  where  $M$  is size of memory in bytes and  $W$  is size of signature file in bits

*Process collection in chunks of k documents*

*For each document, hash terms and set bits in the bitslices*

*Write k-bit slice into appropriate place in pre-allocated signature file on disk*

## Time & Space

*Assume queries with one term, average of one false match per query, and a total of 8 bit slices retrieved...this gives a signature width of 4,100 bits and each term sets 8 bits*

*Assuming 1 microsecond per hash, it would take 7 hours*

*Assuming 10 microseconds per hash, it would take 23 hours*

*Obviously, the hash function is the time critical part of constructing signature files*

# Dynamic collections

## Concepts

*Most collections are not static...*

*How do we maintain indexes in dynamic collections?*

*How do we maintain collection parameters such as document weights in dynamic collections?*

## Insertions and Edits

*Expanding the text*

*Text itself is no problem*

*Compression scheme could be a problem if the new document contains new symbols that have no code in the compression scheme...these symbols will have to be "escaped"*

*Eventually, the collection may have to be recompressed...*

*experiments have shown that a two to four increase in number of documents can be tolerated without significant degradation*

*Expanding the index*

*Stop press file that gets checked for each query*

*Need multipoint expansion of the inverted file entries...*

*need to support a collection of dynamically changing variable-length records (see figure 5.14, p. 258)...*

*also should consider a parameterless compression code to isolate the decompression of the IFE*

*The lexicon also must be designed to allow expansion since the new document may contain new query terms...need to support something like a B-tree (minimal perfect hashing would be impractical)*

*Parameters of the collection, such as document weight, should be chosen carefully to depend only on the contents of each document (to avoid invalidating document weights when new documents are added, etc.)*

# Image Compression

# Concepts

## Terminology

*Pixels*

*Resolution - number of pixels per linear unit (table 6.1, p.265)*

*Bitplanes, pixel depth, number of bits per pixel*

*Bilevel images, continuous tone images (grayscale or color)*

*Anti-aliasing - using grayscales (especially with text)*

*Halftoning - changing grayscale to bilevel by passing through a halftone screen (grid pattern of different sized dots)*

*Lossy versus lossless storage and transmission of images*

*Progressive vs raster transmission of images (fig. 6.1, p.267)*

# CCITT fax standard

## Concepts

*Bilevel images*

*Group 3 - conventional analog telephone circuits*

*Group 4 - digital networks (assumes bit errors detected and corrected at a lower level in the protocol and therefore gives higher compression ratios) (provision for optional grayscale and color images)*

*Based on A4 paper dimensions*

*200x100 dpi (standard) or 200x200 dpi (high resolution)*

*Group 3 specifies two coding methods*

*One-dimensional scheme (fig. 6.2, p. 270)*

*Two-dimensional scheme (fig. 6.3, p. 271)*

*Group 4 just uses the two-dimensional scheme*

*See table 6.2, p. 272 for typical compression results*

# Context-based compression

## Context-based prediction methods coupled with arithmetic coding

*See figures 6.4 and 6.6, p. 274 and 275*

## Context models

*Q. How can we predict upcoming bit values?*

*A. Use as context a template of pixels surrounding the one to code (must precede it in transmission order)*

*See figures 6.7 and 6.8, p. 276 and 278*

## Two-level context models

*Large contexts, in general, do better but it takes a longer period of time to get them "primed" (for them to have enough samples to be reliable predictors)*

*Two-level models only use the full context if that context has previously been seen enough times to be a reliable predictor, else a reduced context is used.*

*See figure 6.9, p. 279*

*This method is preferred if best compression is desired*

## Clairvoyant context models

*Contexts can include pixels from the future as well as from the past*

*Indicate an upper bound on the compression achieved*

*See figure 6.10, p. 280*

*Note that two-level context models compare favorably with the "impossible" clairvoyant context models*

# JBIG

## Joint Binary Image Group

### Concepts

*Lossless compression of bilevel images*

*Can be used with grayscale images by compressing each bitplane individually (say, up to 6 planes or so)*

*Full image or progressive images*

*Progressive images (by higher resolution)*

*Base layer*

*Differential layers*

*Striping (if decode memory is a problem)*

*Context-based encoder using a template model and adaptive arithmetic coding*

*Resolution reduction*

*See figures 6.11, 6.12, and 6.13, p. 283, 284 and 285*

*Implementation is straightforward in a 4k bit table*

*Templates and adaptive pixels for coding*

*See figures 6.14 and 6.15, p. 286 and 287*

*Note:*

*Resolution reduction is from high resolution to low resolution*

*Progressive Transmission is from low resolution to high resolution*

# Lossless compression of continuous-tone images

Exact representations are sometimes essential  
*Medical, legal, archival reasons*

## GIF

*Graphics Interchange Format*  
*8 bit pixel is index into color map for the image*  
*Color table entries are 24 bits*  
*LZW compression of pixel values*

## PNG

*Portable Network Graphics*  
*GZIP compression of pixel values*  
*Difference filters: horizontal, vertical, average*  
*Up to 16 bits of grayscale or 48 bits of color information*

## FELICS

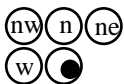
*Fast Efficient Lossless Image Compression System*  
*Grayscale images*  
*Code each pixel based on its two neighbors (fig. 6.20, p. 291)*  
*Rice codes (Golomb code where bucket size is power of 2)*  
*Computes predicted value and then sends correction*

## CALIC

*Context-based Adaptive Lossless Image Codec*  
*Codes in raster scan order using a 12 pixel context*  
*Computes predicted value and then sends correction*  
*Gradients for lines and edges ... texture patterns*  
*Distinguishes between binary and continuous-tone regions*

## JPEG-LS

*Codes in raster scan order using a 4 pixel context*  
*Computes predicted value and then sends correction*  
*Edge-detecting predictor*  
*Gradient contexts*



# JPEG

## Joint Photographic Experts Group

### Concepts

*Lossy (or lossless) compression of continuous tone still images*

*Designed for interactive use*

*Baseline system + optional extended features*

*Works on 8x8 pixel blocks at 8 bits per pixel*

*(higher resolutions are an option)*

*Encodes color image components independently*

*(can be used with different color spaces)*

*See fig. 6.17 and 6.18, p. 299 and 300*

*Progressive transmission*

*Spectral selection - low frequency followed by high frequency*

*Successive approximation - all coefficients sent with reduced precision and followed by higher precision*

*Lossless feature sends final spatial "correction" to coefficients*

*See compression factors on p. 303*

# Progressive transmission of images

## Three basic mechanisms

### *Transform coding*

*Transmit spatial frequency information progressively*

*Image grows sharper as more data is transmitted*

### *JPEG*

*Spectral selection*

*Successive approximation*

*Medium speed encoding; slow decoding*

### *Vector quantization*

*Begin with limited palette of colors or grayscales*

*Color detail increases as more data is transmitted*

*Very slow to encode; fast to decode*

### *Pyramid coding*

*Begin with few pixels ... low resolution*

*Gradually add more pixels ... higher resolution*

*Image resolution increases as more data is transmitted*

*Fast for both encoding and decoding*

*See figure 6.21, p. 305*

### *Compression for pyramid coding*

*Average 2x2 blocks*

*Reduced sum (3 child pixels sent)*

*Difference pyramid (parents as predictor)*

*Reduced difference (siblings as predictor)*

## How much space do "pictures" take?

*Argue point on p. 308 using KMS frame representations*

# Textual Images

# Concepts

## Textual images

*Images of documents that contain mainly typed or typeset text*

## OCR

*Many problems and may not be appropriate (archives)*

## Lossy versus lossless compression

*Compressing text and noise separately*

*Allows for progressive transmission and hierarchical storage*

*Reconstructed text (lossy) may be adequate for browsing*

## Textual image compression steps

- 1. Find, isolate, and extract all the marks (connected groups of black pixels) in the image*
  - 2. Construct a library containing the marks found in the image*
  - 3. Identify the symbol in the library that corresponds the closest to each mark in the image and measure the coordinate offsets between one mark and the next*
  - 4. Compress and store the library, the symbol sequence, and the offsets (allows one to build a reconstructed text)*
  - 5. Store enough additional information to restore the original image from the reconstructed text (specks and halos)*
- See figures 7.1–7.4 and table 7.1, p. 315–318*

## Errors produced by inexact pattern matching

*Errors of substitution (c matches an o template)*

*Errors of omission (two characters match one template)*

*Errors of commision (one template is made from two characters)*

# Extracting marks (step one)

## Procedure

*Image is scanned from left to right, top to bottom*

*First non-white pixel is used as seed to extract mark*

*A boundary tracing algorithm is used*

*See figure 7.5 and 7.6, p. 321 and 322*

*8-connected vs. 4-connected*

*Borders indicate size of bitmap needed to hold mark*

*The mark is extracted*

*Nested marks can be a problem*

*Run-based region fill algorithm (see figure 7.7, 324)*

*Note: boundary tracing is not necessary if you have the space to hold arbitrarily sized marks ... you can find the size at the same time as you extract the mark*

Marks can be sorted into natural reading order if desired

# Template matching (step two)

## Procedure

*As marks are extracted they are matched against those already in the library*

*Each library entry is a set of matching marks*

*The current mark is added to an existing set or starts a new library entry*

# Template matching (step two) (cont)

## Template matching is crucial

*Registration: corners, centroids, etc.*

*Comparison of error map between mark and library template*

*Screening strategies*

*No point in trying to match obviously different library templates*

*Width and height*

*Number of horizontal and vertical white runs enclosed in pattern*

*Registering by centroids, dividing into quadrants, finding centroid*

*of each quadrant, noting difference in position between two marks*

*Could try resolution reduction methods*

*Could sort templates to match based on some criteria to improve chances to match earlier*

*Global matching:*

*Looks at whole error map; weighting; figures 7.8, 7.9, p. 327, 328*

*Local matching*

*Looks at individual areas of error map; figure 7.10, p. 329*

*Compression-based template matching*

*Quantify the information required to transmit a mark using each library symbol as a model and choose the symbol that enables transmission in the smallest number of bits*

*Since library symbol is already stored, all bits can be used in the model and the method is clairvoyant*

*The mark may also be used as a model to code the library symbol*

*It is essentially an entropy measure of one mark wrt another – called the cross-entropy*

*Both are computed and the maximum is taken as the value*

*Evaluation*

*Six template matching methods, p. 334*

*Three types of noise: salt and pepper, edge, extreme edge*

*Five methods of screening*

*See figures 7.12–7.14, p. 335–337*

# From marks to symbols (step three)

## Library construction

*Can average set of marks for each library entry to make the image for the reconstructed text*

*If lossless mode (step five) will be used, then singletons can be removed from the library*

*If lossy mode (with no step five) will be used, then singletons should remain in the library*

*In general, one can choose to place marks in the library or leave them in the residue depending on compression desired and the way in which the textual image will be used*

# Coding the components (steps 4&5)

## Concepts

*All components can be encoded with a single arithmetic coder*

*Library*

*Number of symbols*

*For each symbol: height, width, bitmap*

*Symbol numbers*

*Symbol offsets*

*Residue*

*Difficult to compress since it is essentially noise*

*But can take advantage of the reconstructed text and the part of the residue decoded so far*

*Clairvoyant model can be used since reconstructed text is known*

*See figure 7.15, p. 343*

*Final twist (as they say...)*

*Do not calculate the residue at all...instead, encode the original image based on the clairvoyant reconstructed text and the original image (that is, use the original image instead of the residue image)*

# Performance

## Scanning

*Light contrast setting results in much fragmentation of symbols*

*See figure 7.16, p. 344*

*Too high contrast setting results in bleeding together of symbols*

*See figure 7.17, p. 345*

*See table 7.2, p. 346 (basic stats on two images from the two collections)*

*Table 7.3, p. 347 shows results for lossy and lossless compression on the two collections*

*Table 7.4, p. 348 shows results for coding the same images at different resolutions*

# System considerations

## Residue maps

*An expensive part of the whole operation*

*Could/should be held on slower backing store than reconstructed text*

*OCR for partial full-text retrieval*

# JBIG2

## A standard for textual image compression

*Defines the decoding side, leaves encoding side open*

*Soft pattern matching*

*Mark-based clairvoyant compression*

*Marks are sent by sending identity of best match in library and then encoding the actual mark with respect to this best match*

*Can incrementally transmit the library with the same technique*

# Mixed Text and Images

# Concepts

Separating the textual, line drawing, and halftone components of a document image to enhance compression

## Steps

- 1. Determine orientation of image and correct it for skew*
- 2. Segment the document into visually distinct regions*
- 3. Classify the regions into textual, line drawing, and halftones*
- 4. Code the regions appropriately*

See figures 8.1, 8.2, p. 356, 357

# Orientation

The text assumes constant skew...

## Correcting skew

- Rotate image*
- Shear transformations*
- Use a skewed "horizontal" line for scanning*

## Hough transform

- A line-to-point transformation*
- Applied to two-level images*
- Used to find sets of pixels that lie along a straight line*
  - Actually, approximately colinear sets of points to an accuracy that depends on the number of quantization levels—a parameter to the transform*
- Can be extended to detect any parametrically representable curves*
- Lines in a two-dimensional image are mapped to points in the Hough domain*
- Points in a two-dimensional image are mapped to sinusoidal curves in the Hough domain*

# Orientation (cont)

## Three basic approaches for finding skew

### *Left margin search*

*Scan right along each scan line until the first black pixel*

*Throw away paragraph indentations etc.*

*Calculate skew of left margin line*

*See figure 8.5, p. 362*

*Not a robust method and not reliable in practice*

### *Projection profile*

*Project the pixels onto the vertical axis by counting the number of black pixels on each line*

*This gives a histogram, see figure 8.6, p.363*

*Correct skew by rotating the image until the histogram displays the deepest and sharpest valleys*

### *Algorithms*

*Autocorrelation function of the histogram*

*Sum of squares of gradients at each point*

### *Slope histograms and docstrums*

*Use the marks found in the image*

*Calculate the slope between every pair of marks*

*Plot the slopes as a histogram*

*Should be a spike at 0 degrees due to the baseline of the text*

*If there is a spike at say, 2 degrees, then the image is likely skewed by two degrees (see figures 8.11-8.12, p. 369)*

*A random sample of marks will suffice...*

### *Docstrums*

*k nearest neighbor marks are used instead of all other marks*

*Conveys a sense of the image's overall properties*

*Each pairing of marks contributes a point to the docstrum*

*Radial distance from the origin is the neighbor distance*

*Angle from the horizontal is the neighbor angle*

*See figures 8.13–8.15, p. 370–372*

# Segmentation

## Concepts

*Dividing the document into regions containing text, line drawings, and halftones*

*Critical choices*

*Rectangular regions?*

*Scale (granularity)?*

*DocStyle available?*

## Two common methods

*Bottom-up segmentation*

*Run-length smoothing algorithm*

*Smears the binary image by filling in pixels between any two black pixels that are less than a certain distance apart*

*Applied separately in the horizontal and vertical directions, resulting in two distinct bitmaps*

*See figure 8.18, p. 375*

*A variant of this methods fills in both dimensions simultaneously*

*See figure 8.19, p. 377*

*Top-down segmentation*

*Recursive X–Y cut*

*The projection profile is calculated in the horizontal and vertical directions*

*A cut is made in the most prominent valley in either profile*

*The process is repeated recursively on each part*

*Stops when no sufficiently deep and wide valleys are left*

*See figure 8.20, p. 377*

## Other methods

*Mark-based segmentation*

*Bottom-up policy of enlarging a mark's bounding box*

*See figures 8.22–8.23, p. 379–380*

*Segmenting short text strings*

*Segmenting based upon a document grammar*

*SGML and the disappearance of paper...comments on bottom of p. 384*

# Classification

## Concepts

*Finding text*

*Text is long and thin*

*Height is small*

*Width/height ratio is large*

*Blackness ratio is small*

*Mean black run length is small*

*Measure black and white horizontal runs*

*Measure black, white, black horizontal runs*

*Measure slopes of marks, etc.*

*See figure 8.27, p. 387*

*Comments on p. 388*

# Implementation

# Concepts

Basically discusses their decisions for the  
*mg* system

Algorithmics

Requirements

*Good compression*

*Decoding should be fast, encoding can be slower*

*Individual documents should be decodable with a minimum  
of overhead*

Zero-order word-based semi-static  
model for text

Canonical Huffman coder

Length-limited coding

*Package-merge algorithm*

*32 bit Huffman coding can handle 10 to 20 Gbytes*

Text compression performance

*See table 9.5, p. 406*

Images and textual images

*Two-level context modeling technique*

*FELICS*

*Mark-based textual image compression*

*Compression-based template matching*

*PBM/PGM formats*

Index construction/compression

*Text-based partitioning*

*Local Bernoulli and Golomb codes*

# The Information Explosion

# Concepts

## Comments

*One gigabyte per day of Newsgroup information*

*Cell-phones and wireless, p. 434*

*How many gigabytes on the web? Terabytes?*

*Forseeable future, p. 436*

*Digital journals, p. 437*

*Web search engines*

*Collaborative filtering, p. 441*

*Digital libraries*

*Manual indexes, p. 444*

*Proper filing!, p. 448*

Science is compression?

# Guide to the mg System

# Concepts

## Comments

*Unix, retrieval engine, static document collections*

*mgbuild, mg\_get, mgstat (see figure A.1, p. 454)*

*mgquery (see figures A.2 - A.5, p. 455-458)*

*Database creation*

*See: The Steps of mgbuild in Table A.1, p. 461*

*See: Files used by mg in Table A.2, p. 462*

*Querying an indexed document collection*

**Not** *is a set difference rather than a true set complement*

*Stemmed terms, case independent, ignores punctuation*

*.mgrc*

*See: Query modes in Table A.3, p. 464*

*See: Some query variables in Table A.4, p. 464*

*Non-textual files*

*mg\_get is responsible for associating a textual file  
with the non-textual file*

*Image compression programs*

*mgbilevel, mgfelics, mgtic*

*PGM, PBM formats*

# Guide to the NZDL

# Concepts

## Comments

*Uses mg as its kernel*

*Independent demonstration collections*

## Collections

*Computer Science Technical Reports*

46,000 reports, over a million pages, over 500 million words

Indexes 2.7 Gb extracted from 34 Gb in postscript from hundreds of sites

See figure B.1, p. 471

*Other collections described on p. 472-473*

## Differences among collections

*Source and format of information*

*Updating policy*

*Search granularity*

*Different kinds of indexes*

*Structure and format of output*

## Audio collections

### *Melody index*

*Transcribes melodies automatically from microphone input*

## Video collections

*Full-text indexes versus Library catalogs*

*Digital libraries will be a major force  
for social development and democratization  
throughout the coming millenium*