**Indices and Boolean retrieval**

**Precision**

* Fraction of the retrieved docs that are relevant to the user’s information need.

**Recall**

* Fraction of the docs in collection that are retrieved.

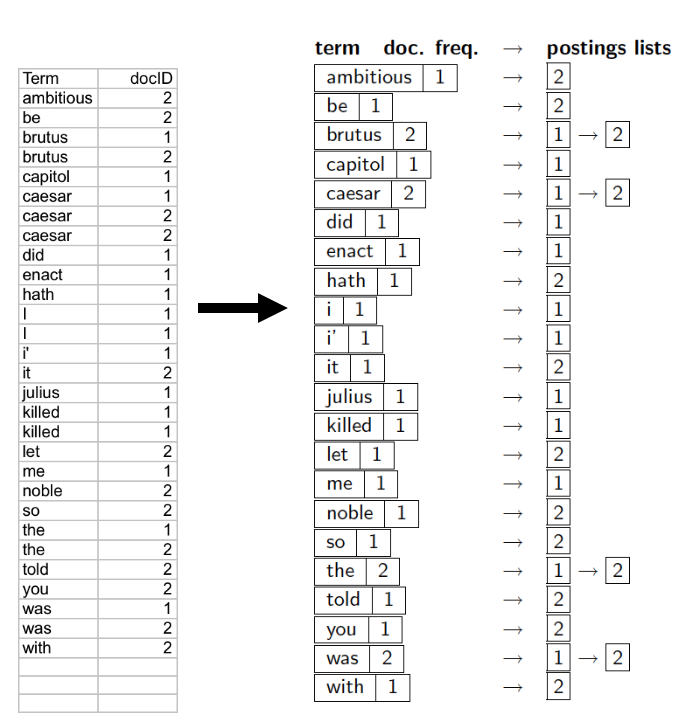
**Term-document incidence matrices:**

* Can generate incidence vectors for each term.
* Answer queries by bitwise AND/OR operation.
* Impractical in real-life due to sparseness.

**Inverted index**

* For each term t, we must store a list of all documents that contain t. (Posting list)
  + This list is sorted by docID.
* Identify each document by docID, a document serial number.

**Inverted index construction**

* **Initial stages**
  + Tokenization
    - Cut character sequence in to word tokens.
  + Normalization
    - Map text and query term to the same form. (**U.S.A** and **USA**)
  + Stemming
    - We may wish different forms of a word root to match. (**Authorize, authorization**)
  + Stop words
    - We may omit very common words. (**the, or, a, of**)
* **Token sequence processing**
  + Sort by terms and then by docID.
* **Construct dictionary and postings**
  + Multiple term entries in a single document are merged.
  + Split into dictionary and postings.
  + Document frequency information is added. (Number of different documents that contains a term)

**Query processing**

* Retrieve the posting lists of the query terms.
* Merge the posting lists together. (AND/OR)
  + AND: Construct a new posting list that only contains the postings that are in both.
  + OR: Construct a new posting list that contains all the postings.

**Boolean retrieval model**

* The **Boolean retrieval model** is being able to ask a query that is a Boolean expression
  + Boolean Queries are queries using AND, OR and NOT to join query terms.
    - Views each document as a set of words.
    - Is precise: document matches condition or not.
  + Perhaps the simplest model to build an information retrieval system on.
* Primary commercial retrieval tool for 3 decades.
* Many search systems are still using the Boolean retrieval model:
  + Email, library catalog, etc.

**Query evaluation optimizations**

* Process query terms in order of increasing frequency/length
  + Eliminate most of the postings during the earlier stages of the process.
* General optimization
  + (A OR B) AND (C OR D)
  + Get document frequency for all terms
  + Estimate the size of each OR clause by the sum of its document frequencies.
  + Process in increasing order of OR clause sizes.

**Phrase queries**

* We want to be able to answer queries such as ‘’Stanford university’’ as a phrase.
* Thus, the sentence ‘’I went to university at Stanford’’ is not a match.
  + The concept of phrase queries has proven easily understood by users; one of the few ‘’advanced search’’ ideas that works.
  + Many other queries are implicit phrase queries.
* For this. It is no longer suffices to store only <term: docs> entries.

**Solution 1: Biword indexes**

* + Index every consecutive pair of terms in the text as a phrase.
  + For example, the text ‘’Friends, Romans, Countrymen’’ would generate the Biwords:
    - Friends romans
    - Romans, countrymen
  + Each of these Biwords is not a dictionary term
  + Two-word phrase query processing is now immediate
  + **Longer phrase queries:**
    - Longer phrases can be processed by breaking them down
    - ‘’Stanford university palo alto’’ can be broken into the Boolean query on Biwords
      * **Stanford university** AND **university palo** AND **palo alto**

**Issues with Biword indexes**

* + False positives
    - Without the document contents, we cannot verify that the documents matching the Boolean query do contain the phrase. The document could contain all the Biwords, but not as a coherent phrase.
  + Index blowup due to bigger dictionary
    - Infeasible for more than Biwords, big even just for Biwords
  + Biwords indexes are not the standard solution but can be part of a compound query evaluation strategy.

**Solution 2: Positional indexes**

* + In the postings, store for each term, the positions in which the tokens of i appear.
    - Term: frequency,
      * Doc1: pos1, pos2...
      * Doc2: pos1, pos2...
  + A position is the order of the token, not the index in the content of the document. Position 15 means the token is the 15th token in the normalized document content.
  + **Processing a phrase query**
    - Phrase **‘’to be or not to be’’**
    - Terms: to, be, or, not
    - Extract inverted index entries for each distinct term
    - Merge their doc:position lists to enumerate all positions with ‘**’to be or not to be**’’.
      * To: 2: 1,17...., 4: 8, **16**, 190.......
      * Be: 1: 17...., 4: **17**, 197......
    - Same general method for proximity searches.
  + **Proximity queries**
    - **LIMIT /3 STATUTE /3 FEDEAL /2 TORT**
      * /k means ‘’within k words of’’.
    - Clearly, positional indexes can be used for such queries, Biword indexes cannot.
  + **Positional index size**
    - A positional index expands postings storage substantially
      * Even though indices can be compressed
    - Nevertheless, a positional index is not standardly used because of the power and usefulness of phrase and proximity queries, whether used explicitly or implicitly in a ranking retrieval system.
    - Need an entry for each occurrence, not just once per document.
    - Index size depends on average document size
      * Average web page has < 1000 terms
      * SEC filings, book, even some epic poems, easily 100,000 terms.
    - **Rule of thumb**
      * Positional index is 2-4 time as large as a non-positional index
      * Positional index size is 35%-50% of volume of original text.
      * **(For all ‘’English like’’ languages)**

**Combination schemes**

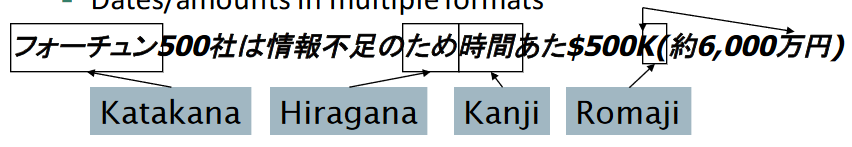
* + Biword indexes and positional indexes can be profitably combined
    - For particular phrases, such as named entities (‘’Michael Jackson’’, ‘’Britney Spears’’) it is inefficient to keep on merging positional posting lists.
      * Even more so for phrases like ‘’The Who’’(band)

**Document ingestion**

**Parsing a document**

* Format (pdf/word/excel/html……)
* Language
* Character set (CP1252, UTF-8, Unicode……)
* **Complications**
  + Documents being indexed can include documents from many different languages
    - A single index may contain terms from many languages.
  + Sometimes a document or its components can contain multiple languages/formats
    - French email with German pdf attachment
    - French email quote clauses from an English-language contract
  + What is a unit document?
    - A file?
    - An email? (Perhaps one of many in a single mbox file)
      * What about an email with 5 attachments?
    - A group of files (e.g., PPT or LaTeX split over HTML pages)

**Tokens – Tokenization**

* A token is an instance of a sequence of characters.
* Each such token is now a candidate for an index entry, after further processing.
* **Issues in tokenization:**
  + “Finland’s capital” ->
    - Finland And s? Finlands? Finland’s?
  + “Hewlett-Packard” ->
    - Hewlett and Packard as tow tokens?
  + San Francisco
    - One token or two?
  + Numbers
    - Often have embedded spaces
    - Older information retrieval systems may not index numbers
      * However often very useful, things like looking up error codes/stack traces on the web
      * Often index “meta-data” separately (creation data, format, etc.)
  + Language issues
    - French
      * L’ensemble
        + One token or two tokens?
    - German
      * “Lebenscersicherungsgesellschaftsangestellter”
      * ‘Life insurance company employee’
      * German retrieval systems benefit greatly from a **compound splitter** module
    - Chine and Japanese have no spaces between words
      * Not always guaranteed a unique tokenization
    - Further complicated in Japanese, with multiple alphabets intermingled.
      * Dates/amounts in multiple formats
    - Arabic (or Hebrew) is basically written right to left, but with certain items like numbers written left to right.
      * Words are separated, but letter forms within a word form complex ligatures.
      * With Unicode, the surface presentation is complex, but the stored form is straight forward.

**Terms – The things indexed in an IR system**

* **Stop words**
  + Words that have little sematic content: the, a, and, to, be
  + There are a lot of them.
  + Can be excluded from the dictionary entirely.
  + However, the trend is away from excluding them
    - Good compression techniques mean the space for including stop words in a system is very small
    - Good query optimization techniques mean you pay little ay query time for including stop words
    - You need them for
      * Phrase queries: “King of Denmark”
      * Various song title, etc.:” Let it be”, “To be or not to be”
      * “Relational queries”: “flights to London”
* **Normalization to terms**
  + We may need to “normalize” words in indexed text as well as query words into the same form.
    - We want to match U.S.A with USA
  + Result is terms: a **term** is a (normalized) word type, which is an entry in our IR system dictionary.
  + We most commonly implicitly define equivalence classes of terms by
    - Deleting periods to form a term
      * U.S.A, USA
    - Deleting hyphens to form a term
      * Anti-discriminatory, antidiscriminatory
  + Other languages
    - Accents: French Résumé vs resume.
    - Umlauts: German: Tuebingen vs Tűbingen
    - Even languages that standardly have accents, users often may not type them
  + **Tokenization and normalization may depend on the language and so is intertwined with language detection**
  + **Crucial: need to “normalize” indexed text as well as query terms identically.**
  + An alternative to equivalence classing is to do asymmetric expansion
    - An example of where this may be useful:
      * Enter: window
        + Search: window, windows
      * Enter: windows
        + Search: Windows, windows, window
      * Enter: Windows
        + Search: Windows
    - Potentially more powerful, but less efficient
* **Case folding**
  + Reduce all letters to lower case
    - Exception: upper case in mid-sentence?
      * General Motors
      * Fed vs fed
      * SAIL vs sail
    - Often best to lower case everything, since users will use lowercase regardless of “correct” capitalization
* **Thesauri and Soundex**
  + Do we handle synonyms and homonyms?
    - Hand-constructed equivalence classes
      * Car = automobile color = colour
    - We can rewrite to form equivalence class terms
      * When the document contains automobile, index is under car-automobile (and vice-versa)
    - We can expand a query
      * When the query contains automobile, look under car and automobile
  + What about spelling mistakes?
    - One approach is Soundex, which forms equivalence classes of words based on phonetic heuristics.

**Stemming and lemmatization**

**Lemmatization**

* Reduce inflectional/variant forms to base form
* Examples:
  + Am, are, is -> be
  + Car, cars, car’s, cars’ -> car
  + The boy’s cars are different colors -> the boy car be different color
* Lemmatization implies doing “proper” reduction to dictionary headword form

**Stemming**

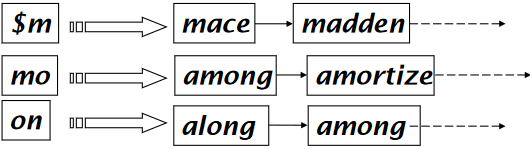
* Reducer terms to their “roots” before indexing
* “Stemming” suggests crude affix chopping
  + Language dependent
  + Examples:
    - Automat, automatic, automation -> automat
* **Porter’s algorithm**
  + Most common algorithm for stemming English
  + Conventions + 5 phases of reduction
    - Phases applied sequentially
    - Each phase consists of a set of commands
    - Sample convention
      * Of the rules in a compound command, select the one that applies to the longest suffix
    - Typical rules in Porter
      * Sses -> ss
      * Ies -> i
      * Ational -> ate
      * Tional -> tion
    - Weight of word sensitive rules
      * (m > 1) EMENT -> ø (EMENT goes to the empty string, if more than one character is left after the removal)
        + Replacement -> replac
        + Cement -> cement (unchanged)
* Other stemmer
  + Lovins stemmer
    - Single-pass, lonest suffix removal (250 rules)
  + Paice/Husk stemmer
  + Snowball
  + Full morphological analysis (lemmatization), At most modest benefits for retrieval.
* Language and application specific
* Does stemming help?
  + English: Very mixed results. Helps recall but harms precision.
  + Useful for Spanish, German Finnish….

**Tolerant retrieval**

**Dictionary**

* The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list
* Two main choices
  + Hash tables
    - Each vocabulary term is hashed to an integer.
    - Pros:
      * Lookup is faster than for a tree O(1)
    - Cons:
      * No easy way to find minor variants (judgment / judgEment)
      * No prefix search (hash is mixed up)
      * If vocabulary keeps growing, expensive rehash operation
  + Trees
    - Simplest: binary tree
    - Most common: B-trees
    - Require a standard ordering of characters and hence strings, but we typically have one
    - Pros:
      * Solves the prefix search problem
    - Cons:
      * Slower: O(logM) search for balanced trees, worse for unbalanced trees
      * Rebalancing binary trees is expensive
        + B-trees mitigate the rebalancing problem

**Wild-card queries**

* Mon\*: Find all docs containing any word beginning with ‘mon’
  + Easy with trees lexicon: retrieve all words in range ‘mon’ < w < ’moo’.
* \*mon: Find all docs that contain any word ending with ‘mon’
  + Maintain an additional B-tree for terms backwards
  + Retrieve all words in range: ‘nom’(mon backwards) < w < non.
* **Query processing**
  + At this point, we have an enumeration of all terms in the dictionary that match the wild-card query
  + We still have to look up the postings for each enumerated term.
  + **How to handle \*’s in the middle of a query term? (co\*tion)**
    - **We could look up co\* and \*tion in a B-tree and intersect the two term sets**
      * **Expensive**
    - **Solution**
      * **Transform wild-card queries so that the \*’s occur at the end.**
      * **Permuterm index**
* **Permuterm index**
  + For term **hello**, index under:
    - hello$, ello$h, llo$he, lo$hel, o$hell
    - $ is a special symbol
  + Each permuterm is mapped back to the original term
  + Queries:
    - X lookup on X$
    - X\* look up on $X\*
    - \*X lookup on X$\*
    - \*X\* look up on X\*
    - X\*Y look up on Y$X\*
    - X\*Y\*Z
      * First look up on Z$X\*
      * Results are all words in the form X\*Z
      * Filter results that are in the form X\*Y\*Z
    - Always push the start to the end
    - Transform all queries into prefix queries
* **Bigram (k-gram) indexes**
  + Enumerate all k-grams (sequence of k chars) occurring in any term
  + From the text ‘April is the cruelest month’, we get the bigrams
    - $a, ap, pr, ri, il, l$, $i, is, s$, $t, th, he, e$, $c, cr, ru,ue, el, le, es, st, t$, $m, mo, on, nt, h$
    - $ is a special symbol that represents the word boundaries
  + Index mappes from bigrams to dictionary terms that matches each bigram
  + **Query processing**
    - Query mon\* can now be run as
      * $m AND mo AND on
    - Get posting lists of the bigram terms and intersect
      * Results in the list of matching dictionary terms
    - **BUT we would enumerate moon**
      * **Must post-filter against query again**
    - Surviving enumerated terms are then looked up in the term document inverted index
    - Fast, space efficient (compared to permuterm index)

**Spelling correction**

* Two principal uses
  + Correct documents being indexed
  + Correcting user queries to retrieve ‘right’ answers
* Two main flavors:
  + Isolated word
    - Check each word on its own misspelling
    - Will not catch typos resulting in correctly spelled words
    - Form -> from
  + Context sensitive
    - Look at surrounding words
    - I flew form Heathrow to Narita.
* **Document correction**
  + Especially needed for OCR’ed documents
  + Can use domain-specific knowledge
    - E.G., OCR can confuse O and D more often than it would confuse O and I
  + But also, web pages and even printed material have typos
  + Goal: the dictionary contains fewer misspellings
  + Often we do NOT change the document contents, but instead fix the query document mapping
* **Query misspellings**
  + We can either
    - Retrieve documents indexed by the correct spelling
    - Return several suggested alternative queries with the corerct spelling
      * *Did you mean...?*
* **Isolated word correction**
  + Fundamental premise
    - There is a lexicon from which the correct spellings come from.
  + Two basic choices for this
    - A standard lexicon such as
      * Webster’s English dictionary
      * An ‘industry-specific’ lexicon (hand maintained)
    - The lexicon of the indexed corpus
      * E.g., all the words on the web
      * All names, acronyms etc.
      * (including the misspellings)
  + **Goal:** 
    - **Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q**
  + **Edit distance (Levenshtein distance)**
    - Given two strings S1 and S2, the minmimum number of operations to convert one to the other
    - Operations are typically character-level
      * Insert, delete, replace, transposition (switch)
      * From dof to dog is 1
      * From cat to act is 2 (or 1 if transposition is permited)
    - Generally found by dynamic programming
  + **Weighted edit distance**
    - Edit distance, but the weight of an operation depends on the cahracter(s) involved
      * Meant to capture OCR or keyboard errors
      * M is more likely to be mistypes as n than as q, thus replacing ,m by n has a smaller edit distance than replacing m by q.
      * This may be formulated as a probability model
    - Requires weight matrix as input
    - Can be implemented by modifying the dynamic programming implementations so that they handle weights

**Using edit distance**

* + - Given a query, first enumerate all cahracter sequences within a preset(weighted) edit distance.
    - Intersect this set with list of ‘correct’ words
    - Show terms you found to user as suggestions
    - Alternatively
      * We can look up all possible corrections in our inverted index and return all docs (SLOW)
      * We can run with a single most likely correction
    - The alternatives disempower the users, but save a round of interaction with the user.
  + **N-gram overlap**
    - Enumerate all the n-grams in the query string as well as in the lexicon
    - Use the n-gram index (wil-card search) to retrieve all lexicon terms matching any of the query n-grams
    - Threshold by number of mathcing n-grams
      * Variants – weight by keyboard layout, etc.
    - Example
      * Suppose the text is **november**
        + Trigrams: nov, ove, vem, **emb, mbe, ber**
      * The query is december
        + Trigrams: dec, ece, cem, **emb, mbe, ber**
      * 3 trigrams overlap (of 6 trigrams in each term)
      * Use Jaccard coefficient to calculate overlap
      * Decide threshold to devide if you have a match, e.g. if J.C. > 0.8.
      * This case the Jaccard coefficient is 0.3333333...
    - Jaccard coefficient
      * A commonly used jmeasure of overlap
      * Let X and Y be two sets
      * Jaccard coefficient is:
        + |X n Y| / |X u Y|
      * Equals 1 when X and Y have the same elements and zero when they are disjoint
      * X and Y do not have to be the same size
      * Always assigns number between 0 and 1
* **Context sensitive spell correction**
  + Need surrounding context to catch this
  + First idea:
    - retrieve dictionary terms close (in weighted edit distance) to each query term
    - now try all possible resulting phrases with one word ‘fixed’ at a time
      * flew from Heathrow
      * fled form Heathrow
      * flea form Heathrow
    - **Hit based spelling correction**
      * **Suggest the alternative that has lots of hits**
  + Another approach
    - Break phrase query into a conjunction of Biwords.
    - Look for Biwords that need only one term corrected.
    - Enumerate only phrases containing ‘common’ Biwords.
* General issues in spell correction
  + We enumerate multiple alternatives for ‘did you mean?’
  + Need to figure out which to present to the user
    - The alternative hitting most docs
    - Query log analysis
  + More generally, rank alternatives by probability.

**Soundex**

* Turn every token to be indexed into 4-character reduced form
* Do the same with query terms
* Build and search in index on the reduced forms
  + When the query calls for a Soundex match
* Typical algorithm

1. Retain the first letter of the word
2. Change all occurrences of the following letters to ‘0’(zero)
   1. A, E, I, O, U, H, W, Y
3. Change letters to digits as follows
   1. B, F, P, V -> 1
   2. C, G, J, K, Q, S, X, Z -> 2
   3. D, T -> 3
   4. L -> 4
   5. M, N -> 5
   6. R -> 6
4. Remove all pairs of consecutive digits
5. Remove all zeros from the resulting string
6. Pad the resulting string with trailing zeros and return the first four positions, which will be in the form
   1. Uppercase letter , digit, digit, digit

E.g, Herman becomes H655

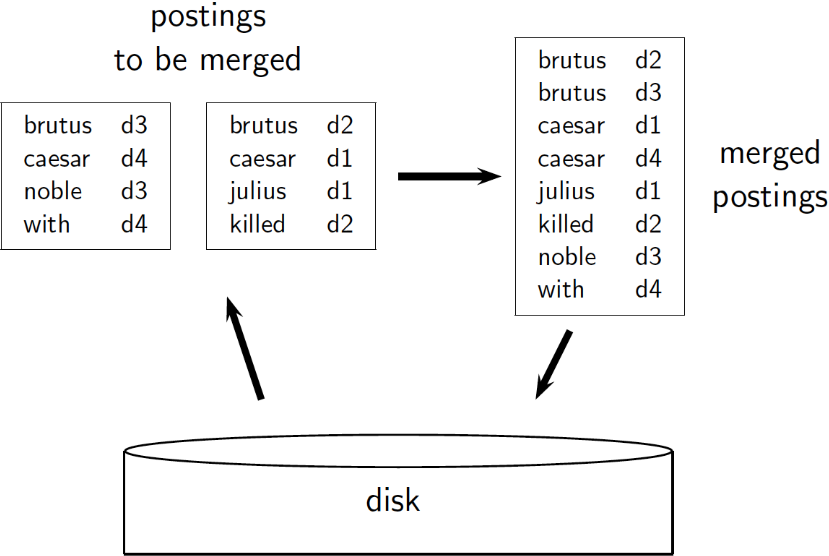
* Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft…)
* OKEY for high recall tasks, e.g. name searching for Interpol, though biased to names of certain nationalities.
* Research shows that other algorithms for phonetic matching perform much better in the context of IR.

**Index construction**

**Sort-based index construction**

* As we build the index, we parse docs one at a time
  + While building the idnex, we cannot easily exploit compression tricks
* The final postings for any term are incomplete until the end.
* At 12 bytes per non-positional postings entry (term, doc, freq), demands a lot of space for large collections
* We need to store intermediate results on disk.

**Sort using disk as memory**

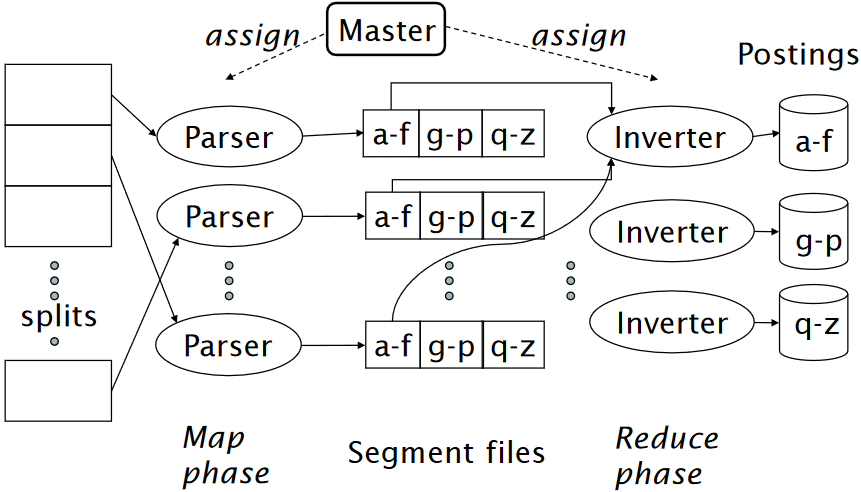
* We need external sorting algorithm
* **BSBI: Blocked sort based indexing (Sorting with fewer disk seeks)**
  + 12-byte records
  + These are generated as we parse docs.
  + Must now sort 100M jsuch 12-byte records by term.
  + Dsefine a Block (~10M) such records
    - Can easily fit a couple blocks into memory
    - We have 10 such blocks to start with
  + Basic idea of algorithm
    - Accumulate postings for each block, sort, write to disk
    - Then merge the blocks into one long sorted order.
    - 
  + Sorting 10 blocks of 10M records
    - First, read each block and sort within
      * Quicksort takes 2NlnN expected steps
      * In our case 2 x (10M ln 10M) steps
    - Done straightforwardly, need 2 copies of data on disk
    - Second, merge sorted blocks, can do in two ways
      * Binary merges, with a merge tree of logN
        + During each layer, read into memory runs in blocks of 10M, merge, write back
        + Multi-way merge, where you are reading from all blocks simultaneously

Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you’re not killed by disk seeks.

* Remaining problem with sort-based algorithm
  + Our assumption: we can keep the dictionary in memory
  + We need the dictionary (which grows dynamically) in order to implement a term to termID mapping
  + We could work with term, docID postings instead of termID, docID postings
  + However, intermediate files become very large. We would end up with a scalable, but very slow index construction method.
* **SPIMI: Single pass in memory indexing**
  + Key ideas
    - Generate separate dictionaries for each block, no need to maintain term-termID mapping across blocks.
    - Do NOT sort. Accumulate postings in posting lists as they occur
  + With these ideas we can generate complete inverted index for each block
  + These separate indexes can then be merged into one big index.
  + Compression makes SPIMI even more efficient
    - Compression of terms and postings

Distributed indexing

* For web-based indexing
* Must have distributed computing cluster
* **Key ideas**
  + Individual machines are fault-prone
  + Can unpredictably slow down or fail
  + How do we exploit a pool of machines?
* We search data centers (Google, Bing, Baidu) mainly contain commodity machines
* Data centers are distributed around the world
* Estimate: Google ~1 mill servers, 3 mill processors/cores (2007)
* In in a non-fault tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the whole system?
  + 63%
* **Basics**
  + Maintain a master machine directing the indexing job
    - This machine is considered to be ‘safe’ (will not fail)
  + Break up indexing into sets of parallel tasks
  + Master machine assigns each task to an idle machine from a pool
* Parallel tasks
  + Example: we will use two sets of parallel tasks
    - Parsers and inverters
  + Break the input document collection into splits
  + Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)
* Parsers
  + Master assigns a split to an idle parser machine
  + Parser reads a document at a time and emits (term, doc) pairs
  + Parser writes pairs into j partitims
  + Each partitim is for a range of terms first letters
    - (a-f, g-p, q-z) – here j = 3
* Inverters
  + An inverter collects all (term, doc) pairs for one term-partition
  + Sorts and writes to postings lists



* One could add another phase to the index construction process
  + Transforming a term partitioned index into a document partitioned index.

**Dynamic indexing**

* Up to now, we have assumed that collections are static
  + They rarely are:
    - Documents come in over time and need to be inserted
    - Documents are deleted and modified
  + This means that the dictionary and postings lists have to be modified
    - Postings updates for terms already in dictionary
    - New terms added to dictionary
* **Simplest approach**
  + Maintain ‘big’ main index
  + New docs go into ‘small’ auxiliary index
  + Search across both, merge results
  + Deletions
    - Invalidateion bit vector for deleted docs
    - Filter docs output on a search results by this invalidation bit vector
  + Periodically, re-index into one main index
  + **Issues:**
    - Probnlem of frequent merges
    - Poor performance during merge
    - Actually:
      * Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
      * Merge is the same as a simple append
      * However, we would need a lot fo files – OS overhead
* **Logarithmic merge**
  + Maintain a series fo indexes, each twice as large as the previous one
    - At anytime, some of these powers of 2 are instantiated
  + Algorithm
    - Keep smallest (Z0) in memory
    - Larger ones (I0, I1, ...) on disk
    - If Z0 gets too big, write to disk as I0
      * Or merge with I0 if I0 already exists, as Z1
    - Either write merged Z1 to disk as I1, or merge with I1 to form Z2
  + Auxiliary and main index: index constuction time is O(T2) as each posting is touched in each merge.
  + Logarithmic merge: Each posting is merged O(log T) times, so complexity is O(T log T)
  + So logarithmis merge is much more efficient for index construction
  + But query processing now requires the merging of O(log T) indexes
    - Where it is O(1) if you just have a main and auxiliary index
  + **Further issues with multiple indexes**
    - Collection-wide statistics are hard to maintain
      * Example: When providing spell correction, we chose the corrected alternative with the most hits.
    - How do we maintain the top ones with multiple indexes and invalidation bits vectors
      * One possibility: ignore everything but the main index for such ordering
* **Other sorts of indexes**
  + Positional indexes
    - Same sort fo sorting problem, just larger.
  + Bulding character n-gram indexes
    - As text is parsed, enumerate n-grams
    - For each n-gram, need pointers to all dictionary terms containing it
    - Note that the same ‘postings entry’ will arise repeatedly in parsing of the docs, need efficient hashing to keep track of this
      * Trigram uou occurs in the term deciduous will be dicovered on each text occurrence of deciduous
      * Only need to process each term once

**Compression**

**Why compression?**

* In general
  + Use less disk space
  + Keep more stuff in memory, thus increasing speed
  + Increase speed of data transfer from disk to memory
    - It is faster to read compressed data, then decompress, than reading uncompressed data
    - This can be achieved if the decompression algorithm is fast
* Regarding search engines (inverted indexes)
  + Dictionary
    - Make it small enough to keep in memory
    - Make it so small that you can keep some postings lists in memory too
  + Posting files
    - Reduce disk space needed
    - Decrease time needed to read postings lists from disk
    - Large search engines keep a significant part of the postings in memory
      * Compression lets you keep more in memory

**Lossless vs lossy compression**

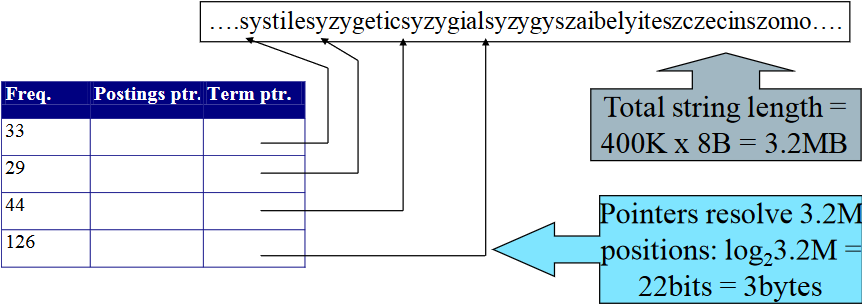
* Lossless compression
  + All information is preserved
  + What we mostly do in information retrieval
* Lossy compression
  + Discard some information
* Several of the preprocessing steps can be viewed as lossy compression
  + Case folding
  + Stop words
  + Stemming
  + Number elimination

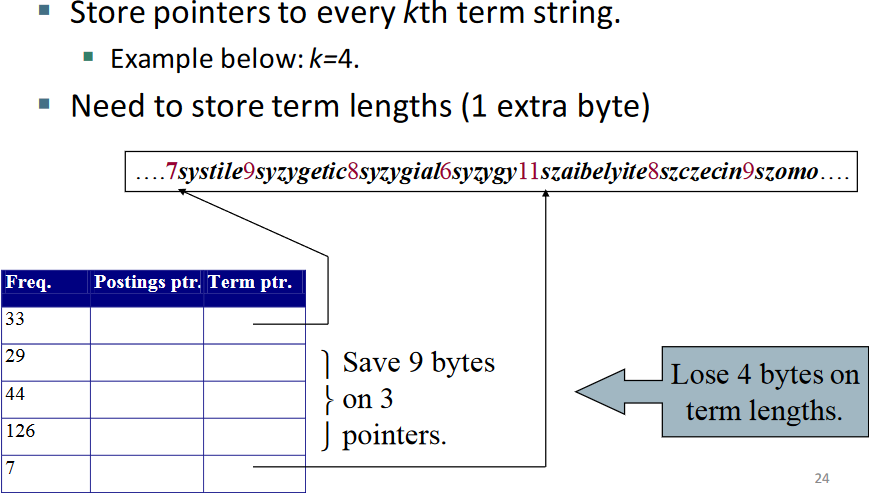
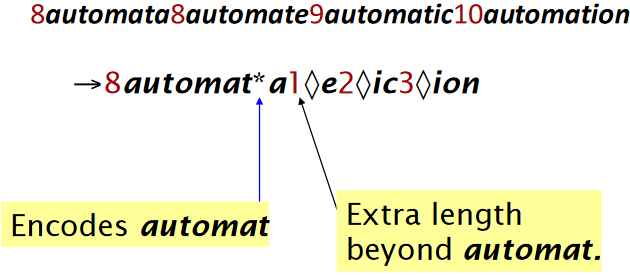
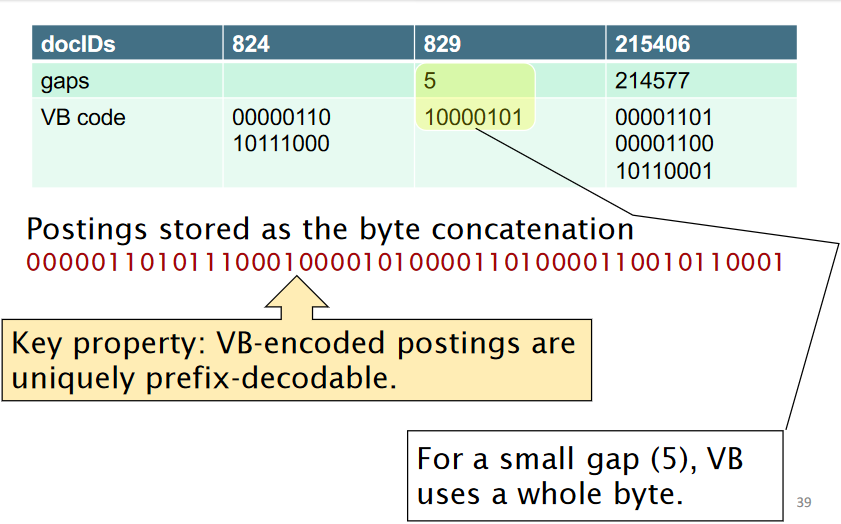
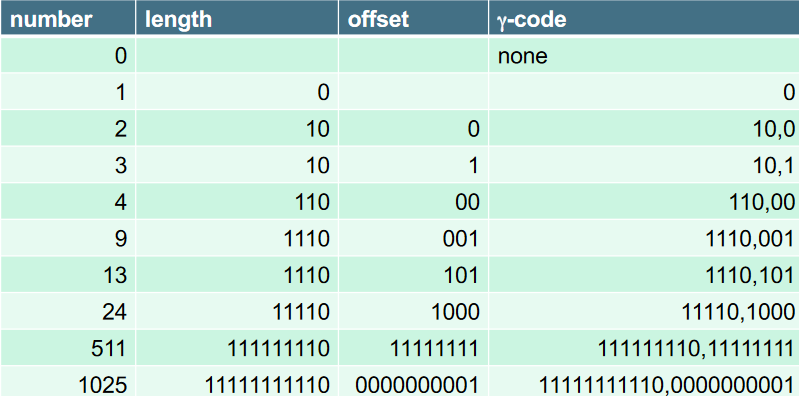
**Vocabulary vs. collection size**

* Heap’s law
  + M = kTb
  + M is the size of the vocabulary
  + T is the number of tokens in the collection
  + Typical values
    - 30 < k < 100 and b = 0.5
  + In a log-log plot of vocabulary size M vs T, Heaps’ law predicts a line with slop about ½
    - It is the simplest possible relationship between the two in log-log space
    - Empirical finding
* Zipf’s law
  + Relative frequencies of terms
  + In natural language, there are a few very frequent terms and very many very rare terms
  + The law:
    - The ith most frequent term has frequency proportional to 1/i
    - Cfi 1/I = K/I, where K is a normalizing constant
    - Cfi is collection frequency
      * The number of occurrences of the term ti in the collection

**Dictionary compression**

* Search begins with the dictionary
* We want to keep it in memory
* Memory footprint competition with other applications
* Embedded/mobile devices may have very little memory
* Even if the dictionary is not in memory, we want it to be small for a fast search startup time
* **Fixed-width terms are wasteful**
  + most of the bytes in the Term column are wasted if we allot 20 bytes for 1 letter terms
    - And we still can not handle supercalifragilisticexpialidocious
  + Written English averages ~4.5 characters / word.
  + Ave. dictionary word in English: ~8 characters
  + Short words dominate token count but not type average
* In real systems, compression is done in chunks (not to be confused with blocking)
  + Each chunk can be individually decompressed
  + This allows nextGEQ to jump forward without uncompressing all entries, by skipping over entire chunks
  + This requires auxiliary table containing the docID of the last posting in each chunk
  + Chunks maybe fixed size or fixed number of postings
* **Compression the term list**
  + **Dictionary as a string**
    - Store dictionary as a long string of characters
      * Pointer to next word show the end of current word
      * Hope to save up to 60% of dictionary space



* + - * 4 bytes per term for Freq.
      * 4 bytes per term for pointer to Postings
      * 3 bytes per term pointer
      * Avg. 8 bytes per term in term string
  + **Blocking**
    - Store pointers to every kth term string
    - Need to store term lengths
    - 
  + **Front coding**
    - Sorted words commonly have long common prefix, store differences only
    - 
* **Postings compression**
  + The postings file is much larger than the dictionary, factor of at least 10
  + Key desideratum: store each posting compactly
  + A posting for our purposes is a docID
  + Two confliction forces
    - A term like **arachnocentric** occurs in maybe one doc out of a million
      * We would like to store this posting using log2 1M ~ 20bits
    - A term like the occurs in virtually every doc, so 20 bits per posting is too expensive
      * Prefer 0/1 bitmap vector in this case
  + **Postings file entry**
    - We store the list of docs containing a term in increasing order of docID
      * Computer: 33, 47, **154, 159,** …
    - It is sufficient to store gaps
      * Computer: 33, 14, **107, 5**, …
    - Hope: most gaps can be encoded/stored with far fewer than 20 bits
  + **Variable length encoding**
    - Aim
      * For **arachnocentric**, we will use ~20bits/gap entry
      * For **the**, we will use ~1 bit/gap entry
    - If the average gap for a term is G, we want to used ~log2G bit / gap entry
    - **Key challenge: encode every inter gap with about as few bits as needed for that integer**
    - **Variable Byte codes (VB codes)**
      * Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit C
      * If G < 127, binary encode it in the 7 available bits and set c = 1
      * Else encode G’s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
      * At the end set the continuation bit of the last byte to 1, for the other byes c = 0
      * 
      * Used by many commercial/research systems
      * Good low-tech blend of variable length coding and sensitivity to computer memory alignment matches vs bit-level codes
    - **Other variable unit codes**
      * Instead of bytes, we can also use a different ‘unit of alignment’
        + 32 bits (words)
        + 16 bits
        + 4 bits (nibbles)
      * Variable byte alignment wastes space if you have many small gaps, nibbles do better in such cases
    - **Unary codes**
      * Represent n as n 1s with a final 0
      * Unary code for 3 is 1110
      * Unary code for 40 is
        + 11111111111111111111111111111111111111110
    - **Gamma codes**
      * We can compress better with bit-level codes
        + Gamma code is the best known of these
      * Represent a gap G as a pair length and offset
      * Offset is G in binary, with the leading bit cut off
        + 13 -> 1101 -> 101
      * Length is the length of offset
        + 13 (offset 101), this is 3
      * We encode length with unary code: 1110
      * Gamma code of 13 is the concatenation of length and offset
        + 1110101
      * 
      * G is encoded using 2 Log G + 1 bits
        + Length of offset is Log G bits
        + Length of length is Log G + 1 bits
      * All gamma codes have an odd number of bit
      * Almost with a factor of 2 of best possible, Log G
      * Gamma code is uniquely prefix decodable, like VB
      * Gamma code can be used for any distribution
      * Gamma code is parameter free
      * **Problem: seldom used in practice**
        + Machines have word boundaries, operations cross word boundaries are slower
        + Compressing and manipulation at the granularity of bits can be slow
        + Variable byte encoding is aligned and thus potentially more efficient
        + Regardless of efficiency, variable byte is conceptually simpler at little additional space cost.
    - **Delta coding**
      * Gamma coding, then gamma coding again on the unary part.
    - **Rice coding**
      * Consider the average or median of the numbers (gaps)
      * Simplified example for a list of 4 docIDs
        + 34, 178, 291, 453 -> 34, 144, 113, 162
        + Average 113.33
      * Round this number to smaller power of 2: b = 64 (6bits)
      * Then for each number x, encode x-1 as
        + (x-1)//b in unary
        + Followed by (x-1) mod b (binary)
      * 33 -> 0\*64 + 33 = 0 100001 (33 in binary)
      * 143 -> 2\*64 + 15 = 110 001111 (15 in binary)
      * 112 -> 1 \* 64 + 48 = 10 110000 (48 in binary)
      * 161 -> 2 \* 64 + 33 = 110 10001 (33 in binary)
    - **Golomb coding**
      * Takes average of the numbers (gaps)
      * Instead of choosing a power of 2 as b, choose b ~ 0.69\*average
        + Usually not a power of 2
      * For each number x, encode x-1 as
        + (x-1)//b in unary
        + Followed by (x-1) mod b (binary)
      * Lets say b is 78
        + Need fixed encoding of number 0 to 77 using 6 or 7 bits
        + e.g. 50 = 110010 0 and 64 = 110010 1
    - **Rice and Golomb coding**
      * Uses parameters b, Either global or local
        + Local

Once for each inverted list

More appropriate for large index structures

* + - * + Global

Entire index

Exploit clustering within a list

* + - **Simple9 (S9) coding**
      * Idea:
        + produce a word-aligned code – basic unit 32 bits
        + try to pack serval numbers into one word (32 bits)
      * Each word is split into 4 control bits and 28 data bits
      * Table
        + 28-bit number
        + 2 14-bit numbers
        + 3 9-bit numbers (1 bit wasted)
        + 4 8-bit numbers
        + 5 5-bit numbers (3 bits wasted)
        + 7 4-bit numbers
        + 9 3-bit numbers (I bit wasted)
        + 14 2-bit numbers
        + 28 1-bit numbers
      * Store and retrieve numbers using fixed bit masks
      * Encoding Algorithm
        + Do the next 28 numbers fit into one bit each?
        + Yes, use that case
        + No

Do the next 14 numbers fit into two bits each?

Yes, use that case

No……

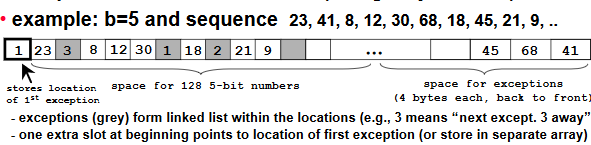
* + - * Fast decoding: only if decision for every 32 bits
      * Decent compression, uses < 1 byte for small numbers
      * **Better version: simple16**
    - **PFOR-DELTA**
      * Idea
        + Compress/decompress many values at a time
        + How many bits per number?

Different choice for each number?

Decoding slow due to decision branches

One size fits all?

Bad compression

* + - * Good compromise
        + Choose size such that 90% fit, code the other 10% as exceptions
      * Suppose in the next 128 numbers
        + 90% < 32
        + Choose b = 5
      * Allocate 128\*5 + exception space bits,
      * Exceptions stored at end as normal 4 bytes int
      * 
      * There may be forces exceptions
        + If there are more than 2b consecutive numbers < 2b, then encode the 2bth number as exception so we can keep a simple linked list structure
      * Very simple and fast decoding
        + Copy the 128 b-bit numbers into integer array
        + Then traverse linked list and patch the exceptions
        + If we keep exceptions < 10%, this will be extremely fast
      * Always uncompress next 128 posts into temp array
        + Do not uncompress entire list into one long array
        + CPU Cache
      * Simple effective improvement
        + Do not use 32 bits per exception
        + Using maximum among next 128 numbers to choose number of bits
        + 10 -20% better compression with basically the same speed (if done correctly)