Web Information Retrieval

A student

March 2019

1 Boolean retrieval

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections. The term *unstructured data* refers to data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database. Almost no data are truly *unstructured*: most text has structure, such as headings and paragraphs and footnotes, which is commonly represented in documents by explicit markup.

The field of information retrieval also covers supporting users in browsing or filtering document collections or further processing a set of retrieved documents. Given a set of documents, clustering is the task of coming up with a good grouping of the documents based on their contents. Given a set of topics, standing information needs, or other categories (such as suitability of texts for different age groups), classification is the task of deciding which class(es), if any, each of a set of documents belongs to.

Information retrieval systems can also be distinguished by the scale at which they operate, and it is useful to distinguish three prominent scales. In **web search**, the system has to provide search over billions of documents stored on millions of computers. At the other extreme is **personal information retrieval**. In the last few years, consumer operating systems have integrated information retrieval (such as Apple's Mac OS X Spotlight). Email programs usually not only provide search but also text classification: they at least provide a spam (junk mail) filter, and commonly also provide either manual or automatic means for classifying mail so that it can be placed directly into particular folders. In between is the space of **enterprise**, **institutional**, and **domain-specific search**, where retrieval might be provided for collections such as a corporation's internal documents.

Our goal is to develop a system to address the *ad hoc* retrieval task. In it, a system aims to provide documents from within the collection that are relevant to an arbitrary user information need. An **information need** is the topic about which the user desires to know more, and is differentiated from a query, which is what the user conveys to the computer in an attempt to communicate the information need. A document is relevant if it is one that the user perceives as containing information of value with respect to their personal information need. To assess the **effectiveness** of an IR system (i.e., the quality of its search results), there are two statistics about the system's returned results for a query:

- **Precision**: What fraction of the returned results are relevant to the information need?
- **Recall**: What fraction of the relevant documents in the collection were returned by the system?

1.1 An example information retrieval problem

Suppose you wanted to determine which plays of Shakespeare contain the words `Brutus AND Caesar AND NOT Calpurnia`. One way to do that is to start at the beginning and to read through all the text (grepping through text). But for many purposes, you do need more:

1. To process large document collections quickly.
2. To allow more flexible matching operations. For example, it is impractical to perform the query `Romans NEAR countrymen` with grep, where `NEAR` might be defined as "within 5 words" or "within the same sentence".
3. To allow ranked retrieval: in many cases you want the best answer to an information need among many documents that contain certain words.

The way to avoid linearly scanning the texts for each query is to **index** the documents in advance. Suppose we record for each document whether it contains each word out of all the words Shakespeare used. The result is a binary term-document incidence matrix. Terms are the indexed units; they are usually words. Depending on whether we look at the matrix rows or columns, we can have a vector for each term, which shows the documents it appears in, or a vector for each document, showing the terms that occur in it.

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

Figure 1: A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

To answer the query `Brutus AND Caesar AND NOT Calpurnia`, we take the vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise `AND`:

```
110100 AND 110111 AND 101111 = 100100
```

The answers for this query are thus *Antony and Cleopatra* and *Hamlet*.

The Boolean retrieval model is a model for information retrieval in which we can pose any query which is in the form of a Boolean expression of terms, that is, in which terms are combined with the operocument as just a set of words. Suppose we have $N = 1$ million documents. Suppose each document is about 1000 words long. If we assume an average of 6 bytes per word including spaces and punctuation, then this is a document collection about 6 GB in size. Typically, there might be about $M = 500,000$ distinct terms in these documents. A $500K \times 1M$ matrix has half-a-trillion 0's and 1's - too many to fit in a computer's memory. But the matrix is extremely sparse, it has

few non-zero entries. Because each document is 1000 words long, the matrix has no more than one billion 1's, so a minimum of 99.8% of the cells are zero.

A much better representation is to record only the 1's occurrences. With **inverted index** we keep a dictionary of terms. Then for each term, we have a list that records which documents the term occurs in. Each item in the list is called **posting**, the list is called **postings list** and all the postings list are called **postings**.

1.2 A first take at building an inverted index

To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in this are:

1. Collect the documents to be indexed.
2. Tokenize the text, turning each document into a list of tokens.
3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms.
4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

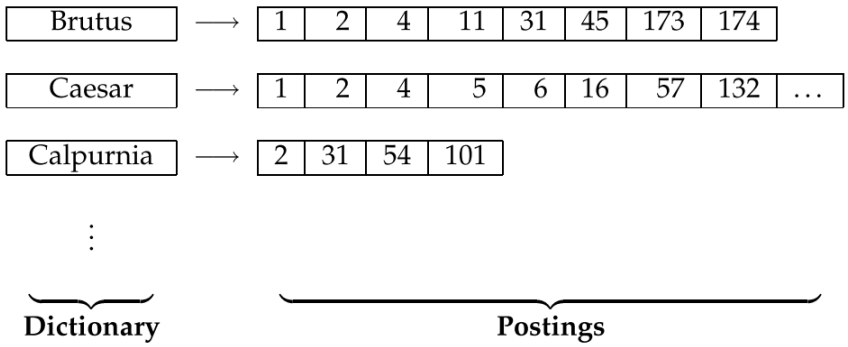

Figure 2: The two parts of an inverted index. The dictionary is kept in memory, with pointers to each postings list, stored on disk.

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (**docID**). The core indexing step is sorting this list so that the terms are alphabetical. Instances of the same term are then grouped, and the result is split into a dictionary and postings. The dictionary also records some statistics, such as the number of documents which contain each term, the **document frequency**. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. The dictionary is commonly kept in memory, while postings lists are normally kept on disk. What data structure should be used for a postings list? A fixed length array would be wasteful as some words occur in many documents, and others in very few. For an in-memory postings list, two good alternatives are singly linked lists or variable length arrays.

1.3 Processing Boolean queries

How do we process a query using an inverted index and the basic Boolean retrieval model? Consider processing the *simple conjunctive query*:

```
Brutus AND Calpurnia
```

over the inverted index:

1. Locate `Brutus` in the Dictionary
2. Retrieve its postings
3. Locate `Calpurnia` in the Dictionary
4. Retrieve its postings
5. Intersect the two postings lists

The **intersection** operation is the crucial one: we need to efficiently intersect postings lists so as to be able to quickly find documents that contain both terms (This operation is sometimes referred to as **merging** postings lists). There is a simple and effective method of intersecting postings lists using the merge algorithm: we maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries.

Algorithm 1 Algorithm for the intersection of two postings lists.

```
1: function INTERSECT(p1, p2)
2:     answer ← ⟨⟩
3:     while p1 ≠ NULL and p2 ≠ NULL do
4:         if docID(p1) = docID(p2) then
5:             ADD(answer, docID(p1))
6:             p1 ← next(p2)
7:             p2 ← next(p1)
8:         else if docID(p1) < docID(p2) then
9:             p1 ← next(p2)
10:            p2 ← next(p1)
11:        end if
12:    end while
13:    return answer
14: end function
```

At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are x and y, the intersection takes $O(x + y)$ operations. To use this algorithm, it is crucial that postings be sorted by a single global ordering.

Query optimization is the process of selecting how to organize the work of answering a query. A major element of this for Boolean queries is the order in which postings lists are accessed. The standard heuristic is to process terms in order of increasing document frequency: if we start by intersecting the two smallest postings lists, then all intermediate results must be no bigger than the smallest postings list.

1.4 The extended Boolean model versus ranked retrieval

The Boolean retrieval model contrasts with **ranked retrieval models** such as the vector space model in which users largely use **free text queries**, just typing one or more words rather than using a precise language. A strict Boolean expression over terms with an unordered results set is too limited for many of the information needs and most systems implemented extended Boolean retrieval models by incorporating additional operators such as term proximity operators. A **proximity operator** is a way of specifying that two terms in a query must occur close to each other.

2 The term vocabulary and postings lists

The major steps in inverted index construction are:

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Do linguistic preprocessing of tokens.
4. Index the documents that each term occurs in.

Tokenization is the process of chopping character streams into tokens, while linguistic preprocessing then deals with building equivalence classes of tokens which are the set of terms that are indexed.

2.1 Document delineation and character sequence decoding

Obtaining the character sequence in a document Digital documents that are the input to an indexing process are typically bytes in a file or on a web server. The first step of processing is to convert this byte sequence into a linear sequence of characters, by determining the correct encoding. This can be regarded as a machine learning classification problem. Once the encoding is determined, we decode the byte sequence to a character sequence.

The characters may have to be decoded out of some binary representation like Microsoft Word DOC files and/or a compressed format such as zip files. Even for plain text documents, additional decoding may need to be done. In XML documents, character entities, such as `&amp;` (&), need to be decoded to give the correct character. Finally, the textual part of the document may need to be extracted out of other material that will not be processed.

Choosing a document unit The next phase is to determine what the **document unit** for indexing is. For a collection of books, it would usually be a bad idea to index an entire book as a document. Instead, we may well wish to index each chapter or paragraph as a mini-document. Matches are thenmore likely to be relevant, and since the documents are smaller it will be much easier for the user to find the relevant passages in the document. We could treat individual sentences as mini-documents. But, if the units get too small, we are likely to miss important passages because terms were distributed over several mini-documents, while if units are too large we tend to get spurious matches and the relevant information is hard for the user to find. The issue of **indexing granularity** is responsible of a precision/recall trade-off.

2.2 Determining the vocabulary of terms

Tokenization Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called **tokens**, throwing away certain characters, such as punctuation.

Figure 3: Example of tokenization.

A **token** is an instance of a sequence of characters in some particular document that are grouped together as a semantic unit for processing. A **type** is the class of all tokens containing the same character sequence. A **term** is a type that is included in the IR system's dictionary.

The major question of the tokenization phase is what are the correct tokens to use? For example, what do you do about the various uses of the apostrophe? For the text *Finland's capital* which is the best tokenization? `Finland AND s`, `Finlands` or `Finland's`?

These issues of tokenization are language-specific. It thus requires the language of the document to be known. **Language identification** based on classifiers that use short character subsequences as features is highly effective; most languages have distinctive signature patterns.

In English, **hyphenation** is used for various purposes ranging from splitting up vowels in words, *co-education*, to joining nouns as names, *Hewlett-Packard*, to a copy editing device to show word grouping, *the hold-him-backand-drag-him-away maneuver*. Handling hyphens automatically can be complex: it can either be done as a classification problem, or more commonly by some heuristic rules, such as allowing short hyphenated prefixes on words, but not longer hyphenated forms.

Splitting on white space can also split what should be regarded as a single token. This occurs most commonly with names, *San Francisco*, but also with borrowed foreign phrases, *au fait*, and compounds that are sometimes written as a single word and sometimes space separated (such as *white space* vs. *whitespace*).

One effective strategy in practice, which is used by some Boolean retrieval systems is to encourage users to enter hyphens wherever they may be possible, and whenever there is a hyphenated form, the system will generalize the query to cover all three of the one word, hyphenated, and two word forms. However, this strategy depends on user training.

Dropping common terms: stop words Sometimes, some extremely common words which would appear to be of little value. These words are called **stop words**. The general strategy for determining a stop list is to sort the terms by **collection frequency** (the total number of times each term appears in the document collection) and then to take the most frequent terms, often hand-filtered for their semantic content relative to the domain of the documents being indexed, as a stop list, the members of which are then discarded during indexing. Using a stop list significantly reduces the number of postings that a system has to store. And a lot of the time not indexing stop words does little harm: keyword searches with terms like the and by don't seem very useful. However, this is not true every time. For example, the meaning of *flights to London* is likely to be lost if the word `to` is stopped out.

The general trend in IR systems over time has been from standard use of quite large stop lists (200-300 terms) to very small stop lists (7-12 terms) to no stop list. Web search engines generally do not use stop lists. Some of the design of modern IR systems has focused precisely on how we can exploit the statistics of language so as to be able to cope with common words in better ways.

For most modern IR systems, the additional cost of including stop words is not that big – neither in terms of index size nor in terms of query processing time.

Normalization (equivalence classing of terms) The easy case is if tokens in the query just match tokens in the token list of the document. However, there are many cases when two character sequences are not quite the same but you would like a match to occur.

Token normalization is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. The most standard way to normalize is to implicitly create **equivalence classes**. For instance, if the tokens *anti-discriminatory* and *antidiscriminatory* are both mapped onto the term `antidiscriminatory`, in both the document text and queries, then searches for one term will retrieve documents that contain either.

An alternative to creating equivalence classes is to maintain relations betweenunnormalized tokens. Thismethod can be extended to hand-constructed lists of synonyms such as *car* and *automobile*. These term relationships can be achieved in two ways. The usual way is to index unnormalized tokens and to maintain a query expansion list of multiple vocabulary entries to consider for a certain query term. A query term becomes a disjunction of several postings lists. The alternative is to perform the expansion during index construction. The alternative is to perform the expansion during index construction. When the document contains automobile, we index it under car as well, and, usually, also vice-versa. Use of either of these methods is considerably less efficient than equivalence classing.

These are some forms of normalization that are commonly used:

- **Accents and diacritics.** Diacritics on characters in English have a fairly marginal status, and we might well want *cliché* and *cliche* to match, or *naive* and *naïve*. This can be done by normalizing tokens to remove diacritics. In many other languages, diacritics are a regular part of the writing system and distinguish different sounds.

- **Capitalization/case-folding.** A common strategy is to do case-folding by reducing all letters to lower case.It will allow instances of *Automobile* at the beginning of a sentence to match with a query of *automobile*. It will also help on a web search engine when most of your users type in *ferrari* when they are interested in a *Ferrari* car. On the other hand, such case folding can equate words that might better be kept apart. Many proper nouns are derived from common nouns and so are distinguished only by case, including companies, government organizations and person names. For English, the simplest heuristic is to convert to lowercase words at the beginning of a sentence. Mid-sentence capitalized words are left as capitalized. This is known as **truecasing**.

- **Other issues in English.** For instance, you might wish to equate *ne'er* and *never* or the British spelling *colour* and the American spelling *color*. Dates, times and similar items come in multiple formats, presenting additional challenges. You might wish to collapse together *3/12/91* and *Mar. 12, 1991.*

- **Other languages.** Other languages present distinctive issues in equivalence classing. The French word for *the* has distinctive forms based not only on the gender (masculine or feminine) and number of the following noun, but also depending on whether the following word begins with a vowel. We may well wish to equivalence class these various forms of the. German has a convention whereby vowels with an umlaut can be rendered instead as a two vowel digraph. We would want to treat *Schütze* and *Schuetze* as equivalent. Document collections being indexed can include documents from many different languages. Or a single document

can easily contain text from multiple languages. Most commonly, the language is detected and language-particular tokenization and normalization rules are applied at a predetermined granularity.

Stemming and lemmatization For grammatical reasons, documents are going to use different forms of a word, such as *organize*, *organizes*, and *organizing*. Additionally, there are families of related words with similar meanings, such as *democracy*, *democratic*, and *democratization*. It's useful for a search for one of these words to return documents that contain another word in the set.
The goal of both stemming and lemmatization is to reduce inflectional forms, for instance:

am, are, is ⇒ be
car, cars, car's, cars' ⇒ car

Stemming usually refers to a heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, includes the removal of derivational affixes. **Lemmatization** usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the **lemma**. For example, the token *saw*, stemming might return just *s*, whereas lemmatization would attempt to return either *see* or *saw*.

The most common algorithm for stemming English is *Porter's algorithm*. Porter's algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules (sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix*). Some typical rules are:

- Substitution
  - SSES → SS
  - IES → I
  - SS → SS
  - S →
- Weight of word sensitive rules
- $(m > 1)$ EMENT →
  - *replacement* → *replac*
  - *cement* → *cement*

Stemmers use language-specific rules, but they require less knowledge than a lemmatizer, which needs a complete vocabulary and morphological analysis to correctly lemmatize words.

A **lemmatizer** is a tool from Natural Language Processing which does full morphological analysis to accurately identify the lemma for each word. Doing full morphological analysis produces at most very modest benefits for retrieval. While it helps a lot for some queries, it equally hurts performance a lot for others. Stemming increases recall while harming precision. For example, the Porter stemmer stems all of the following words:

operate operating operates operation operative operatives operational

to *oper*. However, since operate in its various forms is a common verb, we would expect to lose precision on queries such as:

operative (*dentistry*) ⇒ *oper*

The situation is different for languages with much more morphology, such as Spanish, German, and Finnish.

2.3 Faster postings list intersection via skip pointers

In the basic postings list intersection, ff the list lengths are m and n, the intersection takes $O(m+n)$ operations. One way to do better than this is to use a **skip list** by augmenting postings lists with skip pointers (at indexing time). Skip pointers allow us to avoid processing parts of the postings list that will not figure in the search results.

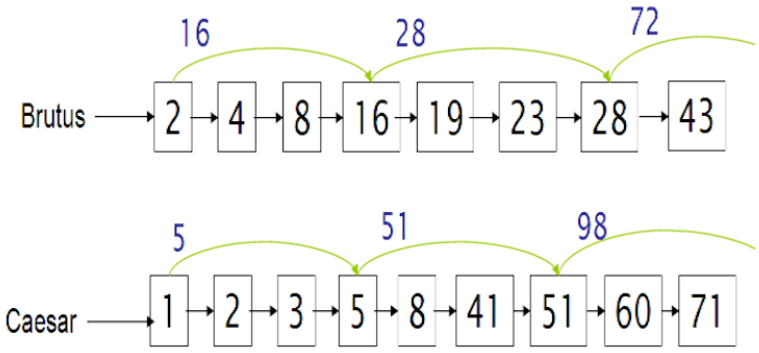

Figure 4: Postings lists with skip pointers.

Where do we place skips? There is a trade-off. More skips means shorter skip spans, and we are more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparisons, but then long skip spans which means that there will be fewer opportunities to skip. A simple heuristic is that for a postings list of length P, use $\sqrt{P}$ evenly-spaced skip pointers. Building effective skip pointers is easy if an index is relatively static; it is harder if a postings list keeps changing because of updates.

2.4 Positional postings and phrase queries

Most recent search engines support a double quotes syntax (*stanford university*) for **phrase queries**, which has proven to be very easily understood and successfully used by users. To be able to support such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms. There are two approaches.

Biword indexes One approach is to consider every pair of consecutive terms in a document as a phrase. For example, the text *Friends*, *Romans*, *Countrymen* would generate the **biwords**:

```
friends romans
romans countrymen
```

We treat each of these biwords as a vocabulary term. First, we tokenize the text and perform part-of-speech-tagging. We can then group terms into nouns, including proper nouns, (N) and function words, including articles and prepositions, (X), among other classes. Any string of terms of the form $NX*N$ is an extended biword. Each such extended biword is made a term in the vocabulary. To process a query using such an extended biword index, we need to also parse it into N's and X's, and then segment the query into extended biwords.

This algorithm does not always work in an intuitively optimal manner when parsing longer queries into Boolean queries. Using the above algorithm, the query

```
cost overruns on a power plant
```

is parsed into

```
“cost overruns” AND “overruns power” AND “power plant”
```

whereas it might seem a better query to omit the middle biword.

Searches for a single term are not naturally handled in a biword index, and so we also need to have an index of single-word terms. But storing longer phrases has the potential to greatly expand the vocabulary size.

Positional indexes A biword index is not the standard solution, a **positional index** is most commonly employed. Here, for each term in the vocabulary, we store postings of the form docID: $\langle position1, position2, \ldots \rangle$ where each position is a token index in the document. Each posting will also usually record the term frequency.

To process a phrase query, you still need to access the inverted index. As before, you would start with the least frequent term and then restrict the list of possible candidates. In the merge operation, the same general technique is used as before, but you also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. The same general method is applied for within k word proximity searches:

```
employment /3 place
```

$/k$ means "within k words of". Positional indexes can be used for such queries; biword indexes cannot.

Adopting a positional index expands required postings storage significantly. Moving to a positional index also changes the asymptotic complexity of a postings intersection operation, because the number of items to check is now bounded not by the number of documents but by the total number of tokens in the document collection T: the complexity of a Boolean query is $\Theta(T)$ rather than $\Theta(N)$.

Combination schemes The strategies of biword indexes and positional indexes can be fruitfully combined. If users commonly query on particular phrases, such as `Michael Jackson`, it is quite inefficient to keep merging positional postings lists. A combination strategy uses a phrase index, or just a biword index, for certain queries and uses a positional index for other phrase queries.

3 Dictionaries and tolerant retrieval

We will discuss techniques that are robust to typographical errors in the query, as well as alternative spellings. A **wildcard query** * symbol indicates any (possibly empty) string of characters: a query such as `*a*e*i*o*u*`, which seeks documents containing any term that includes all the five vowels in sequence.

3.1 Search structures for dictionaries

Given an inverted index and a query, our first task is to determine whether each query term exists in the vocabulary and if so, identify the pointer to the corresponding postings. This vocabulary lookup operation uses a classical data structure called the dictionary (hashing or search trees).

Hashing has been used for dictionary lookup in some search engines. Each vocabulary term (**key**) is hashed into an integer over a large enough space that hash collisions are unlikely. At query time, we hash each query term separately and following a pointer to the corresponding postings. There is no easy way to find minor variants of a query term since these could be hashed to very different integers. In a setting (such as the Web) where the size of the vocabulary keeps growing, a hash function designed for current needs may not scale well.

Search trees overcome many of these issues. The best-known search tree is the **binary tree**: the search for a term begins at the root of the tree. Each internal node (including the root) represents a binary test. Efficient search (with a number of comparisons that is $O(\log M)$) hinges on the tree being balanced: the numbers of terms under the two sub-trees of any node are either equal or differ by one. The principal issue here is that of rebalancing: as terms are inserted into or deleted from the binary search tree, it needs to be rebalanced so that the balance property is maintained.

To mitigate rebalancing, one approach is to allow the number of sub-trees under an internal node to vary in a fixed interval. A search tree commonly used is the **B-tree**, in which every internal node has a number of children in the interval $[a, b]$, where a and b are positive integers.

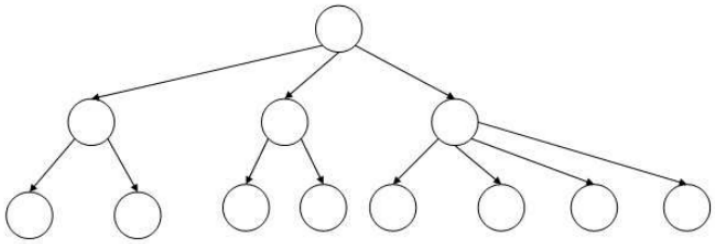

Figure 5: A B-tree. In this example every internal node has between 2 and 4 children.

Unlike hashing, search trees demand that the characters used in the document collection have a prescribed ordering.

3.2 Wildcard queries

Wildcard queries are used in any of the following situations: (1) the user is uncertain of the spelling of a query term (e.g., `Sydney` vs. `Sidney`, which leads to the wildcard query `S*dney`); (2) the user is aware of multiple variants of spelling a term and (consciously) seeks documents containing any of the variants (e.g., `color` vs. `colour`); (3) the user seeks documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming stemming (e.g., `judicial` vs. `judiciary`, the wildcard query `judicia*`); (4) the user is uncertain of the correct rendition of a foreign word or phrase (e.g., the query `Universit* Stuttgart`).

A query such as `mon*` is known as a **trailing wildcard query**, because the * symbol occurs only once, at the end of the search string. We walk down the tree following the symbols `m`, `o` and `n` in turn, for each we can enumerate the set W of terms in the dictionary with the prefix `mon`. Finally, we use $|W|$ lookups on the standard inverted index to retrieve all documents containing any term in W.

Consider now **leading wildcard queries**, queries of the form `*mon`. Consider a **reverse B-tree** on the dictionary, one in which each root-to-leaf path of the B-tree corresponds to a term in the dictionary written **backwards**: the term `lemon` would, in the B-tree, be represented by the path `root-n-o-m-e-l`.

Using a regular B-tree together with a reverse B-tree, we can handle an even more general case: wildcard queries in which there is a single * symbol, such as `se*mon`.The regular B-tree is used to enumerate the set W of dictionary terms beginning with the prefix `se`, then the reverse B-tree to enumerate the set R of terms ending with the suffix `mon`. Next, we take the intersection $W \cap R$ of these two sets. Finally, we use the standard inverted index to retrieve all documents containing any terms in this intersection.

General wildcard queries There are two techniques for handling general wildcard queries. Both techniques share a common strategy: express the given wildcard query q_w as a Boolean query Q on a specially constructed index, such that the answer to Q is a superset of the set of vocabulary terms matching q_w. Then, we check each term in the answer to Q against q_w, discarding those vocabulary terms that do not match q_w.

The first special index for general wildcard queries is the **permuterm index**, a form of inverted index. First, we introduce a special symbol \$, used to mark the end of a term. The term `hello` is shown here as the augmented term `hello$`. Next, we construct a permuterm index, in which the various rotations of each term (augmented with \$) all link to the original vocabulary term. The set of rotated terms in the permuterm index is the permuterm vocabulary.

Consider the wildcard query `m*n`. The key is to rotate such a wildcard query so that the * symbol appears at the end of the string, `n$m*`. Next, we look up this string in the permuterm index, where seeking `n$m*` (via a search tree) leads to rotations of (among others) the terms `man` and `moron`. Now we look up these terms in the standard inverted index to retrieve matching documents. We can thus handle any wildcard query with a single * symbol. In query such as `fi*mo*er`, we first enumerate the terms in the dictionary that are in the permuterm index of `er$fi*`, then we filter out the entries that don't contain `mo`. One disadvantage of the permuterm index is that its dictionary becomes quite large.

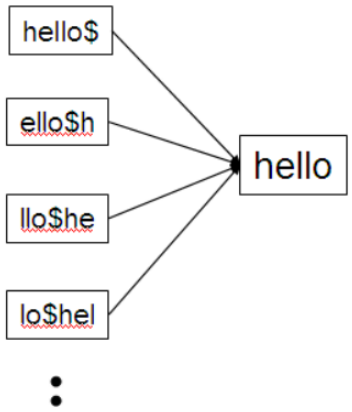

Figure 6: A portion of a permuterm index.

k-gram indexes for wildcard queries A **k-gram** is a sequence of k characters. For example `cas`, `ast` and `stl` are all 3-grams occurring in the term `castle`. A special character \$ is used to denote the beginning or end of a term, so the full set of 3-grams generated for `castle` is: `$ca`, `cas`, `ast`, `stl`, `tle`, `le$`.

In a k-gram index, the dictionary contains all k-grams that occur in any term in the vocabulary. Each postings list points from a k-gram to all vocabulary terms containing that k-gram. For instance, the 3-gram `etr` would point to vocabulary terms such as `metric` and `retrieval`.

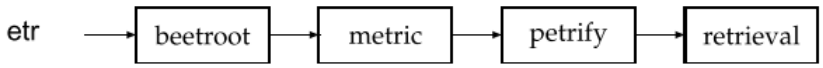

Figure 7: Example of a postings list in a 3-gram index.

There is a difficulty with the use of k-gram indexes, that demands one further step of processing. Consider the query `red*`. We first issue the Boolean query `$re AND red` to the 3-gram index. This leads to a match on terms such as `retired`, which contain the conjunction of the two 3grams `$re` and `red`, yet do not match the original wildcard query `red*`. To handle this, we introduce a **post-filtering** step, in which the terms enumerated by the Boolean query on the 3-gram index are checked individually against the original query. Terms that survive are then searched in the standard inverted index.

The processing of a wildcard query can be quite expensive because of the added lookup in the special index, filtering and finally the standard inverted index. A search engine may support such rich functionality, but most commonly, the capability is hidden behind an interface that most users never use.

3.3 Spelling correction

There are two methods to solve the problem of correcting spelling errors in queries: the first based on **edit distance** and the second based on **k-gram overlap**.

Implementing spelling correction There are two basic principles underlying most spelling correction algorithms.

1. Of various alternative correct spellings for a mis-spelled query, choose the "nearest" one.
2. When two correctly spelled queries are tied (or nearly tied), select the one that is more common.

Spelling correction algorithms are exposed to users in one of several ways:

1. On the query `carot` always retrieve documents containing `carot` as well as any "spell-corrected" version of `carot`, including `carrot` and `tarot`.
2. As in (1) above, but only when the query term `carot` is not in the dictionary.
3. As in (1) above, but only when the original query returned fewer than a preset number of documents.
4. When the original query returns fewer than a preset number of documents, the search interface presents a spelling suggestion to the end user: "Did you mean carrot?"

Forms of spelling correction We focus on two specific forms of spelling correction, **isolated-term correction** and **context-sensitive correction**. In isolated-term correction, we attempt to correct a single query term at a time. Such isolated-term correction would fail to detect, for instance, that the query `flew form Heathrow` contains a mis-spelling of the term `from`, because each term in the query is correctly spelled in isolation.

Edit distance Given two character strings s_1 and s_2, the **edit distance** between them is the minimum number of edit operations required to transform s_1 into s_2. The edit operations allowed for this purpose are: (i) insert a character into a string; (ii) delete a character from a string and (iii) replace a character of a string by another character; edit distance is known as **Levenshtein distance**. For example, the edit distance between `cat` and `dog` is 3.

To compute the (weighted) edit distance between two strings in time $O(|s_1| \times |s_2|)$, where $|s_i|$ denotes the length of a string s_i. The idea is to use the dynamic programming.

The spelling correction problem however demands more than computing edit distance: given a set S of strings and a query string q, we seek the $string(s)$ in V of least edit distance from q. The obvious way of doing this is to compute the edit distance from q to each string in V, before selecting the $string(s)$ of minimum edit distance. This exhaustive search is expensive.

The simplest heuristic is to restrict the search to dictionary terms beginning with the same letter as the query string. A more sophisticated variant of this heuristic is to use a version of the permuterm index.

k-gram indexes for spelling correction To further limit the set of vocabulary terms for which we compute edit distances to the query term we can use the k-gram index to assist with retrieving vocabulary terms with low edit distance to the query q. We will use the k-gram index to retrieve vocabulary terms that have many k-grams in common with the query. The retrieval process is essentially that of a single scan through the postings for the k-grams in the query string q.

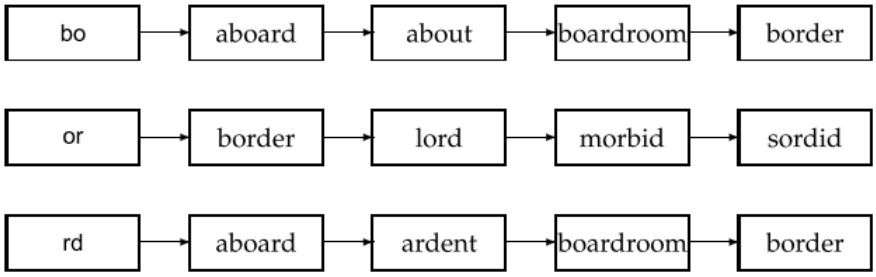

Figure 8: Matching at least two of the three 2-grams in the query `bord`.

The linear scan intersection can be adapted when the measure of overlap is the **Jaccard coefficient** for measuring the overlap between two sets A and B, defined to be $|A \cup B|/|A \cup B|$. The two sets we consider are the set of k-grams in the query q, and the set of k-grams in a vocabulary term. The scan proceeds from one vocabulary term t to the next, computing on the fly the Jaccard coefficient between q and t. If the coefficient exceeds a preset threshold, we add t to the output; if not, we move on to the next term in the postings. To compute the Jaccard coefficient, we only need the length of the string t.

We could replace the Jaccard coefficient by other measures that allow efficient on the fly computation during postings scans. We can use the k-gram index to enumerate a set of candidate vocabulary terms that are potential corrections of q. We then compute the edit distance from q to each term in this set, selecting terms from the set with small edit distance to q.

Context sensitive spelling correction Isolated-term correction would fail to correct typographical errors where all query terms are correctly spelled. The simplest way to correct these errors is to enumerate corrections of each of the query terms even though each query term is correctly spelled,

then try substitutions of each correction in the phrase. For the example `flew form Heathrow`, we enumerate such phrases as `fled form Heathrow` and `flew fore Heathrow`. For each such substitute phrase, the search engine runs the query and determines the number of matching results. This enumeration can be expensive, several heuristics are used to reduce the search space.

3.4 Phonetic correction

Phonetic correction: misspellings that arise because the user types a query that sounds like the target term. Such algorithms are especially applicable to searches on the names of people. The main idea here is to generate, for each term, a **phonetic hash** so that similar-sounding terms hash to the same value.

Algorithms for such phonetic hashing are commonly collectively known as **soundex** algorithms. The common steps are:

1. Turn every term to be indexed into a 4-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index.
2. Do the same with query terms.
3. When the query calls for a soundex match, search this soundex index.

A commonly used conversion results in a 4-character code, with the first character being a letter of the alphabet and the other three being digits between 0 and 9.

1. Retain the first letter of the term.
2. Change all occurrences of the following letters to '0' (zero): 'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:
   B, F, P, V to 1.
   C, G, J, K, Q, S, X, Z to 2.
   D,T to 3.
   L to 4.
   M, N to 5.
   R to 6.
4. Repeatedly remove one out of each pair of consecutive identical digits.
5. Remove all zeros from the resulting string. Pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits.

This algorithm rests on a few observations: (1) vowels are viewed as interchangeable, in transcribing names; (2) consonants with similar sounds are put in equivalence classes.

4 Index construction

The construction process of an inverted index is called **index construction** or **indexing**; the process or machine that performs it the **indexer**.

4.1 Hardware basics

When building an information retrieval (IR) system, many decisions are based on the characteristics of the computer hardware. There are some considerations:

- Access to data in memory is much faster than access to data on disk, so we want to keep as much data as possible in memory. The technique of keeping frequently used disk data in main memory **caching**.
- When doing a disk read or write, it takes a while for the disk head to move to the part of the disk where the data are located. This time is called the **seek time**. No data are being transferred during the seek. To maximize data transfer rates, chunks of data that will be read together should therefore be stored contiguously on disk.
- Operating systems generally read and write entire blocks. Thus, reading a single byte from disk can take as much time as reading the entire block.
- Data transfers from disk to memory are handled by the system bus, not by the processor. This means that the processor is available to process data during disk I/O. We can speed up data transfers by storing compressed data on disk.

4.2 Blocked sort-based indexing

First, we introduce the basic steps in constructing a nonpositional index. We first make a pass through the collection assembling all term–docID pairs. We then sort the pairs with the term as the dominant key and docID as the secondary key. Finally, we organize the docIDs for each term into a postings list and compute statistics like term and document frequency. For small collections, all this can be done in memory.

To make index construction more efficient, we represent terms as termIDs, where each **termID** is a unique. We can build the mapping from terms to termIDs on the fly while we are processing the collection; or, in a two-pass approach, we compile the vocabulary in the first pass and construct the inverted index in the second pass.

We work with the *Reuters-RCV1* collection as our model collection with roughly 1 GB of text. It consists of about 800,000 documents. Reuters-RCV1 has 100million tokens. Collecting all termID–docID pairs of the collection using 4 bytes each for termID and docID therefore requires 0.8 GB of storage.

With main memory insufficient, we need to use an **external sorting algorithm**. For acceptable speed, the central requirement of such an algorithm is that it minimize the number of random disk seeks during sorting. One solution is the **blocked sort-based indexing algorithm** or **BSBI**. BSBI (i) segments the collection into parts of equal size, (ii) sorts the termID–docID pairs of each part in memory, (iii) stores intermediate sorted results on disk, and (iv) merges all intermediate results into the final index.

The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full. The block is then inverted and written to disk. **Inversion** involves two steps. First, we sort the termID–docID pairs. Next, we collect all termID–docID pairs with the same termID into a postings list, where a **posting** is simply a docID. The result is then written to disk. In the final step, the algorithm simultaneously merges the ten blocks into one large merged index.

Algorithm 2 Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.

```
1: function BSBI_CONSTRUCTION
2:     n ← 0
3:     while all documents have not been processed do
4:         n ← n + 1
5:         block ← PARSE_NEXT_BLOCK()
6:         BSBI_INVERT(block)
7:         WRITE_BLOCK_TO_DISK(block, f_n)
8:     end while
9:     MERGE_BLOCKS(f_1, ..., f_n, f_merged)
10: end function
```

How expensive is BSBI? Its time complexity is $\Theta(T \log T)$ because the step with the highest time complexity is sorting and T is an upper bound for the number of items we must sort. But the actual indexing time is usually dominated by the time it takes to parse the documents (`PARSE_NEXT_BLOCK`) and to do the final merge (`MERGE_BLOCKS`).

4.3 Single-pass in-memory indexing

Blocked sort-based indexing has excellent scaling properties, but it needs a data structure for mapping terms to termIDs. For very large collections, this data structure does not fit into memory. A more scalable alternative is **single-pass in-memory indexing** or **SPIMI**. SPIMI uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. `SPIMI_INVERT` is called repeatedly on the token stream until the entire collection has been processed. Tokens are processed one by one during each successive call of `SPIMI_INVERT`. When a term occurs for the first time, it is added to the dictionary, and a new postings list is created. At the end it returns this postings list for subsequent occurrences of the term.

A difference between BSBI and SPIMI is that SPIMI adds a posting directly to its postings list. Instead of first collecting all termID–docID pairs and then sorting them, each postings list is dynamic and it is immediately available to collect postings. This has two advantages: it is faster because there is no sorting, and it saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.

Because we do not know how large the postings list of a term will be when we first encounter it, we allocate space for a short postings list initially and double the space each time it is full.

When memory has been exhausted, we write the index of the block to disk. We have to sort the terms before doing this because we want to write postings lists in lexicographic order to facilitate the final merging step. If each block's postings lists were written in unsorted order, merging blocks could not be accomplished by a simple linear scan.

Each call of `SPIMI_INVERT` writes a block to disk, just as in BSBI. The last step of SPIMI is then to merge the blocks into the final inverted index.

SPIMI has a third important component: compression. Both the postings and the dictionary terms can be stored compactly on disk.

The time complexity of SPIMI is $\Theta(T)$ because no sorting of tokens is required and all operations are at most linear in the size of the collection.

Algorithm 3 Inversion of a block in single-pass in-memory indexing.

```
1: function SPIMI_INVERT(token_stream)
2:     output_file = NEW_FILE()
3:     dictionary = NEW_HASH()
4:     while free memory available do
5:         token ← next(token_stream)
6:         if term(token) ∉ dictionary then
7:             postings_list = ADD_TO_DICTIONARY(dictionary, term(token))
8:         else
9:             postings_list = GET_POSTINGS_LIST(dictionary, term(token))
10:            p2 ← next(p1)
11:        end if
12:        if full(postings_list) then
13:            postings_list = DOUBLE_POSTINGS_LIST(dictionary, term(token))
14:        end if
15:        ADD_TOPOSTINGS_LIST(postings_list, docID(token))
16:    end while
17:    sorted_terms ← SORT_TERMS(dictionary)
18:    WRITE_BLOCK_TO_DISK(sorted_terms, dictionary, output_file)
19:    return output_file
20: end function
```

4.4 Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer **clusters** to construct any reasonably sized web index. Web search engines, therefore, use **distributed indexing algorithms** for index construction. The result of the construction process is a distributed index that is partitioned across several machines.

The distributed index construction method is an application of **MapReduce**, a general architecture for distributed computing. The point of a cluster is to solve large computing problems on **nodes** that are built from standard parts as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in clusters, individual machines can fail at any time. A robust distributed indexing needs the work to be divided in re-assignable chunks. A **master node** directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases split up the computing job into chunks that standard machines can process in a short time. First, the input data, are split into *nsplits* where the size of the split is chosen to ensure that the work can be distributed evenly and efficiently. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: as a machine finishes processing one split, it is assigned the next one. If a machine dies, the split it is working on is simply reassigned to another machine.

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of **key-value pairs**, in the form of $\langle termID, docID \rangle$. In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex. A simple solution is to maintain a mapping for frequent terms that is copied to all nodes and to use terms directly for infrequent terms.

The **map phase** consists of mapping splits of the input data to key-value pairs. The machines that execute the map phase are called **parser**. Each parser writes its output to local intermediate files, the **segment files**.

For the **reduce phase**, we want all values for a given key to be stored close together. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. Each term partition thus corresponds to r segments files, where r is the number of parsers.

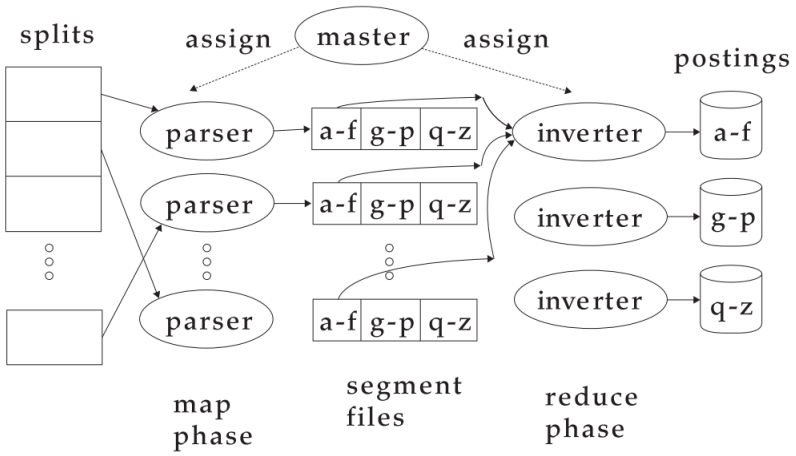

Figure 9: An example of distributed indexing with MapReduce.

Collecting all values for a given key into one list is the task of the **inverters**. The master assigns each term partition to a different inverter. Finally, the list of values is sorted for each key and written to the final sorted postings list.

Parsers and inverters are not separate sets of machines. The master identifies idle machines and assigns tasks to them. The same machine can be a parser in the map phase and an inverter in the reduce phase.

To minimize write times before inverters reduce the data, each parser writes its segment files to its **local disk**. In the reduce phase, the master communicates to an inverter the locations of the relevant segment files. Each segment file only requires one sequential read. This setup minimizes the amount of network traffic needed during indexing. Input and output are often lists of key-value pairs themselves, so that several MapReduce jobs can run in sequence.

Schema of map and reduce functions

map: input	$\rightarrow$ list(k, v)
reduce: (k,list(v))	$\rightarrow$ output

Instantiation of the schema for index construction

map: web collection	$\rightarrow$ list(termID, docID)
reduce: ($\langle$termID$_1$, list(docID)$\rangle$, $\langle$termID$_2$, list(docID)$\rangle$, ...)	$\rightarrow$ (postings_list$_1$, postings_list$_2$, ...)

Example for index construction

map: d_2 : C died. d_1 : C came, C c'ed.	$\rightarrow$ ($\langle$C,$d_2\rangle$, $\langle$died,$d_2\rangle$, $\langle$C,$d_1\rangle$, $\langle$came,$d_1\rangle$, $\langle$C,$d_1\rangle$, $\langle$c'ed,$d_1\rangle$)
reduce: ($\langle$C,(d_2,d_1,d_1)$\rangle$,$\langle$died,(d_2)$\rangle$,$\langle$came,(d_1)$\rangle$,$\langle$c'ed,(d_1)$\rangle$)	$\rightarrow$ ($\langle$C,(d_1:2,d_2:1)$\rangle$,$\langle$died,(d_2:1)$\rangle$,$\langle$came,(d_1:1)$\rangle$,$\langle$c'ed,(d_1:1)$\rangle$)

Figure 10: Map and reduce functions in MapReduce.

4.5 Dynamic indexing

Most collections are modified frequently with documents being added, deleted, and updated. New terms need to be added to the dictionary, and postings lists need to be updated.

The simplest way to achieve this is to periodically reconstruct the index from scratch. This is a good solution if the number of changes over time is small and if enough resources are available to construct a new index while the old one is still available for querying.

If there is a requirement that new documents be included quickly, one solution is to maintain two indexes: a large main index and a small **auxiliary index** that stores new documents. The auxiliary index is kept in memory. Searches are run across both indexes. Deletions are stored in an invalidation bit vector. We can then filter out deleted documents before returning the search result.

Each time the auxiliary index becomes too large, we merge it into the main index. The cost of this merging operation depends on how we store the index. If we store each postings list as a separate file, then the merge consists of extending each postings list of the main index by the corresponding postings list of the auxiliary index.

Unfortunately, the one-file-per-postings-list scheme is infeasible because most file systems cannot efficiently handle very large numbers of files. The simplest alternative is to store the index as one large file as a concatenation of all postings lists. In reality, we often choose a compromise between the two extremes.

In a scheme where the index is one big file, we process each posting $\lfloor T/n \rfloor$ times because we touch it during each of $\lfloor T/n \rfloor$ merges where n is the size of the auxiliary index and T the total number of postings. Thus, the time complexity is $\Theta(T^2/n)$. We can do better with a scheme called **logarithmic merging**. Logarithmic merging amortizes the cost of merging indexes over time. As before, up to n postings are accumulated in an in-memory auxiliary index, which we call Z_0. When the limit n is reached, the $2^0 \times n$ postings in Z_0 are transferred to a new index I_0 that is created on disk. The next time Z_0 is full, it is merged with I_0 to create an index Z_1 of size $2^1 \times n$. Then Z_1 is either stored as I_1 (if there isn't already an I_1) or merged with I_1 into Z_2 (if I_1 exists); and so on.

Overall index construction time is $\Theta(T \log(T/n))$ because each posting is processed only once on each of the $\log(T/n)$ levels. We trade this efficiency gain for a slow down of query processing; we now need to merge results from $\log(T/n)$. We still need to merge very large indexes occasionally, but this happens less frequently.

Because of this complexity of dynamic indexing, some large search engines adopt a reconstruction-from-scratch strategy. They do not construct indexes dynamically. Instead, a new index is built from scratch periodically. Query processing is then switched from the new index and the old index is deleted.

5 Index compression

Compression techniques for dictionary and inverted index are essential for efficient IR systems. One benefit of compression is straightforward: we need less disk space.

There are two more benefits of compression. The first is increased use of caching. Search systems use some parts of the dictionary much more than others. If we cache the postings list of a frequently used query term t, then the computations necessary for responding to t can be entirely done in memory. With compression, we can fit a lot more information into main memory.

The second advantage of compression is faster transfer of data from disk to memory. Efficient decompression algorithms run so fast on modern hardware that the total time of transferring a compressed chunk of data from disk and then decompressing it is usually less than transferring

the same chunk of data in uncompressed form. In most cases, the retrieval system runs faster on compressed postings lists than on uncompressed postings lists.

We define a **posting** as a docID in a postings list. For example, the postings list (6; 20, 45, 100), where 6 is the termID, contains three postings.

5.1 Statistical properties of terms in information retrieval

Reuters-RCV1 will be our model collection. 11 shows the preprocessing affects the size of the dictionary and the number of nonpositional postings. Stemming and case folding reduce the number of (distinct) terms by 17% each and the number of nonpositional postings by 4% and 3%, respectively. Eliminating the 150 most common words from indexing cuts 25% to 30% of the nonpositional postings. But this size reduction does not carry over to the size of the compressed index. The postings lists of frequent words require only a few bits per posting after compression. However, the the percentage reductions can be very different for some text collections. For example, for a collection of web pages with a high proportion of French text, a lemmatizer for French reduces vocabulary size much more than the Porter stemmer does for an English-only collection because French is a morphologically richer language than English.

size of	word types (term)			non-positional postings			positional postings (word tokens)		
	dictionary			non-positional index			positional index		
	size	Δ	cml..	size	Δ	cml..	size	Δ	cml..
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2%	-2%	100,680,242	-8%	-8%	179,158,204	-9%	-9%
case folding	391,523	-17%	-19%	96,969,056	-3%	-12%	179,158,204	-0%	-9%
30 stop w's	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop w's	391,373	-0%	-19%	67,001,847	-30%	-39%	94,516,599	-47%	-52%
stemming	322,383	-17%	-33%	63,812,300	-4%	-42%	94,516,599	-0%	-52%

Figure 11: The effect of preprocessing on the number of terms, nonpositional postings, and tokens for Reuters-RCV1.

The compression techniques are **lossless**, where all information is preserved, and **lossy**, which discards some information. Case folding, stemming, and stop word elimination are forms of lossy compression. The vector space model and dimensionality reduction techniques like latent semantic indexing create compact representations from which we cannot fully restore the original collection. Lossy compression makes sense when the "lost" information is unlikely ever to be used.

Before introducing techniques for compressing the dictionary, it's useful to estimate the number of distinct terms M in a collection. The second edition of the *Oxford English Dictionary* (OED) defines more than 600,000 words, but it does not include most names of people, locations, products, etc. These names need to be included in the inverted index.

5.2 Heaps' law: Estimating the number of terms

A better way of getting a handle on M is **Heaps' law**, which estimates vocabulary size as a function of collection size:

$$M = kT^b$$

where T is the number of tokens in the collection. Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$.

Heaps' law suggests that (i) the dictionary size continues to increase with more documents in the collection and (ii) the size of the dictionary is quite large for large collections.

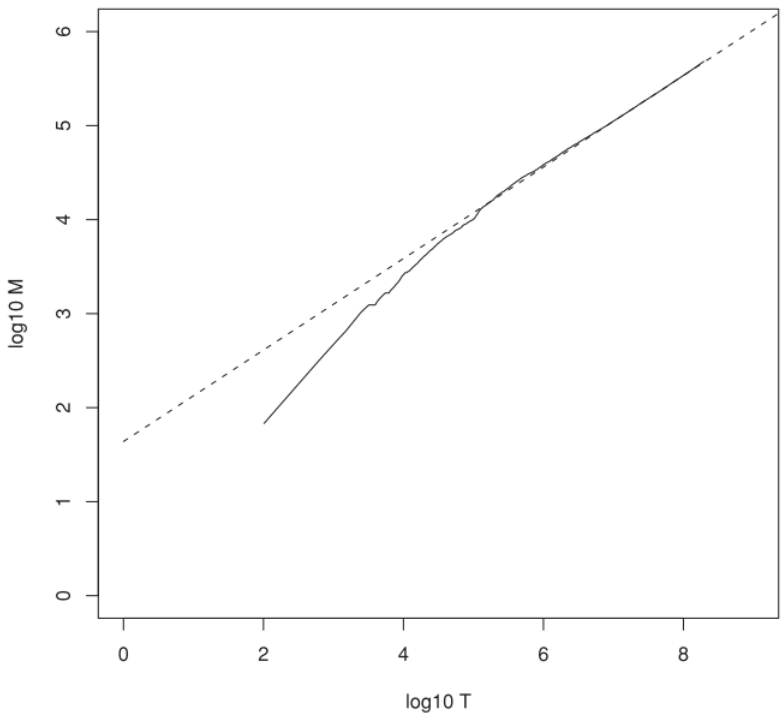

Figure 12: Heaps' law. Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1.

5.3 Zipf's law: Modeling the distribution of terms

We also want to understand how terms are distributed across documents. A commonly used model of the distribution of terms in a collection is **Zipf's law**. It states that, if t_1 is the most common term in the collection, t_2 is the next most common, and so on, then the collection frequency cf_i of the ith most common term is proportional to $1/i$:

$$cf_i \propto \frac{1}{i}.$$

So if the most frequent term occurs cf_i times, then the second most frequent term has half as many occurrences, the third most frequent term a third as many occurrences, and so on. The frequency decreases very rapidly with rank.

Equivalently, we can write Zipf's law as $cf_i = ci^k$ or as $\log cf_i = \log c + k \log i$ where $k = -1$ and c is a constant. It is therefore a power law with exponent $k = -1$.

The fit of the data to the law is not particularly good, but good enough to serve as a model for term distributions.

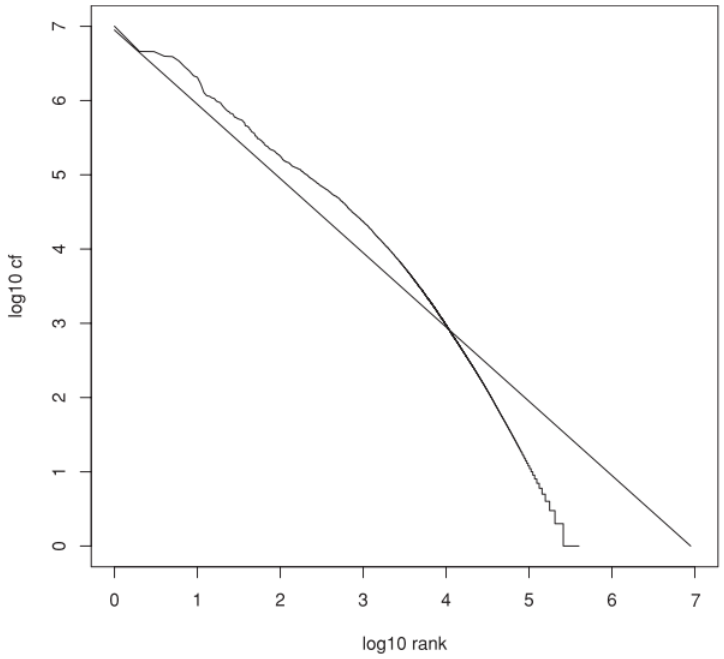

Figure 13: Zipf's law for Reuters-RCV1. Frequency is plotted as a function of frequency rank for the terms in the collection. The line is the distribution predicted by Zipf's law.

5.4 Dictionary compression

One of the primary factors in determining the response time of an IR system is the number of disk seeks necessary to process a query. If parts of the dictionary are on disk, then many more disk seeks are necessary in query evaluation. Thus, the main goal of compressing the dictionary is to fit it in main memory,

Dictionary as a string The simplest data structure for the dictionary is to sort the vocabulary lexicographically and store it in an array of fixed-width entries. We allocate 20 bytes for the term itself, 4 bytes for its document frequency, and 4 bytes for the pointer to its postings list. Four-byte pointers resolve a 4 GB address space. For large collections like the web, we need to allocate more bytes per pointer. We look up terms in the array by binary search. For Reuters-RCV1, we need $M \times (20 + 4 + 4) = 400,000 \times 28 = 11.2$ MB for storing the dictionary in this scheme.

	term	document frequency	pointer to postings list
	a	656,265	⟶
	aachen	65	⟶
	...	...	...
	zulu	221	⟶
space needed:	20 bytes	4 bytes	4 bytes

Figure 14: Storing the dictionary as an array of fixed-width entries.

Using fixed-width entries for terms is wasteful. The average length of a term in English is about eight characters. Also, we have no way of storing terms with more than twenty characters like

hydrochlorofluorocarbons and supercalifragilisticexpialidocious. We can overcome these shortcomings by storing the dictionary terms as one long string of characters. The pointer to the next term is also used to demarcate the end of the current term. This scheme saves us 60% compared to fixed-width storage (12 bytes on average of the 20 bytes). However, we now also need to store term pointers. The term pointers resolve $400,000 \times 8 = 3.2 \times 10^6$ positions, so they need to be $\log_2 3.2 \times 10^6 \approx 22$ bits or 3 bytes long.

In this new scheme, we need $400,000 \times (4+4+3+8) = 7.6$ MB for the Reuters-RCV1 dictionary: 4 bytes each for frequency and postings pointer, 3 bytes for the term pointer, and 8 bytes on average for the term. So we have reduced the space from 11.2 to 7.6 MB.

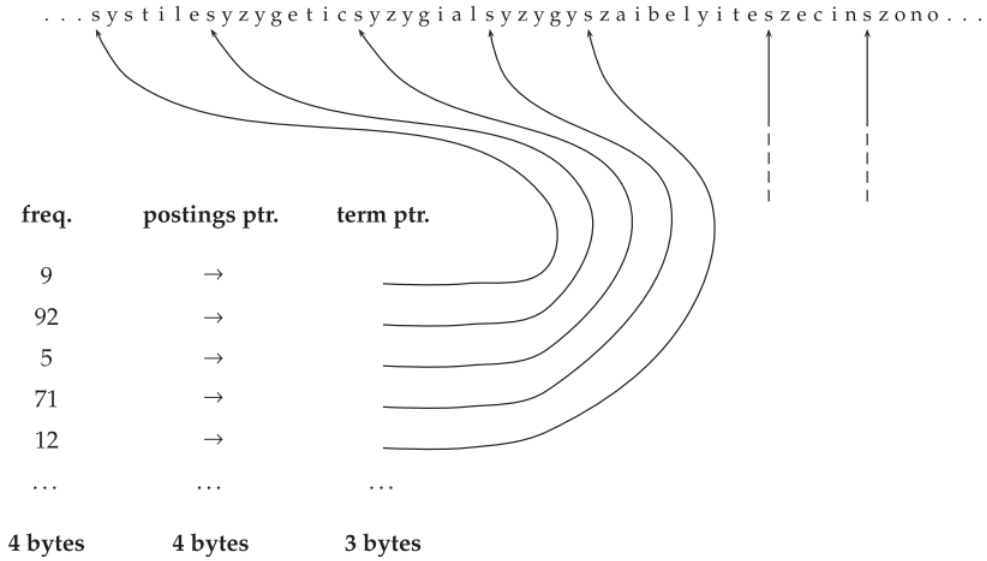

Figure 15: Dictionary-as-a-string storage. Pointers mark the end of the preceding term and the beginning of the next. For example, the first three terms in this example are systile, syzygetic, and syzygial.

Blocked storage We can further compress the dictionary by grouping terms in the string into blocks of size k and keeping a term pointer only for the first term of each block. We store the length of the term in the string as an additional byte at the beginning of the term. We thus eliminate $k-1$ term pointers, but need an additional k bytes for storing the length of each term. For $k = 4$, we save $(k-1) \times 3 = 9$ bytes for term pointers, but need an additional $k = 4$ bytes for term lengths. So the total space requirements for the dictionary of Reuters-RCV1 are reduced by 0.5 MB, to 7.1 MB.

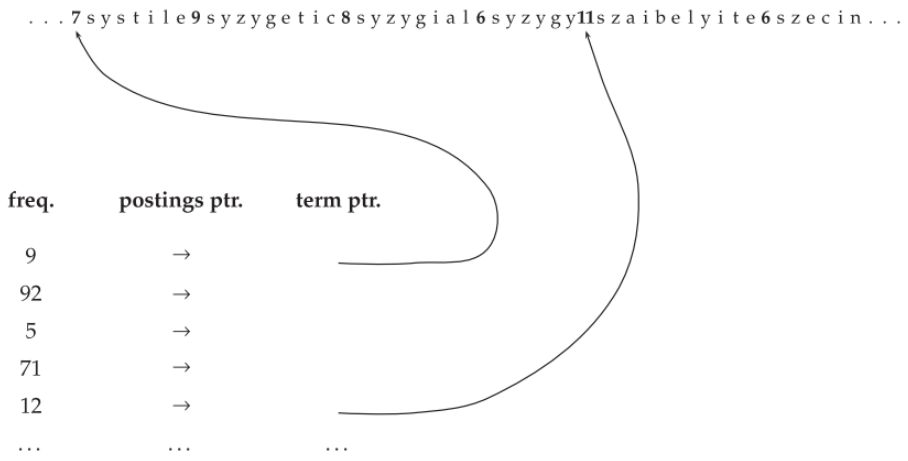

Figure 16: Blocked storage with four terms per block. The first block consists of `systile`, `syzygetic`, `syzygial`, and `syzygy` with lengths of seven, nine, eight, and six characters, respectively. Each term is preceded by a byte encoding its length that indicates how many bytes to skip to reach subsequent terms.

By increasing the block size k, we get better compression. However, there is a tradeoff between compression and the speed of term lookup. We search for terms in the uncompressed dictionary by binary search (a). In the compressed dictionary, we first locate the term's block by binary search and then its position within the list by linear search through the block (b). Searching the uncompressed dictionary in (a) takes on average $(0+1+2+3+2+1+2+2)/8 \approx 1.6$ steps. With blocks of size k $= 4$ in (b), we need $(0+1+2+3+4+1+2+3)/8 = 2$ steps on average, $\approx 25\%$ more.

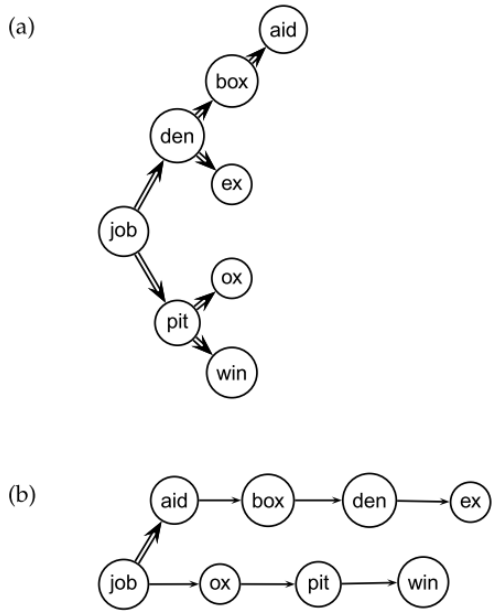

Figure 17: Search of the uncompressed dictionary (a) and a dictionary compressed by blocking with $k = 4$ (b).

Consecutive entries in an alphabetically sorted list share common prefixes. This observation leads to **front coding**. A common prefix is identified for a subsequence of the term list and then referred to with a special character. In the case of Reuters, front coding saves another 1.2 MB.

One block in blocked compression ($k = 4$) ...
8automata**8**automate**9**automatic**10**automation

⇓

...further compressed with front coding.
8automat∗a**1**⋄e**2**⋄ic**3**⋄ion

Figure 18: Front coding. A sequence of terms with identical prefix ("automat") is encoded by marking the end of the prefix with ∗ and replacing it with ⋄ in subsequent terms. As before, the first byte of each entry encodes the number of characters.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9

Figure 19: Dictionary compression for Reuters-RCV1.

5.5 Postings file compression

Reuters-RCV1 has 800,000 documents, 200 tokens per document, six characters per token, and 100,000,000 postings. Document identifiers are $\log_2 800,000 \approx 20$ bits long. Thus, the size of the collection is about $800,000 \times 200 \times 6$ bytes = 960 MB and the size of the uncompressed postings file is $100,000,000 \times 20/8 = 250$ MB.

To devise a more efficient representation of the postings file, one that uses fewer than 20 bits per document, we observe that the postings for frequent terms are close together. The key idea is that the **gaps** between postings are short. In fact, gaps for the most frequent terms are mostly equal to 1. But the gaps for a rare term need 20 bits. For an economical representation of this distribution of gaps, we need a variable encoding method that uses fewer bits for short gaps. To encode small numbers in less space than large numbers, we look at two types of methods: bytewise compression and bitwise compression.

Variable byte codes **Variable byte** (**VB**) **encoding** uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a **continuation bit**. It is set to 1 for the last byte of the encoded gap and to 0 otherwise. To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9

Figure 20: VB encoding. Gaps are encoded using an integral number of bytes. The first bit, the continuation bit, of each byte indicates whether the code ends with this byte (1) or not (0).

With VB compression, the size of the compressed index for Reuters-RCV1 is 116 MB. This is a more than 50% reduction of the size of the uncompressed index.

The idea of VB encoding can also be applied to larger or smaller units than bytes: 32-bit words, 16-bit words, and 4-bit words or *nibbles*. Bytes offer a good compromise between compression ratio and speed of decompression. They are also simple to implement.

We can achieve better compression ratios by using bit-level encodings, in particular two closely related encodings: γ codes, which we will turn to next, and δ codes.

γ codes VB codes use an adaptive number of bytes depending on the size of the gap. Bit-level codes adapt the length of the code on the finer grained bit level. The simplest bit-level code is **unary code**. The unary code of n is a string of n 1s followed by a 0. This is not a very efficient code.

Assuming the 2^n gaps G with $1 \leq G \leq 2^n$ are all equally likely, the optimal encoding uses n bits for each G. So some gaps cannot be encoded with fewer than $\log_2 G$ bits. Our goal is to get as close to this lower bound as possible.

γ codes implement variable-length encoding by splitting the representation of a gap G into a pair of length and offset. **Offset** is G in binary, but with the leading 1 removed. For example, for 13 (binary 1101) offset is 101. **Length** encodes the length of offset in unary code. For 13, the length of offset is 3 bits, which is 1110 in unary. The γ code of 13 is therefore 1110101.

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

Figure 21: Some examples of unary and γ codes. Unary codes are only shown for the smaller numbers. Commas in γ codes are for readability only and are not part of the actual codes.

A γ code is decoded by first reading the unary code up to the 0 that terminates it, for example, the four bits 1110 when decoding 1110101. Now we know how long the offset is: 3 bits. The offset 101 can then be read correctly and the 1 that was chopped off in encoding is prepended: $101 \rightarrow 1101 = 13$. The length of offset is $\lfloor \log_2 G \rfloor$ bits and the length of length is $\lfloor \log_2 G + 1 \rfloor$ bits,

so the length of the entire code is $2 \times \lfloor \log_2 G + 1 \rfloor 1$ bits. γ codes are always of odd length and they are within a factor of 2 of what we claimed to be the optimal encoding length $\log_2 G$.

γ codes have two other properties. First, they are **prefix free**, no γ code is the prefix of another. This means that there is always a unique decoding of a sequence of γ codes – and we do not need delimiters between them. The second property is that γ codes are parameter free.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
term incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

Figure 22: Index and dictionary compression for Reuters-RCV1. The compression ratio depends on the proportion of actual text in the collection. Reuters-RCV1 contains a large amount of XML markup. Using the two best compression schemes, γ encoding and blocking with front coding, the ratio compressed index to collection size is therefore especially small for Reuters-RCV1: $(101 + 5.9)/3600 \approx 0.03$.

6 Scoring, term weighting and the vector space model

In the case of large document collections, the resulting number of matching documents can far exceed the number a human user could possibly sift through. It is essential for a search engine to rank-order the documents matching a query. To do this, the search engine computes a score with respect to the query.

6.1 Parametric and zone indexes

Digital documents generally encode certain **metadata** associated with each document. This metadata would generally include fields such as the date of creation and the format of the document.

Consider queries of the form "find documents authored by William Shakespeare in 1601, containing the phrase `alas poor Yorick`". Query processing then consists as usual of postings intersections from standard inverted as well as **parametric indexes**. There is one parametric index for each field; it allows us to select only the documents matching a date specified in the query.

Zones are similar to fields, except the contents of a zone can be arbitrary free text. A zone can be thought of as an arbitrary, unbounded amount of text. For instance, document titles and abstracts are generally treated as zones. We may build a separate inverted index for each zone of a document, to support queries such as "find documents with `merchant` in the title and `william` in the author list and the phrase `gentle rain` in the body". Whereas the dictionary for a parametric index comes from a fixed vocabulary, the dictionary for a zone index must structure whatever vocabulary stems from the text of that zone.

We can reduce the size of the dictionary by encoding the zone in which a term occurs in the postings, a technique called **weighted zone scoring**.

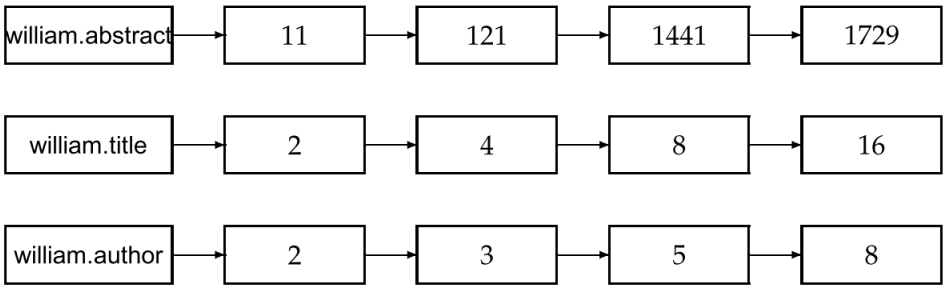

Figure 23: Basic zone index ; zones are encoded as extensions of dictionary entries.

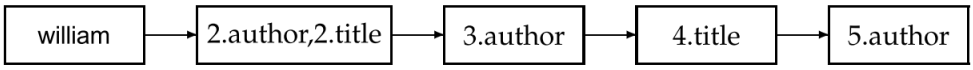

Figure 24: Zone index in which the zone is encoded in the postings rather than the dictionary.

Weighted zone scoring Given a Boolean query q and a document d, weighted zone scoring assigns to the pair (q, d) a score in the interval $[0, 1]$, by computing a linear combination of **zone scores**. Consider a set of documents each of which has ℓ zones. Let $g_1, \ldots, g_\ell \in [0, 1]$ such that $\sum_{i=1}^{\ell} g_i = 1$. For $1 \leq i \leq \ell$, let s_i be the Boolean score denoting a match between q and the ith zone. The weighted zone score is defined to be

$$\sum_{i=1}^{\ell} g_i s_i$$

Weighted zone scoring is sometimes referred to also as **ranked Boolean retrieval**.

A simple approach to implement the computation of weighted zone scores would be to compute scores directly from inverted indexes. The algorithm scan the postings traversal adding a document to the set of results for a Boolean AND query and computing a score for each such document.

Learning weights The weights g_i for weighted zone scoring could be specified by an expert, or learned using training examples that have been judged editorially (**machine-learned relevance**). The basic steps of a machine-learned relevance approach are:

- We are provided with a set of training examples, a set of tuple consisting of a query q and a document d, with a relevance judgment for d on q.
- The weights g_i are then "learned" from these examples, in order that the learned scores approximate the relevance judgments in the training examples.

The process may be viewed as learning a linear function of the Boolean match scores contributed by the various zones.

6.2 Term frequency and weighting

A document or zone that mentions a query term more often has more to do with that query and therefore should receive a higher score.We assign to each term in a document a weight for that term, that depends on the number of occurrences of the term in the document. We would like to compute a score between a query term t and a document d, based on the weight of t in d. The simplest approach is to assign the weight to be equal to the number of occurrences of t in d. This weighting scheme is referred to as **term frequency** and is denoted $\text{tf}_{t,d}$.

For a document d, the set of weights determined by the tf weights may be viewed as a quantitative digest of that document. In this view of a document, called **bag of words model**, the exact ordering of the terms in a document is ignored.

Inverse document frequency Raw term frequency as above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query. For instance, a collection of documents on the auto industry is likely to have the term auto in almost every document. Thus we introduce a mechanism for attenuating the effect of terms that occur too often.

The **document frequency** df_t, defined to be the number of documents in the collection that contain a term t. It is better to use a document-level statistic than to use a collection-wide statistic for the term. 25 shows that collection frequency (cf) and document frequency (df) can behave rather differently. In particular, the cf values for both `try` and `insurance` are roughly equal, but their df values differ significantly. We want the few documents that contain `insurance` to get a higher boost for a query on `insurance` than the many documents containing `try` get from a query on `try`.

Word	cf	df
try	10422	8760
insurance	10440	3997

Figure 25: Collection frequency (cf) and document frequency (df) behave differently, as in this example from the Reuters collection.

How is the document frequency df of a term used to scale its weight? Denoting total number of documents in a collection by N, we define the **inverse document frequency** (idf) of a term t as follows:

$$\text{idf}_t = \log \frac{N}{\text{df}_t}$$

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

Tf-idf weighting We now combine the definitions of term frequency and inverse document frequency, to produce a composite weight for each term in each document. The **tf-idf** weighting scheme assigns to term t a weight in document d given by

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

$\text{tf-idf}_{t,d}$ assigns to term t a weight in document d that is

1. highest when t occurs many times within a small number of documents
2. lower when the term occurs fewer times in a document, or occurs in many documents

3. lowest when the term occurs in virtually all documents.

We may view each document as a **vector** with one component corresponding to each term in the dictionary, together with a weight for each component. For dictionary terms that do not occur in a document, this weight is zero.

Now we introduce the **overlap score measure**: the score of a document d is the sum, over all query terms, of the number of times each of the query terms occurs in d. We add up not the number of occurrences of each query term t in d, but instead the tf-idf weight of each term in d.

$$\text{Score}(q, d) = \sum_{t \in q} \text{tf-idf}_{t,d}.$$

6.3 The vector space model for scoring

The representation of a set of documents as vectors in a common vector space is known as the **vector space model**.

Dot products We denote by $\vec{V}(d)$ the vector derived from document d, with one component in the vector for each dictionary term. The set of documents in a collection then may be viewed as a set of vectors in a vector space, in which there is one axis for each term. This representation loses the relative ordering of the terms in each document.

How do we quantify the similarity between two documents in this vector space? A first attempt might consider the magnitude of the vector difference between two document vectors, but this measure suffers from a drawback: two documents with very similar content can have a significant vector difference simply because one is much longer than the other.

To compensate for the effect of document length, the standard way of quantifying the similarity between two documents d_1 and d_2 is to compute the cosine similarity of their vector representations $\vec{V}(d_1)$ and $\vec{V}(d_2)$

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|}$$

where the numerator represents the dot product of the vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$, while the denominator is the product of their **Euclidean lengths**. The effect of the denominator is to length-normalize the vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$ to unit vectors $\vec{v}(d_1) = \vec{V}(d_1)/|\vec{V}(d_1)|$ and $\vec{v}(d_2) = \vec{V}(d_2)/|\vec{V}(d_2)|$. We can then rewrite the equation as

$$\text{sim}(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2).$$

It can be viewed as the dot product of the normalized versions of the two document vectors. This measure is the cosine of the angle Θ between the two vectors. Given a document d, consider searching for the documents in the collection most similar to d. Such a search is useful in a system where a user may identify a document and seek others like it. We reduce the problem of finding the document(s) most similar to d to that of finding the d_i with the highest dot products $\vec{v}(d) \cdot \vec{v}(d_i)$.

Viewing a collection of N documents as a collection of vectors leads to a natural view of a collection as a term-document matrix: this is an $M \times N$ matrix whose rows represent the M terms (dimensions) of the N columns, each of which corresponds to a document.

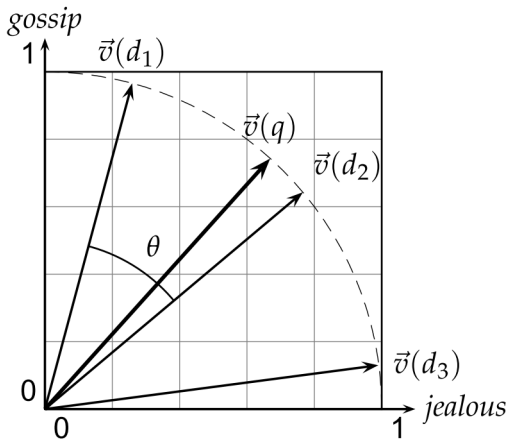

Figure 26: Cosine similarity illustrated. $\text{sim}(d_1, d_2) = \cos\Theta$.

Queries as vectors We can also view a query as a vector. Consider a query q. This query turns into the unit vector $\vec{v}(q)$. The idea is to assign to each document d a score equal to the dot product

$$\vec{v}(q) \cdot \vec{v}(d).$$

By viewing a query as a "bag of words", we are able to treat it as a very short document. We can use the cosine similarity between the query vector and a document vector as a measure of the score of the document for that query. The resulting scores can then be used to select the top-scoring documents for a query. Thus we have

$$\text{Score}(q, d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)|}$$

A document may have a high cosine score for a query even if it does not contain all query terms.

Computing the cosine similarities between the query vector and each document vector in the collection, sorting the resulting scores and selecting the top K documents can be expensive.

Computing vector scores In a typical setting we have a collection of documents each represented by a vector, a free text query represented by a vector, and a positive integer K. We seek the K documents of the collection with the highest vector space scores on the given query. Typically, we seek these K top documents in ordered by decreasing score (for instance many search engines use $K = 10$).

The Algorithm 4 computes the vector space scores. The array Length holds the lengths for each of the N documents, whereas the array Scores holds the scores for each of the documents. When the scores are finally computed, all that remains is to pick off the K documents with the highest scores. The outermost loop repeats the updating of Scores, iterating over each query term t in turn. We calculate the weight in the query vector for term t. We update the score of each document by adding in the contribution from term t. This process is known as **term-at-a-time** scoring or accumulation, and the N elements of the array Scores are therefore known as **accumulators**. It is wasteful to store the weight $\text{wf}_{t,d}$ of term t in document d since storing this weight may require a floating point number. Two ideas help. First, if we are using inverse document frequency, we need not precompute idf_t; it suffices to store N/df_t at the head of the postings for t. Second, we store the term frequency $\text{tf}_{t,d}$ for each postings entry. Finally extracts the top K scores: this requires a

priority queue data structure, often implemented using a heap. Each of the K top scores can be extracted from the heap at a cost of $O(\log N)$ comparisons.

Algorithm 4 The basic algorithm for computing vector space scores.

```
1: function COSINE_SCORE(q)
2:     float Scores[N] = 0
3:     Initialize Length[N]
4:     for all query term t do calculate w_{t,q} and fetch postings list for t
5:         for all pair(d, tf_{t,d}) in postings list do Scores[d]+ = wf_{t,d} × w_{t,q}
6:         end for
7:     end for
8:     Read the array Length[d]
9:     for all d do Scores[d] = Scores[d]/Length[d]
10:    end for
11:    return Top K components of Scores[]
12: end function
```

6.4 Variant tf-idf functions

For assigning a weight for each term in each document, a number of alternatives to tf and tf-idf have been considered.

Sublinear tf scaling It seems unlikely that twenty occurrences of a term in a document truly carry twenty times the significance of a single occurrence. We can use instead the logarithm of the term frequency, which assigns a weight given by

$$\mathrm{wf}_{t,d} = \begin{cases} 1 + \log \mathrm{tf}_{t,d} & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

In this form, we may replace tf by wf, to obtain:

$$\text{wf-idf}_{t,d} = \mathrm{wf}_{t,d} \times \mathrm{idf}_t.$$

Maximum tf normalization One well-studied technique is to normalize the tf weights of all terms occurring in a document by the maximum tf in that document. For each document d, let $\mathrm{tf}_{max}(d) = \max_{\tau \in d} \mathrm{tf}_{\tau,d}$, where τ ranges over all terms in d. Then we compute a normalized term frequency for each term t in document d by

$$\mathrm{ntf}_{t,d} = a + (1-a)\frac{\mathrm{tf}_{t,d}}{\mathrm{tf}_{\max}(d)},$$

where a is the **smoothing term**, a value between 0 and 1, whose role is to damp the contribution of the second term, which may be viewed as a scaling down of tf by the largest tf value in d.

Maximum tf normalization does suffer from the following issues:

1. The method is unstable in the following sense: a change in the stop word list can dramatically alter term weightings. It is hard to tune.

2. A document may contain an outlier term with an unusually large number of occurrences of that term, not representative of the content of that document.

3. a document in which the most frequent term appears roughly as often as many other terms should be treated differently from one with a more skewed distribution.

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1+\log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5+\frac{0.5\times\mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0,\log\frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$ (Section 6.4.4)
b (boolean)	$\begin{cases}1 & \text{if } \mathrm{tf}_{t,d}>0\\ 0 & \text{otherwise}\end{cases}$			b (byte size)	$1/\mathit{CharLength}^{\alpha}$, $\alpha<1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

Figure 27: SMART notation for tf-idf variants. Here *CharLength* is the number of characters in the document.

7 Computing scores in a complete search system

7.1 Efficient scoring and ranking

For a query such as $q =$ `jealous gossip`, two observations are immediate:

1. The unit vector $\vec{v}(q)$ has only two non-zero components.

2. In the absence of any weighting for query terms, these non-zero components are equal

We are interested in the relative (rather than absolute) scores of the documents in the collection. To this end, it suffices to compute the cosine similarity from each document unit vector $\vec{v}(d)$ to $\vec{V}(q)$ (in which all non-zero components of the query vector are set to 1), rather than to the unit vector $\vec{v}(q)$. For any two documents d_1, d_2

$$\vec{V}(q)\cdot\vec{v}(d_1) > \vec{V}(q)\cdot\vec{v}(d_2) \iff \vec{v}(q)\cdot\vec{v}(d_1) > \vec{v}(q)\cdot\vec{v}(d_2)$$

For any document d, the cosine similarity $\vec{V}(q)\cdot\vec{v}(d)$ is the weighted sum, over all terms in the query q, of the weights of those terms in d. We walk through the postings in the inverted index for the terms in q, accumulating the total score for each document. We maintain an idf value for each dictionary term and a tf value for each postings entry. This scheme computes a score for every document in the postings of any of the query terms.

Given these scores, the final step before presenting results to a user is to pick out the K highest-scoring documents. While one could sort the complete set of scores, a better approach is to use a heap to retrieve only the top K documents in order. Where J is the number of documents with non-zero cosine scores, constructing such a heap can be performed in $2J$ comparison steps, following which each of the K highest scoring documents can be "read off" the heap with $\log J$ comparison steps.

Inexact top K document retrieval Consider schemes by which we produce K documents that are likely to be among the K highest scoring documents for a query. We want to lower the cost of computing the K documents we output, without materially altering the user's perceived relevance of the top K results. Consequently, in most applications it suffices to retrieve K documents whose scores are very close to those of the K best.

The **inexact top-K retrieval** is not necessarily, from the user's perspective, a bad thing. The top K documents by the cosine measure are in any case not necessarily the K best for the query: cosine similarity is only a proxy for the user's perceived relevance. The heuristics have the following two-step scheme:

1. Find a set A of documents that are contenders, where $K < |A| < N$. A does not necessarily contain the K top-scoring documents for the query, but is likely to have many documents with scores near those of the top K.
2. Return the K top-scoring documents in A.

Index elimination For a multi-term query q:

1. We only consider documents containing terms whose idf exceeds a preset threshold. In the postings traversal, we only traverse the postings for terms with high idf. The postings lists of low-idf terms are generally long, thus the set of documents for which we compute cosines is greatly reduced. Low-idf terms are treated as stop words and do not contribute to scoring.
2. We only consider documents that contain many of the query terms. A danger of this scheme is that by requiring all (or even many) query terms to be present in a document before considering it for cosine computation, we may end up with fewer than K candidate documents in the output.

Champion lists The idea of **champion lists** is to precompute, for each term t in the dictionary, the set of the r documents with the highest weights for t. For tf-idf weighting, these would be the r documents with the highest tf values for term t. We call this set of r documents the champion list for term t.

Given a query q we create a set A as follows: we take the union of the champion lists for each of the terms comprising q. We restrict cosine computation to only the documents in A. A critical parameter is the value r. Intuitively, r should be large compared with K. One issue here is that the value r is set at the time of index construction, whereas K is application dependent and may not be available until the query is received; as a result we may find ourselves with a set A that has fewer than K documents.

Static quality scores and ordering In many search engines, we have available a measure of quality $g(d) \in [0, 1]$ for each document d that is query-independent and thus **static**.

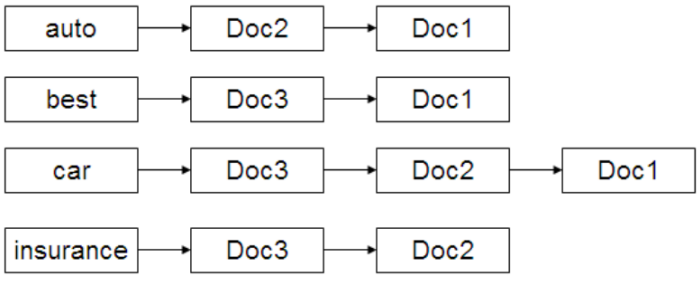

Figure 28: A static quality-ordered index. In this example we assume that Doc1, Doc2 and Doc3 respectively have static quality scores $g(1) = 0.25$, $g(2) = 0.5$, $g(3) = 1$.

The net score for a document d is some combination of $g(d)$ together with the query-dependent score.

$$\text{net-score}(q, d) = g(d) + \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)|}$$

In this simple form, the static quality $g(d)$ and the query-dependent score have equal contributions. Other relative weightings are possible.

First, consider ordering the documents in the postings list for each term by decreasing value of $g(d)$. The first idea is a direct extension of champion lists: for a well-chosen value r, we maintain for each term t a **global champion list** of the r documents with the highest values for $g(d) + \text{tf-idf}_{t,d}$. The list itself is sorted by a common order. Then at query time, we only compute the net scores for documents in the union of these global champion lists.

We maintain for each term t two postings lists consisting of disjoint sets of documents, each sorted by $g(d)$ values. The first list, which we call **high**, contains the m documents with the highest tf values for t. The second list, which we call **low**, contains all other documents containing t. We first scan only the high lists of the query terms, if we obtain scores for K documents in the process, we terminate. If not, we continue the scanning into the low lists.

Impact ordering In all the postings lists described thus far, we order the documents consistently by some common ordering. Such a common ordering supports the concurrent traversal of all of the query terms' postings lists, computing the score for each document as we encounter it. Computing scores in this manner is sometimes referred to as document-at-a-time scoring. A inexact top-K retrieval in which the postings are not all ordered by a common ordering requires scores to be "accumulated" one term at a time, so that we have term-at-a-time scoring.

The idea is to order the documents d in the postings list of term t by decreasing order of $\text{tf}_{t,d}$. Thus, the ordering of documents will vary from one postings list to another. Two ideas to significantly lower the number of documents for which we accumulate scores:

1. When traversing the postings list for a query term t, we stop after considering a prefix of the postings list – either after a fixed number of documents r have been seen, or after the value of $\text{tf}_{t,d}$ has dropped below a threshold

2. When accumulating scores, we consider the query terms in decreasing order of idf, so that the query terms likely to contribute the most to the final scores are considered first.

Cluster pruning In **cluster pruning** we have a preprocessing step during which we cluster the document vectors. Then at query time, we consider only documents in a small number of clusters as candidates for which we compute cosine scores. The preprocessing step is as follows:

1. Pick $\sqrt{N}$ documents at random from the collection. Call these **leaders**.

2. For each document that is not a leader, we compute its nearest leader.

We refer to documents that are not leaders as **followers**. The expected number of followers for each leader is $\approx N/\sqrt{N} = \sqrt{N}$. Next, the query processing proceeds as follows:

1. Given a query q, find the leader L that is closest to q. This entails computing cosine similarities from q to each of the $\sqrt{N}$ leaders.

2. The candidate set A consists of L together with its followers. We compute the cosine scores for all documents in this candidate set.

Variations of cluster pruning introduce additional parameters b_1 and b_2, both of which are positive integers. In the pre-processing step we attach each follower to its b_1 closest leaders. At query time we consider the b_2 leaders closest to the query q.

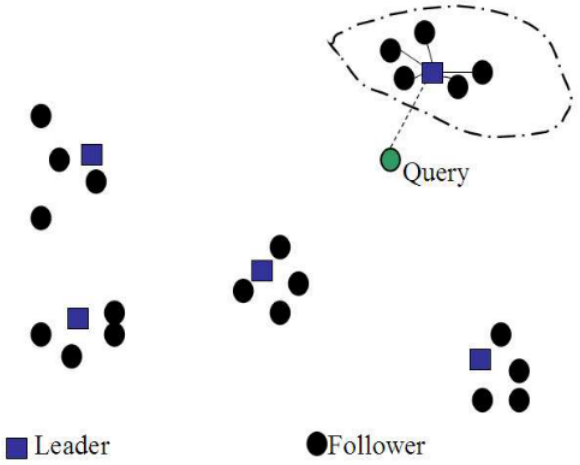

Figure 29: Cluster pruning.

7.2 Components of an information retrieval system

In this section we combine the ideas developed so far to describe a rudimentary search system that retrieves and scores documents.

Tiered indexes Heuristics such as index elimination for inexact top-K retrieval, we may occasionally find ourselves with a set A of contenders that has fewer than K documents. A common solution to this issue is the user of **tiered indexes**. In this example we set a tf threshold of 20 for tier 1 and 10 for tier 2, meaning that the tier 1 index only has postings entries with tf values exceeding 20, while the tier 2 index only has postings entries with tf values exceeding 10.

Query-term proximity Especially for free text queries, users prefer a document in which most or all of the query terms appear close to each other. Consider a query with two or more query terms, $t_1, t_2, \ldots, t_k$. Let ω be the width of the smallest window in a document d that contains all the query terms, measured in the number of words in the window. For instance, `The quality of mercy is not strained`, the smallest window for the query `strained mercy` would be 4. The smaller that ω is, the better that d matches the query. In cases where the document does not contain all of the query terms, we can set ω to be some enormous number. Such proximity-weighted scoring functions are a departure from pure cosine similarity and closer to the *soft conjunctive* semantics.

How can we design such a proximity-weighted scoring function to depend on ω? The simplest answer relies on a *hand coding* technique. In a more scalable approach, we treat the integer ω as yet another feature in the scoring function, whose importance is assigned by machine learning.

Designing parsing and scoring functions Common search interfaces tend to mask query operators from the end user. The intent is to hide the complexity of these operators from the largely non-technical audience, inviting **free text queries**. How should a search equipped with indexes for various retrieval operators treat a query such as `rising interest rates`? Typically, a **query**

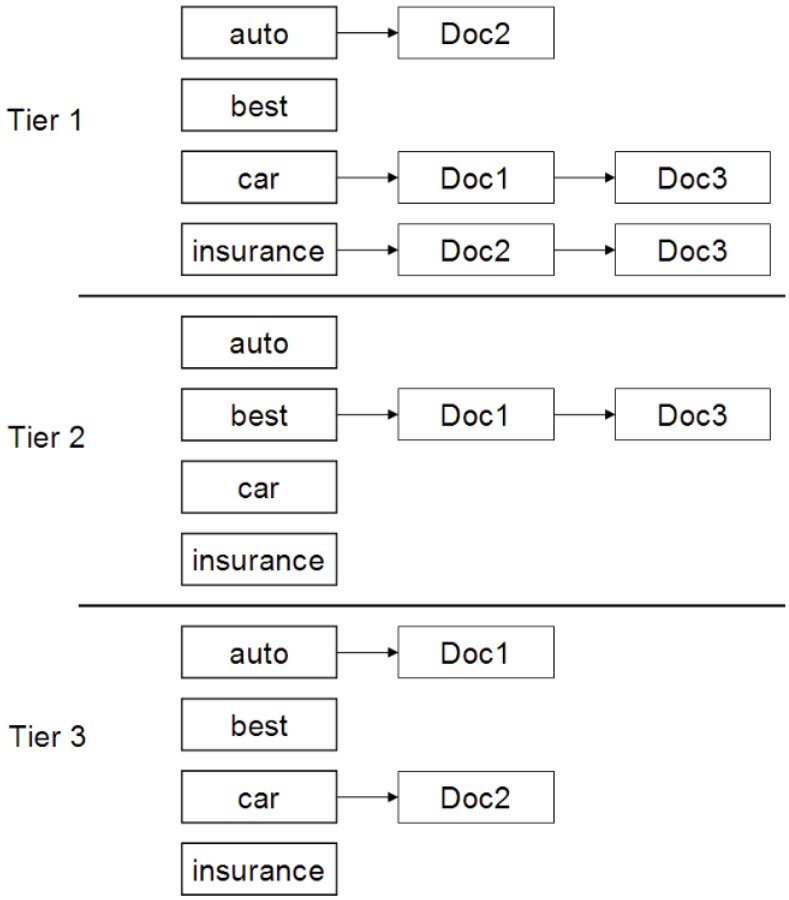

Figure 30: Tiered indexes. If we fail to get K results from tier 1, query processing "falls back" to tier 2, and so on. Within each tier, postings are ordered by document ID.

parser is used to translate the user-specified keywords into a query with various operators that is executed against the underlying indexes. Sometimes, this execution can entail multiple queries against the underlying indexes; for example, the query parser may issue a stream of queries:

1. Run the user-generated query string as a phrase query. Rank them by vector space scoring using as query the vector consisting of the 3 terms `rising interest rates`.
2. If fewer than ten documents contain the phrase `rising interest rates`, run the two 2-term phrase queries `rising interest` and `interest rates`; rank these using vector space scoring.
3. If we still have fewer than ten results, run the vector space query consisting of the three individual query terms.

Each of these steps may yield a list of scored documents, for each of which we compute a score. This score must combine contributions from vector space scoring, static quality, proximity weighting and potentially other factors. This demands an aggregate scoring function that **accumulates evidence** of a document's relevance from multiple sources.

Putting it all together In 31, documents stream in from the left for parsing and linguistic processing (language and format detection, tokenization and stemming). The resulting stream of tokens feeds into two modules. First, we retain a copy of each parsed document in a document cache. A second copy of the tokens is fed to a bank of indexers that create a bank of indexes including zone and field indexes that store the metadata for each document, (tiered) positional indexes, indexes for spelling correction and other tolerant retrieval, and structures for accelerating inexact top-K retrieval. A free text user query (top center) is sent down to the indexes both directly and through a module for generating spelling-correction candidates. Retrieved documents (dark arrow) are passed to a scoring module that computes scores based on machine-learned ranking (MLR). Finally, these ranked documents are rendered as a results page.

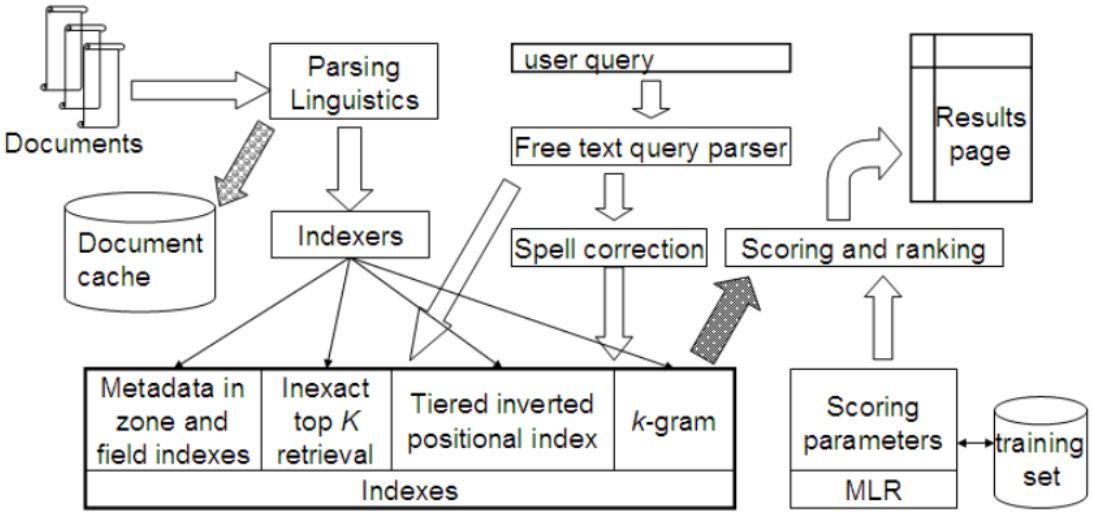

Figure 31: A complete search system. Data paths are shown primarily for a free text query.

7.3 Vector space scoring and query operator interaction

Vector space scoring supports so-called free text retrieval, in which a query is specified as a set of words without any query operators connecting them. It allows documents matching the query to be scored and thus ranked, unlike the Boolean, wildcard and phrase queries. The interpretation of such free text queries was that at least one of the query terms be present in any retrieved document. However more recently, web search engines such as Google have popularized the notion that a set of terms typed into their query boxes carries the semantics of a conjunctive query that only retrieves documents containing all or most query terms.