1 *Information retrieval using the Boolean model*

INFORMATION RETRIEVAL

The meaning of the term *information retrieval* can be very broad. Just getting a credit card out of your wallet so that you can type in the card number is a form of information retrieval. However, as an academic field of study, *information retrieval* might be defined thus:

> Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfy an information need from within large collections (usually stored on computers).

As defined in this way, information retrieval used to be an activity that only a few people engaged in: reference librarians, paralegals, and similar professional searchers. Now the world has changed, and hundreds of millions of people engage in information retrieval every day when they use a web search engine or search their email.[1] Information retrieval is fast becoming the dominant form of information access, overtaking traditional database-style searching (the sort that is going on when a clerk says to you: "I'm sorry, I can only look up your order if you can give me your Order ID").

IR can also cover other kinds of data and information problems beyond that specified in the core definition above. 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, of the sort companies usually use to maintain product inventories and personnel records. In reality, almost no data is truly "unstructured". This is definitely true of all text data if you count the latent linguistic structure of human languages. But even accepting that the intended notion of structure is overt structure, most text has structure, such as headings and paragraphs and footnotes, which is commonly represented in the documents by explicit markup (such as the coding underlying web

1. In modern parlance, the word "search" has tended to replace "(information) retrieval"; the term "search" is quite ambiguous, but in context we use the two synonymously.

pages). IR is also used to facilitate "semi-structured" search such as finding a document where the Title contains Java and the Body contains threading.

The field of information retrieval also covers operations typically done in browsing 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. It is similar to arranging books on a bookshelf according to their topic. Given a set of topics (or standing information needs), classification is the task of deciding which topic(s), if any, each of a set of documents belongs to. It is often approached by first manually classifying some documents and then hoping to be able to classify new documents automatically.

In this chapter we will begin with an information retrieval problem, and introduce the idea of a term-document matrix (Section 1.1) and the central inverted index data structure (Section 1.2). We will then examine the Boolean retrieval model and how queries are processed (Section 1.3).

1.1 An example information retrieval problem

A fat book which many people own is Shakespeare's Collected Works. 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, noting for each play whether it contains Brutus and Caesar and excluding it from consideration if it contains Calpurnia. The simplest form of document retrieval is for a computer to do this sort of linear scan through documents. This process is commonly referred to as *grepping* through text, after the Unix command `grep`, which performs this process. Grepping through text can be a very effective process, especially given the speed of modern computers, and often allows useful possibilities for wildcard pattern matching through the use of *regular expressions*. With modern computers, for simple querying of modest collections (the size of Shakespeare's Collected Works is a bit under one million words of text in total), you really need nothing more.

GREPPING

But for many purposes, you do need more:

1. To process large document collections quickly. The amount of online data has grown at least as quickly as the speed of computers, and we would now like to be able to search collections that total in the order of billions to trillions of words.

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".

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	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 1.1** A term-document incidence matrix. Matrix element (i, j) is 1 if the play in column j contains the word in row i, and is 0 otherwise.

3. To allow ranked retrieval: in many cases you want the best answer to an information need among many documents that contain certain words.

INDEX

The way to avoid linearly scanning the texts for each query is to *index* the documents in advance. Let us stick with Shakespeare's Collected Works, and use it to introduce the basics of the Boolean retrieval model. Suppose we record for each document – here a play of Shakespeare's – whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000 different words). The result is a binary term-document *incidence matrix*, as in Figure 1.1. Here the word *term* roughly means "word" but is normally preferred in the information retrieval literature since some of the indexed units, such as perhaps 3.2 or & are not usually thought of as words. Now, 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.[2]

INCIDENCE MATRIX

TERM

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 *Anthony and Cleopatra* and *Hamlet* (Figure 1.1).

BOOLEAN RETRIEVAL MODEL

The *Boolean retrieval model* refers to being able to ask any query which is in the form of a Boolean expression of terms. That is, terms can be combined with the operators AND, OR, and NOT. Such queries effectively view each document as a set of words.

2. Formally, we take the transpose of the matrix to be able to get the terms as column vectors.

Antony and Cleopatra, Act III, Scene ii
Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii
Lord Polonius: I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

► **Figure 1.2** Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia.

Let us now consider a more realistic scenario, simultaneously using the opportunity to introduce some key terminology and notation. Suppose we have $N = 1$ million documents. By *documents* we mean whatever units we have decided to build a retrieval system over. They might be individual memos, or chapters of a book (see Section 2.1.2 (page 18) for further discussion). We will refer to the group of documents over which we perform retrieval as the (document) *collection*. It is sometimes also refered to as a *corpus* (a *body* of texts). Suppose each document is about 1000 words long (2–3 book pages). 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. There is nothing special about the numbers we have chosen, and they might vary by an order of magnitude or more, but they give us some idea of the dimensions of the kinds of problems we need to handle. We will discuss and model these size assumptions in Section 5.1 (page 68).

DOCUMENT

COLLECTION
CORPUS

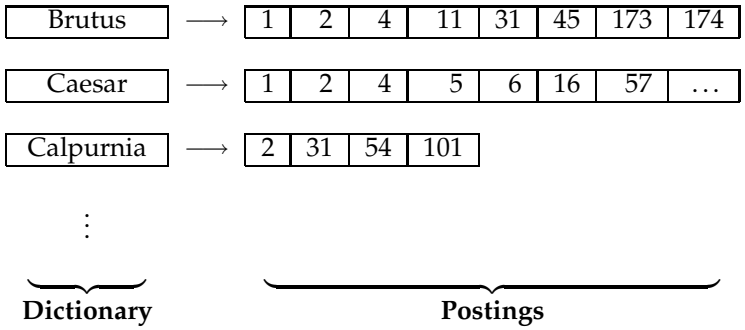
AD HOC RETRIEVAL

Our goal is to develop a system to address the *ad hoc retrieval* task. This is the most standard IR task. Un it, a system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query. 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.

INFORMATION NEED
QUERY

We now cannot build a term-document matrix in a naive way. A 500K $\times$ 1M matrix has half-a-trillion 0's and 1's. But the crucial observation is that the matrix is extremely sparse, that is, 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 things that do occur, that is, the 1 positions.

This idea is central to the first major concept in information retrieval, the



▶ **Figure 1.3** The two parts of an inverted index. The dictionary is usually kept in memory, with pointers to each postings list, which is stored on disk.

INVERTED INDEX

inverted index. The name is actually redundant: an index always maps back from terms to the parts of a document where they occur. Nevertheless, *inverted index*, or sometimes *inverted file*, has become the standard term in information retrieval.[3] The basic idea of an inverted index is shown in Figure 1.3.

DICTIONARY
VOCABULARY
LEXICON

We keep a *dictionary* of terms (sometimes also referred to as a *vocabulary* or *lexicon*), and then for each term, we have a list that records which documents the term occurs in. Each item in the list – a term combined with a document ID (and, later, often a position in the document) – is conventially called a *posting*. The list is then called a *postings list* (or inverted list), and all the postings lists taken together are referred to as the *postings*. The dictionary in Figure 1.3 has been sorted alphabetically and the postings list is sorted by document ID. We will see why this is useful in Section 1.3, below, but later we will also consider alternatives to doing this (Section 7.2.1).

POSTING
POSTINGS LIST
POSTINGS

1.2 A first take at building an inverted index

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

1. Collect the documents to be indexed.
   Friends, Romans, countrymen. | So let it be with Caesar | ...
2. Tokenize the text , turning each document into a list of tokens: Friends | Romans | countrymen | So | ...

3. Some information retrieval researchers prefer the term inverted file, but expressions like index construction and index compression are much more common than inverted file construction and inverted file compression. For consistency, we use (inverted) index throughout this book.

3. Linguistic preprocessing. The result is that each document is a list of normalized tokens: friend roman countryman so ...

4. Index the documents that each token occurs in by creating an inverted index, consisting of a dictionary and postings.

We will define and discuss the earlier stages of processing, that is, steps 1–3, in Chapter 2, and until then you can think of *tokens*, *normalized tokens*, and *terms* loosely as just *words*. Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index.

DOCID

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (*docID*). During index construction, we can simply assign successive integers to each new document when it is first encountered. The input to indexing will be a list of normalized tokens for each document, which we can equally think of as a list of pairs of term and docID, as in Figure 1.4. The core indexing step is *sorting* this list so that the terms are alphabetical, giving us the representation in the middle column of Figure 1.4. Multiple occurrences of the same term from the same document are then merged, and an extra column is added to record the frequency of the term in the document, giving the righthand column of Figure 1.4.[4] Frequency information is unnecessary for the basic Boolean search engine with which we begin our discussion, but as we will see soon, it is useful or needed in many extended searching paradigms, and it also allows us to improve efficiency at query time, even with a basic Boolean retrieval model.

SORTING

DICTIONARY
POSTINGS

Instances of the same term are then grouped, and the result is split into a *dictionary* and *postings*, as shown in Figure 1.5. The postings are secondarily sorted by docID. Since a term generally occurs in a number of documents, this organization already reduces the storage requirements of the index and, moreover, it provides the basis for efficient retrieval. Indeed, this inverted index structure is essentially without rivals as the most efficient structure for supporting ad hoc text search.

In the resulting index, we pay for storage of both the dictionary and the postings lists. The latter are much larger, but the dictionary is normally kept in memory, while postings lists are normally kept on disk, so the size of each is important, and in Chapter 5 we will examine how each can be optimized for storage and access efficiency. 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. Singly

4. Unix users can note that these steps are equivalent to use of the `sort` and then `uniq -c` commands.

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	**docID**
I	1
did	1
enact	1
julius	1
casear	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	**docID**
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

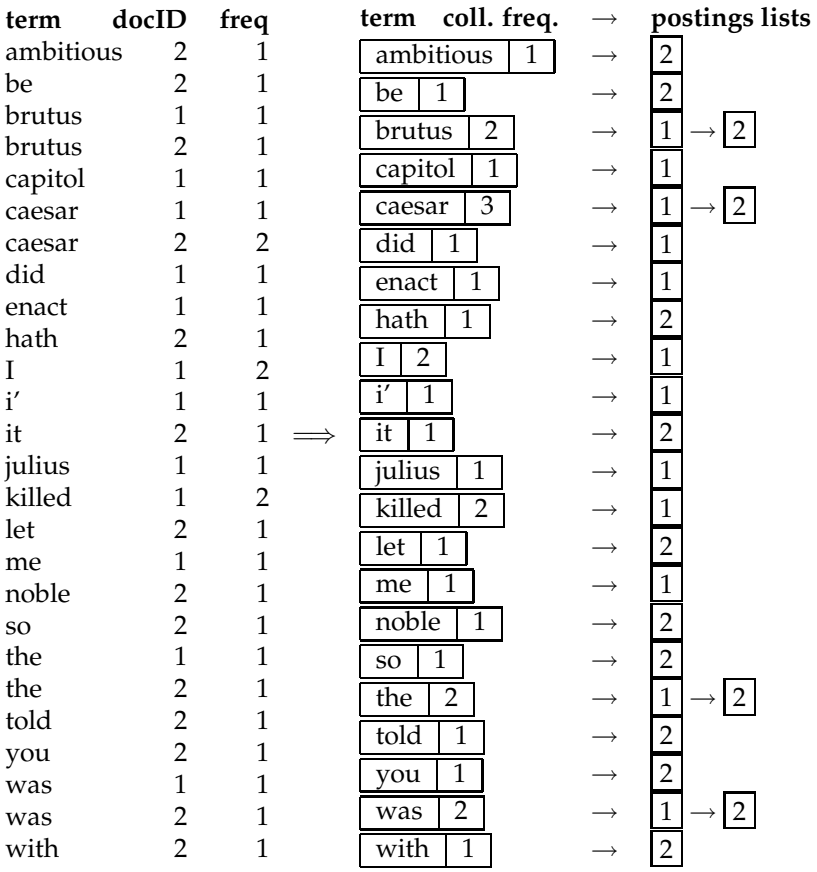
$\Longrightarrow$

term	**docID**	**term freq.**
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

► **Figure 1.4** Indexing, part one. The initial list of terms in each document (tagged by their documentID) is sorted alphabetically, and duplicate occurrences of a term in a document are collapsed, but the term frequency is preserved. (We will use it later in weighted retrieval models.)

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:



► **Figure 1.5** Indexing, part two. The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as the total frequency of each term in the collection, and/or the number of documents in which each term occurs. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency in each document and the position(s) of the term in the document.

linked lists allow cheap insertion of documents into postings lists (following updates, such as when recrawling the web for updated documents), and naturally extend to more advanced indexing strategies such as skip lists (Section 2.3.1), which require additional pointers. Variable length arrays gain in space requirements by avoiding the space overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches. Extra pointers can in practice be encoded into the lists as offsets. If updates are relatively infrequent, variable length arrays will be more compact and faster to traverse. We can also use a hybrid scheme with a linked list of fixed length arrays for each term. When postings lists are stored on disk, they are stored (perhaps compressed) as a contiguous run of postings without explicit pointers (as in Figure 1.3), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

1.3 Processing Boolean queries

How do we process a query in the basic Boolean retrieval model? Consider processing the *simple conjunctive query*:

SIMPLE CONJUNCTIVE QUERIES

(1.1) Brutus AND Calpurnia

over the inverted index partially shown in Figure 1.3 (page 5). This is fairly straightforward:

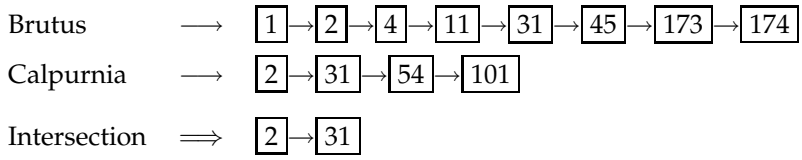
1. Locate Brutus in the Dictionary
2. Retrieve its postings
3. Locate Calpurnia in the Dictionary
4. Retrieve its postings
5. Intersect the two postings lists, as shown in Figure 1.6.

POSTINGS LIST INTERSECTION

POSTINGS MERGE

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: this slightly counterintuitive name reflects using the term *merge algorithm* for a general family of algorithms that combine multiple sorted lists; here we are merging the lists with a logical AND operation.)

There is a simple and effective method of intersecting postings lists using the merge algorithm (see Figure 1.7): we maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to



► **Figure 1.6** Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3.

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID[p1] = docID[p2]
 4        then ADD(answer, docID[p1])
 5             p1 ← next[p1]
 6             p2 ← next[p2]
 7        else if docID[p1] < docID[p2]
 8                then p1 ← next[p1]
 9                else p2 ← next[p2]
10  return answer
```

► **Figure 1.7** Algorithm for the intersection of two postings lists p_1 and p_2.

by both pointers. If they are the same, we put that docID in the result 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, this merge takes $O(x + y)$ operations. Formally, the complexity of querying is $\Theta(T)$, where T is the number of tokens in the document collection.[5] Our indexing methods gain us just a constant, not a difference in Θ time complexity compared to a linear scan, but in practice the constant is huge. To use this algorithm, it is crucial that postings be sorted by a single global ordering. Using a numeric sort by docID is one simple way to achieve this.

We would like to be able to extend the intersection operation to process more complicated queries like:

(1.2) (Brutus OR Caesar) AND NOT Calpurnia

QUERY OPTIMIZATION *Query optimization* is the process of selecting how to organize the work of answering a query so that the least total amount of work needs to be done by the system. A major element of this for Boolean queries is the order in which

5. The notation $\Theta(\cdot)$ is used to express an asymptotically tight bound on the complexity of an algorithm. Informally, this is often written as $O(\cdot)$, but this notation really expresses an asymptotic upper bound, which need not be tight (Cormen et al. 1990).

```
INTERSECT(⟨t_i⟩)
1  terms ← SORTBYINCREASINGFREQUENCY(⟨t_i⟩)
2  result ← postings[FIRST(terms)]
3  terms ← REST(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do list ← postings[FIRST(terms)]
6     terms ← REST(terms)
7     result ← INTERSECT(result, list)
8  return result
```

▶ **Figure 1.8** Algorithm for intersection of a set of postings lists.

postings lists are accessed. What is the best order for query processing? Consider a query that is an AND of t terms, for instance:

(1.3) Brutus AND Calpurnia AND Caesar

For each of the t terms, we need to get its postings, then AND them together. The standard heuristic is to process terms in order of increasing term frequency: if we start by intersecting the two smallest postings lists, then all intermediate results must be no bigger than the smallest postings list, and we are therefore likely to do the least amount of total work. So, for the postings lists in Figure 1.3, we execute the above query as:

(1.4) (Calpurnia AND Brutus) AND Caesar

This is a first justification for keeping the frequency of terms in the dictionary: it allows us to make this ordering decision based on in-memory data before accessing any postings list.

Consider now the optimization of more general queries, such as:

(1.5) (madding OR crowd) AND (ignoble OR strife)

As before, we will get the frequencies for all terms, and we can then (conservatively) estimate the size of each OR by the sum of the frequencies of its disjuncts. We can then process the query in increasing order of the size of each disjunctive term.

For arbitrary Boolean queries, we have to evaluate and temporarily store the answers for intermediate terms in a complex expression. However, in many circumstances, either because of the nature of the query language, or just because this is the most common type of query that users submit, a query is purely conjunctive. In this case, rather than viewing merging postings lists as a function with two inputs and a distinct output, it is more efficient to

intersect each retrieved postings list with the current intermediate result in memory, where we initialize the intermediate result by loading the postings list of the least frequent term. This algorithm is shown in Figure 1.8. The intersection operation is then asymmetric: the intermediate result list is in memory while the list it is being intersected with is being read from disk. Moreover the intermediate result list is always shorter than the other list, and in many cases it is orders of magnitude shorter. The postings intersection can still be done by the algorithm in Figure 1.7, but when the difference between the list lengths is very large, opportunities to use alternative techniques open up. The intersection can be calculated in place by destructively modifying or marking invalid items in the intermediate result list. The intersection can instead be done as a sequence of binary searches in the long postings lists for each term in the intermediate result list. Another possibility is to store the long postings list as a hashtable, so that membership of an intermediate result item can be calculated in constant rather than linear or log time. However, such alternative techniques are difficult to combine with postings list compression of the sort discussed in Chapter 5. Moreover, standard postings list intersection operations remain necessary when both terms of a query are very common.

1.4 Boolean querying, extended Boolean querying, and ranked retrieval

RANKED RETRIEVAL MODELS
FREE-TEXT QUERIES

The Boolean retrieval model contrasts with *ranked retrieval models* such as the vector space model (Chapter 7), in which users largely use *free-text queries*, that is, just typing one or more words rather than using a precise language with operators for building up query expressions, and the system decides which documents best satisfy the query. Despite decades of academic research on the advantages of ranked retrieval, systems implementing the Boolean retrieval model were the main or only search option provided by large commercial information providers for three decades until approximately the arrival of the World Wide Web (in the early 1990s). However, these systems did not have just the basic Boolean operations (AND, OR, and NOT) which we have presented so far. A strict Boolean expression over terms with an unordered results set is too limited for many of the information needs that people have, and these systems implemented extended Boolean retrieval models by incorporating additional operators such as term proximity operators. A

PROXIMITY OPERATOR

proximity operator is a way of specifying that two terms in a query must occur in a document close to each other, where closeness may be measured by limiting the allowed number of intervening words or by reference to a structural unit such as a sentence or paragraph.

Information need: What is the statute of limitations in cases involving the federal tort claims act?
Query: limit! /3 statute action /s federal /2 tort /3 claim

Information need: Requirements for disabled people to be able to access a workplace.
Query: disab! /p access! /s work-site work-place (employment /3 place)

Information need: Is a host responsible for drunk guests' behavior?
Query: host! /p (responsib! or liab!) /p (intoxicat! or drunk!) /p guest

► **Figure 1.9** Example Boolean queries on WestLaw. Note the long, precise queries and the use of proximity operators, both uncommon in web search. Unlike web search conventions, a space between words represents disjunction (the tightest binding operator), & is AND and /s, /p, and /*n* ask for matches in the same sentence, same paragraph or within *n* words respectively. The exclamation mark (!) gives a trailing wildcard query (see Chapter 3); thus liab! matches all words starting with liab. Queries are usually incrementally developed until they obtain what look to be good results to the user.

✎ **Example 1.1: Commercial Boolean searching: WestLaw.** WestLaw (http://www.westlaw.com/) is the largest commercial legal search service (in terms of the number of paying subscribers), with over half a million subscribers performing millions of searches a day over tens of terabytes of text data. The service was started in 1975. In 2005, Boolean search (called "Terms and Connectors" by WestLaw) was still the default, and used by a large percentage of users, although ranked free-text querying (called "Natural Language" by WestLaw) was added in 1992. Example Boolean queries are shown in Figure 1.9. Typical expert queries are carefully defined and refined to try to match suitable documents. Submitted queries average about ten words in length. Many users, particularly professionals, prefer Boolean query models. Boolean queries are precise: a document either matches the query or it does not. This offers the user greater control and transparency over what is retrieved. And some domains, such as legal materials, allow an effective means of document ranking within a Boolean model: WestLaw returns documents in reverse chronological order, which is in practice quite effective. This does not mean though that Boolean queries are more effective for professional searchers. Indeed, experimenting on a WestLaw subcollection, Turtle (1994) found that free-text queries produced better results than Boolean queries prepared by WestLaw's own reference librarians for the majority of the information needs in his experiments.

In later chapters we will consider in detail richer query languages and the sort of augmented index structures that are needed to handle them efficiently. Here we just mention a few of the main additional things we would like to be able to do:

1. It is often useful to search for compounds or phrases that denote a concept such as "operating system". And as the WestLaw example shows, we might

also wish to do proximity queries such as Gates NEAR Microsoft. To answer such queries, the index has to be augmented to capture the proximities of terms in documents.

2. A Boolean model only records term presence or absence, but often we would like to accumulate evidence, giving more weight to documents that have a term several times as opposed to ones that contain it only once. To be able to do this we need term frequency information for documents in postings lists.

3. Boolean queries just retrieve a set of matching documents, but commonly we wish to have an effective method to order (or "rank") the returned results. This requires having a mechanism for determining a document score which encapsulates how good a match a document is for a query.

In this chapter, we have looked at the structure and construction of a basic inverted index, comprising a dictionary and postings list. We introduced the Boolean retrieval model, and examined how to do efficient retrieval via linear time merges and simple query optimization. The following chapters provide more details on selecting the set of terms in the dictionary and building indexes that support these kinds of options. These are the basic operations that support ad hoc searching over unstructured information. Ad hoc searching over documents has recently conquered the world, powering not only web search engines but the kind of unstructured search that lies behind modern large eCommerce websites.

1.5 References and further reading

The practical pursuit of information retrieval began in the late 1940s (Cleverdon 1991, Liddy 2005). A great increase in the production of scientific literature, much in the form of less formal technical reports rather than traditional journal articles, coupled with the availability of computers, led to interest in automatic document retrieval. However, in those days, document retrieval was always based on author, title, and keywords; full-text search came much later.

The article of Bush (1945) provided lasting inspiration for the new field:

> "Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, 'memex' will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory."

The term *Information Retrieval* was coined by Calvin Mooers in 1948/1950.

In 1958, much newspaper attention was paid to demonstrations at a conference (see Taube and Wooster 1958) of IBM "auto-indexing" machines, based primarily on the work of H. P. Luhn. Commercial interest quickly gravitated towards Boolean retrieval systems, but the early years saw a heady debate over various disparate technologies for retrieval systems. For example Mooers (1961) dissented:

> "It is a common fallacy, underwritten at this date by the investment of several million dollars in a variety of retrieval hardware, that the algebra of George Boole (1847) is the appropriate formalism for retrieval system design. This view is as widely and uncritically accepted as it is wrong."

Witten et al. (1999) is the standard reference for an in-depth comparison of the space and time efficiency of the inverted index versus other possible data structures; a more succinct and up-to-date presentation appears in Zobel and Moffat (2006). We further discuss a number of approaches in Chapter 5.

REGULAR EXPRESSIONS

Friedl (2006) covers the practical usage of *regular expressions* for searching.

1.6 Exercises

Exercise 1.1 [⋆]

For the queries below, can we still run through the merge in time $O(x + y)$, where x and y are the lengths of the postings lists for Brutus and Caesar? If not, what can we achieve?

a. Brutus AND NOT Caesar

b. Brutus OR NOT Caesar

Exercise 1.2 [⋆]

Consider the time complexity for satisfying an arbitrary Boolean query formula. For instance, consider:

c. (Brutus OR Caesar) AND NOT (Anthony OR Cleopatra)

Can we always merge in linear time? Linear in what? Can we do better than this?

Exercise 1.3

We can use distributive laws for AND and OR to rewrite queries. Rewrite the above query into disjunctive normal form using the distributive laws. Is this result more or less efficiently evaluated than the original form?

Exercise 1.4

Recommend a query processing order for

d. (tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes)

given the following postings list sizes:

Term	Postings size
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

Exercise 1.5

If the query is:

e. friends AND romans AND (NOT countrymen)

how could we use the frequency of countrymen in evaluating the best query evaluation order?

Exercise 1.6

Extend the merge algorithm to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size? Hint: Begin with the case of a Boolean formula query: each query term appears only once in the query.

Exercise 1.7

For a conjunctive query, is processing postings lists in order of size guaranteed to be optimal? Explain why it is, or give an example where it isn't.