UNIT1

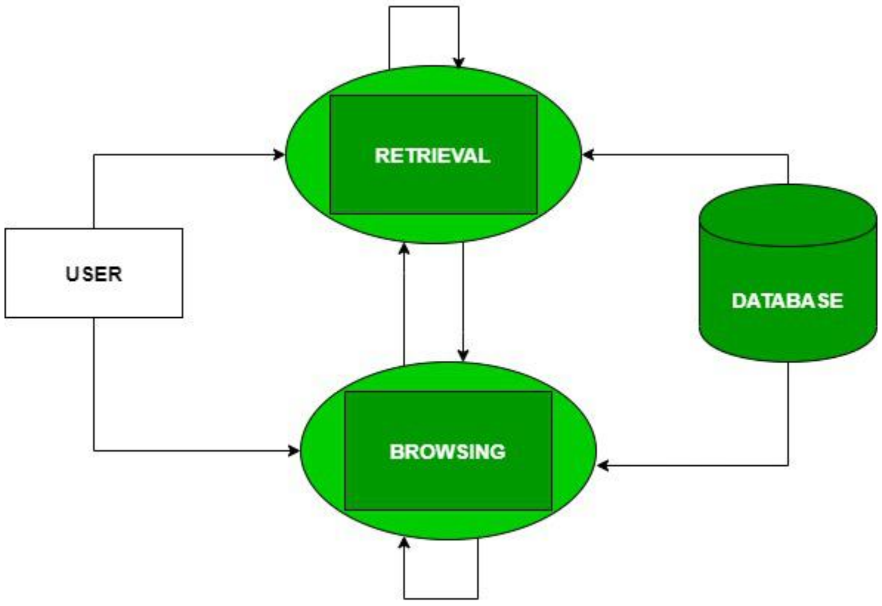
Basics of IR

Introduction- Basics of Information Retrieval and Introduction to Search Engines - Boolean Retrieval-: Boolean queries, Building simple indexes, Processing Boolean queries : Term Vocabulary and Posting Lists- Choosing document units, Selection of terms, Skip lists, Positional postings and Phrase queries

Introduction

- **Information Retrieval (IR)** can be defined as a software program that deals with the organization, storage, retrieval, and evaluation of information from document repositories, particularly textual information.
- Information Retrieval is the activity of obtaining material that can usually be documented on an unstructured nature **i.e.** usually text which satisfies an information need from within large collections which is stored on computers.

For example, Information Retrieval can be when a user enters a query into the system.



Purpose/role of an IR system

An information retrieval system is designed to retrieve the documents or information required by theuser community.

It should make the right information available to the right user. Thus, an information retrieval system aims at collecting and organizing information in one or more subject areas in order to provide it to the user as soon as possible.

Thus it serves as a bridge between the world of creators or generators of information and the users of that information.

Application areas within IR

- Cross language retrieval
- Speech/broadcast retrieval
- Text categorization
- Text summarization
- Structured document element retrieval (XML)

Types of Information Retrieval

- Text
- XML and structured documents
- Images
- Audio
- Video
- Source Code
- Applications/Web services

Features of an information retrieval system

Liston and Schoene suggest that an effective information retrieval system must have provisions for:

- Prompt dissemination of information
- Filtering of information
- The right amount of information at the right time
- Active switching of information
- Receiving information in an economical way
- Browsing
- Getting information in an economical way
- Current literature
- Access to other information systems
- Interpersonal communications, and
- Personalized help.

IR and Related Areas

1. Database Management
2. Library and Information Science
3. Artificial Intelligence
4. Natural Language Processing
5. Machine Learning

1.Database Management

- Focused on *structured* data stored in relational tables rather than free-form text.
- Focused on efficient processing of well-defined queries in a formal language (SQL).

2. Library and Information Science

- Focused on the human user aspects of information retrieval (human-computer interaction, user interface, visualization).

3. Artificial Intelligence

- Focused on the representation of knowledge, reasoning, and intelligent action.

4. Natural Language Processing

- Focused on the syntactic, semantic, and pragmatic analysis of natural language text and discourse.
- Ability to analyze syntax (phrase structure) and semantics could allow retrieval based on *meaning* ratherthan keywords.

Natural Language Processing: IR Directions

- Methods for determining the sense of an ambiguous word based on context (*word sensedisambiguation*).
- Methods for identifying specific pieces of information in a document (*information extraction*).

5. Machine Learning

- Focused on the development of computational systems that improve their performance with experience.

Machine Learning: IR Directions

- Text Categorization
  – Automatic hierarchical classification (Yahoo).
  – Adaptive filtering/routing/recommending.
  – Automated spam filtering.
- Text Clustering
  – Clustering of IR query results.
  – Automatic formation of hierarchies (Yahoo).
- Learning for Information Extraction
- Text Mining
- Learning to Rank

COMPONENTS OF INFORMATION RETRIEVAL

Information retrieval is concerned with representing, searching, and manipulating large collections of electronic text and other human-language data.

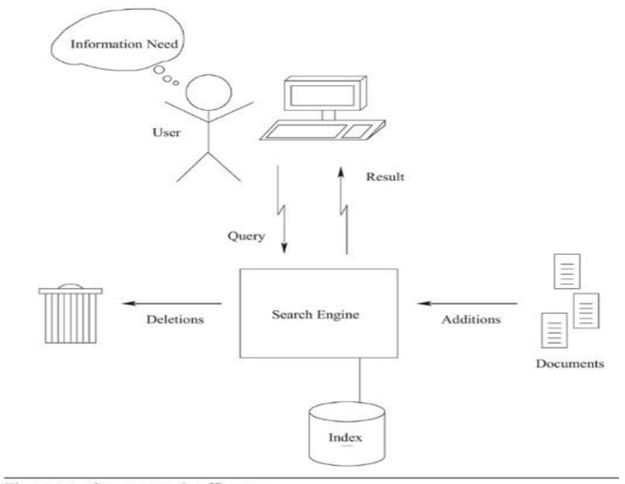
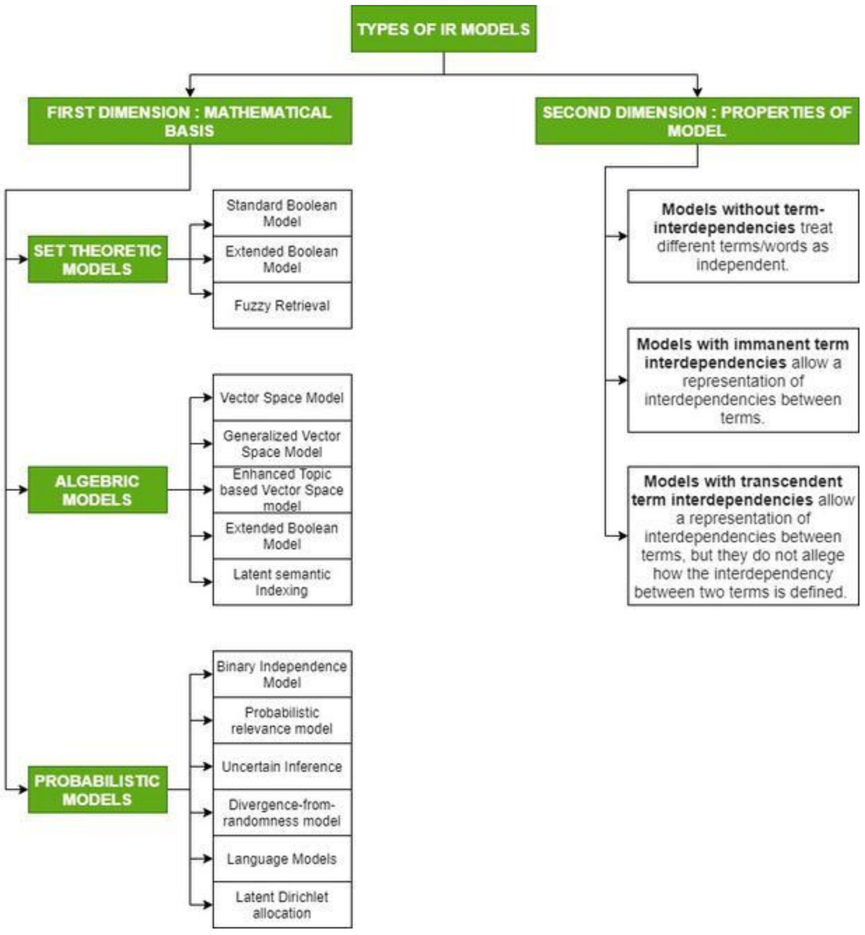


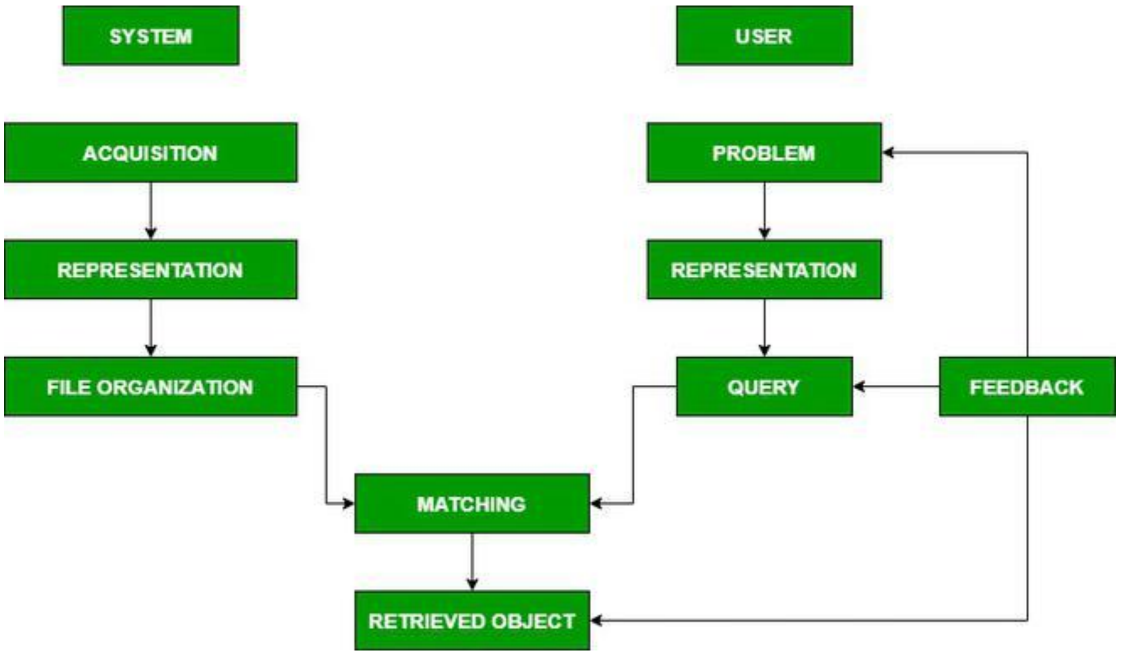
Figure 1.1 Components of an IR system.

The user's query is processed by a **search engine**, which may be running on the user's local machine, ona large cluster of machines in a remote geographic location, or anywhere in between. A major task of a search engine is to maintain and manipulate an **inverted index** for a **document collection**

Types of IR Models


TYPES OF IR MODELS
FIRST DIMENSION : MATHEMATICAL BASIS
SECOND DIMENSION : PROPERTIES OF MODEL
SET THEORETIC MODELS
Standard Boolean Model
Extended Boolean Model
Fuzzy Retrieval
ALGEBRIC MODELS
Vector Space Model
Generalized Vector Space Model
Enhanced Topic based Vector Space model
Extended Boolean Model
Latent semantic Indexing
PROBABILISTIC MODELS
Binary Independence Model
Probabilistic relevance model
Uncertain Inference
Divergence-from-randomness model
Language Models
Latent Dirichlet allocation
Models without term-interdependencies treat different terms/words as independent.
Models with immanent term interdependencies allow a representation of interdependencies between terms.
Models with transcendent term interdependencies allow a representation of interdependencies between terms, but they do not allege how the interdependency between two terms is defined.

Components of Information Retrieval/ IR Model



- **Acquisition:** In this step, the selection of documents and other objects from various web resources that consist of text-based documents takes place. The required data is collected by web crawlers and stored in the database.
- **Representation:** It consists of indexing that contains free-text terms, controlled vocabulary, manual & automatic techniques as well. example: Abstracting contains summarizing and Bibliographic description that contains author, title, sources, data, and metadata.
- **File Organization:** There are two types of file organization methods. i.e. *Sequential*: It contains documents by document data. *Inverted*: It contains term by term, list of records under each term. *Combination* of both.
- **Query:** An IR process starts when a user enters a query into the system. Queries are formal statements of information needs, for example, search strings in web search engines. In information retrieval, a query does not uniquely identify a single object in the collection. Instead, several objects may match the query, perhaps with different degrees of relevancy

SEARCH ENGINE

A search engine is the practical application of information retrieval techniques to large-scale text collections. Search engines come in a number of configurations that reflect the applications they are designed for. Web search engines, such as Google and Yahoo!, must be able to capture, or *crawl*, many terabytes of data, and then provide subsecond response times to millions of queries submitted every day from around the world.

Search engine **components support two major functions:**

1. **Indexing process**
2. **Query process**

1.Indexing process

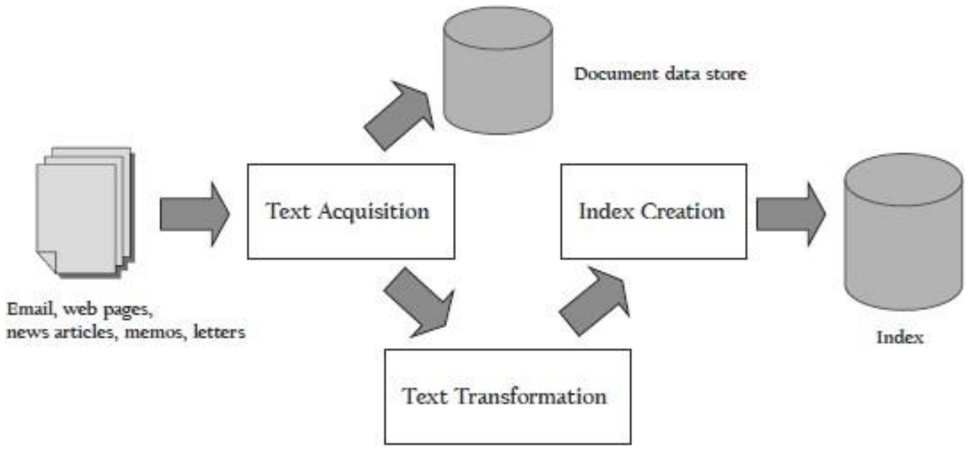


Fig. 2.1. The indexing process

The indexing process builds the structures that enable searching, and the query process uses those structures and a person's query to produce a ranked list of documents. Figure 2.1 shows the high-level "buildingblocks" of the indexing process.

These major components are

a) **Text acquisition**
b) **Text transformation**
c) Index creation

d) a)Text acquisition

The task of the text acquisition component is to identify and make available the documents that will be searched. Although in some cases this will involve simply using an existing collection, text acquisition willmore often require building a collection by *crawling* or scanning the Web, a corporate intranet, a desktop, or other sources of information.

(e.g., email or web page), document structure, and other features, such as document length.

b) Text transformation

The text transformation component transforms documents into *index terms* or *features*. Index terms, as the name implies, are the parts of a document that are stored in the index and used in searching. The simplest index term is a word, but not every word may be used for searching.

Index creation

The index creation component takes the output of the text transformation component and creates the indexes or data structures that enable fast searching. Given the large number of documents in many search applications, index creation must be efficient, both in terms of time and space. Indexes must also be able to be efficiently *updated* when new documents are acquired.

2. Query process

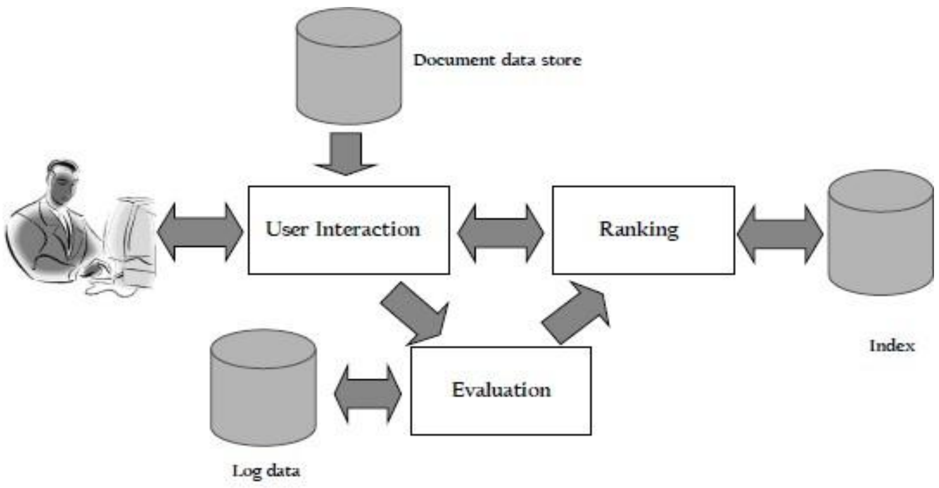


Fig. 2.2. The query process

Figure 2.2 shows the building blocks of the query process. The major components are

a) **User interaction**

b) **Ranking**

c) Evaluation

a)User interaction

The user interaction component provides the interface between the person doing the searching and the search engine.

for example, generating the *snippets* used to summarize documents. need.

b) **Ranking**

The ranking component is the core of the search engine. It takes the transformed query from the user interaction component and generates a ranked list of documents using scores based on a retrieval model.

Ranking must be both efficient, since many queries may need to be processed in a short time, and effective, since the quality of the ranking determines whether the search engine accomplishes the goal of finding relevant information.

The efficiency of ranking depends on the indexes, and the effectiveness depends on the retrieval model.

c) **Evaluation**

The task of the evaluation component is to measure and monitor effectiveness and efficiency. An important part of that is to record and analyze user behavior using *log data*.

The results of evaluation are used totune and improve the ranking component.

Most of the evaluation component is not part of the online search engine, apart from logging user and system data. Evaluation is primarily an offline activity, but it is a critical part of any search application.

BOOLEAN RETRIEVAL

Boolean Model

It is a simple retrieval model based on set theory and boolean algebra. Queries are designed as boolean expressions which have precise semantics. The retrieval strategy is based on binary decision criterion. The boolean model considers that index terms are present or absent in a document.

Problem:

Consider 5 documents with a vocabulary of 6 terms

document 1 = ‘ term1 term3 ‘

document 2 = ‘ term 2 term4 term6 ‘

document 3 = ‘ term1 term2 term3 term4 term5 ‘

document 4 = ‘ term1 term3 term6 ‘

document 5 = ‘ term3 term4 ‘

Our documents in a boolean model

	term1	**term2**	**term3**	**term4**	**term5**	**term6**
document1	1	0	1	0	0	0
document2	0	1	0	1	0	1
document3	1	1	1	1	1	0
document4	1	0	1	0	0	1
document5	0	0	1	1	0	0

Consider the query: *Find the document consisting of term1 and term3 and not term2 (* **term1 ∧ term3 ∧ ¬ term2)**

	term1	**¬term2**	**term3**	term4	term5	term6
document1	**1**	**1**	**1**	0	0	0
document2	**0**	**0**	**0**	1	0	1
document3	**1**	**0**	**1**	1	1	0
document4	**1**	**1**	**1**	0	0	1
document5	**0**	**1**	**1**	1	0	0

document 1 : 1 ∧ 1∧ 1 = 1

document 2 : 0 ∧ 0 ∧ 0 = 0

document 3 : 1 ∧ 1 ∧ 0 = 0

document 4 : 1 ∧ 1 ∧ 1 = 1

document 5 : 0 ∧ 1 ∧ 1 = 0

Based on the above computation **document1 and document4** are relevant to the given query

The CSV file which is given as input:

documents , term1, term2, term3

document1, ice cream, mango, litchi

document2 , hockey, cricket, sport

document3, litchi, mango, chocolate

document4 , nice, good, cute

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* :

(A) Brutus AND Calpurnia

over the inverted index partially shown in Figure 1.3 (page ❑). We:

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 1.5 .

The *intersection* 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 by interleaved advancing of pointers through each; here we are merging the lists with a logical AND operation.)

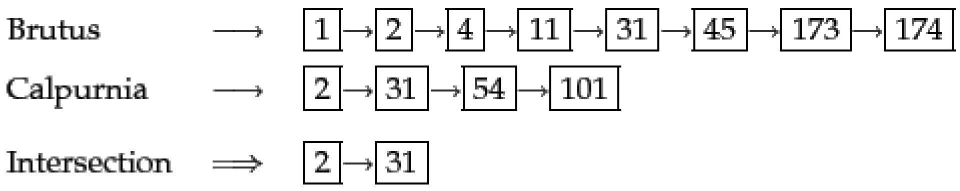

Brutus ⟶ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174
Calpurnia ⟶ 2 → 31 → 54 → 101
Intersection ⟹ 2 → 31

Figure: Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3 .

```
INTERSECT($p_1, p_2$)
  answer $\leftarrow \langle\, \rangle$
  while $p_1 \neq$ NIL and $p_2 \neq$ NIL
  do if $docID(p_1) = docID(p_2)$
        then ADD($answer, docID(p_1)$)
             $p_1 \leftarrow next(p_1)$
             $p_2 \leftarrow next(p_2)$
        else if $docID(p_1) < docID(p_2)$
               then $p_1 \leftarrow next(p_1)$
               else $p_2 \leftarrow next(p_2)$
  return answer
```

Figure 1.6: Algorithm for the intersection of two postings lists p_1 and p_2.

The term vocabulary and postings lists

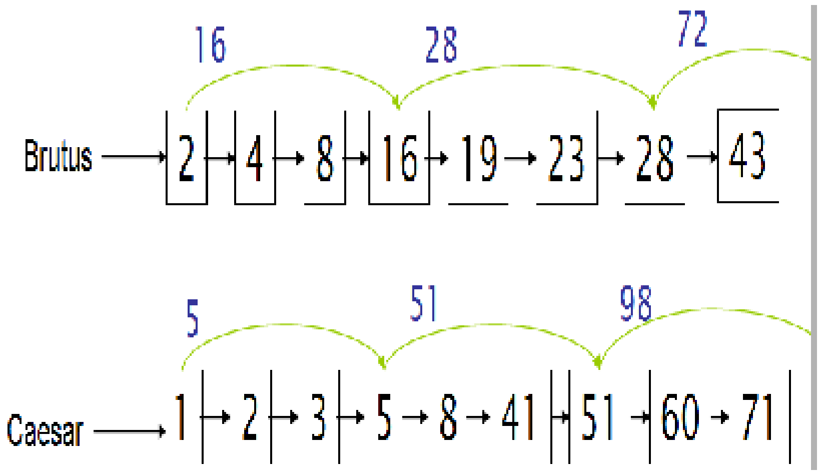
Recall the major steps in inverted index construction:

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.

Faster postings list intersection via skip pointers

In the remainder of this chapter, we will discuss extensions to postings list data structures and ways to increase the efficiency of using postings lists. Recall the basic postings list intersection operation from Section 1.3 (page): we walk through the two postings lists simultaneously, in time linear in the total number of postings entries. If the list lengths are m and n, the intersection takes $O(m+n)$ operations. Can we do better than this? That is, empirically, can we usually process postings list intersection in sublinear time? We can, if the index isn't changing too fast.

One way to do this is to use a *skip list* by augmenting postings lists with skip pointers (at indexing time), as shown in Figure 2.9 . Skip pointers are effectively shortcuts that allow us to avoid processing parts of the postings list that will not figure in the search results. The two questions are then where to place skip pointers and how to do efficient merging using skip pointers.



Postings lists with skip pointers.The postings intersection can use a skip pointer when the end point is still less than the item on the other list.

```
INTERSECTWITHSKIPS($p_1, p_2$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD($answer, docID(p_1)$)
            $p_1 \leftarrow next(p_1)$
            $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then if $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
                      then while $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
                              do $p_1 \leftarrow skip(p_1)$
                      else $p_1 \leftarrow next(p_1)$
              else if $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
                      then while $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
                              do $p_2 \leftarrow skip(p_2)$
                      else $p_2 \leftarrow next(p_2)$
return answer
```

Figure 2.10: Postings lists intersection with skip pointers.

Positional postings and phrase queries

Many complex or technical concepts and many organization and product names are multiword compounds or phrases. We would like to be able to pose a query such as Stanford University by treating it as a phrase so that a sentence in a document like *The inventor Stanford Ovshinsky never went to university.* is not a match. 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. As many as 10% of web queries are phrase queries, and many more are implicit phrase queries (such as person names), entered without use of double quotes.

Biword indexes

One approach to handling phrases 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

In this model, we treat each of these biwords as a vocabulary term. Being able to process two-word phrase queries is immediate. Longer phrases can be processed by breaking them down. The query stanford university palo alto can be broken into the Boolean query on biwords:

``stanford university'' AND ``university palo'' AND ``palo alto''

This query could be expected to work fairly well in practice, but there can and will be occasional false positives. Without examining the documents, we cannot verify that the documents matching the above Boolean query do actually contain the original 4 word phrase.

. These needs can be incorporated into the biword indexing model in the following way. 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. Now deem any string of terms of the form NX*N to be an extended biword. Each such extended biword is made a term in the vocabulary. For example:

renegotiation of the constitution

N X X N

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, which can be looked up in the index.

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

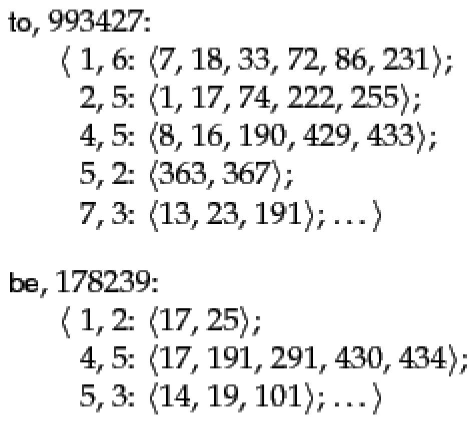
is parsed into

``cost overruns" AND ``overruns power" AND ``power plant"

whereas it might seem a better query to omit the middle biword. Better results can be obtained by using more precise part-of-speech patterns that define which extended biwords should be indexed.

Positional indexes

For the reasons given, a biword index is not the standard solution. Rather, a *positional index* is most commonly employed. Here, for each term in the vocabulary, we store postings of the form docID: ⟨ position1, position2, ... ⟩ , as shown in Figure 2.11 , where each position is a token index in the document. Each posting will also usually record the term frequency, for reasons discussed in Chapter 6 .



```
to, 993427:
    ⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;
      2, 5: ⟨1, 17, 74, 222, 255⟩;
      4, 5: ⟨8, 16, 190, 429, 433⟩;
      5, 2: ⟨363, 367⟩;
      7, 3: ⟨13, 23, 191⟩; . . . ⟩

be, 178239:
    ⟨ 1, 2: ⟨17, 25⟩;
      4, 5: ⟨17, 191, 291, 430, 434⟩;
      5, 3: ⟨14, 19, 101⟩; . . . ⟩
```

► **Figure 2.1** Positional index example. The word to has a document frequency 993,477, and occurs 6 times in document 1 at positions 7, 18, 33, etc.

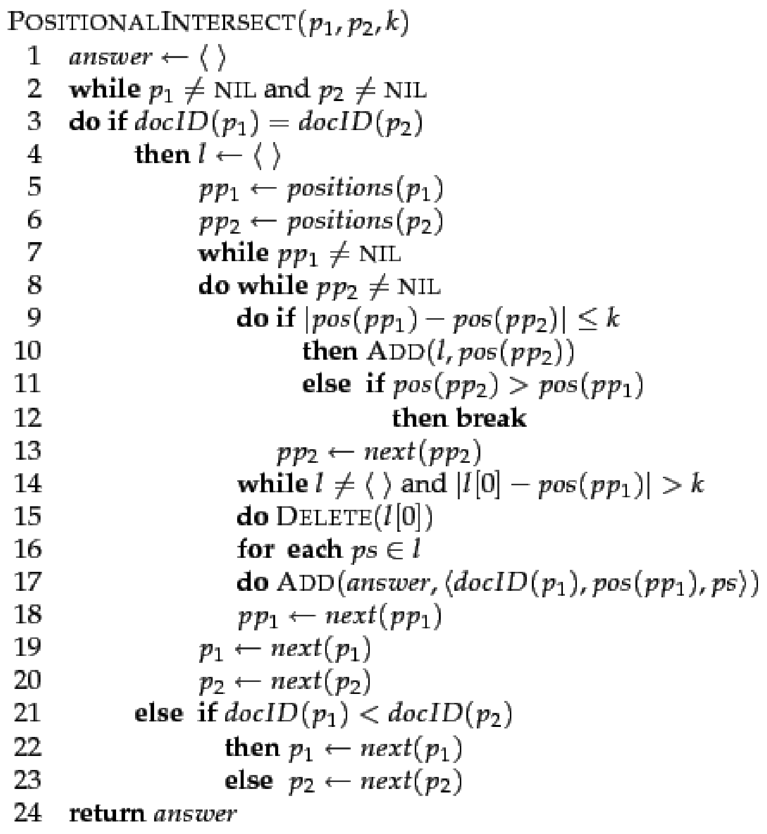
To process a phrase query, you still need to access the inverted index entries for each distinct term.

Worked example. Satisfying phrase queries.phrasequery Suppose the postings lists for to and be are as in Figure 2.11 , and the query is ``to be or not to be". The postings lists to access are: to, be, or, not. We will examine intersecting the postings lists for to and be.

to: ⟨ ...; 4: ⟨ ...,429,433 ⟩ ; ... ⟩

be: ⟨ ...; 4: ⟨ ...,430,434 ⟩ ; ... ⟩

End worked example.



```
PositionalIntersect($p_1, p_2, k$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then $l \leftarrow \langle\ \rangle$
           $pp_1 \leftarrow positions(p_1)$
           $pp_2 \leftarrow positions(p_2)$
           while $pp_1 \neq$ NIL
           do while $pp_2 \neq$ NIL
              do if $|pos(pp_1) - pos(pp_2)| \leq k$
                    then Add($l, pos(pp_2)$)
                    else if $pos(pp_2) > pos(pp_1)$
                            then break
                 $pp_2 \leftarrow next(pp_2)$
              while $l \neq \langle\ \rangle$ and $|l[0] - pos(pp_1)| > k$
              do Delete($l[0]$)
              for each $ps \in l$
              do Add($answer, \langle docID(p_1), pos(pp_1), ps \rangle$)
              $pp_1 \leftarrow next(pp_1)$
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
             then $p_1 \leftarrow next(p_1)$
             else $p_2 \leftarrow next(p_2)$
return answer
```

► **Figure 2.2** An algorithm for proximity intersection of postings lists p_1 and p_2. The algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in p_1 and p_2.

Positional index size.

Adopting a positional index expands required postings storage significantly, even if we compress position values/offsets as we will discuss in Section 5.3 (page). Indeed, 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. That is, the complexity of a Boolean query is $\Theta(T)$ rather than $\Theta(N)$. However, most applications have little choice but to accept this, since most users now expect to have the functionality of phrase and proximity searches.

Let's examine the space implications of having a positional index. A posting now needs an entry for each occurrence of a term. The index size thus depends on the average document size. The average web page has less than 1000 terms, but documents like SEC stock filings, books, and even some epic poems easily reach 100,000 terms. Consider a term with frequency 1 in 1000 terms on average. The result is that large documents cause an increase of two orders of magnitude in the space required to store the postings list:

	Expected	Expected entries
Document size	postings	in positional posting
1000	1	1
	1	100