Chapter 5 Databases and Information Retrieval

5.1 Database System

5.1.1 Database

A database is a collection of data that is stored for a specific purpose and organized in a manner that allows its contents to be easily accessed, managed, and updated. Although this definition includes stored data collections such as libraries, file cabinets, and address books, when we talk about databases we almost invariably mean a collection of data that is stored on a computer. There are two basic categories of database. The most commonly encountered category is the **transactional database**, used to store dynamic data, such as inventory contents, which is subject to change on an ongoing basis. The other category is the **analytical database**, used to store static data, such as geographical or chemical test results, which is rarely altered.

Strictly speaking, a database is just the stored data itself, although the term is often used, erroneously, to refer to a database and its management system (DBMS).

The first attempts at computer databases arose around the mid-twentieth century. Early versions were file-oriented. A database file became known as a table because its structure was the same as a paper-based data **table**. For the same reason, the columns within a table were called **fields** and the rows were called **records**. Computers were evolving during that same time period, and their potential for data storage and retrieval was becoming recognized.

The earliest computer databases were based on a flat file model, in which records were stored in text format. In this model, no relationships are defined between records. Without defining such relationships, records can only be accessed sequentially. For example, if you wanted to find the record for the fiftieth customer, you would have to go through the first 49 customer records in sequence first. The flat file model works well for situations in which you want to process all the records, but not for situations in which you want to find specific records within the database.

The **hierarchical model**, widely used in mainframe environments, was designed to allow structured relationships that would facilitate **data retrieval**. Within an inverted tree structure, relationships in the hierarchical model are parent-child and one-to-many. Each parent table may be related to multiple child tables, but each child table can only be related to a single parent table. Because table structures are permanently and explicitly linked in this model, data retrieval was fast. However, the model's rigid structure causes some problems. For example, you can't add a child table that is not linked to a parent table: if the parent table was "Doctors" and the child table was "Patients," you could not add a patient record independently. That would mean that if a new patient came into a community's health care system, under this system, their record could not be added until they had a doctor. The hierarchical structure also means that if a record is deleted in a parent table, all the records linked to it in child tables will be deleted as well.

Also based on an inverted tree structure, the next approach to database design was the **network model**. The network model allowed more complex connections than the hierarchical model: several inverted trees might share branches, for example. The model connected tables in sets, in which a record in an owner table could link to multiple records in a member table. Like the hierarchical model, the network model enabled very fast data retrieval. However it also had a number of problems. For example, a user would need a clear understanding of the database structure to be able to get information from the data. Furthermore, if a set structure was changed, any reference to it from an external program would have to be changed as well.

In the 1970s, the relational database was developed to deal with data in more complex ways. The **relational model** eventually dominated the industry and has continued to do so through to the present day. We'll explore the relational database in some detail in the next segment.

5.1.2 Relational Database①

<u>(5-1) In the relational database model, data is stored in relations, more commonly known as tables. Tables, records (sometimes known as **tuples**), and fields (sometimes known as **attributes**) are the basic components. Each individual piece of data, such as a last name or a telephone number, is stored in a table field and each record comprises a complete set of field data for a particular table.</u> In the following example, the table maintains customer shipping address information. Last_Name and other column headings are the fields. A record, or row, in the table comprises the complete set of field data in that context: all the address information that is required to ship an order to a specific customer. Each record can be identified by, and accessed through, a unique identifier called a primary key. In the Customer_Shipping table, for example, the Customer_ID field could serve as a primary key because each record has a unique value for that field's data.

<u>Customer_Shipping</u>

ID	First	Last	Apt.	Address	City	State	Zip
101	John	Smith	147	123 1st Street	Chicago	IL	60635
102	Jane	Doe	13 C	234 2nd Street	Chicago	IL	60647

① http://whatis.techtarget.com/reference/Learn-IT-The-Power-of-the-Database

103	June	Doe	14A	243 2nd Street	Chicago	IL	60647
104	George	Smith	N/A	345 3rd Street	Chicago	IL	60625

The term relational comes from set theory, rather than the concept that relationships between data drive the database. However, the model does, in fact, work through defining and exploiting the relationships between table data. Table relationships are defined as one-to-one (1 ∶ 1), one- to-many (1 ∶ *N*), or (uncommonly) many-to-many (*N* ∶ *M*).

If a pair of tables has a one-to-one relationship, each record in Table A relates to a single record in Table B. For example, in a table pairing consisting of a table of customer shipping addresses and a table of customer account balances, each single customer ID number would be related to a single identifier for that customer's account balance record. The one-to-one relationship reflects the fact that each individual customer has a single account balance.

If a pair of tables has a one-to-many relationship, each individual record in Table A relates to one or more records in Table B. For example, in a table pairing consisting of a table of university courses (Table A) and a table of student contact information (Table B), each single course number would be related to multiple records of student contact information. The one-to-many relationship reflects the fact that each individual course has multiple students enrolled in it.

If a pair of tables has a many-to-many relationship, each individual record in Table A relates to one or more records in Table B, and each individual record in Table B relates to one or more records in Table A. For example, in a table pairing consisting of a table of employee information and a table of project information, each employee record could be related to multiple project records and each project record could be related to multiple employee records. The many-to-many relationship reflects the fact that each employee may be involved in multiple projects and that each project involves multiple employees.

The relational database model developed from the proposals in "**A Relational Model of Data for Large Shared Databanks**", a paper presented by Dr. E. F. Codd in 1970. Codd, a research scientist at IBM, was exploring better ways to manage large amounts of data than were currently available. The heirarchical and network models of the time tended to suffer from problems with data redundancy and poor data integrity. By applying relational calculus, algebra, and logic to data storage and retrieval, Codd enabled the development of a more complex and fully articulated model than had previously existed.

One of Codd's goals was to create an English-like language that would allow non-technical users to interact with a database. Based on Codd's article, IBM started their System R research group to develop a relational database system. The group developed SQL/DS, which eventually became DB2. The system's language, SQL, became the industry's **de facto standard**. In 1985, Dr. Codd published a list of twelve rules for an ideal relational database. Although the rules may never have been fully implemented, they have provided a guideline for database developers for the last several decades.

Codd's 12 Rules:

(1) The Information Rule: Data must be presented to the user in table format.

(2) Guaranteed Access Rule: Data must be reliably accessible through a reference to the table name, **primary key**, and field name.

(3) Systematic Treatment of Null Values: Fields that are not primary keys should be able to remain empty (contain a null value).

(4) Dynamic On-Line Catalog Based on the Relational Model: The database structure should be accessible through the same tools that provide data access.

(5) Comprehensive Data Sublanguage Rule: The database must support a language that can be used for all interactions (SQL was developed from Codd's rules).

(6) View Updating Rule: Data should be available in different combinations (views) that can also be updated and deleted.

(7) High-level Insert, Update, and Delete: It should be possible to perform all these tasks on any set of data that can be retrieved.

(8) Physical Data Independence: Changes made to the architecture underlying the database should not affect the user interface.

(9) Logical Data Independence: If the logical structure of a database changes, that should not be reflected in the way the user views it.

(10) Integrity Independence: The language used to interact with the database should support user constraints that will maintain data integrity.

(11) Distribution Independence: If the database is distributed (physically located on multiple computers) that fact should not be apparent to the user.

(12) Nonsubversion Rule: It should not be possible to alter the database structure by any other means than the database language.

Although the relational model is by far the most prevalent one, there are a number of other models that are better suited to particular types of data. Alternatives to the relational model include:

Flat-File Databases: Data is stored in files consisting of one or more readable files, usually in text format.

Hierarchical Databases: Data is stored in tables with parent/child relationships with a strictly hierarchical structure.

Network Databases: Similar to the hierarchical model, but allows more flexibility; for example, a child table can be related to more than one parent table.

Object-Oriented Databases: The object-oriented database model was developed in the late 1980s and early 1990s to deal with types of data that the relational model was not well-suited for. Medical and multimedia data, for example, required a more flexible system for data representation and manipulation.

Object-Relational Databases: A hybrid model, combining features of the relational and object-oriented models.

Normalization is a guiding process for database table design that ensures, at four levels of stringency, increasing confidence that results of using the database are unambiguous and as intended. Basically a refinement process, normalization tests a table design for the way it stores data, so that it will not lead to the unintentional deletion of records, for example, and that it will reliably return the data requested.

Normalization degrees of relational database tables are defined as follows.

First normal form (1NF). This is the "basic" level of normalization and generally corresponds to the definition of any database, namely:

(1) It contains two-dimensional tables with rows and columns corresponding, respectively, to records and fields.

(2) Each field corresponds to the concept represented by the entire table: for example, each field in the Customer_Shipping table identifies some component of the customer's shipping address.

(3) No duplicate records are possible.

(4) All field data must be of the same kind. For example, in the "Zip" field of the Customer_Shipping table, only five consecutive digits will be accepted.

Second normal form (2NF). In addition to 1NF rules, each field in a table that does not determine the contents of another field must itself be a function of the other fields in the table. For example, in a table with three fields for customer ID, product sold, and price of the product when sold, the price would be a function of the customer ID (entitled to a discount) and the specific product.

Third normal form (3NF). In addition to 2NF rules, each field in a table must depend on the primary key. For example, using the customer table just cited, removing a record describing a customer purchase (because of a return perhaps) will also remove the fact that the product has a certain price. In the third normal form, these tables would be divided into two tables so that product pricing would be tracked separately. The customer information would depend on the primary key of that table, Customer_ID, and the pricing information would depend on the primary key of that table, which might be Invoice_Number.

Domain/key normal form (DKNF). In addition to 3NF rules, a key, which is a field used for sorting, uniquely identifies each record in a table. A domain is the set of permissible values for a field. By enforcing key and domain restrictions, the database is assured of being freed from modification anomalies. DKNF is the normalization level that most designers aim to achieve.

5.1.3 Database Management System

(5-2) A **Database Management System** is also known as DBMS, sometimes called a database manager or database system, which is a set of computer programs that controls the creation, organization, maintenance, and retrieval of data from the database stored in a computer. An excellent database system helps the end users to easily access and uses the data and also stores the new data in a systematic way.

A DBMS is a system software package that ensures the integrity and security of the data.

The most typical DBMS is a relational database management system (RDBMS). A newer kind of DBMS is the object-oriented database management system (ODBMS). The DBMS are categorized according to their data types and structure. It accepts the request for the data from an application program and instructs the operating system to transfer the appropriate data to the end user. A standard user and program interface is the **Structured Query Language** (SQL).

There are many Data Base Management System like MySQL, PostgreSQL, Microsoft Access, SQL Server, FileMaker, Oracle, RDBMS, dBASE, Clipper, FoxPro and many more that work independently and freely but also allow other database systems to be integrated with them. For this DBMS software comes with an **Open Database Connectivity** (ODBC) driver ensuring the databases to be integrated with it.

A DBMS includes four main parts: **modeling language**, data structure, **database query language**, and **transaction mechanisms modeling language**.

(1) Modeling language: A data modeling language to define the schema (the overall structure of the database) of each database hosted in the DBMS, according to the DBMS database model. The schema specifies data, data relationships, data semantics, and consistency constraints on the data. The four most common types of models are the:

① Hierarchical model

② Network model

③ Relational model

④ Object model

The optimal structure depends on the natural organization of the application’s data, and on the application’s requirements that include transaction rate (speed), reliability, maintainability, scalability, and cost.

(2) Data structures: Data structures which include fields, record, files and objects optimized to deal with very large amounts of data stored on a permanent data storage device like hard disks, CDs, DVDs, Tape etc.

(3) Database query language: Using the Database Query Language (DQL) users can formulate requests and generate reports. It also controls the security of the database. The DQL and the report writer allows users to interactively interrogate the database, analyze its data and update it according to the users privileges on data. For accessing and using personal records there is a need of password to retrieve the individual records among the bunch of records. For example: the individual records of each employee in a factory.

(4) Transaction mechanisms modeling language: The transaction mechanism modeling language ensures about data integrity despite concurrent user accesses and faults. It maintains the integrity of the data in the database by not allowing more than one user to update the same record at the same time. The unique index constraints prevent to retrieve the duplicate records like no two customers with the same customer numbers (key fields) can be entered into the database.

Among several types of DBMS, Relational Database Management System (RDBMS) and

Object-Oriented Database Management System (OODBMS) are the most commonly used DBMS software.

The RDBMS is a Database Management System (DBMS) based on the relational model in which data is stored in the form of tables and the relationship among the data is also stored in the form of tables. It was introduced by E. F. Codd, which is the most popular commercial and open source databases nowdays. The most popular RDBMS is: MySQL,PostgreSQL, Firebird,SQLite,DB2,Oracle Tutorials.

Object-Oriented Database Management System (OODBMS) in short Object Database Management System (ODBMS) is a Database Management System (DBMS) that supports the modeling and creation of data as objects. It includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects. An ODBMS must satisfy two conditions: it should be an object-oriented programming language and a DBMS too.

OODBMS extends the object programming language with transparently persistent data, concurrency control, data recovery, associative queries, and other database capabilities. At present it is on its development stage and used in Java and other object oriented programming languages.

Earlier it was introduced to replace the RDBMS due to its better performance and **scalability** but the inclusion of object-oriented features in RDBMS and the origin of Object-Relational Mappers (ORMs) made it powerful enough to defend its persistence. The higher switching cost also played a vital role to defend the existence of RDBMS. Now it is being used as a complement, not a replacement for relational databases.

Now it is being used in embedded persistence solutions in devices, on clients, in packaged software, in real-time control systems, and to power websites. The open source community has created a new wave of enthusiasm that's now fueling the rapid growth of ODBMS installations.

5.1.4 SQL

SQL is short for Structured Query Language, it is a domain-specific language used in programming and designed for managing data held in a Relational Database Management System (RDBMS).

(5-3) Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language, data manipulation language, and data control language. The scope of SQL includes data insert, query, update and delete, schema creation and modification, and data access control. Although SQL is often described as, and to a great extent is, a **declarative language**, it also includes procedural elements.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987. Since then, the standard has been revised to include a larger set of features. Despite the existence of such standards, most SQL code is not completely portable among different database systems without adjustments.

Data Definition Language: The Data Definition Language is used to create, alter, drop or delete a table in database. This includes a index through which you can make a link between tables in database. The list of Data Definition Language is given below.

(1) Create Table is used to create a table in database.

(2) Update is used to alter the table.

(3) Drop Table is used to drop a table in database.

Data Manipulation Language: The Data Manipulation Language is used to select, insert into, delete, update from database table. The keywords perform the particular task are given below.

(1) Select is used to display the records available in table.

(2) Update is used to modify or change in table.

(3) Delete is used to delete record from table.

(4) Insert into is used to add records into table.

The Data Control Language (DCL) authorizes users to access and manipulate data. Its two main statements are:

(1) GRANT authorizes one or more users to perform an operation or a set of operations on an object.

(2) REVOKE eliminates a grant, which may be the default grant.

SQL is subdivided into several language elements, including:

(1) **Clauses**, which are constituent components of statements and queries. (In some cases, these are optional) .

(2) Expressions, which can produce either scalar values, or tables consisting of columns and rows of data.

(3) **Predicates**, which specify conditions that can be evaluated to SQL three-valued logic (3VL) (true/false/unknown) or Boolean truth values and are used to limit the effects of statements and queries, or to change program flow.

(4) Queries, which retrieve the data based on specific criteria. This is an important element of SQL.

(5) Statements, which may have a persistent effect on **schemata** and data, or may control transactions, program flow, connections, sessions, or diagnostics.

(6) SQL statements also include the semicolon (“;”) statement terminator. Though not required on every platform, it is defined as a standard part of the SQL grammar.

(7) Insignificant whitespace is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

5.2 Information Retrieval

5.2.1 Introduction

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 satisfies an information need from within large collections (usually stored on computers).

(5-4) Information retrieval is concerned with representing, searching, and manipulating large collections of electronic text and other human-language data. IR systems and services are now widespread, with millions of people depending on them daily to facilitate business, education, and entertainment. Web search engines — Google, Bing, and others — are by far the most popular and heavily used IR services, providing access to up-to-date technical information, locating people and organizations, summarizing news and events, and simplifying comparison shopping. Digital library systems help medical and academic researchers learn about new journal articles and conference presentations related to there are as of research. Consumers turn to local search services to find retailers providing desired products and services. Within large companies, enterprise search systems act as repositories for e-mail, memos, technical reports, and other business documents, providing corporate memory by preserving these documents and enabling access to the knowledge contained within them. Desktop search systems permit users to search their personal e-mail, documents, and files.

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 are 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 documents by explicit markup (such as the coding underlying Web pages). IR is also used to facilitate “**semistructured**” search such as finding a document where the title contains Java and the body contains threading.

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. It is similar to arranging books on a bookshelf according to their topic. 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. It is often approached by first manually classifying some documents and then hoping to be able to classify new documents automatically.

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. Distinctive issues are needed to gather documents for indexing, being able to build systems that work efficiently at this enormous scale, and handling particular aspects of the Web, such as the exploitation of hypertext and not being fooled by site providers manipulating page content in an attempt to boost their search engine rankings, given the commercial importance of the Web. 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 or Windows Vista's Instant Search). E-mail 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. Distinctive issues here include handling the broad range of document types on a typical personal computer, and making the search system maintenance free and sufficiently lightweight in terms of startup, processing, and disk space usage that it can run on one machine without annoying its owner. 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, a database of patents, or research articles on biochemistry. In this case, the documents will typically be stored on centralized file systems and one or a handful of dedicated machines will provide search over the collection.

5.2.2 An Example of Information Retrieval①

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. 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.

But for many purposes, you do need more:

(1) To process large document collections quickly. The amount of online data has grown at

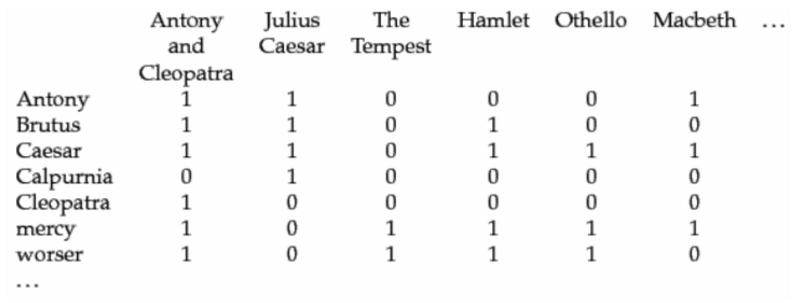
① Christopher D et al. Introduction to Information Retrieval. Cambridge University Press, 2008. http://nlp. stanford. edu/ IR-book/.

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".

(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. 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 Fig.5-1. Terms are the indexed units; they are usually words, and for the moment you can think of them as words, but the information retrieval literature normally speaks of terms because some of them, such as perhaps I-9 or Hong Kong 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.



	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	
...							

Fig.5-1 A term-document incidence matrix

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 (Fig.5-1).

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 operators and, or, and not. The model views each document as just a set of words.

Let us now consider a more realistic scenario, simultaneously using the opportunity to introduce some **terminology** and **notation**. Suppose we have **N**=1million 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. We will refer to the group of documents over which we perform retrieval as the (document) collection. It is sometimes also referred to as a **corpus** (a body of texts). Suppose each document is about 1,000 words long (2~3 book pages). If we assume an average of 6 B 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.

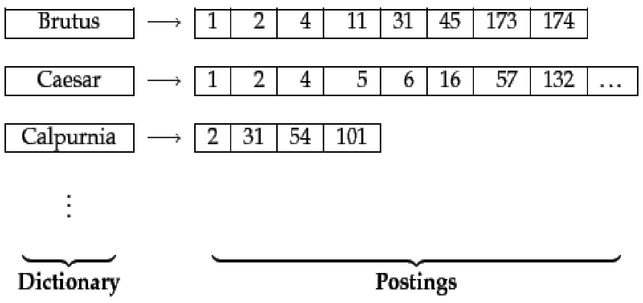
Our goal is to develop a system to address the ad hoc retrieval task. This is the most standard IR task. In it, a system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query. 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. A document is relevant if it is one that the user perceives as containing information of value with respect to their personal information need. Our example above was rather artificial in that the information need was defined in terms of particular words, whereas usually a user is interested in a topic like "pipeline leaks" and would like to find relevant documents regardless of whether they precisely use those words or express the concept with other words such as pipeline rupture. (5-5)To assess the effectiveness of an IR system (i.e., the quality of its search results), a user will usually want to know two key statistics about the system's returned results for a query:

(1) **Precision** : What fraction of the returned results are relevant to the information need?

(2) **Recall** : What fraction of the relevant documents in the collection were returned by the system?

We now cannot build a term-document matrix in a naive way. A 500K×1M matrix has half-a-trillion 0's and 1's - too many to fit in a computer's memory. But the crucial observation is that the matrix is extremely **sparse**, that is, it has few non-zero entries. Because each document is 1,000 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's positions.

This idea is central to the first major concept in information retrieval, the **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. The basic idea of an inverted index is shown in Fig.5-2. We keep a dictionary of terms (sometimes also referred to as a vocabulary or **lexicon**; in this book, we use dictionary for the data structure and vocabulary for the set of terms). Then for each term, we have a list that records which documents the term occurs in. Each item in the list — which records that a term appeared in a document (and, later, often, the positions in the document) — is conventionally called a posting. The list is then called a **postings list**, and all the postings lists taken together are referred to as the postings. The dictionary in Fig.5-2 has been sorted alphabetically and each postings list is sorted by document ID.



Brutus → 1 2 4 11 31 45 173 174
Caesar → 1 2 4 5 6 16 57 132 …
Calpurnia → 2 31 54 101
Dictionary
Postings

Fig.5-2 the two parts of 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:

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 let it …

(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.

You can think of tokens and normalized tokens as also loosely equivalent to words. Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index by sort-based indexing as Fig.5-3.

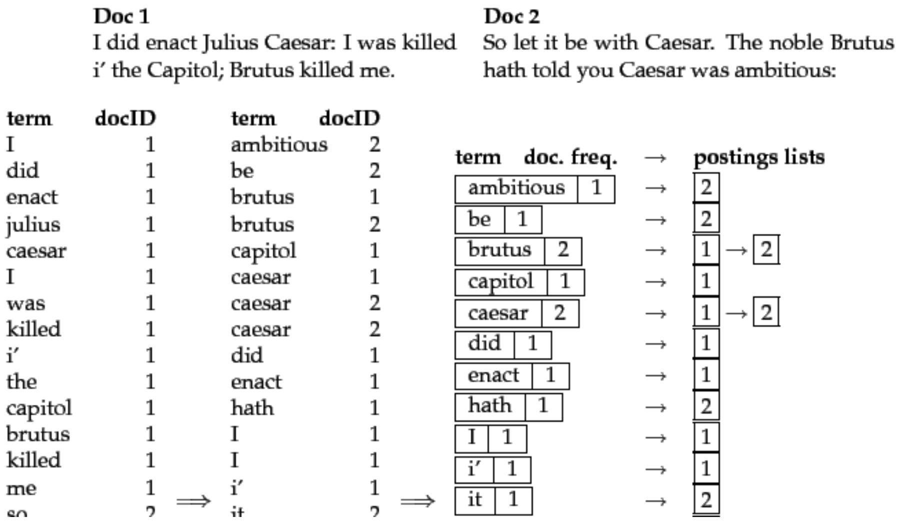


Fig.5-3 Building an index by sorting and grouping

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 is 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 Fig.5-3. The core indexing step is sorting this list so that the terms are alphabetical, giving us the representation in the middle column of Fig.5-3. Multiple occurrences of the same term from the same document are then merged. Instances of the same term are then grouped, and the result is split into a dictionary and postings, as shown in the right column of Fig.5-3. Since a term generally occurs in a number of documents, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the number of documents which contain each term (the document frequency, which is here also the length of each postings list). This information is not vital for a basic Boolean search engine, but it allows us to improve the efficiency of the search engine at query time, and it is a statistic later used in many ranked retrieval models. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. 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 commonly kept in memory, while postings lists are normally kept on disk, so the size of each is important, we need 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 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, which require additional pointers. Variable length arrays win in space requirements by avoiding the 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 Fig.5-2), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

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 partially shown in Fig.5-3. 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 Fig.5-4.

```
INTERSECT($p_1$, $p_2$)
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 $p_1 \leftarrow next(p_1)$
                 else $p_2 \leftarrow next(p_2)$
return answer
```

Fig.5-4 Algorithm for the intersection of two postings lists $p_1$and p_2

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.)

5.2.3 Open Source IR System[①]

There exists a wide variety of open-source information retrieval systems that you may use for exercises in this book and to start conducting your own information retrieval experiments. As always, a (non-exhaustive) list of open-source IR systems can be found in Wikipedia. Since this list of available systems is so long, we do not even try to cover it in detail. Instead, we restrict ourselves to a very brief overview of three particular systems that were chosen because of their popularity, their influence on IR research, or their intimate relationship with the contents of this book. All three systems are available for download from the Web and may be used free of charge, according to their respective licenses.

1. Lucene

Lucene is an indexing and search system implemented in Java, with ports to other programming languages. The project was started by Doug Cutting in 1997. Since then, it has grown from a single-developer effort to a global project involving hundreds of developers in various countries. It is currently hosted by the Apache Foundation. Lucene is by far the most successful open-source search engine. Its largest installation is quite likely Wikipedia: All queries entered into Wikipedia's search form are handled by Lucene. A list of other projects relying on its indexing and search capabilities can be found on Lucene's "PoweredBy" page.

Known for its modularity and extensibility, Lucene allows developers to define their own

① http://www.ir.uwaterloo.ca/book/01-introduction.pdf indexing and retrieval rules and **formulae**. Under the hood, Lucene's retrieval framework is based on the concept of fields: Every document is a collection of fields, such as its title, body, URL, and so forth. This makes it easy to specify **structured search requests** and to give different weights to different parts of a document.

Due to its great popularity, there is a wide variety of books and tutorials that help you get started with Lucene quickly. Try the query "Lucene tutorial" in your favorite Web search engine.

2. Indri

Indri is an academic information retrieval system written in C++. It is developed by researchers at the University of Massachusetts and is part of the Lemur project, a joint effort of the University of Massachusetts and Carnegie Mellon University.

Indri is well known for its high retrieval effectiveness and is frequently found among the top-scoring search engines at TREC. Its retrieval model is a combination of the language modeling approaches. Like Lucene, Indri can handle multiple fields per document, such as title, body, and anchor text, which is important in the context of Web search. It supports automatic query expansion by means of **pseudo-relevance feedback**, a technique that adds related terms to an initial search query, based on the contents of an initial set of search results. It also supports query-independent document scoring that may, for instance, be used to prefer more recent documents over less recent ones when ranking the search results.

3. Wumpus

Wumpus is an academic search engine written in C++ and developed at the University of Waterloo. Unlike most other search engines, Wumpus has no built-in notion of "documents" and does not know about the beginning and the end of each document when it builds the index. Instead, every part of the text collection may represent a potential unit for retrieval, depending on the structural search constraints specified in the query. This makes the system particularly attractive for search tasks in which the ideal search result may not always be a whole document, but may be a section, a paragraph, or a sequence of paragraphs within a document.

Wumpus supports a variety of different retrieval methods, including the proximity ranking function, the BM25 algorithm, and the language modeling and divergence from randomness approaches. In addition, it is able to carry out real-time index updates (i.e., adding/removing files to /from the index) and provides support for multi-user security restrictions that are useful if the system has more than one user, and each user is allowed to search only parts of the index.

5.2.4 Performance Measure

To measure ad hoc information retrieval effectiveness in the standard way, we need a test collection consisting of three things:

(1) A document collection.

(2) A test suite of information needs, expressible as queries.

(3) A set of relevance judgments, standardly a binary assessment of either relevant or nonrelevant for each query-document pair.

The standard approach to information retrieval system evaluation revolves around the notion of **relevant** and **nonrelevant** documents. With respect to a user information need, a document in the test collection is given a binary classification as either relevant or nonrelevant. This decision is referred to as the **gold standard** or **ground truth** judgment of relevance. The test document collection and suite of information needs have to be of a reasonable size: you need to average performance over fairly large test sets, as results are highly variable over different documents and information needs. As a rule of thumb, 50 information needs has usually been found to be a sufficient minimum.

Relevance is assessed relative to an information need, not a query. For example, an information need might be:

Information on whether drinking red wine is more effective at reducing your risk of heart attacks than white wine.

This might be translated into a query such as:

wine AND red AND white AND heart AND attack AND effective

A document is relevant if it addresses the stated information need, not because it just happens to contain all the words in the query. This distinction is often misunderstood in practice, because the information need is not overt. If a user types python into a Web search engine, they might want to know where they can purchase a pet python. Or they might want information on the programming language Python. From a one word query, it is very difficult for a system to know what the information need is. But, nevertheless, the user has one, and can judge the returned results on the basis of their relevance to it. To evaluate a system, we require an overt expression of an information need, which can be used for judging returned documents as relevant or nonrelevant. At this point, we make a simplification: relevance can reasonably be thought of as a scale, with some documents highly relevant and others marginally so. But for the moment, we will use just a binary decision of relevance.

Many systems contain various weights (often known as parameters) that can be adjusted to tune system performance. It is wrong to report results on a test collection which were obtained by tuning these parameters to maximize performance on that collection. That is because such tuning overstates the expected performance of the system, because the weights will be set to maximize performance on one particular set of queries rather than for a random sample of queries. In such cases, the correct procedure is to have one or more development test collections, and to tune the parameters on the development test collection. The tester then runs the system with those weights on the test collection and reports the results on that collection as an unbiased estimate of performance.

5.3 Web Search Basics

5.3.1 Background and History

The Web is unprecedented in many ways: unprecedented in scale, unprecedented in the almost-complete lack of coordination in its creation, and unprecedented in the diversity of backgrounds and motives of its participants. Each of these contributes to making Web search different — and generally far harder — than searching “traditional” documents.

The invention of hypertext, envisioned by Vannevar Bush in the 1940’s and first realized in working systems in the 1970’s, significantly precedes the formation of the World Wide Web (which we will simply refer to as the Web), in the 1990’s. Web usage has shown tremendous growth to the point where it now claims a good fraction of humanity as participants, by relying on a simple, open client-server design: ① the server communicates with the client via a protocol (the http or hypertext transfer protocol) that is lightweight and simple, **asynchronously** carrying a variety of payloads (text, images and — over time — richer media such as audio and video files) encoded in a simple markup language called HTML (for Hypertext Markup Language); ② the client — generally a browser, an application within a graphical user environment — can ignore what it does not understand. Each of these seemingly innocuous features has contributed enormously to the growth of the Web, so it is worthwhile to examine them further.

The basic operation is as follows: a client (such as a browser) sends an http request to a Web server. The browser specifies a URL (for Uniform Resource Locator) such as http://www.stanford.edu/home/atoz/contact.html. In this example URL, the string “http” refers to the protocol to be used for transmitting the data. The string “www.stanford.edu” is known as the domain and specifies the root of a hierarchy of Web pages (typically mirroring a filesystem hierarchy underlying the Web server). In this example, “/home/atoz/contact.html” is a path in this hierarchy with a file “contact.html” that contains the information to be returned by the Web server at www.stanford.edu in response to this request. The HTML — encoded file contact.html holds the hyperlinks and the content (in this instance, contact information for Stanford University), as well as formatting rules for rendering this content in a browser. Such an http request thus allows us to fetch the content of a page, something that will prove to be useful to us for crawling and indexing documents.

The designers of the first browsers made it easy to view the HTML markup tags on the content of a URL. This simple convenience allowed new users to create their own HTML content without extensive training or experience; rather, they learned from example content that they liked. As they did so, a second feature of browsers supported the rapid proliferation of Web content creation and usage: browsers ignored what they did not understand. This did not, as one might fear, lead to the creation of numerous incompatible dialects of HTML. What it did promote was amateur content creators who could freely experiment with and learn from their newly created Web pages without fear that a simple syntax error would “bring the system down”. Publishing on the Web became a mass activity that was not limited to a few trained programmers, but rather open to tens and eventually hundreds of millions of individuals. For most users and for most information needs, the Web quickly became the best way to supply and consume information on everything from rare ailments to subway schedules.

The mass publishing of information on the Web is essentially useless unless this wealth of information can be discovered and consumed by other users. Early attempts at making Web information "discoverable" fell into two broad categories: ① full-text index search engines such as Altavista, Excite and Infoseek and ② **taxonomies** populated with Web pages in categories, such as Yahoo!. The former presented the user with a keyword search interface supported by inverted indexes and ranking mechanisms. The latter allowed the user to browse through a hierarchical tree of category labels. While this is at first blush a convenient and intuitive metaphor for finding Web pages, it has a number of drawbacks: first, accurately classifying Web pages into taxonomy tree nodes is for the most part a manual editorial process, which is difficult to scale with the size of the Web. Arguably, we only need to have "high-quality" Web pages in the taxonomy, with only the best Web pages for each category. However, just discovering these and classifying them accurately and consistently into the taxonomy entails significant human effort. Furthermore, in order for a user to effectively discover Web pages classified into the nodes of the taxonomy tree, the user's idea of what sub-tree(s) to seek for a particular topic should match that of the editors performing the classification. This quickly becomes challenging as the size of the taxonomy grows; the Yahoo! taxonomy tree surpassed 1,000 distinct nodes fairly early on. Given these challenges, the popularity of taxonomies declined over time, even though variants (such as About.com and the Open Directory Project) sprang up with subject-matter experts collecting and annotating Web pages for each category.

The first generation of Web search engines transported classical search techniques to the Web domain, focusing on the challenge of scale. The earliest Web search engines had to contend with indexes containing tens of millions of documents, which were a few orders of magnitude larger than any prior information retrieval system in the public domain. Indexing, query serving and ranking at this scale required the harnessing together of tens of machines to create highly available systems, again at scales not witnessed hitherto in a consumer-facing search application. The first generation of Web search engines was largely successful at solving these challenges while continually indexing a significant fraction of the Web, all the while serving queries with sub-second response times. However, the quality and relevance of Web search results left much to be desired owing to the idiosyncrasies of content creation on the Web. This necessitated the invention of new ranking and spam-fighting techniques in order to ensure the quality of the search results. While classical information retrieval techniques continue to be necessary for Web search, they are not by any means sufficient. A key aspect is that whereas classical techniques measure the relevance of a document to a query, there remains a need to gauge the authoritativeness of a document based on cues such as which Web site hosts it.

5.3.2 Web Search Features

The essential feature that led to the explosive growth of the Web-decentralized content publishing with essentially no central control of authorship — turned out to be the biggest challenge for Web search engines in their quest to index and retrieve this content. Web page authors created content in dozens of (natural) languages and thousands of dialects, thus demanding many different forms of stemming and other linguistic operations. Because publishing was now open to tens of millions, Web pages exhibited heterogeneity at a daunting scale, in many crucial aspects. First, content-creation was no longer the privy of editorially-trained writers; while this represented a tremendous democratization of content creation, it also resulted in a tremendous variation in grammar and style (and in many cases, no recognizable grammar or style). Indeed, Web publishing in a sense unleashed the best and worst of desktop publishing on a planetary scale, so that pages quickly became riddled with wild variations in colors, fonts and structure. Some Web pages, including the professionally created home pages of some large corporations, consisted entirely of images (which, when clicked, led to richer textual content) — and therefore, no **indexable text**.

What about the substance of the text in Web pages? The democratization of content creation on the Web meant a new level of **granularity** in opinion on virtually any subject. This meant that the Web contained truth, lies, contradictions and suppositions on a grand scale. This gives rise to the question: which Web page does one trust? In a simplistic approach, one might argue that some publishers are trustworthy and others not — begging the question of how a search engine is to assign such a measure of trust to each Web site or Web page. More subtly, there may be no universal, user-independent notion of trust; a Web page whose contents are trustworthy to one user may not be so to another. In traditional (non-Web) publishing this is not an issue: users self-select sources they find trustworthy. Thus one reader may find the reporting of **The New York Times** to be reliable, while another may prefer **The Wall Street Journal**. But when a search engine is the only viable means for a user to become aware of (let alone select) most content, this challenge becomes significant.

While the question "how big is the Web?" has no easy answer, the question "how many Web pages are in a search engine's index" is more precise, although, even this question has issues. By the end of 1995, Altavista① reported that it had crawled and indexed approximately 30 million static Web pages. Static Web pages are those whose content does not vary from one request for that page to the next. For this purpose, a professor who manually updates his home page every week is considered to have a static Web page, but an airport's flight status page is considered to be dynamic. Dynamic pages are typically mechanically generated by an application server in response to a query to a database, as show in Fig.5-5. One sign of such a page is that the URL has the character "?" in it. Since the number of static Web pages was believed to be doubling every few months in 1995, early Web search engines such as Altavista had to constantly add hardware and bandwidth for crawling and indexing Web pages.

① AltaVista was a Web search engine established in 1995. It became one of the most-used early search engines, but lost ground to Google and was purchased by Yahoo! in 2003.

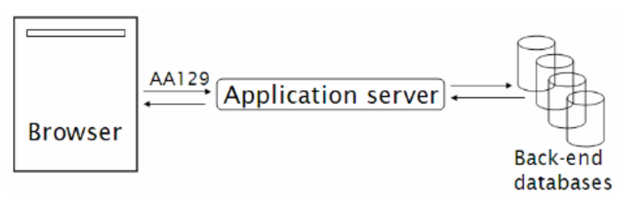


Fig.5-5 Dynamically generated Web page

It is crucial that we understand the users of Web search as well. This is again a significant change from traditional information retrieval, where users were typically professionals with at least some training in the art of phrasing queries over a well-authored collection whose style and structure they understood well. In contrast, Web search users tend to not know (or care) about the heterogeneity of Web content, the syntax of query languages and the art of phrasing queries; indeed, a mainstream tool (as Web search has come to become) should not place such onerous demands on billions of people. A range of studies has concluded that the average number of keywords in a Web search is somewhere between 2 and 3. Syntax operators (Boolean connectives, wildcards, etc.) are seldom used, again a result of the composition of the audience – "normal" people, not information scientists.

It is clear that the more user traffic a Web search engine can attract, the more revenue it stands to earn from **sponsored search**. How do search engines differentiate themselves and grow their traffic? Here Google identified two principles that helped it grow at the expense of its competitors: ① a focus on relevance, specifically precision rather than recall in the first few results; ② a user experience that is lightweight, meaning that both the search query page and the search results page are uncluttered and almost entirely textual, with very few graphical elements. The effect of the first was simply to save users time in locating the information they sought. The effect of the second is to provide a user experience that is extremely responsive, or at any rate not bottlenecked by the time to load the search query or results page.

There appear to be three broad categories into which common Web search queries can be grouped: ① informational, ② navigational and ③ transactional. We now explain these categories; it should be clear that some queries will fall in more than one of these categories, while others will fall outside them.

Informational queries seek general information on a broad topic, such as leukemia or Provence. There is typically not a single Web page that contains all the information sought; indeed, users with informational queries typically try to assimilate information from multiple Web pages.

Navigational queries seek the website or home page of a single entity that the user has in mind, say Lufthansa airlines. In such cases, the user's expectation is that the very first search result should be the home page of Lufthansa. The user is not interested in a plethora of documents containing the term Lufthansa; for such a user, the best measure of user satisfaction is precision at 1.

A transactional query is one that is a prelude to the user performing a transaction on the Web — such as purchasing a product, downloading a file or making a reservation. In such cases, the search engine should return results listing services that provide form interfaces for such transactions.

Discerning which of these categories a query falls into can be challenging. The category not only governs the algorithmic search results, but the suitability of the query for sponsored search results (since the query may reveal an intent to purchase). For navigational queries, some have argued that the search engine should return only a single result or even the target Web page directly. Nevertheless, Web search engines have historically engaged in a battle of bragging rights over which one indexes more Web pages. Does the user really care? Perhaps not, but the media does highlight estimates (often statistically indefensible) of the sizes of various search engines. Users are influenced by these reports and thus, search engines do have to pay attention to how their index sizes compare to competitors'. For informational (and to a lesser extent, transactional) queries, the user does care about the comprehensiveness of the search engine.

Fig.5-6 shows a composite picture of a Web search engine including the crawler, as well as both the Web page and advertisement indexes. The portion of the figure under the curved dashed line is internal to the search engine.

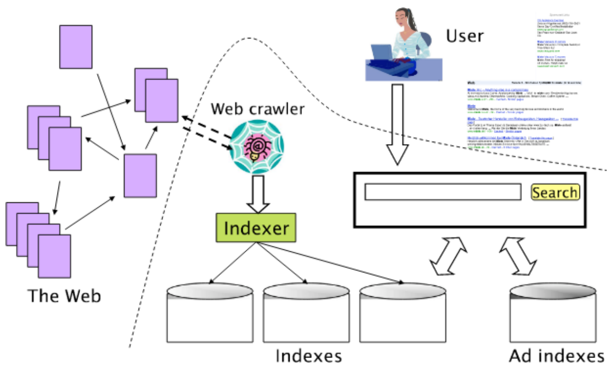


Fig.5-6 The various components of a Web search engine

5.3.3 Web Crawling and Indexes

Web crawling is the process by which we gather pages from the Web, in order to index them and support a search engine. The objective of crawling is to quickly and efficiently gather as many useful Web pages as possible, together with the link structure that interconnects them. The complexities of the Web stem from its creation by millions of uncoordinated individuals. In this section we study the resulting difficulties for crawling the Web. The focus is the component shown in Fig.5-6 as Web crawler ; it is sometimes referred to as a spider.

1. Features a Crawler Must Provide

1) **Robustness**

The Web contains servers that create spider traps, which are generators of Web pages that mislead crawlers into getting stuck fetching an infinite number of pages in a particular domain. Crawlers must be designed to be resilient to such traps. Not all such traps are malicious; some are the inadvertent side-effect of faulty Web site development.

2) Politeness

Web servers have both implicit and explicit policies regulating the rate at which a crawler can visit them. These politeness policies must be respected.

2. Features a Crawler Should Provide

1) Distributed

The crawler should have the ability to execute in a distributed fashion across multiple machines.

2) Scalable

The crawler architecture should permit scaling up the crawl rate by adding extra machines and bandwidth.

3) Performance and Efficiency

The crawl system should make efficient use of various system resources including processor, storage and network bandwidth.

4) Quality

Given that a significant fraction of all Web pages are of poor utility for serving user query needs, the crawler should be biased towards fetching "useful" pages first.

5) Freshness

In many applications, the crawler should operate in continuous mode: it should obtain fresh copies of previously fetched pages. A search engine crawler, for instance, can thus ensure that the search engine's index contains a fairly current representation of each indexed Web page. For such continuous crawling, a crawler should be able to crawl a page with a frequency that approximates the rate of change of that page.

6) Extensible

Crawlers should be designed to be extensible in many ways — to cope with new data formats, new fetch protocols, and so on. This demands that the crawler architecture be modular.

3. Crawling

The basic operation of any hypertext crawler, whether for the Web, an intranet or other hypertext document collection, is as follows. The crawler begins with one or more URLs that constitute a seed set. It picks a URL from this seed set, and then fetches the Web page at that URL. The fetched page is then **parsed**, to extract both the text and the links from the page (each of which points to another URL). The extracted text is fed to a text indexer. The extracted links (URLs) are then added to a URL frontier, which at all times consists of URLs whose corresponding pages have yet to be fetched by the crawler. Initially, the URL frontier contains the seed set; as pages are fetched, the corresponding URLs are deleted from the URL frontier. The entire process may be viewed as traversing the Web graph. In continuous crawling, the URL of a fetched page is added back to the frontier for fetching again in the future.

This seemingly simple **recursive traversal** of the Web graph is complicated by the many demands on a practical Web crawling system: the crawler has to be distributed, scalable, efficient, polite, robust and extensible while fetching pages of high quality. We examine the effects of each of these issues. Our treatment follows the design of the Mercator crawler that has formed the basis of a number of research and commercial crawlers. As a reference point, fetching a billion pages (a small fraction of the static Web at present) in a month-long crawl requires fetching several hundred pages each second. We will see how to use a multi-threaded design to address several bottlenecks in the overall crawler system in order to attain this fetch rate.

Before proceeding to this detailed description, we reiterate for readers who may attempt to build crawlers of some basic properties any non-professional crawler should satisfy:

(1) Only one connection should be open to any given host at a time.

(2) A waiting time of a few seconds should occur between successive requests to a host.

(3) Politeness restrictions should be obeyed.

5.4 Key Terms and Review Questions

1. Technical Terms

analytical database	分析型数据库
table	表
field	字段
record	记录
hierarchical model	层次模型
data retrieval	数据检索
network model	网状模型
relational model	关系模型
tuple	元组
attribute	属性
de facto standard	事实标准
primary key	主键
normalization	标准化、规范化
first normal form (1NF)	第一范式
Database Management System	数据库管理系统
Structured Query Language	结构化查询语言
Open Database Connectivity	开放数据库连接
modeling language	建模语言
database query language	数据库查询语言
transaction mechanisms modeling language	事务机制建语言

declarative language	说明性语言
Data Definition Language	数据定义语言
Data Manipulation Language	数据操纵语言
Data Control Language	数据控制语言
clause	子句
predicate	谓词
schemata	纲要
information retrieval	信息检索
unstructured data	非结构化数据
structured data	结构化数据
canonical	标准
semistructured	半结构化的
Web search	网页搜索
personal information retrieval	个人信息检索
spam	垃圾邮件
junk mail	垃圾邮件
grep	UNIX 工具程序；可做文件内的字符串查找
wildcard	通配符
index	索引
Boolean retrieval model	布尔检索模型
terminology	术语
notation	符号
corpus	语料库
magnitude	数量级
precision	准确率
recall	召回率（查全率）
inverted index	倒排索引
sparse	稀疏
lexicon	词典
postings list	位置表
linked list	链表
ariable length array	变长数组
offset	偏移量
formulae	规则
structured search requests	结构化的搜索请求
pseudo-relevance feedback	伪关联性反馈
relevant	相关
nonrelevant	不相关
ground truth	基本事实，机器学习中指正确的标注信息

asynchronously	异步地
taxonomy	分类学
indexable text	可索引的文本
granularity	粒度
sponsored search	付费搜索
Web crawling	网页信息采集
robustnet	健壮性
recursive traversal	递归遍历
parse	解析

2. Translation Exercises

(5-1) In the relational database model, data is stored in relations, more commonly known as tables. Tables, records (sometimes known as **tuples**), and fields (sometimes known as *attributes*) are the basic components. Each individual piece of data, such as a last name or a telephone number, is stored in a table field and each record comprises a complete set of field data for a particular table.

(5-2) A **Database Management System** is also know as DBMS, sometimes called a database manager or database system, which is a set of computer programs that controls the creation, organization, maintenance, and retrieval of data from the database stored in a computer. An excellent database system helps the end users to easily access and uses the data and also stores the new data in a systematic way.

(5-3) Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language, data manipulation language, and data control language. The scope of SQL includes data insert, query, update and delete, schema creation and modification, and data access control. Although SQL is often described as, and to a great extent is, a declarative language, it also includes procedural elements.

(5-4) Information retrieval is concerned with representing, searching, and manipulating large collections of electronic text and other human-language data. IR systems and services are now widespread, with millions of people depending on them daily to facilitate business, education, and entertainment. Web search engines — Google, Bing, and others — are by far the most popular and heavily used IR services, providing access to up-to-date technical information, locating people and organizations, summarizing news and events, and simplifying comparison shopping.

(5-5) To assess the effectiveness of an IR system (i.e., the quality of its search results), a user will usually want to know two key statistics about the system's returned results for a query:

(1) **Precision**: What fraction of the returned results are relevant to the information need?

(2) **Recall**: What fraction of the relevant documents in the collection were returned by the system?

References

[1] Structured Query Language (SQL)[A/OL]. [2017-03-22].https://docs.microsoft.com/en-us/sql/odbc/reference/structured-query-language-sql.

[2] Manning C D, Raghavan P, Hinrich Schütze. Introduction to Information Retrieval[M]. Cambridge University Press, 2008.

[3] Stefan Büttcher, Clarke C L, Cormack G V. Information Retrieval: Implementing and Evaluating Search Engines[M/OL]. MIT Press, 2010. http://www.ir.uwaterloo.ca/book/bib.html

[4] Learn IT: The Power of the Database[EB/OL]. 2017-03-21. http://whatis.techtarget.com/reference/ Learn-IT-The-Power-of-the-Database.

[5] Baeza-Yates R A, Ribeiro-Neto B. Modern Information Retrieval[M]. 2nd ed. Reading, Massachusetts: Addison-Wesley, 2010.

[6] Barroso L A, Dean J, Hölzle U. Web search for a planet: The Google cluster architecture[J]. IEEE Micro, 2003, 23(2):22–28.