SEARCHING MULTIMEDIA DATABASES BY CONTENT

SEARCHING MULTIMEDIA DATABASES BY CONTENT

Christos FALOUTSOS
University of Maryland
College Park, MD, USA

KLUWER ACADEMIC PUBLISHERS
Boston/London/Dordrecht

Dedication

To my wife Christina and my parents Sophia and Nikos.

CONTENTS

PREFACE

The problem on target is the searching of large multimedia databases by content. For example, '*given a collection of color images, find the ones that look like a sunset*'. Research on a specific domain (eg., machine vision, voice processing, text retrieval) typically focuses on feature extraction and similarity functions, with little regard to the efficiency of the search. Conversely, database research has focused on fast searching for a set of numbers or strings or vectors.

The main goal of this book is to try to bridge the gap between the database and signal processing communities. The book provides enough background information for both areas, presenting the intuition and the mechanics of the best tools in each area, as well as discussing when and how these tools can work together.

The structure of the book reflects its goal. The first half of the book reviews the most successful database access methods, in increasing complexity. It starts from primary-key access methods, where B-trees and hashing are the industry work-horses, and continues with methods that handle n-dimensional vectors. A chapter is also devoted to text retrieval, because text is important on its own right, and because it has led to some extremely useful ideas, like relevance feedback, clustering and the vector-space model. In all the sections, the emphasis is on practical approaches that have been incorporated in commercial systems, or that seem very promising.

The second half of the book uses the above access methods to achieve fast searching in a database of signals. In all cases, the underlying idea is to extract n features from each signal (eg, the first n Discrete Fourier Transform (DFT) coefficients), to map a signal into a point in n-dimensional space; subsequently, the access methods of the first part can be applied to search for similar signals in time that is much faster than sequential scanning, *without* missing any signals that sequential scanning would find ('complete' searching). Then, the book presents some recent, successful applications of this approach on time series and color images. It also describes methods to extract automatically features from a distance function, using the so-called Multidimensional Scaling (MDS), as well as a newer, faster approximation, called 'FastMap'.

Finally, the appendix gives some background information on fundamental signal processing and linear algebra techniques: the traditional Discrete Fourier Transform (DFT), the Discrete Cosine Transform (used in the JPEG standard), the Discrete Wavelet transform, which is the state-of-the-art in signal processing, the Karhunen-Loeve transform for optimal dimensionality reduction, and the closely related Singular Value Decomposition (SVD), which is a powerful tool for approximation problems. In all the above discussions, the emphasis is on the physical intuition behind each technique, as opposed to the mathematical properties. Source code is also provided for several of them.

The book is targeted towards researchers and developers of multimedia systems. It can also serve as a textbook for a one-semester graduate course on multimedia searching, covering both access methods as well as the basics of signal processing. The reader is expected to have an undergraduate degree in engineering or computer science, and experience with some high-level programming language (eg., 'C'). The exercises at the end of each chapter are rated according to their difficulty. The rating follows a logarithmic scheme similar to the one by Knuth [Knu73]:

00 Trivial - it should take a few seconds

10 Easy - it should take a few minutes

20 It should take a few hours. Suitable for homework exercises.

30 It should take a few days. Suitable for a week-long class project.

40 It should take weeks. Suitable for a semester class project.

50 Open research question.

Acknowledgements Several friends and colleagues have helped in this effort. In alphabetic order: Rakesh Agrawal, Howard Elman, Will Equitz, Myron Flickner, H.V. Jagadish, Philip (Flip) Korn, King-Ip (David) Lin, Yannis Manolopoulos, Wayne Niblack, Douglas Oard, Dragutin Petkovic, M. Ranganathan, Arun Swami, and Kuansan Wang. The research funding of the National Science Foundation (NSF) is also gratefully acknowledged (IRI-8958546, IRI-9205273).

1 INTRODUCTION

As a working definition of a Multimedia Database System we shall consider a system that can store and retrieve multimedia objects, such as 2-dimensional color images, gray-scale medical images in 2-d or 3-d (eg., MRI brain scans), 1-dimensional time series, digitized voice or music, video clips, traditional data types, like 'product-id', 'date', 'title', and any other user-defined data types. For such a system, what this book focuses on is the design of fast searching methods by content. A typical query by content would be, eg., '*in a collection of color photographs, find ones with a same color distribution like a sunset photograph*'.

Specific applications include the following:

- Image databases, where we would like to support queries on color, shape and texture [NBE+93].
- Financial, marketing and production time series, such as stock prices, sales numbers etc. In such databases, typical queries would be '*find companies whose stock prices move similarly*', or '*find other companies that have similar sales patterns with our company*', or '*find cases in the past that resemble last month's sales pattern of our product*'
- Scientific databases, with collections of sensor data. In this case, the objects are time series, or, more general, *vector fields*, that is, tuples of the form, eg., $< x, y, z, t, pressure, temperature, \ldots >$. For example, in weather data [CoPES92], geological, environmental, astrophysics [Vas93] databases, etc., we want to ask queries of the form, e.g., '*find past days in which the solar magnetic wind showed patterns similar to today's pattern*' to help in predictions of the earth's magnetic field [Vas93].

- multimedia databases, with audio (voice, music), video etc. [NC91]. Users might want to retrieve, eg., similar music scores, or video clips.
- Medical databases, where 1-d objects (eg., ECGs), 2-d images (eg., X-rays) [PF94] and 3-d images (eg., MRI brain scans) [ACF+93] are stored. Ability to retrieve quickly past cases with similar symptoms would be valuable for diagnosis, as well as for medical teaching and research purposes.
- text and photograph archives [Nof86], digital libraries [TSW+85] [Har94] with ASCII text, bitmaps, gray-scale and color images.
- office automation [MRT91], electronic encyclopedias [ST84] [GT87], electronic books [YMD85].
- DNA databases [AGM+90] [WZ96] where there is a large collection of long strings (hundred or thousand characters long) from a four-letter alphabet (A,G,C,T); a new string has to be matched against the old strings, to find the best candidates. The distance function is the editing distance (smallest number of insertions, deletions and substitutions that are needed to transform the first string to the second).

It is instructive to classify the queries of interest in increasing complexity. Consider, for illustration, a set of employee records, where each record contains the employee number `emp#`, `name`, `salary`, `job-title`, a resume (ASCII text), a greeting (digitized voice clip) and a photograph (2-d color image). Then, the queries of interest form the following classes.

primary key '*Find the employee record with* `emp#`*= 123*'. That is, the specified attribute has no duplicates.

secondary key '*Find the employee records with* `salary`*=40K and* `job-title` *= engineer*'. That is, the queries involve attributes that may have duplicate values.

text '*Find the employee records containing the words* 'manager', 'marketing' *in their resume*'. A text attribute contains an unspecified number of alphanumeric strings.

signals For example, a query on 1-d signals could be '*Find the employee records whose greeting sounds similar to mine*'. Similarly, for 2-d signals, a query could be '*Find employee photos that look like a desirable photo*'.

Acronym	Definition
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DNA	DeoxyriboNucleic Acid
DWT	Discrete Wavelet Transform
GEMINI	GEneric Multimedia INdexIng method
GIS	Geographic Information Systems
IR	Information Retrieval
LSI	Latent Semantic Indexing
MBR	Minimum Bounding Rectangle
MDS	Multi-Dimensional Scaling
MRI	Magnetic Resonance Imaging
PAM	Point Access Method
SAM	Spatial Access Method
SVD	Singular Value Decomposition
SWFT	Short-Window Fourier Transform
WWW	World-Wide-Web

Table 1.1 Summary of Acronyms and Definitions

The book is organized in two parts and an appendix, following the above classification of queries. In the first part, we present the most successful methods for indexing traditional data (primary, secondary, and text data). Most of these methods, like B-trees and hashing, are textbook methods and have been successfully incorporated in commercial products. In the second part we examine methods for indexing signals. The goal is to adapt the previously mentioned tools and to make them work in this new environments. Finally, in the appendix we present some fundamental techniques from signal processing and matrix algebra, such as the Fourier transform and the Singular Value Decomposition (SVD).

Table 1.1 gives a list of the acronyms that we shall use in the book.

PART I

DATABASE INDEXING METHODS

2 INTRODUCTION TO RELATIONAL DBMS

This chapter presents the minimal necessary set of concepts from relational database management systems. Excellent textbooks include, eg., [KS91] [Dat86]. The chapter also describes how the proposed methods will fit in an extensible DBMS, customized to support multimedia datatypes.

Traditional, relational database management systems ('RDBMS' or just 'DBMS') are extremely popular. They use the *relational model* [Cod70] to describe the data of interest. In the relational model, the information is organized in *tables* ('relations'); the rows of the tables correspond to records, while the columns correspond to attributes. The language to store and retrieve information from such tables is the *Structured Query Language* (SQL).

For example, if we want to create a table with employee records, so that we can store their employee number, name, age and salary, we can use the following SQL statement:

```
create table EMPLOYEE (
    emp# integer,
    name char(50),
    age float,
    salary float);
```

The result of the above statement is to notify the DBMS about the EMPLOYEE table (see Figure 2.1). The DBMS will create a table, which will be empty, but ready to hold EMPLOYEE records.

Tables can be populated with the SQL `insert` command. E.g.

EMPLOYEE	emp#	name	age	salary

Figure 2.1 An empty EMPLOYEE table ('relation').

```
insert into EMPLOYEE values (
   123, "Smith, John", 30, 38000.00);
```

will insert a row in the EMPLOYEE table, recording the information about the employee 'Smith'. Similarly, the command to insert the record for another employee, say, 'Johnson', is:

```
insert into EMPLOYEE values (
   456, "Johnson, Tom", 25, 55000.00);
```

The result is shown in Figure 2.2

EMPLOYEE	emp#	name	age	salary
	123	Smith, John	30	38000.00
	456	Johnson, Tom	25	55000.00

Figure 2.2 The EMPLOYEE table, after two `insertions`

We can retrieve information using the `select` command. E.g., if we want to find all the employees with salary less than 50,000, we issue the following query:

```
select *
from EMPLOYEE
where salary <= 50000.00
```

In the absence of indices, the DBMS will perform a sequential scanning, checking the salary of each and every employee record against the desired threshold of 50,000. To accelerate queries, we can create an index (usually, a B-tree index, as described in Chapters 3 and 4), with the command `create index`. For example, to build an index on the employee's salary, we would issue the SQL statement:

```
create index salIndex on EMPLOYEE (salary);
```

SQL provides a large number of additional, valuable features, such as the ability to retrieve information from several tables ('joins') and the ability to perform aggregate operations (sums, averages). However, we restrict the discussion to the above few features of SQL, which are the absolutely essential ones for this book.

Every commercial DBMS offers the above functionalities, supporting numerical and string datatypes. Additional, user-defined datatypes, like images, voice etc., need an *extensible DBMS*. Such a system offers the facility to provide new data types, along with functions that operate on them. For example, one datatype could be 'voiceClip', which would store audio files in some specified format; another datatype could be 'image', which would store, eg., JPEG color images. The definition of new datatypes and the associated functions for them ('display', 'compare' etc.) are typically implemented by a specialist. After such datatypes have been defined, we could create tables that can hold multimedia employee records, with the command, eg.:

```
create table EMPLOYEE (
    emp# fixed,
    name char(50),
    salary float,
    age float,
    greeting voiceClip,
    face image);
```

Assuming that the predicate `similar` has been appropriately defined for the 'image' datatype, we can look for employees that look like given person, as follows:

```
select name
from EMPLOYEE
where EMPLOYEE.face similar desirableFace
```

where 'desirableFace' is the object-id of the desirable JPEG image.

Providing the ability to answer such queries is exactly the focus of this book. The challenges are two: (a) how to measure 'similarity' and (b) how to search efficiently. In this part of the book we examine older database access methods that can help accelerate the search. In the second part we discuss some similarity measures for multimedia data types, like time sequences and color images.

Before we continue with the discussion of database access methods, we should notice that they are mainly geared towards a two-level storage hierarchy:

- The first level is fast, small, and expensive. Typically, it is the *main memory* or *core* or *RAM*, with an access time of micro-seconds or faster.
- The second level (*secondary store*) is much slower, but much larger and cheaper. Typically, it is a magnetic disk, with ≈5-10 msec access time.

Typically, database research has focused on large databases, which do not fit in main memory and thus have to be store on secondary store. A major characteristic of the secondary store is that it is organized into *blocks* (= *pages*). The reason is that, accessing data from the disk involves the mechanical move of the read/write head of the disk above the appropriate track on the disk. Exactly because these moves ('seeks') are slow and expensive, every time we do a disk-read we bring into main memory a whole disk *block*, of the order of 1Kb-8Kb. Thus, it makes a huge performance difference if we manage to group similar data in the same disk blocks. Successful access methods (like the B-trees) try exactly to achieve good clustering, to minimize the number of disk-reads.

3

PRIMARY KEY ACCESS METHODS

Here we give a brief overview of the traditional methods to handle queries on primary (ie, unique) keys. Considering the running example of EMPLOYEE records, a typical query is, eg., '*find the employee with* `emp#` *= 344*'. Notice that the `emp#` is assumed to have no duplicates.

Primary key access methods are useful for multimedia data for two reasons:

1. primary keys will be part of the information: for example, in an employee database, we may have the `emp#` as the primary key; in a video database, we may have the title or the ISBN as the primary key, etc.
2. The primary key access methods provide fundamental ideas, like the hierarchical organization of records that the B-trees suggest. These ideas were the basis for newer, more general and more powerful access methods, like the R-trees (see Section 5.2), that can be used for multimedia data as well.

For primary keys on secondary store, the textbook approaches are two: B-trees and hashing [Knu73].

3.1 HASHING

The idea behind hashing is the *key-to-address transformation.* For example, consider a set of 40,000 employee records, with unique 9-digit `emp#`. Also assume that we are interested in providing fast responses for queries on `emp#`.

Suppose we have decided to use 1,000 consecutive disk pages (=blocks = buckets), each capable of holding 50 records. Then, we can use a *hashing function* $h()$, to map each key to a bucket. For example:

$$h(emp\#) = (emp\#) \bmod 1000 \tag{3.1}$$

is a function that maps each **emp#** to its last three digits, and therefore, to the corresponding bucket, as shown in Figure 3.1

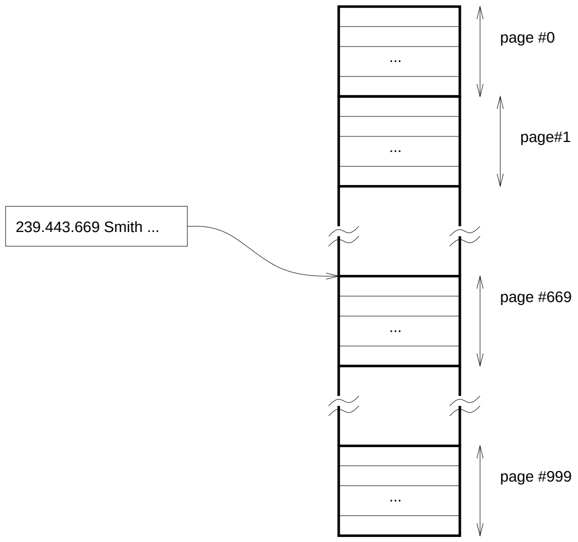


Figure 3.1 Illustration of a hash table, with 'division hashing' ($h(emp\#) = (emp\#) \bmod 1000$).

The first step in the design of a hashing scheme is the choice of the hashing function. There are several classes of them, the most successful ones being (a) the *division hashing*, like the function of Eq. 3.1 and (b) the *multiplication hashing*.

The second step in the design of a hashing scheme is to choose a collision resolution method. Notice that we deliberately allocate more space in the hash table than needed: in our example, 50,000 slots, versus 40,000 records; in general, we opt for 80%-90% load factor. However, due to the almost random nature of the hashing function, there is always the possibility for bucket-overflows. In such a case, we have several choices, the most popular being: (a) using a separate overflow area (*'separate chaining'*) and (b) re-hashing to another bucket (*'open addressing'*).

There are numerous surveys, variations and analyses of hashing [Kno75, SD76, Sta80, Lar85]. An easily accessible hash-table implementation is the `ndbm` package of UNIX$^{\mathrm{TM}}$, which uses hashing with 'open addressing'.

3.1.1 Extensible hashing

The original hashing suffered from the fact that the hash table can not grow or shrink, to adapt to the volume of insertions and deletions. The reason is that the size of the table is 'hardwired' in the hashing function; changing the size implies changing the hashing function, which may force relocation of each and every record, a very expensive operation.

Relatively recent developments tried to alleviate this problem by allowing the hash table to grow and shrink without expensive reorganizations. These methods come under the name of extensible hashing: extendible hashing [FNPS79], dynamic hashing [Lar78], spiral hashing [Mar79], linear hashing [Lit80], linear hashing with partial expansions [Lar82]. See [Lar88] for a recent survey and analysis of such methods.

3.2 B-TREES

B-trees and variants are among the most popular methods for physical file organization on disks [BM72]. Following Knuth [Knu73], we have:

Definition 3.1 *A B-tree of order m is a multiway tree, with the key-ordering property, satisfying the following restrictions:*

1. *Every node has $\leq m$ sons.*
2. *Every node, except for the root, has $\geq m/2$ sons.*
3. *The root has at least 2 sons, unless it is a leaf.*
4. *All leaves appear at the same level*
5. *A non-leaf node with k sons contains k-1 keys.*

The key-ordering property means that, for every sub-tree, the root of the sub-tree is greater than all the key values at the left and smaller than all the key values at the right sub-tree. Notice that the leaves are empty; in practice, the leaves and the pointers pointing to them are omitted, to save space.

To achieve a high fan-out, the tree does not store the full records; instead, it stores pointers to the actual records. More specifically, the format of a B-tree node of order m is as follows:

$$(p_1, key_1, ptr_1, p_2, key_2, ptr_2, \ldots, p_m)$$

where ptr_i is the pointer to the record that corresponds to key_i; p_i is a pointer to another B-tree node (or null). Figure 3.2 shows a B-tree of order m=3 and height 2. Notice that (a) the pointers to the records are not shown (b) the tree fulfills the B-tree properties.

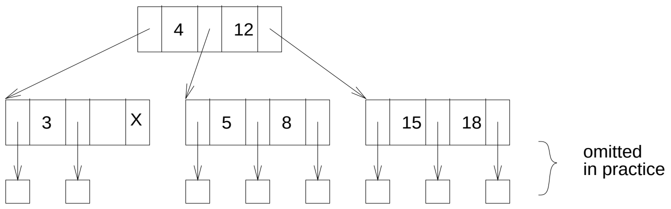


Figure 3.2 Illustration of a B-tree of order m=3. 'X' indicates null pointers.

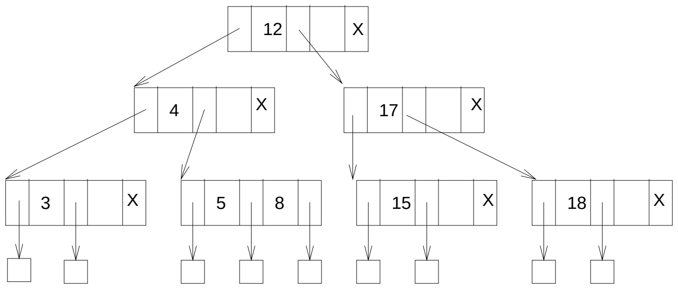


Figure 3.3 Insertion of key #17 in the previous B-tree.

Because of the above definition, a B-tree has the following desirable characteristics:

- it is always balanced, thus leading to logarithmic search time $O(\log_m(N+1))$ and few disk accesses.

- it has guaranteed 50% space utilization, while the average is $\approx$ 69% [Yao78, LW89, JS89]
- it also has logarithmic insertion and deletion times.

The insertion and deletion algorithms are masterfully designed [BM72] to maintain the B-tree properties. A full discussion is outside the scope of this book (see, eg., [Knu73]). A brief sketch of them is instructive, though:

The insertion algorithm works as follows: given a key to be inserted, we find the appropriate leaf node; if there is not enough space there, then we split the node in two, pushing the middle key to the parent node; the parent node may recursively overflow and split again. Figure 3.3 shows the resulting B-tree, after key '17' is inserted into the B-tree of Figure 3.2. Notice the propagated split, which created a new root, thus increasing the height of the B-tree. Thus, the B-tree 'grows from the leaves'. Notice that a node has the lowest possible utilization (50%) immediately after a split, exactly because we split a 100%-full node into 2 new ones.

Deletion is done in the reverse way: omitting several details, if a node underflows, it either borrows keys from one of its siblings, or it merges with a sibling into a new node.

B-trees are very suitable for disks: each node of the B-tree is stored in one page; typically, the fanout is large, and thus the B-tree has few levels, requiring few disk ($\equiv$node) accesses for a search.

This concludes the quick introduction to the basic B-trees. There are two very successful variations, namely the B^+-trees and the B^*-trees:

- The B^+-trees keep a copy of all the keys at the leaves, and string the leaves together with pointers. Thus the scanning of the records in sorted order is accelerated: after we locate the first leaf node, we just follow the pointer to the next leaf.
- The B^*-trees introduce the idea of *deferred splitting*. Splits hurt the performance, because they create new, half-empty nodes, and potentially they can make the tree taller (and therefore slower). The goal is to try to postpone splits: instead of splitting in two an overflowing node, we check to see if there is a sibling node that could host some of the overflowing keys. If the sibling node is also full, *only then* we do a split. The split though

involves *both* of the full nodes, whose entries are divided among *three* new nodes. This clever idea results in much fewer splits and in guaranteed 66% (= 2/3) space utilization of the nodes, with a higher average than that. These splits are called '2-to-3' splits; obviously, we can have '3-to-4' and 's-to-$(s+1)$' splits. However, the programming complexity and the additional disk accesses on insertion time reach a point of diminishing returns. Thus, the '2-to-3' split policy usually provides a good trade-off between search time and insertion effort.

3.3 CONCLUSIONS

B-trees and hashing are the industry work-horses. Each commercial system provides at least one of them. Such an index is built, eg., by the `create index` command of SQL, as discussed in Chapter 2. The two methods compare as follows:

- B-trees guarantee logarithmic performance for any operation (insertion, deletion, search), while hashing gives constant search on the *average* (with linear performance, in the worst case). Depending on the specific version, the insertion and update times for hashing can be constant, or grow linearly with the relation size.
- B-trees can expand and shrink gracefully, as the relation sizes grows or shrinks; hashing requires expensive reorganization unless an extensible hashing method is used (such as the 'linear hashing').
- B-trees preserve the key order, which allows them to answer range queries, nearest neighbor queries, as well as to support ordered sequential scanning. (Eg., consider the query '*print all the employees' paychecks, in increasing* `emp#` *order*').

Exercises

Exercise 3.1 **[15]** *In the B-tree of Figure 3.2, insert the key '7'.*

Exercise 3.2 **[15]** *In the B-tree of Figure 3.3, delete the key '15'.*

Exercise 3.3 [32] *Design and implement the algorithm for insertion and deletion in B-trees.*

Exercise 3.4 [34] *Design and implement the algorithm for insertion and deletion in B^*-trees (i.e., with deferred splitting).*

Exercise 3.5 [25] *Using an existing B-tree package, analyze its average-case space utilization through simulation.*

Exercise 3.6 [25] *Implement a phone-book database, using the* `ndbm` *library of* UNIX$^{\text{TM}}$. *Treat the phone number as the primary key.*

4 SECONDARY KEY ACCESS METHODS

Access methods for secondary key retrieval have attracted much interest. The problem is stated as follows: Given a file, say, `EMPLOYEE( name, salary, age)`, organize the appropriate indices so that we can answer efficiently queries on any and all of the available attributes. Rivest [Riv76] classified the possible queries into the following classes, in increasing order of complexity:

- *exact match* query, when the query specifies all the attribute values of the desired record, e.g.:

  $$\texttt{name} = \text{`Smith'} \ \texttt{and} \ \texttt{salary} = 40{,}000 \ \texttt{and} \ \texttt{age} = 45$$

- *partial match* query, when only some of the attribute values are specified, e.g.:

  $$\texttt{salary} = 40{,}000 \ \texttt{and} \ \texttt{age} = 35$$

- *range queries*, when ranges for some or all of the attributes are specified, e.g.:

  $$35{,}000 \leq \texttt{salary} \leq 45{,}000 \ \texttt{and} \ \texttt{age} = 45$$

- *Boolean queries*:

  $$(\ (\texttt{not name} = \text{`Smith'}) \ \texttt{and} \ \texttt{salary} \geq 40{,}000\ ) \ \texttt{or} \ \texttt{age} \geq 50$$

In the above classification, each class is a special case of the next class. A class of queries outside the above hierarchy is the *nearest neighbor* query:

- *nearest neighbor* or *best match* query, eg.:

$$\texttt{salary} \approx 45{,}000 \texttt{ and age} \approx 55$$

where the user specifies some of the attribute values, and asks for the best match(es), according to some pre-specified distance/dis-similarity function.

In this chapter, first we mention the inverted files, which is the industry work-horse. Then we describe some methods that treat records as points in k-d space (where k is the number of attributes); these methods are known as *point access methods* or PAMs, and are closely related to the upcoming *spatial access methods* (SAMs).

4.1 INVERTED FILES

This is the most popular approach in database systems. An inverted file on a given attribute (say, '`salary`') is built as follows: For each distinct attribute value, we store:

1. A list of pointers to records that have this attribute value (*postings list*).
2. Optionally, the length of this list.

The set of distinct attribute values is typically organized as a B-tree or as a hash table. The postings lists may be stored at the leaves, or in a separate area on the disk. Figure 4.1 shows an index on the `salary` of an EMPLOYEE table. A list of unique salary values is maintained, along with the 'postings' lists.

Given indices on the query attributes, complex boolean queries can be resolved by manipulating the lists of record-pointers, before accessing the actual records.

A interesting variation that can handle conjunctive queries has been proposed by Lum [Lum70], by using *combined indices*: We can build an index on the *concatenation* of two or more attributes, for example (`salary`, `age`). Such an index can answer easily queries of the form '`salary`=*40000* `and age`=*30*', without the need of merging any lists. Such an index will contain all the unique, existing pairs of (`salary`, `age`) values, sorted on lexicographical order. For each pair, it will have a list of pointers to the EMPLOYEE records with the specified combination of `salary` and `age`.

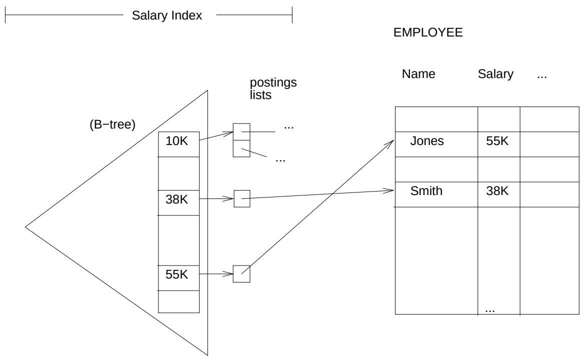


Figure 4.1 Illustration of inversion: a B-tree index on `salary`.

As mentioned in Chapter 2, plain or combined indices can be created automatically by a relational DBMS, with the SQL command `create index`.

4.2 POINT ACCESS METHODS (PAMS)

A fruitful point of view is to envision a record with k attributes as a point in k-dimensional space. Then, there are several methods that can handle points, the so-called *Point Access Methods* (PAMs) [SK90]. Since most of them can also handle spatial objects (rectangles, polygons, etc.) in addition to points, we postpone their description for the next chapter. Here we briefly describe two of the PAMs, the *grid files*, and the *k-d-trees*. They both are mainly designed for points and they have proposed important ideas that several SAMs have subsequently used.

4.2.1 Grid File

The grid file [NHS84] can be envisioned as the generalization of extendible hashing [FNPS79] in multiple dimensions. The idea is that it imposes a grid on the address space; the grid adapts to the data density, by introducing more divisions on areas of high data density. Each grid cell corresponds to one disk page, although two or more cells may share a page. To simplify the 'record-keeping', the cuts are allowed only on predefined points (1/2, 1/4, 3/4 etc. of each axis) and they cut all the way through, to form a grid. Thus, the grid file needs only a list of cut-points for every axis, as well as a directory. The directory has one entry for every grid cell, containing a pointer to the disk page that contains the elements of the grid cell.

The grid file has the following desirable properties: it guarantees 2 disk accesses for exact match queries; it is symmetric with respect to the attributes; and it adapts to non-uniform distributions. However, it suffers from two disadvantages: (a) it does not work well if the attribute values are correlated (eg., '`age`' and '`salary`' might be linearly correlated in an EMPLOYEE file) and (b) it might need a large directory, if the dimensionality of the address space is high ('dimensionality curse'). However, for a database with low-dimensionality points and un-correlated attributes, the grid file is a solution to consider.

Several variations have been proposed, trying to avoid these problems: the rotated grid file [HN83] rotates the address space, trying to de-correlate the attributes; the tricell method [FR89a] uses triangular as opposed to rectangular grid cells; the twin grid file [HSW88] uses a second, auxiliary grid file, to store some points, in an attempt to postpone the directory growth of the main grid file.

4.2.2 K-d-trees

This is the only main-memory access method that we shall describe in this book. The exception is due to the fact that k-d-trees propose elegant ideas that have been used subsequently in several access methods for disk-based data. Moreover, extensions of the original k-d-tree method have been proposed [Ben79] to group and store k-d-tree nodes on disk pages, at least for static data.

The k-d-tree [Ben75] divides the address space in disjoint regions, through 'cuts' on alternating dimensions/attributes. Structurally, it is a binary tree, with every node containing (a) a data record (b) a left pointer and (c) a right pointer. At every level of the tree, a different attribute is used as the 'discriminator', typically in a round-robin fashion.

Let n be a node, r be the record in this node, and A be the discriminator for this node. Then, the left subtree of the node n will contain records with smaller A values, while the right subtree will contain records with greater or equal A values. Figure 4.2(a) illustrates the partitioning of the address space by a k-d-tree: the file has 2 attributes (eg., '`age`' and '`salary`'), and it contains the following records (in insertion order): (30,50), (60,10), (45, 20). Figure 4.2(b) shows the equivalent k-d-tree as a binary tree.

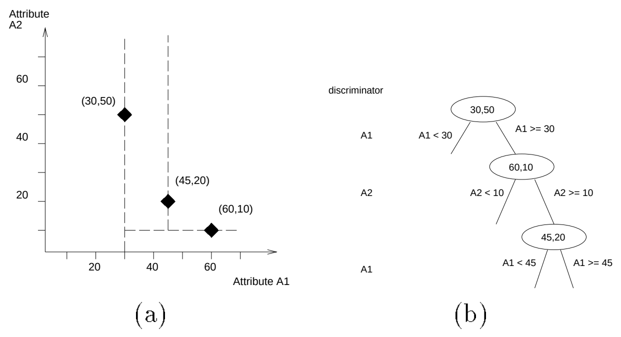


Figure 4.2 Illustration of a k-d tree with three records: (a) the divisions in the address space and (b) the tree itself.

The k-d tree can easily handle exact-match queries, range queries and nearest-neighbor queries [Ben75]. The algorithms are elegant and intuitive, and they typically achieve good response times, thanks to the efficient 'pruning' of the search space that the k-d-tree leads to.

Several disk-based PAMs have been inspired by or used k-d-trees. The k-d-B-trees [Rob81] divide the address space in m regions for every node (as opposed to just 2 that the k-d-tree does), where m is fanout of the tree. The hB-tree [LS90] divides the address space in regions that may have 'holes'; moreover, the contents of every node/disk-page are organized into a k-d-tree.

4.3 CONCLUSIONS

With respect to secondary-key methods, inversion with a B-tree (or hashed) index is automatically provided by commercial DBMS with the `create index` SQL command. The rest of the point access methods are typically used in stand-alone systems. Their extensions, the *spatial access methods*, are examined next.

Exercises

Exercise 4.1 [33] *Implement a grid-file package, for n=2 dimensions with insertion and range search routines.*

Exercise 4.2 [33] *Modify the previous package, so that the number of dimensions n is user-defined.*

Exercise 4.3 [30] *For each of the above packages, implement a 'nearest neighbor' search algorithm.*

Exercise 4.4 [20] *Extend your nearest neighbor algorithms to search for k nearest neighbors, where k is user-defined.*

Exercise 4.5 [30] *Populate each of the above packages with N=10,000-100,000 points; issue 100 nearest-neighbor queries, and plot the response time of each method, as well as the time for the sequential scanning.*

Exercise 4.6 [30] *Using a large database (real or synthetic, such as the Wisconsin benchmark), and any available RDBMS, ask selections queries before and after building an index on the query attributes; time the results.*

5 SPATIAL ACCESS METHODS (SAMS)

In the previous section we examined the so-called 'secondary key' access methods, which handle queries on keys that may have duplicates (eg., '`salary`', or '`age`', in an **`EMPLOYEE`** file). As mentioned, records with k numerical attributes can be envisioned as k-dimensional points. Here we examine *spatial access methods*, which are designed to handle multidimensional points, lines, rectangles and other geometric bodies.

There are numerous applications that require efficient retrieval of spatial objects:

- Traditional relational databases, where, as we mentioned, records with k-attributes become points in k-d spaces (see Figure 5.1(a)).
- Geographic Information Systems (GIS), which contain, eg., point data, such as cities on a two-dimensional map (see Figure 5.1(b)).
- Medical image databases with, for example, three-dimensional MRI brain scans, require the storage and retrieval of point-sets, such as digitized surfaces of brain structures [ACF+93].
- Multimedia databases, where multi-dimensional objects can be represented as points in feature space [Jag91, FRM94]. For example, 2-d color images correspond to points in (R,G,B) space (where R,G,B are the average amount of red, green and blue [FBF+94]). See Figure 5.1(c).
- Time-sequences analysis and forecasting [WG94, CE92], where k successive values are treated as a point in k-d space; correlations and regularities in this k-d space help in characterizing the dynamical process that generates the time series.

- Rule indexing in expert database systems [SSH86] where rules can be represented as ranges in address space (eg., '*all the employees with salary in the range (10K-20K) and age in the rage (30-50) are entitled to specific health benefits*'). See Figure 5.1(d).

In a collection of spatial objects, there are additional query types that are of interest. The following query types seem to be the most frequent:

1. *range* queries, a slight generalization of the range queries we saw in secondary key retrieval. Eg., '*find all cities within 10 miles from Washington DC*'; or '*find all rivers in Canada*'. Thus the user specifies a region (a circle around Washington, or the region covered by Canada) and asks for all the objects that intersect this region. The *point query* is a special case of the range query, when the query region collapses to a point. Typically, the range query request all the spatial objects that intersect a region; similarly, it could request the spatial objects that *are completely contained*, or that *completely contain* the query region. We mainly focus on the 'intersection' variation; the rest two can usually be easily answered, by slightly modifying the algorithm for the 'intersection' version.
2. *nearest neighbor* queries, again a slight generalization of the nearest neighbor queries for secondary keys. Eg., '*find the 5 closest post-offices to our office building*'. The user specifies a point or a region, and the system has to return the k closest objects. The distance is typically the Euclidean distance (L_2 norm), or some other distance function (eg., city-block distance L_1, or the L_∞ norm etc).
3. *spatial joins*, or *overlays*: eg., in a CAD design, '*find the pairs of elements that are closer than ϵ*' (and thus create electromagnetic interference to each other). Or, given a collection of lakes and a collection of cities, '*find all the cities that are within 10km from a lake*'.

The proposed methods in the literature form the following classes. For a recent, extensive survey, see [GG95].

- Methods that use *space filling curves* (also known as *z-ordering* or *linear quadtrees*).
- Methods that use tree-like structures: R-trees and its variants.

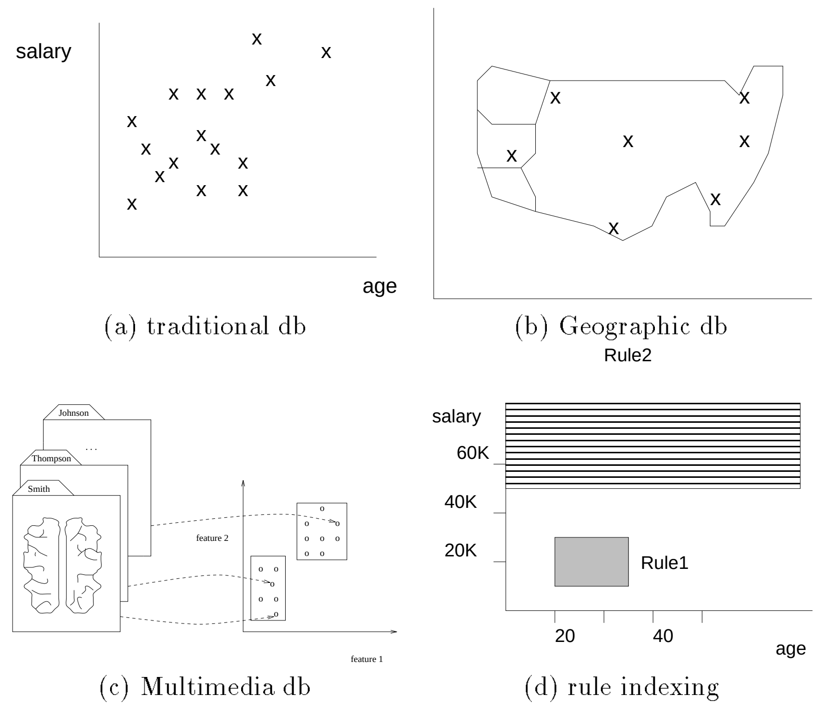


Figure 5.1 Applications of Spatial Access Methods

The next two sections are dedicated to each of the above classes. For each class we discuss the main idea, its most successful variations, and sketch the algorithms to handle the above query types. In the third section we present the idea that transforms spatial objects into higher-dimensionality points. In the last section we give the conclusions for this chapter.

5.1 SPACE FILLING CURVES

The method has attracted a lot of interest, under the names of N-trees [Whi81], linear quadtrees [Gar82], z-ordering [Ore86] [OM88] [Ore89] [Ore90] etc. The fundamental assumption is that there is a finite precision in the representation of each co-ordinate, say K bits. The terminology is easiest described in 2-d address space; the generalizations to n dimensions should be obvious. Following the quadtree literature, the address space is a square, called an <u>*image*</u> , and it is represented as a $2^K \times 2^K$ array of 1×1 squares. Each such square is called a *pixel* .

Figure 5.2 gives an example for n=2 dimensional address space, with K=2 bits of precision. Next, we describe how the method handles points and regions.

5.1.1 Handling points

The space filling curve tries to impose a linear ordering on the resulting pixels of the address space, so that to translate the problem into a primary-key access problem.

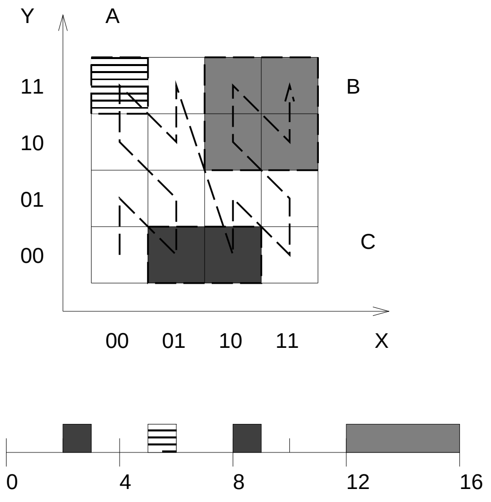


Figure 5.2 Illustration of Z-ordering

One such obvious mapping is to visit the pixels in a row-wise order. A better idea is to use *bit interleaving* [OM84]. Then, the *z-value* of a pixel is the value of the resulting bit string, considered as a binary number. For example, consider the pixel labeled 'A' in Figure 5.2, with coordinates x_A= `00` and y_A= `11`. Suppose that we decide to shuffle the bits, starting from the x-coordinate first, that is, the order with which we pick bits from the coordinates is '1,2,1,2' ('1' corresponds to the x coordinate and '2' to the y coordinate). Then, the z-value z_A of pixel 'A' is computed as follows:

$$z_A = \text{Shuffle} (\text{‘1,2,1,2’}, x_A, y_A) = \text{Shuffle} (\text{‘1,2,1,2’}, \texttt{00}, \texttt{11}) = \texttt{0101} = (5)_{10}$$

Visiting the pixels in ascending z-value order creates a self-similar trail as depicted in Figure 5.2 with a dashed line; the trail consists of 'N' shapes, organized to form larger 'N' shapes recursively. Rotating the trail by 90 degrees gives 'z' shapes, which is probably the reason that the method was named *z-ordering.* Figure 5.3 shows the trails of the z-ordering for a 2×2, a 4×4 and an 8×8 grid. Notice that each larger grid contains four miniature replicas of the smaller grids, joined in an 'N' shape.

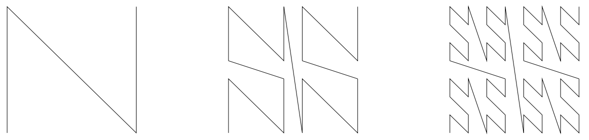


Figure 5.3 Z-order curves for 2x2, 4x4 and 8x8 grids.

We have just described one method to compute the z-value of a point in 2-d address space. The extension to n-d address spaces is obvious: we just shuffle the bits from each of the n dimensions, visiting the dimensions in a round-robin fashion. The inverse is also obvious: given a z-value, we translate it to a binary number and un-shuffle its bits, to derive the n coordinate values.

5.1.2 Handling Regions

The z-value of a region is more complicated. In fact, a region typically breaks into one or more pieces, each of which can be described by a z-value. For example, the region labeled 'C' in Figure 5.2 breaks into two pixels C_1 and C_2, with z-values

$$z_{C_1} = \texttt{0010} = (2)_{10}$$
$$z_{C_2} = \texttt{1000} = (8)_{10}$$

The region labeled 'B' consists of four pixels, which have the common prefix `11` in their z-values; in this case, the z-value of 'B' is exactly this common prefix:

$$z_B = \texttt{11}$$

A conceptually easier and computationally more efficient way to derive the z-values of a region is through the concept of 'quadtree blocks'. Consider the four equal squares that the image can be decomposed into. Each such square is called a *level-1 block* ; a *level-i block* can be recursively defined as one of the four equal squares that constitute a level-$(i-1)$ block. Thus, the pixels are level-K blocks; the image is the (only) level-0 block. Notice that for a level-i block, all its pixels have the same prefix up to $2i$ bits; this common prefix is defined as the z-value of the given block.

We obtain the quadtree decomposition of an object (region) by recursively dividing it into blocks, until the blocks are homogeneous or until we reach the pixel level (level-K blocks). For a 2-dimensional object, the decomposition can be represented as a 4-way tree, as shown in Figure 5.4(b). Blocks that are empty/full/partially-full are represented as white, black and gray nodes in the quadtree, respectively.

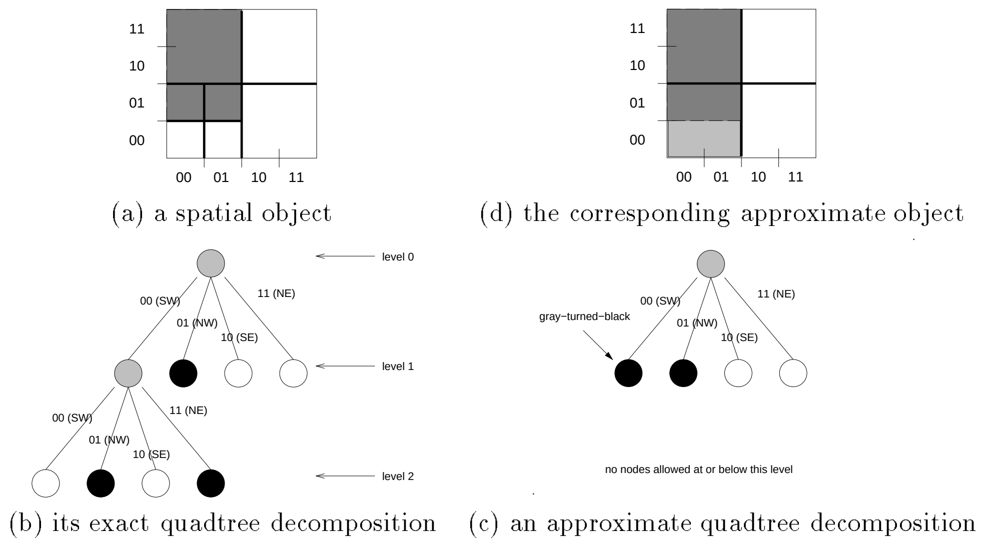
For efficiency reasons (eg., see [Ore89, Ore90]), we typically approximate an object with a 'coarser resolution' object. In this case, we stop the decomposition earlier, eg., when we reach level-i blocks ($i < K$), or when we have a large enough number of pieces. Figure 5.4(c) and (d) give an example.

The quadtree representation gives an easy way to obtain the z-value for a quadtree block: Let '0' stand for 'south' and for 'west', and '1' stand for 'north' and for 'east'. Then, each edge of the quadtree has a unique, 2-bit label, as shown in Figure 5.4; the z-value of a block is defined as the concatenation of the labels of the edges from the root of the quadtree to the node that corresponds to the given block.

Since every block has a unique z-value, we can represent the quadtree decomposition of an object by listing the corresponding z-values. Thus, the z-values of the shaded rectangle in figure 5.4(a) are '0001' (for 'WS WN') '0011' (for 'WS EN') and '01' (for 'WN').

As described above, quadtrees have been used to store objects in main memory. For disk storage, the prevailing approach is the so-called *linear* quadtree [Gar82], or, equivalently the z-ordering method [Ore86]. Each object (and range query) can be uniquely represented by the z-values of its blocks. Each such z-value can be treated as a primary-key of a record of the form (z-value, object-id, *other attributes* ...), and it can be inserted in a primary-key file structure, such as a B^+-tree. Table 5.1 illustrates such a relation, containing the z-values of the shaded rectangle of Figure 5.4(a).

Additional objects in the same address space can be handled in the same way; their z-values will be inserted into the same B^+-tree.



(a) a spatial object (d) the corresponding approximate object

(b) its exact quadtree decomposition (c) an approximate quadtree decomposition

Figure 5.4 Counter-clockwise, from top-left: (a) The shaded rectangle is decomposed into three blocks. (b) the corresponding quadtree, with z-values `01`, `0001` and `0011` (c) an approximate quadtree, with z-values `01`, `00` (d) the corresponding approximate spatial object - the lightly-shaded region is the enlargement, due to the approximation.

z-value	object id	(other attributes)
...	...	...
0001	'ShadedRectangle'	...
...	...	...
0011	'ShadedRectangle'	...
...	...	...
01	'ShadedRectangle'	...
...	...	...

Table 5.1 Illustration of the relational table that will store the z-values of the sample shaded rectangle.

5.1.3 Algorithms

The z-ordering method can handle all the queries that we have listed earlier.

Range Queries: The query shape is translated into a set of z-values, as if it were a data region. Typically, we opt for an approximate representation of it, trying to balance the number of z-values and the amount of extra area in the approximation [Ore90]. Then, we search the B^+-tree with the z-values of the data regions, for matching z-values. Orenstein and Manola [OM88] describe in detail the conditions for matching.

Nearest neighbor queries: The sketch of the basic algorithm is as follows: Given a query point P, we compute its z-value and search the B^+-tree for the closest z-value; we compute the actual distance r, and then issue a range query centered at P with radius r.

Spatial joins: The algorithm for spatial joins is a generalization of the algorithm for the range query. Let S be a set of spatial objects (eg., lakes) and R be another set (eg., railways line segments). The spatial join '*find all the railways that cross lakes*' is handled as follows: the elements of set S are translated into z-values, sorted; the elements of set R are also translated into a sorted list of z-values; the two lists of z-values are merged. The details are in [Ore86] [OM88].

5.1.4 Variations - improvements

We have seen that if we traverse the pixels on ascending z-value order, we obtain a trail as shown in Figure 5.2. This trail imposes a mapping from n-d space onto a 1-d space; ideally, we would like a mapping with distance preserving properties, that is, pixels that are near in address space should have nearby z-values. The reason is that good clustering will make sure that 'similar' pixels will end up in the same or nearby leaf pages of the B^+-tree, thus greatly accelerating the retrieval on range queries.

The z-ordering indeed imposes a good such mapping: It does not leave a quadrant, unless it has visited all its pixels. However, it has some long, diagonal jumps, which maybe could be avoided. This observation prompted the search for better space filling curves. Alternatives included a curve using Gray codes [Fal88]; the best performing one is the Hilbert curve [FR89b], which has been shown to achieve better clustering than the z-ordering and the gray-codes curve, and it is the only one that we shall describe.

Figure 5.5 shows the Hilbert curves of order 1, 2 and 3: The order k curve is derived from the original, order 1 curve, by substituting each of its four points with an order (k-1) curve, appropriately rotated or reflected. In the limit, the resulting curve has fractal dimension=2 [Man77], which intuitively means that it is so inter-twined and dense that it 'behaves' like a 2-d object. Notice also that the trail of a Hilbert curve does *not* have any abrupt jumps, like the z-ordering does. Thus, intuitively it is expected to have better distance-preserving properties than the z-ordering. Experiments in [FR89b] showed that the claim holds for the reported settings.

Algorithms to compute the Hilbert value of an n-d pixel have been published [Bia69, But71]; source code in the 'C' programming language is available in [Jag90a] for n=2 dimensions. The complexity of all these algorithms, as well as their inverses, is $O(b)$ where b is the total number of bits of the z/Hilbert value. The proportionality constant is small (a few operations per bit for the z-value, a few more for the Hilbert value). For both curves, the time to compute a z/Hilbert value is negligible compared to the disk access time.

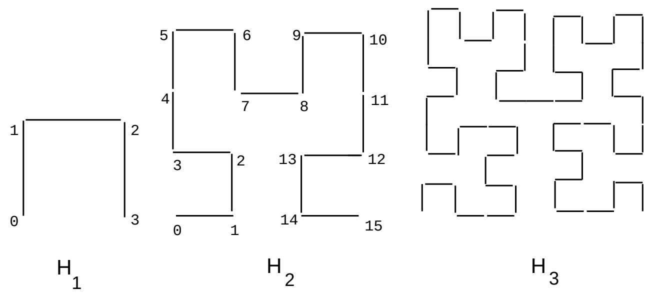


Figure 5.5 *Hilbert Curves of order 1,2 and 3*

There are several analytical and simulation studies of space filling curves: in [FR89b] we used exhaustive enumeration to study the clustering properties of several curves, showing that the Hilbert curve is best; Jagadish [Jag90a] provides analysis for partial match and 2×2 range queries; in [RF91] we derive closed formulas for the z-ordering; Moon et al. [MJFS96] derive closed formulas for range queries on the Hilbert curve.

Also related is the analytical study for quadtrees, trying to determine the number of quadtree blocks that a spatial object will be decomposed into [HS79], [Dye82, Sha88], [Fal92a], [FJM94], [Gae95], [FG96]. The common observation is that the number of quadtree blocks and the number of z/Hilbert values that a spatial object requires is *proportional to the measure of its boundary* (eg., perimeter for 2-d objects, surface area for 3-d etc.). As intuitively expected, the constant of proportionality is smaller for the Hilbert curve, compared to the z-ordering.

5.2 R-TREES

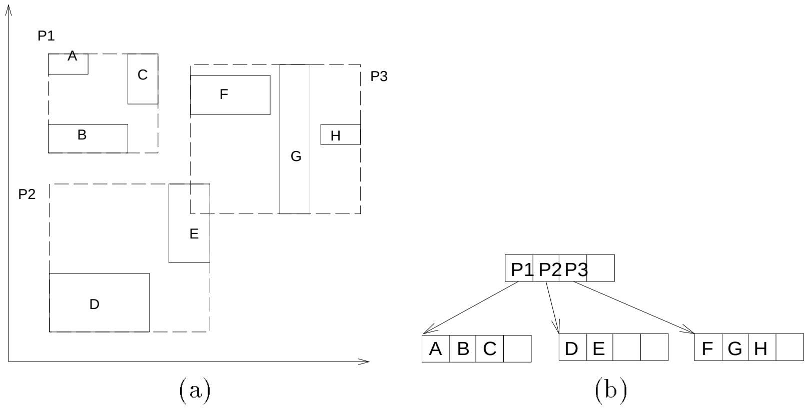


Figure 5.6 (a) Data (solid-line rectangles) organized in an R-tree with fanout=4 (b) the resulting R-tree, on disk.

The R-tree was proposed by Guttman [Gut84]. It can be thought of as an extension of the B-tree for multidimensional objects. A spatial object is represented by its minimum bounding rectangle (MBR). Non-leaf nodes contain entries of the form (ptr, R), where ptr is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form ($obj-id$, R) where $obj-id$ is a pointer to the object description, and R is the MBR of the object. The main innovation in the R-tree is that parent nodes are allowed to overlap. This way, the R-tree can guarantee good space utilization and remain balanced. Figure 5.6(a) illustrates data rectangles (solid boundaries), organized in an R-tree with fanout 3, while Figure 5.6(b) shows the file structure for the same R-tree, where nodes correspond to disk pages.

The R-tree inspired much subsequent work, whose main focus was to improve the search time. A packing technique is proposed in [RL85] to minimize the overlap between different nodes in the R-tree for static data. The idea was to order the data in, say, ascending x-low value, and scan the list, filling each leaf node to capacity. Figure 5.7(a) illustrates 200 points in 2-d, and Figure 5.7(b) shows the resulting R-tree parents according to the x-low sorting. An improved packing technique based on the Hilbert Curve is proposed in [KF93]: the idea is to sort the data rectangles on the Hilbert value of their centers. Figure 5.7(c) shows the resulting R-tree parents; notice that their MBRs are closer to a square shape, which was shown to give better search performance than the elongated shapes of Figure 5.7(b).

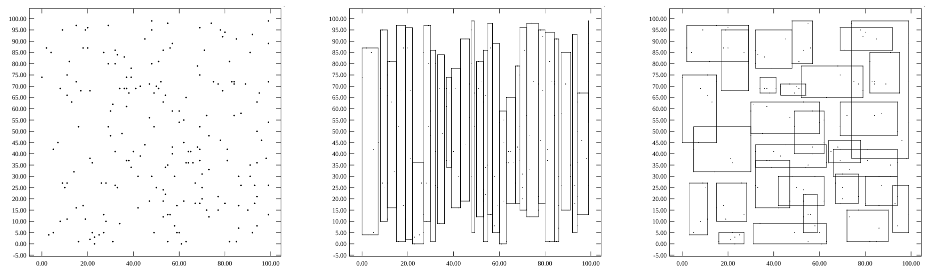


Figure 5.7 Packing algorithms for R-trees: (a) 200 points (b) their parent MBRs, when sorting on x-low (c) the parent MBRs when sorting on the Hilbert value.

A class of variations consider more general minimum bounding shapes, trying to minimize the 'dead space' that an MBR may cover. Gunther proposed the cell trees [Gun86], which introduce diagonal cuts in arbitrary orientation. Jagadish proposed the polygon trees (P-trees) [Jag90b], where the minimum bounding shapes are polygons with slopes of sides 0, 45, 90, 135 degrees. Minimum bounding shapes that are concave or even have holes have been suggested, eg., in the hB-tree [LS90].

One of the most successful ideas in R-tree research is the idea of deferred splitting: Beckmann et. al. proposed the R^*-tree [BKSS90], which was reported to outperform Guttman's R-trees by $\approx$30%. The main idea is the concept of *forced re-insert*, which tries to defer the splits, to achieve better utilization: When a node overflows, some of its children are carefully chosen and they are deleted and re-inserted, usually resulting in a better structured R-tree. The idea of deferred splitting was also exploited in the Hilbert R-tree [KF94]; there, the Hilbert curve is used to impose a linear ordering on rectangles, thus defining who the sibling of a given rectangle is, and subsequently applying the 2-to-3 (or s-to-$(s+1)$) splitting policy of the B^*-trees (see section 3.2). Both methods achieve higher space utilization than Guttman's R-trees, as well as better response time (since the tree is shorter and more compact).

Finally, analysis of the R-tree performance has attracted a lot of interest: in [FSR87] we provide formulas, assuming that the spatial objects are uniformly distributed in the address space. The uniformity assumption was relaxed in [FK94], where we showed that the *fractal dimension* is an excellent measure of the non-uniformity, and we provided accurate formulas to estimate the average number of disk accesses of the resulting R-tree. The fractal dimension also helps estimate the selectivity of spatial joins, as shown in [BF95]. When the sizes of the MBRs of the R-tree are known, formulas for the expected number of disk access are given in [PSTW93] and [KF93].

5.2.1 Algorithms

Since the R-tree is one of the most successful spatial access methods, we describe the related algorithms in some more detail:

Insertion: When a new rectangle is inserted, we traverse the tree to find the most suitable leaf node; we extend its MBR if necessary, and store the new rectangle there. If the leaf node overflows, we split it, as discussed next:

Split: This is one of the most crucial operations for the performance of the R-tree. Guttman suggested several heuristics to divide the contents of an overflowing node into two sets, and store each set in a different node. Deferred splitting, as mentioned in the R^*-tree and in the Hilbert R-tree, will improve performance. Of course, as in B-trees, a split may propagate upwards.

Range Queries: The tree is traversed, comparing the query MBR with the MBRs in the current node; thus, non-promising (and potentially large) branches of the tree can be 'pruned' early.

Nearest Neighbors: The algorithm follows a 'branch-and-bound' technique similar to [FN75] for nearest-neighbor searching in clustered files. Roussopoulos et al. [RKV95] give the detailed algorithm for R-trees.

Spatial Joins: Given two R-trees, the obvious algorithm builds a list of pairs of MBRs, that intersect; then, it examines each pair in more detail, until we hit the leaf level. Significantly faster methods than the above straightforward method have been suggested in [BKS93] [BKSS94]. Lo and Ravishankar [LR94] proposed an efficient method to perform a spatial join when only one of the two spatial datasets has an R-tree index on it.

5.2.2 Conclusions

R-trees is one of the most promising SAMs. Among its variations, the R^*-trees and the Hilbert R-trees seem to achieve the best response time and space utilization, in exchange for more elaborate splitting algorithms.

5.3 TRANSFORMATION TO HIGHER-D POINTS

The idea is to transform 2-d rectangles into 4-d points [HN83], by using the low- and high-values for each axis; then, any point access method (PAM) can be used. In general, an n-dimensional rectangle will become a $2n$-dimensional point. The original and final space are called 'native' and 'parameter' space, respectively [Ore90]. Figure 5.8 illustrates the idea for 1-d address space: line segments A(0:0.25) and B(0.75:1) are mapped into 2-d points. A range query Q(0.25:0.75) in native space becomes a range query in parameter space, as illustrated by the shaded region in Figure 5.8.

The strong point of this idea is that we can turn any Point Access Method (PAM) into a Spatial Access Method (SAM) with very little effort. This approach has been used or suggested in several settings, eg., with grid files [HN83], B-trees [FR91], hB-trees [LS90] as the underlying PAM.

The weak points are the following: (a) the parameter space has high dimensionality, inviting 'dimensionality curse' problems earlier on (see the discussion on page 39). (b) except for range queries, there are no published algorithms for nearest-neighbor and spatial join queries. Nevertheless, it is a clever idea, which can be valuable for a stand-alone, special purpose system, operating on a low-dimensionality address space. Such an application could be, eg., a temporal database system [SS88], where the spatial objects are 1-dimensional time segments [KTF95].

5.4 CONCLUSIONS

From a practical point of view, the most promising methods seem to be:

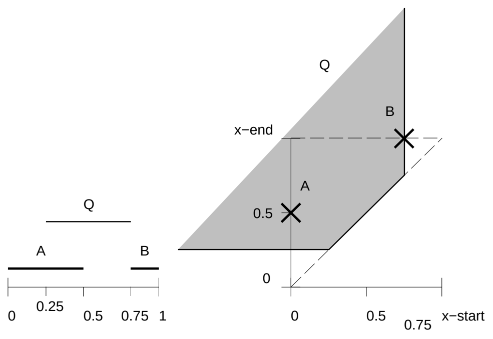


Figure 5.8 Transformation of 1-d rectangles into points in higher dimensionality.

- Z-ordering: Z-ordering and, equivalently, linear quadtrees have been very popular for 2-dimensional spaces. One of the major application is in geographic information systems: linear quadtrees have been used both in production systems, like the TIGER system at the U.S. Bureau of Census [Whi81] (`http://tiger.census.gov/tiger/tiger.html`), which stores the map and statistical data of the U.S.A., as well as research prototypes such as QUILT [SSN87], PROBE [OM88], and GODOT [GR94]. For higher dimensions, oct-trees have been used in 3-d graphics and robotics [BB82]; in databases of 3-d medical images [ACF+94], etc. Z-ordering performs very well for a low dimensionality and for points. It is particularly attractive because it can be implemented on top of a B-tree with relatively little effort. However, for objects with non-zero area (= hyper-volume), the practitioner should be aware of the fact that each such object may require a large number of z-values for its exact representation; the recommended approach is to approximate each object with a small number of z-values [Ore89, Ore90].

- R-trees and variants: they operate on the native space (requiring no transforms to high-dimensionality spaces), and they can handle rectangles and other shapes without the need to divide them into pieces. Cutting data into pieces results in an artificially increased database size (linear on the number of *pieces*); moreover, it requires a duplicate-elimination step, because a query may retrieve the same object-id several times (once for each piece of the qualifying object) R-trees have been tried successfully for 20-30 dimensional address spaces [FBF+94, PF94]. Thanks to the above properties, R-trees have been incorporated in academic as well as commercial systems, like POSTGRES (`http: //s2k-ftp.cs. berkeley.edu: 8000 /postgres95/`) and ILLUSTRA (`http: //www.illustra.com/`).

Before we close this chapter, we should mention about the 'dimensionality curse'. Unfortunately, all the SAMs will suffer for high dimensionalities n: For the z-ordering, the range queries of radius r will require effort proportional to the hyper-surface of the query region $O(r^{(n-1)})$ as mentioned on page 33. Similarly, for the R-trees as the dimensionality n grows, each MBR will require more space; thus, the fanout of each R-tree page will decrease. This will make the R-tree taller and slower. However, as mentioned, R-trees have been successfully used for 20-30 dimensions [FBF+94] [PF94]. To the best of this author's knowledge, performance results for the z-ordering method are available for low dimensionalities only (typically, n=2). A comparison of R-trees versus z-ordering for high dimensionalities is an interesting research question.

Exercises

Exercise 5.1 [07] *What is the z-value of the pixel (*`11, 00`*) in Figure 5.2? What is its Hilbert value?*

Exercises

Exercise 5.2 [20] *Design an algorithm for the spatial join in R-trees*

Exercise 5.3 [30] *Design an algorithm for the k nearest neighbors in R-trees*

Exercise 5.4 [30] *Repeat the two previous exercises for the Z-ordering*

Exercise 5.5 [38] *Implement the code for the Hilbert curve, for 2 dimensions; for n dimensions.*

6

ACCESS METHODS FOR TEXT

6.1 INTRODUCTION

In this chapter we present the main ideas for text retrieval methods. For more details, see the survey in [Fal85] and the book by Frakes and Baeza-Yates [FBY92]. Access methods for text are interesting for three reasons: (a) multimedia objects often have captions in free text; exploiting these captions may help retrieve some additional relevant objects [OS95] (b) research in text retrieval has led to some extremely useful ideas, like the *relevance feedback* and the *vector space* model that we discuss next and (c) text retrieval has several applications in itself.

Such applications include the following:

- Library automation [SM83] [Pri84] and distributed digital libraries [GGMT94], where large amounts of text data are stored on the world-wide-web (WWW). In this setting, search engines are extremely useful and popular, like 'veronica' [ODL93], 'lycos' (`http: //lycos. cs.cmu.edu/`), 'inktomi' (`http: //inktomi. berkeley. edu/`), etc..
- Automated law and patent offices [Hol79] [HH83]; electronic office filing [CTH$^+$86]; electronic encyclopedias [EMS$^+$86] and dictionaries [GT87].
- Information filtering (eg., the *RightPages* project [SOF$^+$92] and the *Latent Semantic Indexing* project [DDF$^+$90] [FD92a]); also, the 'selective dissemination of information' (SDI) [YGM94].

In text retrieval, the queries can be classified as follows [Fal85]:

- Boolean queries, eg. '*(data* `or` *information)* `and` *retrieval* `and` *(*`not` *text)*'. Here, the user specifies terms, connected with Boolean operators. Some additional operators are also supported, like `adjacent`, or `within` <*n*> `words` or `within sentence`, with the obvious meanings. For example the query '*data* `within sentence` *retrieval*' will retrieve documents which contain a sentence with both the words 'data' and 'retrieval'.
- Keyword search: here, the user specifies a set of keywords, like, eg., '*data, retrieval, information*'; the retrieval system should return the documents that contain as many of the above keywords as possible. This interface offers less control to the user, but it is more user-friendly, because it does not require familiarity with Boolean logic.

Several systems typically allow for prefix matches, eg., '*organ**' will match all the words that start with 'organ', like 'organs', 'organization', 'organism' etc. We shall use the star '*' as the variable-length don't care character.

The rest of this chapter is organized as follows: in the next three sections we discuss the main three methods for text retrieval, namely (a) full text scanning, (b) inversion and (c) signature files. In the last section we discuss the clustering approach.

6.2 FULL TEXT SCANNING

According to this method, no preprocessing of the document collection is required. When a query arrives, the whole collection is inspected, until the matching documents are found.

When the user specifies a pattern that is a *regular expression*, the textbook approach is to use a finite state automaton (FSA) [HU79, pp. 29-35]. If the search pattern is a single string with no *don't care* characters, faster methods exist, like the Knuth, Morris and Pratt algorithm [KMP77], and the fastest of all, the Boyer and Moore algorithm [BM77] and its recent variations [Sun90, HS91].

For multiple query strings, the algorithm by Aho and Corasick [AC75] builds a finite state automaton in time linear on the total length of the strings, and reports all the matches in a single pass over the document collection.

Searching algorithms that can tolerate typing errors have been developed by Wu and Manber [WM92] and Baeza-Yates and Gonnet [BYG92]. The idea is to scan the database one character at a time, keeping track of the currently matched characters. The algorithm can retrieve all the strings within a desired *editing distance* from the query string. The editing distance of two strings is the minimum number of insertions, deletions and substitutions that are needed to transform the first string into the second [HD80, LW75, SK83]. The method is flexible and fast, requiring a few seconds for a few Megabytes of text on a SUN-class workstation. Moreover, its source code is available (`ftp://cs.arizona.edu/agrep`).

In general, the advantage of every full text scanning method is that it requires no space overhead and minimal effort on insertions and updates, since no indices have to be changed. The price is that the response time is slow for large data bases. Therefore, full text scanning is typically used for small databases (a few Mbytes in size), or in conjunction with another access method (e.g., inversion) that would restrict the scope of searching.

6.3 INVERSION

In inversion, each document can be represented by a list of (key)words, which describe the contents of the document for retrieval purposes. Fast retrieval can be achieved if we invert on those keywords: The keywords are stored, eg., alphabetically, in the 'index file'; for each keyword we maintain a list of pointers to the qualifying documents in the 'postings file'. Figure 6.1 illustrates the file structure, which is very similar to the inverted index for secondary keys (see Figure 4.1).

Typically, the index file is organized using sophisticated primary-key access methods, such as B-trees, hashing, TRIEs [Fre60] or variations and combinations of these (e.g., see [Knu73, pp. 471-542], or Chapter 3). For example, in an early version of the UNIX^{TM} utility `refer`, Lesk used an over-loaded hash table with separate chaining, in order to achieve fast searching in a database of bibliographic entries [Les78]; the Oxford English Dictionary uses an extension of the PATRICIA trees [Mor68], called PAT trees [GBYS92].

The advantages are that inversion is relatively easy to implement, it is fast, and it supports synonyms easily (e.g., the synonyms can be organized as a threaded list within the dictionary). For the above reasons, the inversion method has

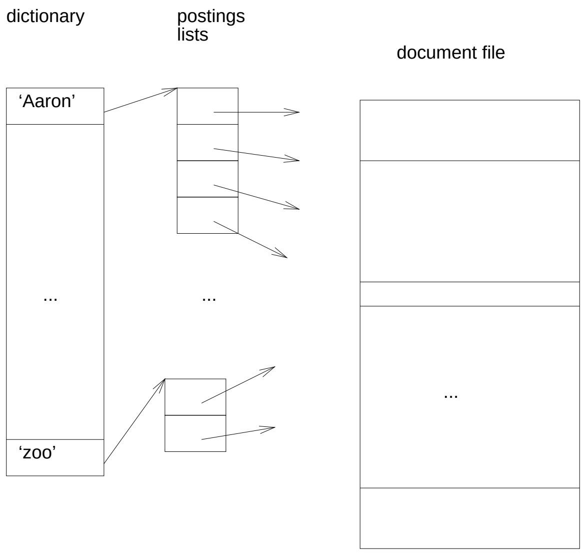


Figure 6.1 Illustration of inversion

been adopted in most of the commercial systems such as DIALOG, BRS, MEDLARS, ORBIT, STAIRS [SM83, ch. 2].

The disadvantages of this method are the storage overhead (which can reach up to 300% of the original file size [Has81] if too much information is kept about each word occurrence), and the cost of updating and reorganizing the index, if the environment is dynamic.

Recent work exactly focuses on these problems: Techniques to achieve fast insertions incrementally include the work by Tomasic et al., [TGMS94]; Cutting and Pedersen [CP90] and Brown et. al. [BCC94]. These efforts typically exploit the skewness of the distribution of postings lists, treating the short lists differently than the long ones.

It is important to elaborate on the skewness, because it has serious implications for the design of fast text retrieval methods. The distribution typically follows *Zipf's law* [Zip49], which states that the occurrence frequency of a word is inversely proportional to its rank (after we sort the vocabulary words in decreasing frequency order). More specifically, we have [Sch91]:

$$f(r) = \frac{1}{r \ln(1.78V)} \tag{6.1}$$

where r is the rank of the vocabulary word, $f(r)$ is the percentage of times it appears, and V is the vocabulary size. This means that a few vocabulary words will appear very often, while the majority of vocabulary words will appear once or twice. Figure 6.2 plots the frequency versus the rank of the words in the Bible, in logarithmic scales. The Figure also plots the predictions, according to Eq. 6.1. Notice that the first few most common words appear tens of thousands of times, while the vast majority of the vocabulary words appear less than 10 times. Zipf reported that similar skeweness is observed in several other languages, in addition to English.

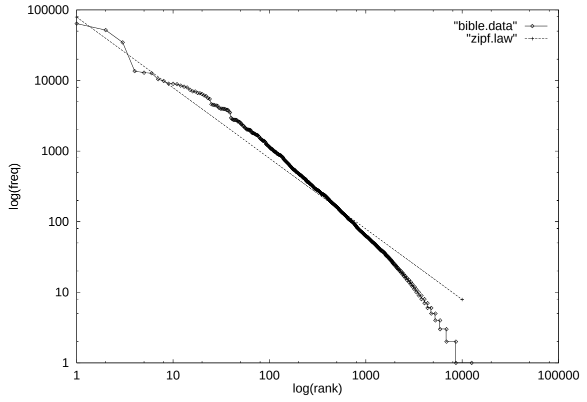


Figure 6.2 Rank-Frequency plot for the words in the Bible - both scales are logarithmic. The line corresponds to Zipf's law.

Given the large size of indices, compression methods have also been investigated: Zobel et al. [ZMSD92] use Elias's [Eli75] compression scheme for postings lists, reporting small space overheads for the index. Finally, the `glimpse` package [MW94] uses a coarse index plus the `agrep` package [WM92] for approximate matching. Like the `agrep` package, `glimpse` is also available from the University of Arizona (`ftp: //cs.arizona.edu/ glimpse`).

6.4 SIGNATURE FILES

The idea behind signature files is to create a 'quick and dirty' filter, which will be able to quickly eliminate most of the non-qualifying documents. As we shall see next and in Chapter 7, this idea has been used several times in very different contexts, often with excellent results. A recent survey on signature files is in [Fal92b].

The method works as illustrated in Figure 6.3: For every document, a short, typically hash-coded version of it is created (its *document signature*); document signatures are typically stored sequentially, and are searched upon a query. The signature test returns *all* the qualifying documents, plus (hopefully, few) false matches, or '*false alarms*' or '*false drops*'. The documents whose signatures qualify are further examined, to eliminate the false drops.

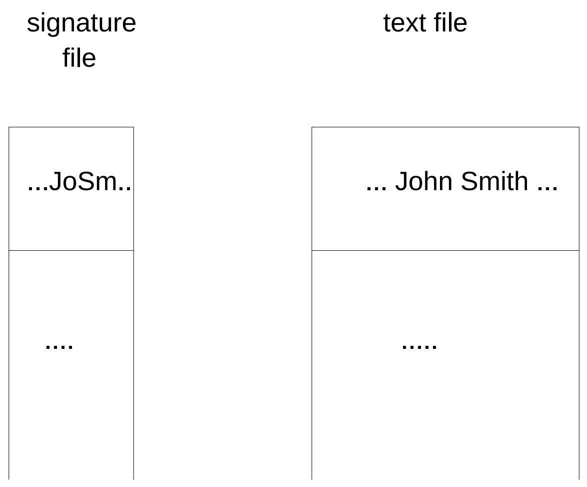


Figure 6.3 Example of signature files. For illustration, the signature of a word is decided to be its first two letters.

Figure 6.3 shows a naive (and not recommended) method of creating signatures, namely, by keeping the first 2 letters of every word in the document. One of the best methods to create signatures is *superimposed coding* [Moo49]. Following the notation in [CF84], each word yields a bit pattern (*word signature*) of size F, with m bits set to '`1`' and the rest left as '`0`'. These bit patterns are OR-ed to form the document signature. Table 6.3 gives an example for a toy document, with two words: '*data*' and '*base*'.

Word	Signature
data	001 000 110 010
base	000 010 101 001
doc. signature	001 010 111 011

Table 6.1 Illustration of superimposed coding: F=12 bits for a signature, with m=4 bits per word set to `1`.

On searching for a word, say '*data*', we create its word signature, and exclude all the document signatures that do not have a '`1`' at the corresponding bit positions. The choice of the signature length F depends on the desirable false-drop rate; the m parameter is chosen such that, on the average, half of the bits should be '`1`' in a document signature [Sti60].

The method has been studied and used extensively in text retrieval: on bibliographic entries [FH69]; for fast substring matching [Har71] [KR87]; in office automation and message filing [TC83] [FC87]. It has been used in academic [SDR83, SDKR87] as well as commercial systems [SK86, Kim88].

Moreover, signature files with some variation of superimposed coding have been used in numerous other applications: For indexing of formatted records [Rob79, CS89]; for indexing of images [RS92]; for set-membership testing in the so-called *Bloom filters*, which have been used for spelling checking in UNIX^{TM} [McI82], in differential files [SL76], and in semi-joins for distributed query optimization [ML86]. A variation of superimposed coding has even been used in chess-playing programs, to alert for potentially dangerous combinations of conditions [ZC77].

In concluding this discussion on the signature file approach, its advantages are the simplicity of its implementation, and the efficiency in handling insertions. In addition, the method is trivially parallelizable [SK86]. The disadvantage is that it may be slow for large databases, unless the sequential scanning of the signature file is somehow accelerated [SDR83, LL89, ZRT91].

6.5 VECTOR SPACE MODEL AND CLUSTERING

The Vector Space Model is very popular in information retrieval [Sal71b] [SM83] [VR79], and it is well suited for 'keyword queries'. The motivation behind the approach is the so-called cluster hypothesis: closely associated documents tend to be relevant to the same requests. Grouping similar documents accelerates the searching.

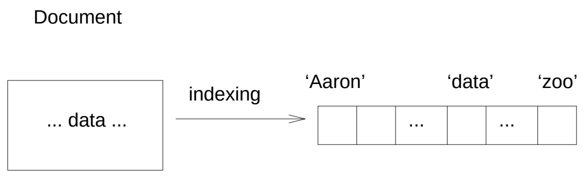


Figure 6.4 Illustration of the 'indexing' process in IR

An important contribution of the vector space model is to envision each document as a V-dimensional vector, where V is the number of terms in the document collection. The procedure of mapping a document into a vector is called *indexing* (overloading the word!); 'indexing' can be done either manually (by trained experts), or automatically, using a stop list of common words, some stemming algorithm, and possibly a thesaurus of terms. The final result of the 'indexing' process is that each document is represented by a V-dimensional vector, where V is the number of permissible index terms. Absence of a term is indicated by a 0 (or by -1 [Coo70]); presence of a term is indicated by 1 (for binary document vectors) or by a positive number (term weight), which reflects the importance of the term for the document.

The next step in the vector space model is to decide how to group similar vectors together ('*cluster generation*'); the last step is to decide how to search a cluster hierarchy for a given query ('*cluster search*'). For both the above problems, we have to decide on a *document-to-document similarity function* and on a *document-to-cluster similarity function*. For the document-to-document similarity function, several choices are available, with very similar performance ([SM83, pp. 202-203] [VR79, p. 38]). Thus, we present only the *cosine similarity function* [Sal71b] which seems to be the most popular:

$$\cos(\vec{x}, \vec{y}) = \vec{x} \circ \vec{y} / (\| \vec{x} \| \, \| \vec{y} \|) \tag{6.2}$$

where $\vec{x}$ and $\vec{y}$ are two V-dimensional document vectors, '$\circ$' stands for the inner product of two vectors and $\| . \|$ for the Euclidean norm of its argument.

There are also several choices for the document-to-cluster distance/similarity function. The most popular seems to be the method that treats the centroid of the cluster as a single document, and then applies the document-to-document similarity/distance function. Other choices include the 'single link' method, which estimates the minimum distance (= dis-similarity) of the document from all the members of the cluster, and the 'all link' method which computes the maximum of the above distances.

An interesting and effective recent development is the 'Latent Semantic Indexing' (LSI), which applies the Singular Value Decomposition (SVD) on the document-term matrix and it automatically groups co-occurring terms. These groups can be used as a thesaurus, to expand future queries. Experiments [FD92b] showed up to 30% improvement over the traditional vector model. More details on the method are in Appendix D.3, along with the description of the SVD.

In the next subsections we briefly describe the main ideas behind (a) the cluster generation algorithms, (b) the cluster search algorithms and (c) the evaluation methods of the clustering schemes.

6.5.1 Cluster generation

Several cluster generation methods have been proposed; recent surveys can be found in [Ras92] [Mur83] [VR79]. Following Van-Rijsbergen [VR79], we distinguish two classes:

- 'sound' methods, that are based on the document-to-document similarity matrix and
- 'iterative' methods, that are more efficient and proceed directly from the document vectors.

Sound Methods: If N is the number of documents, these methods usually require $O(N^2)$ time (or more) and apply graph theoretic techniques. A simplified version of such a clustering method would work as follows ([DH73b, p. 238]): An appropriate threshold is chosen and two documents with a similarity measure that exceeds the threshold are assumed to be connected with an edge. The connected components (or the maximal cliques) of the resulting graph are the proposed clusters.

The problem is the selection of the appropriate threshold: different values for the threshold give different results. The method proposed by Zahn [Zah71] is an attempt to circumvent this problem. He suggests finding a minimum spanning tree for the given set of points (documents) and then deleting the 'inconsistent' edges. An edge is inconsistent if its length l is much larger than the average length l_{avg} of its incident edges. The connected components of the resulting graph are the suggested clusters. Figure 6.5 gives an illustration of the method. Notice that the long edges with solid lines are *not* inconsistent, because, although long, they are not significantly longer than their adjacent edges.

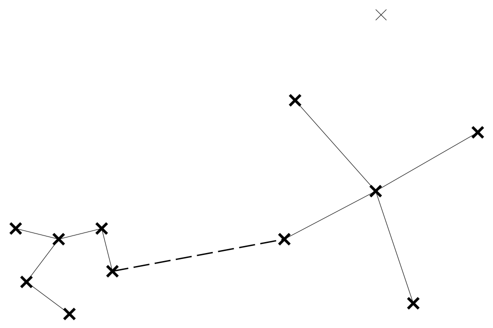


Figure 6.5 Illustration of Zahn's method: the dashed edge of the MST is 'inconsistent' and therefore deleted; the connected components are the clusters.

Iterative methods: This class consists of methods that are faster: $O(N \log N)$ or $O(N^2/\log N)$) on the average. They are based directly on the object (document) descriptions and they do not require the similarity matrix to be computed in advance. The typical iterative method works as follows:

- Choose some seeds (eg., from sound clustering on a sample)
- Assign each vector to the closest seed (possibly adjusting the cluster centroid)
- Possibly, re-assign some vectors, to improve clusters

Several iterative methods have been proposed along these lines, The simplest and fastest one seems to be the 'single pass' method [SW78]: Each document is processed once and is either assigned to one (or more, if overlap is allowed) of the existing clusters, or it creates a new cluster.

In conclusion, as mentioned before, the iterative ones are fast and practical, but they are sensitive to the insertion order.

6.5.2 Cluster searching

Searching in a clustered file is much simpler than cluster generation. The input query is represented as a V-dimensional vector and it is compared with the cluster-centroids. The searching proceeds in the most similar clusters, i.e., those whose similarity with the query vector exceeds a threshold. As mentioned, a typical cluster-to-query similarity function is the cosine function - see Eq. 6.2

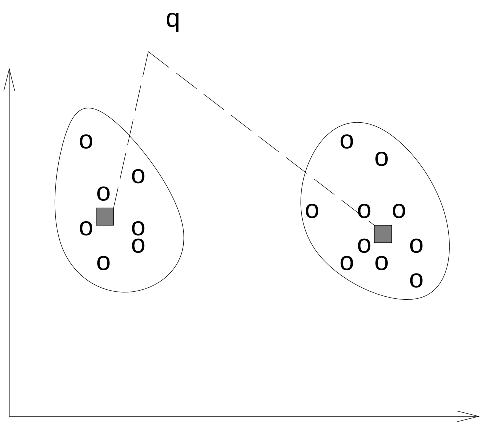


Figure 6.6 Searching in clustered files: For the query vector q, searching continues in the closest cluster at the left.

The vector representation of queries and documents has led to two important ideas:

- *ranked output* and
- *relevance feedback*

The first idea, ranked output, comes naturally, because we can always compute the distance/similarity of the retrieved documents to the query, and we can sort them ('most similar first') and present only the first screenful to the user.

The second idea, the relevance feedback, is even more important, because it can easily increase the effectiveness of the search [Roc71]: The user pinpoints the relevant documents among the retrieved ones and the system re-formulates the query vector and starts the searching from the beginning. To carry out the query re-formulation, we operate on the query vector and add (vector addition) the vectors of the relevant documents and subtract the vectors of the non-relevant ones. Experiments indicate that the above method gives excellent results after only two or three iterations [Sal71a].

6.5.3 Evaluation of clustering methods

The standard way to evaluate the 'goodness' of a clustering method is to use the so-called *precision* and *recall* concepts. Thus, given a collection of documents, a set of queries and a human expert's responses to the above queries, the ideal system is the one that will retrieve exactly what the human dictated, and nothing more. The deviations from the above ideal situation are measured by the precision and recall: consider the set of documents that the computerized system returned; then the precision is defined as the percentage of relevant documents among the retrieved ones:

$$\text{precision} \equiv \frac{\text{retrieved and relevant}}{\text{retrieved}}$$

and recall is the percentage of relevant documents that we retrieved, over the total number of relevant documents in the document collection:

$$\text{recall} \equiv \frac{\text{retrieved and relevant}}{\text{relevant}}$$

Thus, high precision means that we have few false alarms; high recall means that we have few false dismissals.

The popular precision-recall plot gives the scatter-plot of the precision-recall values for several queries. Figure 6.7 shows such a plot, for fictitious data. When comparing the precision-recall plots of competing methods, the one closer to the (1.0,1.0) point is the winner.

The annual *Text REtrieval Conference* (TREC) provides a test-bed for an open competition of text retrieval methods. See `http: //potomac. ncsl. nist.gov/ TREC/` for more details.

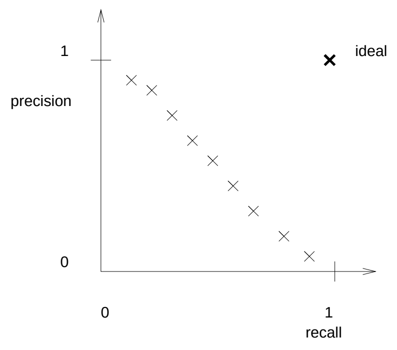


Figure 6.7 A fictitious, typical recall-precision diagram

6.6 CONCLUSIONS

We have discussed several methods for text searching and information retrieval (IR). Among them, the conclusions for a practitioner are as follows:

- Full text scanning is suitable for small databases (up to a few Megabytes); '`agrep`' is a recent, excellent free-ware search package. Its scaled-up version, '`glimpse`', allows fast searching for up to a few Gigabytes of size.
- Inversion is the industry work-horse, for larger databases.
- Signatures can provide a 'quick-and-dirty' test, when false alarms are tolerable or when they can be quickly discarded.
- The major ideas from the vector space model are two: (a) the relevance feedback and (b) the ability to provide ranked output (i.e., documents sorted on relevance order)

Exercises

Exercise 6.1 [20] *Produce the (rank, frequency) plot for a text file. (Hint: use the commands* `sort -u`*,* `tr` *and* `awk` *from* UNIX^{TM} *).*

Exercise 6.2 [30] *For queries with a single string and no 'don't care' characters, implement the straightforward method for full text scanning; also, type-in* *the Boyer-Moore code from Sunday [Sun90]; time them on some large files; compare them to* `grep` *and* `agrep`*.*

Exercise 6.3 [25] *Implement an algorithm that will compute the editing distance of two strings, that is, the minimum number of insertions, deletions or substitutions that are needed to transform one string to the other. (Hint: see the discussion by Hall and Dowling [HD80]).*

Exercise 6.4 [40] *Develop a B-tree package and use it to build an index for text files.*

Exercise 6.5 [40] *Develop and implement algorithms to do insertion and search in a cluster hierarchy, as described in [SW78].*