CHAPTER 22

Advanced Querying and Information Retrieval

Businesses have begun to exploit the burgeoning data online to make better decisions about their activities, such as what items to stock and how best to target customers to increase sales. Many of their queries are rather complicated, however, and certain types of information cannot be extracted even by using SQL.

Several techniques and tools are available to help with decision support. Several tools for data analysis allow analysts to view data in different ways. Other analysis tools precompute summaries of very large amounts of data, in order to give fast responses to queries. The SQL:1999 standard now contains additional constructs to support data analysis. Another approach to getting knowledge from data is to use *data mining*, which aims at detecting various types of patterns in large volumes of data. Data mining supplements various types of statistical techniques with similar goals.

Textual data, too, has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as *information retrieval*. Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage. However, the emphasis in the field of information systems is different from that in database systems, concentrating on issues such as querying based on keywords; the relevance of documents to the query; and the analysis, classification, and indexing of documents.

This chapter covers decision support, including online analytical processing and data mining and information retrieval.

22.1 Decision-Support Systems

Database applications can be broadly classified into transaction processing and decision support, as we have seen earlier in Section 21.3.2. Transaction-processing systems are widely used today, and companies have accumulated a vast amount of information generated by these systems.

For example, company databases often contain enormous quantities of information about customers and transactions. The size of the information storage required may range up to hundreds of gigabytes, or even terabytes, for large retail chains. Transaction information for a retailer may include the name or identifier (such as credit-card number) of the customer, the items purchased, the price paid, and the dates on which the purchases were made. Information about the items purchased may include the name of the item, the manufacturer, the model number, the color, and the size. Customer information may include credit history, annual income, residence, age, and even educational background.

Such large databases can be treasure troves of information for making business decisions, such as what items to stock and what discounts to offer. For instance, a retail company may notice a sudden spurt in purchases of flannel shirts in the Pacific Northwest, may realize that there is a trend, and may start stocking a larger number of such shirts in shops in that area. As another example, a car company may find, on querying its database, that most of its small sports cars are bought by young women whose annual incomes are above $50,000. The company may then target its marketing to attract more such women to buy its small sports cars, and may avoid wasting money trying to attract other categories of people to buy those cars. In both cases, the company has identified patterns in customer behavior, and has used the patterns to make business decisions.

The storage and retrieval of data for decision support raises several issues:

- Although many decision support queries can be written in SQL, others either cannot be expressed in SQL or cannot be expressed easily in SQL. Several SQL extensions have therefore been proposed to make data analysis easier. The area of *online analytical processing* (OLAP) deals with tools and techniques for data analysis that can give nearly instantaneous answers to queries requesting summarized data, even though the database may be extremely large. In Section 22.2, we study SQL extensions for data analysis, and techniques for online analytical processing.
- Database query languages are not suited to the performance of detailed **statistical analyses** of data. There are several packages, such as SAS and S++, that help in statistical analysis. Such packages have been interfaced with databases, to allow large volumes of data to be stored in the database and retrieved efficiently for analysis. The field of statistical analysis is a large discipline on its own; see the references in the bibliographical notes for more information.
- Knowledge-discovery techniques attempt to discover automatically statistical rules and patterns from data. The field of *data mining* combines knowledge discovery techniques invented by artificial intelligence researchers and statistical analysts, with efficient implementation techniques that enable them to be used on extremely large databases. Section 22.3 discusses data mining.
- Large companies have diverse sources of data that they need to use for making business decisions. The sources may store the data under different schemas. For performance reasons (as well as for reasons of organization control), the

data sources usually will not permit other parts of the company to retrieve data on demand.

To execute queries efficiently on such diverse data, companies have built *data warehouses*. Data warehouses gather data from multiple sources under a unified schema, at a single site. Thus, they provide the user a single uniform interface to data. We study issues in building and maintaining a data warehouse in Section 22.4.

The area of **decision support** can be broadly viewed as covering all the above areas, although some people use the term in a narrower sense that excludes statistical analysis and data mining.

22.2 Data Analysis and OLAP

Although complex statistical analysis is best left to statistics packages, databases should support simple, commonly used, forms of data analysis. Since the data stored in databases are usually large in volume, they need to be summarized in some fashion if we are to derive information that humans can use.

OLAP tools support interactive analysis of summary information. Several SQL extensions have been developed to support OLAP tools. There are many commonly used tasks that cannot be done using the basic SQL aggregation and grouping facilities. Examples include finding percentiles, or cumulative distributions, or aggregates over sliding windows on sequentially ordered data. A number of extensions of SQL have been recently proposed to support such tasks, and implemented in products such as Oracle and IBM DB2.

22.2.1 Online Analytical Processing

Statistical analysis often requires grouping on multiple attributes. Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their item-name, color, and size, and that we have a relation *sales* with the schema *sales*(*item-name, color, size, number*). Suppose that *item-name* can take on the values (skirt, dress, shirt, pant), *color* can take on the values (dark, pastel, white), and *size* can take on values (small, medium, large).

Given a relation used for data analysis, we can identify some of its attributes as **measure** attributes, since they measure some value, and can be aggregated upon. For instance, the attribute *number* of the *sales* relation is a measure attribute, since it measures the number of units sold. Some (or all) of the other attributes of the relation are identified as **dimension attributes**, since they define the dimensions on which measure attributes, and summaries of measure attributes, are viewed. In the *sales* relation, *item-name, color*, and *size* are dimension attributes. (A more realistic version of the *sales* relation would have additional dimensions, such as time and sales location, and additional measures such as monetary value of the sale.)

Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.

size: **all**

item-name \ *color*	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

Figure 22.1 Cross tabulation of *sales* by *item-name* and *color*.

To analyze the multidimensional data, a manager may want to see data laid out as shown in the table in Figure 22.1. The table shows total numbers for different combinations of *item-name* and *color*. The value of *size* is specified to be **all**, indicating that the displayed values are a summary across all values of *size*.

The table in Figure 22.1 is an example of a **cross-tabulation** (or **cross-tab**, for short), also referred to as a **pivot-table**. In general, a cross-tab is a table where values for one attribute (say A) form the row headers, values for another attribute (say B) form the column headers, and the values in an individual cell are derived as follows. Each cell can be identified by (a_i, b_j), where a_i is a value for A and b_j a value for B. If there is at most one tuple with any (a_i, b_j) value, the value in the cell is derived from that single tuple (if any); for instance, it could be the value of one or more other attributes of the tuple. If there can be multiple tuples with an (a_i, b_j) value, the value in the cell must be derived by aggregation on the tuples with that value. In our example, the aggregation used is the sum of the values for attribute *number*. In our example, the cross-tab also has an extra column and an extra row storing the totals of the cells in the row/column. Most cross-tabs have such summary rows and columns.

A cross-tab is different from relational tables usually stored in databases, since the number of columns in the cross-tab depends on the actual data. A change in the data values may result in adding more columns, which is not desirable for data storage. However, a cross-tab view is desirable for display to users. It is straightforward to represent a cross-tab without summary values in a relational form with a fixed number of columns. A cross-tab with summary rows/columns can be represented by introducing a special value **all** to represent subtotals, as in Figure 22.2. The SQL:1999 standard actually uses the **null** value in place of **all**, but to avoid confusion with regular null values, we shall continue to use **all**.

Consider the tuples (skirt, **all**, 53) and (dress, **all**, 35). We have obtained these tuples by eliminating individual tuples with different values for *color*, and by replacing the value of *number* by an aggregate—namely, sum. The value **all** can be thought of as representing the set of all values for an attribute. Tuples with the value **all** only for the *color* dimension can be obtained by an SQL query performing a group by on the column *item-name*. Similarly, a group by on *color* can be used to get the tuples with the value **all** for *item-name*, and a group by with no attributes (which can simply be omitted in SQL) can be used to get the tuple with value **all** for *item-name* and *color*.

item-name	*color*	*number*
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	**all**	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	**all**	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	**all**	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	**all**	27
all	dark	62
all	pastel	54
all	white	48
all	**all**	164

Figure 22.2 Relational representation of the data in Figure 22.1.

The generalization of a cross-tab, which is 2-dimensional, to n dimensions can be visualized as an n-dimensional cube, called the **data cube**. Figure 22.3 shows a data cube on the *sales* relation. The data cube has three dimensions, namely *item-name*, *color*, and *size*, and the measure attribute is *number*. Each cell is identified by values for these three dimensions. Each cell in the data cube contains a value, just as in a cross-tab. In Figure 22.3, the value contained in a cell is shown on one of the faces of the cell; other faces of the cell are shown blank if they are visible.

The value for a dimension may be **all**, in which case the cell contains a summary over all values of that dimension, as in the case of cross-tabs. The number of different ways in which the tuples can be grouped for aggregation can be large. In fact, for a table with n dimensions, aggregation can be performed with grouping on each of the 2^n subsets of the n dimensions.[1]

An online analytical processing or OLAP system is an interactive system that permits an analyst to view different summaries of multidimensional data. The word *online* indicates that the an analyst must be able to request new summaries and get responses online, within a few seconds, and should not be forced to wait for a long time to see the result of a query.

With an OLAP system, a data analyst can look at different cross-tabs on the same data by interactively selecting the attributes in the cross-tab. Each cross-tab is a

1. Grouping on the set of all n dimensions is useful only if the table may have duplicates.

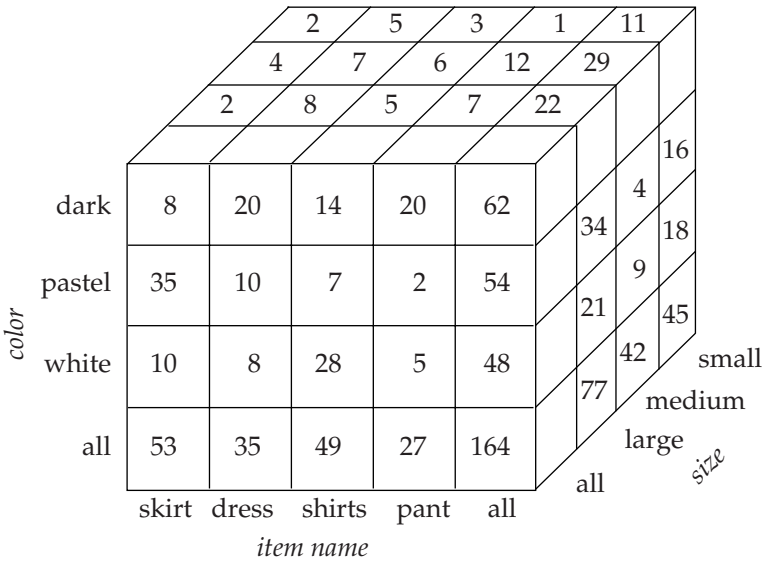


Figure 22.3 Three-dimensional data cube.

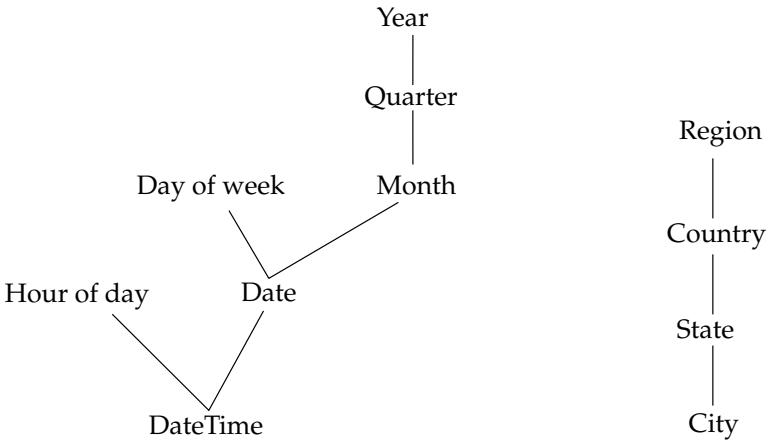
two-dimensional view on a multidimensional data cube. For instance the analyst may select a cross-tab on *item-name* and *size*, or a cross-tab on *color* and *size*. The operation of changing the dimensions used in a cross-tab is called **pivoting**.

An OLAP system provides other functionality as well. For instance, the analyst may wish to see a cross-tab on *item-name* and *color* for a fixed value of *size*, for example, large, instead of the sum across all sizes. Such an operation is referred to as **slicing**, since it can be thought of as viewing a slice of the data cube. The operation is sometimes called **dicing**, particularly when values for multiple dimensions are fixed.

When a cross-tab is used to view a multidimensional cube, the values of dimension attributes that are not part of the cross-tab are shown above the cross-tab. The value of such an attribute can be **all**, as shown in Figure 22.1, indicating that data in the cross-tab are a summary over all values for the attribute. Slicing/dicing simply consists of selecting specific values for these attributes, which are then displayed on top of the cross-tab.

OLAP systems permit users to view data at any desired level of granularity. The operation of moving from finer-granularity data to a coarser granularity (by means of aggregation) is called a **rollup**. In our example, starting from the data cube on the *sales* table, we got our example cross-tab by rolling up on the attribute *size*. The opposite operation—that of moving from coarser-granularity data to finer-granularity data—is called a **drill down**. Clearly, finer-granularity data cannot be generated from coarse-granularity data; they must be generated either from the original data, or from even finer-granularity summary data.

Analysts may wish to view a dimension at different levels of detail. For instance, an attribute of type **datetime** contains a date and a time of day. Using time precise to a second (or less) may not be meaningful: An analyst who is interested in rough time



a) Time Hierarchy

b) Location Hierarchy

Figure 22.4 Hierarchies on dimensions.

of day may look at only the hour value. An analyst who is interested in sales by day of the week may map the date to a day-of-the-week and look only at that. Another analyst may be interested in aggregates over a month, or a quarter, or for an entire year.

The different levels of detail for an attribute can be organized into a **hierarchy**. Figure 22.4(a) shows a hierarchy on the **datetime** attribute. As another example, Figure 22.4(b) shows a hierarchy on location, with the city being at the bottom of the hierarchy, state above it, country at the next level, and region being the top level. In our earlier example, clothes can be grouped by category (for instance, menswear or womenswear); *category* would then lie above *item-name* in our hierarchy on clothes. At the level of actual values, skirts and dresses would fall under the womenswear category and pants and shirts under the menswear category.

An analyst may be interested in viewing sales of clothes divided as menswear and womenswear, and not interested in individual values. After viewing the aggregates at the level of womenswear and menswear, an analyst may *drill down the hierarchy* to look at individual values. An analyst looking at the detailed level may *drill up the hierarchy*, and look at coarser-level aggregates. Both levels can be displayed on the same cross-tab, as in Figure 22.5.

22.2.2 OLAP Implementation

The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems. Later, OLAP facilities were integrated into relational systems, with data stored in a relational database. Such systems are referred to as **relational OLAP (ROLAP)** systems. Hybrid

category	*item-name*	dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	skirt	14	14	28	49	
	dress	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Figure 22.5 Cross tabulation of *sales* with hierarchy on *item-name*.

systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.

Many OLAP systems are implemented as client–server systems. The server contains the relational database as well as any MOLAP data cubes. Client systems obtain views of the data by communicating with the server.

A naïve way of computing the entire data cube (all groupings) on a relation is to use any standard algorithm for computing aggregate operations, one grouping at a time. The naïve algorithm would require a large number of scans of the relation. A simple optimization is to compute an aggregation on, say, (*item-name*, *color*) from an aggregation (*item-name*, *color*, *size*), instead of from the original relation. For the standard SQL aggregate functions, we can compute an aggregate with grouping on a set of attributes A from an aggregate with grouping on a set of attributes B if $A \subseteq B$; you can do so as an exercise (see Exercise 22.1), but note that to compute **avg**, we additionally need the **count** value. (For some non-standard aggregate functions, such as median, aggregates cannot be computed as above; the optimization described here do not apply to such "*non-decomposable*" aggregate functions.) The amount of data read drops significantly by computing an aggregate from another aggregate, instead of from the original relation. Further improvements are possible; for instance, multiple groupings can be computed on a single scan of the data. See the bibliographical notes for references to algorithms for efficiently computing data cubes.

Early OLAP implementations precomputed and stored entire data cubes, that is, groupings on all subsets of the dimension attributes. Precomputation allows OLAP queries to be answered within a few seconds, even on datasets that may contain millions of tuples adding up to gigabytes of data. However, there are 2^n groupings with n dimension attributes; hierarchies on attributes increase the number further. As a result, the entire data cube is often larger than the original relation that formed the data cube and in many cases it is not feasible to store the entire data cube.

Instead of precomputing and storing all possible groupings, it makes sense to precompute and store some of the groupings, and to compute others on demand. Instead of computing queries from the original relation, which may take a very long time, we can compute them from other precomputed queries. For instance, suppose a query requires summaries by (*item-name*, *color*), which has not been precomputed. The query result can be computed from summaries by (*item-name*, *color*, *size*), if that

has been precomputed. See the bibliographical notes for references on how to select a good set of groupings for precomputation, given limits on the storage available for precomputed results.

The data in a data cube cannot be generated by a single SQL query using the basic **group by** constructs, since aggregates are computed for several different groupings of the dimension attributes. Section 22.2.3 discusses SQL extensions to support OLAP functionality.

22.2.3 Extended Aggregation

The SQL-92 aggregation functionality is limited, so several extensions were implemented by different databases. The SQL:1999 standard, however, defines a rich set of aggregate functions, which we outline in this section and in the next two sections. The Oracle and IBM DB2 databases support most of these features, and other databases will no doubt support these features in the near future.

The new aggregate functions on single attributes are standard deviation and variance (**stddev** and **variance**). Standard deviation is the square root of variance.[2] Some database systems support other aggregate functions such as median and mode. Some database systems even allow users to add new aggregate functions.

SQL:1999 also supports a new class of **binary aggregate functions**, which can compute statistical results on pairs of attributes; they include correlations, covariances, and regression curves, which give a line approximating the relation between the values of the pair of attributes. Definitions of these functions may be found in any standard textbook on statistics, such as those referenced in the bibliographical notes.

SQL:1999 also supports generalizations of the **group by** construct, using the **cube** and **rollup** constructs. A representative use of the **cube** construct is:

select *item-name, color, size*, **sum**(*number*)
from *sales*
group by cube(*item-name, color, size*)

This query computes the union of eight different groupings of the *sales* relation:

{ (*item-name, color, size*), (*item-name, color*), (*item-name, size*),
(*color, size*), (*item-name*), (*color*), (*size*), () }

where () denotes an empty **group by** list.

For each grouping, the result contains the null value for attributes not present in the grouping. For instance, the table in Figure 22.2, with occurrences of **all** replaced by *null*, can be computed by the query

select *item-name, color*, **sum**(*number*)
from *sales*
group by cube(*item-name, color*)

2. The SQL:1999 standard actually supports two types of variance, called population variance and sample variance, and correspondingly two types of standard deviation. The definitions of the two types differ slightly; see a statistics textbook for details.

A representative **rollup** construct is

select *item-name, color, size*, **sum**(*number*)
from *sales*
group by rollup(*item-name, color, size*)

Here, only four groupings are generated:

{ (*item-name, color, size*), (*item-name, color*), (*item-name*), () }

Rollup can be used to generate aggregates at multiple levels of a hierarchy on a column. For instance, suppose we have a table *itemcategory(item-name, category)* giving the category of each item. Then the query

select *category, item-name*, **sum**(*number*)
from *sales*, *category*
where *sales.item-name = itemcategory.item-name*
group by rollup(*category, item-name*)

would give a hierarchical summary by *item-name* and by *category*.

Multiple **rollup**s and **cube**s can be used in a single group by clause. For instance, the following query

select *item-name, color, size*, **sum**(*number*)
from *sales*
group by rollup(*item-name*), **rollup**(*color, size*)

generates the groupings

{ (*item-name, color, size*), (*item-name, color*), (*item-name*),
(*color, size*), (*color*), () }

To understand why, note that **rollup**(*item-name*) generates two groupings, {(*item-name*), ()}, and **rollup**(*color, size*) generates three groupings, {(*color, size*), (*color*), () }. The cross product of the two gives us the six groupings shown.

As we mentioned in Section 22.2.1, SQL:1999 uses the value **null** to indicate the usual sense of null as well as **all**. This dual use of **null** can cause ambiguity if the attributes used in a rollup or cube clause contain null values. The function **grouping** can be applied on an attribute; it returns 1 if the value is a null value representing **all**, and returns 0 in all other cases. Consider the following query:

select *item-name, color, size*, **sum**(*number*),
grouping(*item-name*) **as** *item-name-flag*,
grouping(*color*) **as** *color-flag*,
grouping(*size*) **as** *size-flag*
from *sales*
group by cube(*item-name, color, size*)

The output is the same as in the version of the query without **grouping**, but with three extra columns called *item-name-flag*, *color-flag*, and *size-flag*. In each tuple, the value of a flag field is 1 if the corresponding field is a null representing **all**.

Instead of using tags to indicate nulls that represent **all**, we can replace the null value by a value of our choice:

decode(**grouping**(*item-name*), 1, 'all', *item-name*)

This expression returns the value "all" if the value of *item-name* is a null corresponding to **all**, and returns the actual value of *item-name* otherwise. This expression can be used in place of *item-name* in the select clause to get "all" in the output of the query, in place of nulls representing **all**.

Neither the **rollup** nor the **cube** clause gives complete control on the groupings that are generated. For instance, we cannot use them to specify that we want only groupings {(*color, size*), (*size, item-name*)}. Such restricted groupings can be generated by using the **grouping** construct in the **having** clause; we leave the details as an exercise for you.

22.2.4 Ranking

Finding the position of a value in a larger set is a common operation. For instance, we may wish to assign students a rank in class based on their total marks, with the rank 1 going to the student with the highest marks, the rank 2 to the student with the next highest marks, and so on. While such queries can be expressed in SQL-92, they are difficult to express and inefficient to evaluate. Programmers often resort to writing the query partly in SQL and partly in a programming language. A related type of query is to find the percentile in which a value in a (multi)set belongs, for example, the bottom third, middle third, or top third. We study SQL:1999 support for these types of queries here.

Ranking is done in conjunction with an **order by** specification. Suppose we are given a relation *student-marks(student-id, marks)* which stores the marks obtained by each student. The following query gives the rank of each student.

select *student-id*, **rank**() **over** (**order by** (*marks*) **desc**) **as** *s-rank*
from *student-marks*

Note that the order of tuples in the output is not defined, so they may not be sorted by rank. An extra **order by** clause is needed to get them in sorted order, as shown below.

select *student-id*, **rank** () **over** (**order by** (*marks*) **desc**) **as** *s-rank*
from *student-marks* **order by** *s-rank*

A basic issue with ranking is how to deal with the case of multiple tuples that are the same on the ordering attribute(s). In our example, this means deciding what to do if there are two students with the same marks. The **rank** function gives the same

rank to all tuples that are equal on the **order by** attributes. For instance, if the highest mark is shared by two students, both would get rank 1. The next rank given would be 3, not 2, so if three students get the next highest mark, they would all get rank 3, and the next student(s) would get rank 5, and so on. There is also a **dense_rank** function that does not create gaps in the ordering. In the above example, the tuples with the second highest value all get rank 2, and tuples with the third highest value get rank 3, and so on.

Ranking can be done within partitions of the data. For instance, suppose we have an additional relation *student-section(student-id, section)* that stores for each student the section in which the student studies. The following query then gives the rank of students within each section.

select *student-id*, *section*,
    rank () **over** (**partition by** *section* **order by** *marks* **desc**) **as** *sec-rank*
from *student-marks*, *student-section*
where *student-marks.student-id* = *student-section.student-id*
order by *section*, *sec-rank*

The outer **order by** clause orders the result tuples by section, and within each section by the rank.

Multiple **rank** expressions can be used within a single select statement; thus we can obtain the overall rank and the rank within the section by using two **rank** expressions in the same **select** clause. An interesting question is what happens when ranking (possibly with partitioning) occurs along with a **group by** clause. In this case, the **group by** clause is applied first, and partitioning and ranking are done on the results of the group by. Thus aggregate values can then be used for ranking. For example, suppose we had marks for each student for each of several subjects. To rank students by the sum of their marks in different subjects, we can use a **group by** clause to compute the aggregate marks for each student, and then rank students by the aggregate sum. We leave details as an exercise for you.

The ranking functions can be used to find the top n tuples by embedding a ranking query within an outer-level query; we leave details as an exercise. Note that bottom n is simply the same as top n with a reverse sorting order. Several database systems provide nonstandard SQL extensions to specify directly that only the top n results are required; such extensions do not require the rank function, and simplify the job of the optimizer, but are (currently) not as general since they do not support partitioning.

SQL:1999 also specifies several other functions that can be used in place of **rank**. For instance, **percent_rank** of a tuple gives the rank of the tuple as a fraction. If there are n tuples in the partition[3] and the rank of the tuple is r, then its percent rank is defined as $(r - 1)/(n - 1)$ (and as null if there is only one tuple in the partition). The function **cume_dist**, short for cumulative distribution, for a tuple is defined as p/n where p is the number of tuples in the partition with ordering values preceding or equal to the ordering value of the tuple, and n is the number of tuples in the parti-

3. The entire set is treated as a single partition if no explicit partition is used.

tion. The function **row_number** sorts the rows and gives each row a unique number corresponding to its position in the sort order; different rows with the same ordering value would get different row numbers, in a nondeterministic fashion.

Finally, for a given constant n, the ranking function **ntile**(n) takes the tuples in each partition in the specified order, and divides them into n buckets with equal numbers of tuples.[4] For each tuple, **ntile**(n) then gives the number of the bucket in which it is placed, with bucket numbers starting with 1. This function is particularly useful for constructing histograms based on percentiles. For instance, we can sort employees by salary, and use **ntile**(3) to find which range (bottom third, middle third, or top third) each employee is in, and compute the total salary earned by employees in each range:

```
select threetile, sum(salary)
from (
    select salary, ntile(3) over (order by (salary)) as threetile
    from employee) as s
group by threetile.
```

The presence of null values can complicate the definition of rank, since it is not clear where they should occur first in the sort order. SQL:1999 permits the user to specify where they should occur by using **nulls first** or **nulls last**, for instance

```
select student-id, rank () over (order by marks desc nulls last) as s-rank
from student-marks
```

22.2.5 Windowing

An example of a *window* query is query that, given sales values for each date, calculates for each date the average of the sales on that day, the previous day, and the next day; such moving average queries are used to smooth out random variations. Another example of a window query is one that finds the cumulative balance in an account, given a relation specifying the deposits and withdrawals on an account. Such queries are either hard or impossible (depending on the exact query) to express in basic SQL.

SQL:1999 provides a windowing feature to support such queries. In contrast to **group by**, the same tuple can exist in multiple windows. Suppose we are given a relation *transaction*(*account-number, date-time, value*), where *value* is positive for a deposit and negative for a withdrawal. We assume there is at most one transaction per *date-time* value.

Consider the query

4. If the total number of tuples in a partition is not divisible by n, then the number of tuples in each bucket can differ by at most 1. Tuples with the same value for the ordering attribute may be assigned to different buckets, nondeterministically, in order to make the number of tuples in each bucket equal.

```
select account-number, date-time,
    sum(value) over
        (partition by account-number
         order by date-time
         rows unbounded preceding)
    as balance
from transaction
order by account-number, date-time
```

The query gives the cumulative balances on each account just before each transaction on the account; the cumulative balance of the account is the sum of values of all earlier transactions on the account.

The **partition by** clause partitions tuples by account number, so for each row only the tuples in its partition are considered. A window is created for each tuple; the keywords **rows unbounded preceding** specify that the window for each tuple consists of all tuples in the partition that precede it in the specified order (here, increasing order of *date-time*). The aggregate function **sum**(*value*) is applied on all the tuples in the window. Observe that the query does not use a **group by** clause, since there is an output tuple for each tuple in the *transaction* relation.

While the query could be written without these extended constructs, it would be rather difficult to formulate. Note also that different windows can overlap, that is, a tuple may be present in more than one window.

Other types of windows can be specified. For instance, to get a window containing the previous 10 rows for each row, we can specify **rows** 10 **preceding**. To get a window containing the current, previous, and following row, we can use **between rows** 1 **preceding and** 1 **following**. To get the previous rows and the current row, we can say **between rows unbounded preceding and current**. Note that if the ordering is on a nonkey attribute, the result is not deterministic, since the order of tuples is not fully defined.

We can even specify windows by ranges of values, instead of numbers of rows. For instance, suppose the ordering value of a tuple is v; then **range between** 10 **preceding and current row** would give tuples whose ordering value is between $v - 10$ and v (both values inclusive). When dealing with dates, we can use **range interval** 10 **day preceding** to get a window containing tuples within the previous 10 days, but not including the date of the tuple.

Clearly, the windowing functionality of SQL:1999 is very rich and can be used to write rather complex queries with a small amount of effort.

22.3 Data Mining

The term **data mining** refers loosely to the process of semiautomatically analyzing large databases to find useful patterns. Like knowledge discovery in artificial intelligence (also called machine learning), or statistical analysis, data mining attempts to discover rules and patterns from data. However, data mining differs from machine learning and statistics in that it deals with large volumes of data, stored primarily on disk. That is, data mining deals with "knowledge discovery in databases."

Some types of knowledge discovered from a database can be represented by a set of **rules**. The following is an example of a rule, stated informally: "Young women with annual incomes greater than $50,000 are the most likely people to buy small sports cars." Of course such rules are not universally true, and have degrees of "support" and "confidence," as we shall see. Other types of knowledge are represented by equations relating different variables to each other, or by other mechanisms for predicting outcomes when the values of some variables are known.

There are a variety of possible types of patterns that may be useful, and different techniques are used to find different types of patterns. We shall study a few examples of patterns and see how they may be automatically derived from a database.

Usually there is a manual component to data mining, consisting of preprocessing data to a form acceptable to the algorithms, and postprocessing of discovered patterns to find novel ones that could be useful. There may also be more than one type of pattern that can be discovered from a given database, and manual interaction may be needed to pick useful types of patterns. For this reason, data mining is really a semiautomatic process in real life. However, in our description we concentrate on the automatic aspect of mining.

22.3.1 Applications of Data Mining

The discovered knowledge has numerous applications. The most widely used applications are those that require some sort of **prediction**. For instance, when a person applies for a credit card, the credit-card company wants to predict if the person is a good credit risk. The prediction is to be based on known attributes of the person, such as age, income, debts, and past debt repayment history. Rules for making the prediction are derived from the same attributes of past and current credit card holders, along with their observed behavior, such as whether they defaulted on their credit-card dues. Other types of prediction include predicting which customers may switch over to a competitor (these customers may be offered special discounts to tempt them not to switch), predicting which people are likely to respond to promotional mail ("junk mail"), or predicting what types of phone calling card usage are likely to be fraudulent.

Another class of applications looks for **associations**, for instance, books that tend to be bought together. If a customer buys a book, an online bookstore may suggest other associated books. If a person buys a camera, the system may suggest accessories that tend to be bought along with cameras. A good salesperson is aware of such patterns and exploits them to make additional sales. The challenge is to automate the process. Other types of associations may lead to discovery of causation. For instance, discovery of unexpected associations between a newly introduced medicine and cardiac problems led to the finding that the medicine may cause cardiac problems in some people. The medicine was then withdrawn from the market.

Associations are an example of **descriptive patterns**. **Clusters** are another example of such patterns. For example, over a century ago a cluster of typhoid cases was found around a well, which led to the discovery that the water in the well was contaminated and was spreading typhoid. Detection of clusters of disease remains important even today.

22.3.2 Classification

As mentioned in Section 22.3.1, prediction is one of the most important types of data mining. We outline what is classification, study techniques for building one type of classifiers, called decision tree classifiers, and then study other prediction techniques.

Abstractly, the **classification** problem is this: Given that items belong to one of several classes, and given past instances (called **training instances**) of items along with the classes to which they belong, the problem is to predict the class to which a new item belongs. The class of the new instance is not known, so other attributes of the instance must be used to predict the class.

Classification can be done by finding rules that partition the given data into disjoint groups. For instance, suppose that a credit-card company wants to decide whether or not to give a credit card to an applicant. The company has a variety of information about the person, such as her age, educational background, annual income, and current debts, that it can use for making a decision.

Some of this information could be relevant to the credit worthiness of the applicant, whereas some may not be. To make the decision, the company assigns a credit-worthiness level of excellent, good, average, or bad to each of a sample set of *current* customers according to each customer's payment history. Then, the company attempts to find rules that classify its current customers into excellent, good, average, or bad, on the basis of the information about the person, other than the actual payment history (which is unavailable for new customers). Let us consider just two attributes: education level (highest degree earned) and income. The rules may be of the following form:

$$\forall person\ \ P,\ P.degree = masters \textbf{ and } P.income > 75,000 \Rightarrow\ P.credit = excellent$$

$$\forall\ person\ \ P,\ P.degree = bachelors \textbf{ or } (P.income \geq 25,000 \textbf{ and } P.income \leq 75,000)\ \Rightarrow\ P.credit = good$$

Similar rules would also be present for the other credit worthiness levels (average and bad).

The process of building a classifier starts from a sample of data, called a **training set**. For each tuple in the training set, the class to which the tuple belongs is already known. For instance, the training set for a credit-card application may be the existing customers, with their credit worthiness determined from their payment history. The actual data, or population, may consist of all people, including those who are not existing customers. There are several ways of building a classifier, as we shall see.

22.3.2.1 Decision Tree Classifiers

The decision tree classifier is a widely used technique for classification. As the name suggests, **decision tree classifiers** use a tree; each leaf node has an associated class, and each internal node has a predicate (or more generally, a function) associated with it. Figure 22.6 shows an example of a decision tree.

To classify a new instance, we start at the root, and traverse the tree to reach a leaf; at an internal node we evaluate the predicate (or function) on the data instance,

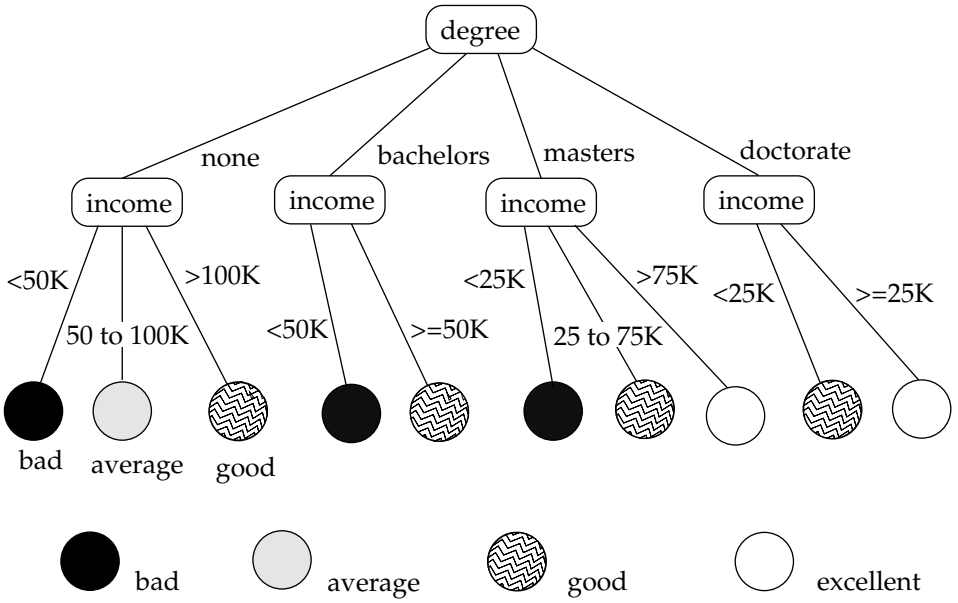


Figure 22.6 Classification tree.

to find which child to go to. The process continues till we reach a leaf node. For example, if the degree level of a person is masters, and the persons income is 40K, starting from the root we follow the edge labeled "masters," and from there the edge labeled "25K to 75K," to reach a leaf. The class at the leaf is "good," so we predict that the credit risk of that person is good.

Building Decision Tree Classifiers

The question then is how to build a decision tree classifier, given a set of training instances. The most common way of doing so is to use a **greedy** algorithm, which works recursively, starting at the root and building the tree downward. Initially there is only one node, the root, and all training instances are associated with that node.

At each node, if all, or "almost all" training instances associated with the node belong to the same class, then the node becomes a leaf node associated with that class. Otherwise, a **partitioning attribute** and **partitioning condition**s must be selected to create child nodes. The data associated with each child node is the set of training instances that satisfy the partitioning condition for that child node. In our example, the attribute *degree* is chosen, and four children, one for each value of degree, are created. The conditions for the four children nodes are *degree* = none, *degree* = bachelors, *degree* = masters, and *degree* = doctorate, respectively. The data associated with each child consist of training instances satisfying the condition associated with that child. At the node corresponding to masters, the attribute *income* is chosen, with the range of values partitioned into intervals 0 to 25,000, 25,000 to 50,000, 50,000 to 75,000, and over 75,000. The data associated with each node consist of training instances with the *degree* attribute being masters, and the *income* attribute being in each of these ranges,

respectively. As an optimization, since the class for the range 25,000 to 50,000 and the range 50,000 to 75,000 is the same under the node *degree* = masters, the two ranges have been merged into a single range 25,000 to 75,000.

Best Splits

Intuitively, by choosing a sequence of partitioning attributes, we start with the set of all training instances, which is "impure" in the sense that it contains instances from many classes, and end up with leaves which are "pure" in the sense that at each leaf all training instances belong to only one class. We shall see shortly how to measure purity quantitatively. To judge the benefit of picking a particular attribute and condition for partitioning of the data at a node, we measure the purity of the data at the children resulting from partitioning by that attribute. The attribute and condition that result in the maximum purity are chosen.

The purity of a set S of training instances can be measured quantitatively in several ways. Suppose there are k classes, and of the instances in S the fraction of instances in class i is p_i. One measure of purity, the **Gini measure** is defined as

$$\text{Gini}(S) = 1 - \sum_{i-1}^{k} p_i^2$$

When all instances are in a single class, the Gini value is 0, while it reaches its maximum (of $1 - 1/k$) if each class has the same number of instances. Another measure of purity is the **entropy measure**, which is defined as

$$\text{Entropy}(S) = -\sum_{i-1}^{k} p_i \log_2 p_i$$

The entropy value is 0 if all instances are in a single class, and reaches its maximum when each class has the same number of instances. The entropy measure derives from information theory.

When a set S is split into multiple sets $S_i, i = 1, 2, \ldots, r$, we can measure the purity of the resultant set of sets as:

$$\text{Purity}(S_1, S_2, \ldots, S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} \text{purity}(S_i)$$

That is, the purity is the weighted average of the purity of the sets S_i. The above formula can be used with both the Gini measure and the entropy measure of purity.

The **information gain** due to a particular split of S into $S_i, i = 1, 2, \ldots, r$ is then

$$\text{Information-gain}(S, \{S_1, S_2, \ldots, S_r\}) = \text{purity}(S) - \text{purity}(S_1, S_2, \ldots, S_r)$$

Splits into fewer sets are preferable to splits into many sets, since they lead to simpler and more meaningful decision trees. The number of elements in each of the sets S_i may also be taken into account; otherwise, whether a set S_i has 0 elements or 1 element would make a big difference in the number of sets, although the split is the same for almost all the elements. The **information content** of a particular split can be

defined in terms of entropy as

$$\text{Information-content}(S, \{S_1, S_2, \ldots, S_r\})) = -\sum_{i-1}^{r} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

All of this leads to a definition: The **best split** for an attribute is the one that gives the maximum **information gain ratio**, defined as

$$\frac{\text{Information-gain}(S, \{S_1, S_2, \ldots, S_r\})}{\text{Information-content}(S, \{S_1, S_2, \ldots, S_r\})}$$

Finding Best Splits

How do we find the best split for an attribute? How to split an attribute depends on the type of the attribute. Attributes can be either **continuous valued**, that is, the values can be ordered in a fashion meaningful to classification, such as age or income, or can be **categorical**, that is, they have no meaningful order, such as department names or country names. We do not expect the sort order of department names or country names to have any significance to classification.

Usually attributes that are numbers (integers/reals) are treated as continuous valued while character string attributes are treated as categorical, but this may be controlled by the user of the system. In our example, we have treated the attribute *degree* as categorical, and the attribute *income* as continuous valued.

We first consider how to find best splits for continuous-valued attributes. For simplicity, we shall only consider **binary splits** of continuous-valued attributes, that is, splits that result in two children. The case of **multiway splits** is more complicated; see the bibliographical notes for references on the subject.

To find the best binary split of a continuous-valued attribute, we first sort the attribute values in the training instances. We then compute the information gain obtained by splitting at each value. For example, if the training instances have values $1, 10, 15$, and 25 for an attribute, the split points considered are 1, 10, and 15; in each case values less than or equal to the split point form one partition and the rest of the values form the other partition. The best binary split for the attribute is the split that gives the maximum information gain.

For a categorical attribute, we can have a multiway split, with a child for each value of the attribute. This works fine for categorical attributes with only a few distinct values, such as degree or gender. However, if the attribute has many distinct values, such as department names in a large company, creating a child for each value is not a good idea. In such cases, we would try to combine multiple values into each child, to create a smaller number of children. See the bibliographical notes for references on how to do so.

Decision-Tree Construction Algorithm

The main idea of decision tree construction is to evaluate different attributes and different partitioning conditions, and pick the attribute and partitioning condition that results in the maximum information gain ratio. The same procedure works recur-

```
procedure GrowTree(S)
    Partition(S);

procedure Partition (S)
    if (purity(S) > δ_p or |S| < δ_s ) then
        return;
    for each attribute A
        evaluate splits on attribute A;
    Use best split found (across all attributes) to partition
            S into S_1, S_2, ..., S_r;
    for i = 1, 2, ..., r
        Partition(S_i);
```

Figure 22.7 Recursive construction of a decision tree.

sively on each of the sets resulting from the split, thereby recursively constructing a decision tree. If the data can be perfectly classified, the recursion stops when the purity of a set is 0. However, often data are noisy, or a set may be so small that partitioning it further may not be justified statistically. In this case, the recursion stops when the purity of a set is "sufficiently high," and the class of resulting leaf is defined as the class of the majority of the elements of the set. In general, different branches of the tree could grow to different levels.

Figure 22.7 shows pseudocode for a recursive tree construction procedure, which takes a set of training instances S as parameter. The recursion stops when the set is sufficiently pure or the set S is too small for further partitioning to be statistically significant. The parameters δ_p and δ_s define cutoffs for purity and size; the system may give them default values, that may be overridden by users.

There are a wide variety of decision tree construction algorithms, and we outline the distinguishing features of a few of them. See the bibliographical notes for details. With very large data sets, partitioning may be expensive, since it involves repeated copying. Several algorithms have therefore been developed to minimize the I/O and computation cost when the training data are larger than available memory.

Several of the algorithms also prune subtrees of the generated decision tree to reduce **overfitting**: A subtree is overfitted if it has been so highly tuned to the specifics of the training data that it makes many classification errors on other data. A subtree is pruned by replacing it with a leaf node. There are different pruning heuristics; one heuristic uses part of the training data to build the tree and another part of the training data to test it. The heuristic prunes a subtree if it finds that misclassification on the test instances would be reduced if the subtree were replaced by a leaf node.

We can generate classification rules from a decision tree, if we so desire. For each leaf we generate a rule as follows: The left-hand side is the conjunction of all the split conditions on the path to the leaf, and the class is the class of the majority of the training instances at the leaf. An example of such a classification rule is

$$degree = masters \textbf{ and } income > 75,000 \Rightarrow excellent$$

22.3.2.2 Other Types of Classifiers

There are several types of classifiers other than decision tree classifiers. Two types that have been quite useful are *neural net classifiers* and *Bayesian classifiers*. Neural net classifiers use the training data to train artificial neural nets. There is a large body of literature on neural nets, and we do not consider them further here.

Bayesian classifiers find the distribution of attribute values for each class in the training data; when given a new instance d, they use the distribution information to estimate, for each class c_j, the probability that instance d belongs to class c_j, denoted by $p(c_j|d)$, in a manner outlined here. The class with maximum probability becomes the predicted class for instance d.

To find the probability $p(c_j|d)$ of instance d being in class c_j, Bayesian classifiers use **Bayes' theorem**, which says

$$p(c_j|d) = \frac{p(d|c_j)p(c_j)}{p(d)}$$

where $p(d|c_j)$ is the probability of generating instance d given class c_j, $p(c_j)$ is the probability of occurrence of class c_j, and $p(d)$ is the probability of instance d occurring. Of these, $p(d)$ can be ignored since it is the same for all classes. $p(c_j)$ is simply the fraction of training instances that belong to class c_j.

Finding $p(d|c_j)$ exactly is difficult, since it requires a complete distribution of instances of c_j. To simplify the task, **naive Bayesian classifiers** assume attributes have independent distributions, and thereby estimate

$$p(d|c_j) = p(d_1|c_j) * p(d_2|c_j) * \ldots * p(d_n|c_j)$$

That is, the probability of the instance d occurring is the product of the probability of occurrence of each of the attribute values d_i of d, given the class is c_j.

The probabilities $p(d_i|c_j)$ derive from the distribution of values for each attribute i, for each class class c_j. This distribution is computed from the training instances that belong to each class c_j; the distribution is usually approximated by a histogram. For instance, we may divide the range of values of attribute i into equal intervals, and store the fraction of instances of class c_j that fall in each interval. Given a value d_i for attribute i, the value of $p(d_i|c_j)$ is simply the fraction of instances belonging to class c_j that fall in the interval to which d_i belongs.

A significant benefit of Bayesian classifiers is that they can classify instances with unknown and null attribute values—unknown or null attributes are just omitted from the probability computation. In contrast, decision tree classifiers cannot meaningfully handle situations where an instance to be classified has a null value for a partitioning attribute used to traverse further down the decision tree.

22.3.2.3 Regression

Regression deals with the prediction of a value, rather than a class. Given values for a set of variables, $X_1, X_2, \ldots, X_n$, we wish to predict the value of a variable Y. For instance, we could treat the level of education as a number and income as another number, and, on the basis of these two variables, we wish to predict the likelihood of

default, which could be a percentage chance of defaulting, or the amount involved in the default.

One way is to infer coefficients $a_0, a_1, a_1, \ldots, a_n$ such that

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \cdots + a_n * X_n$$

Finding such a linear polynomial is called **linear regression**. In general, we wish to find a curve (defined by a polynomial or other formula) that fits the data; the process is also called **curve fitting**.

The fit may only be approximate, because of noise in the data or because the relationship is not exactly a polynomial, so regression aims to find coefficients that give the best possible fit. There are standard techniques in statistics for finding regression coefficients. We do not discuss these techniques here, but the bibliographical notes provide references.

22.3.3 Association Rules

Retail shops are often interested in **associations** between different items that people buy. Examples of such associations are:

- Someone who buys bread is quite likely also to buy milk
- A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.

Association information can be used in several ways. When a customer buys a particular book, an online shop may suggest associated books. A grocery shop may decide to place bread close to milk, since they are often bought together, to help shoppers finish their task faster. Or the shop may place them at opposite ends of a row, and place other associated items in between to tempt people to buy those items as well, as the shoppers walk from one end of the row to the other. A shop that offers discounts on one associated item may not offer a discount on the other, since the customer will probably buy the other anyway.

Association Rules

An example of an association rule is

$$bread \Rightarrow milk$$

In the context of grocery-store purchases, the rule says that customers who buy bread also tend to buy milk with a high probability. An association rule must have an associated **population**: the population consists of a set of **instances**. In the grocery-store example, the population may consist of all grocery store purchases; each purchase is an instance. In the case of a bookstore, the population may consist of all people who made purchases, regardless of when they made a purchase. Each customer is an instance. Here, the analyst has decided that when a purchase is made is not significant, whereas for the grocery-store example, the analyst may have decided to concentrate on single purchases, ignoring multiple visits by the same customer.

Rules have an associated *support*, as well as an associated *confidence*. These are defined in the context of the population:

- **Support** is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.

  For instance, suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule

  $$milk \Rightarrow screwdrivers$$

  is low. The rule may not even be statistically significant—perhaps there was only a single purchase that included both milk and screwdrivers. Businesses are usually not interested in rules that have low support, since they involve few customers, and are not worth bothering about.

  On the other hand, if 50 percent of all purchases involve milk and bread, then support for rules involving bread and milk (and no other item) is relatively high, and such rules may be worth attention. Exactly what minimum degree of support is considered desirable depends on the application.

- **Confidence** is a measure of how often the consequent is true when the antecedent is true. For instance, the rule

  $$bread \Rightarrow milk$$

  has a confidence of 80 percent if 80 percent of the purchases that include bread also include milk. A rule with a low confidence is not meaningful. In business applications, rules usually have confidences significantly less than 100 percent, whereas in other domains, such as in physics, rules may have high confidences.

  Note that the confidence of $bread \Rightarrow milk$ may be very different from the confidence of $milk \Rightarrow bread$, although both have the same support.

Finding Association Rules

To discover association rules of the form

$$i_1, i_2, \ldots, i_n \Rightarrow i_0$$

we first find sets of items with sufficient support, called **large itemsets**. In our example we find sets of items that are included in a sufficiently large number of instances. We will shortly see how to compute large itemsets.

For each large itemset, we then output all rules with sufficient confidence that involve all and only the elements of the set. For each large itemset S, we output a rule $S - s \Rightarrow s$ for every subset $s \subset S$, provided $S - s \Rightarrow s$ has sufficient confidence; the confidence of the rule is given by support of s divided by support of S.

We now consider how to generate all large itemsets. If the number of possible sets of items is small, a single pass over the data suffices to detect the level of support for all the sets. A count, initialized to 0, is maintained for each set of items. When a purchase record is fetched, the count is incremented for each set of items such that

all items in the set are contained in the purchase. For instance, if a purchase included items a, b, and c, counts would be incremented for $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$, and $\{a, b, c\}$. Those sets with a sufficiently high count at the end of the pass correspond to items that have a high degree of association.

The number of sets grows exponentially, making the procedure just described infeasible if the number of items is large. Luckily, almost all the sets would normally have very low support; optimizations have been developed to eliminate most such sets from consideration. These techniques use multiple passes on the database, considering only some sets in each pass.

In the **a priori** technique for generating large itemsets, only sets with single items are considered in the first pass. In the second pass, sets with two items are considered, and so on.

At the end of a pass all sets with sufficient support are output as large itemsets. Sets found to have too little support at the end of a pass are eliminated. Once a set is eliminated, none of its supersets needs to be considered. In other words, in pass i we need to count only supports for sets of size i such that all subsets of the set have been found to have sufficiently high support; it suffices to test all subsets of size $i - 1$ to ensure this property. At the end of some pass i, we would find that no set of size i has sufficient support, so we do not need to consider any set of size $i + 1$. Computation then terminates.

22.3.4 Other Types of Associations

Using plain association rules has several shortcomings. One of the major shortcomings is that many associations are not very interesting, since they can be predicted. For instance, if many people buy cereal and many people buy bread, we can predict that a fairly large number of people would buy both, even if there is no connection between the two purchases. What would be interesting is a **deviation** from the expected co-occurrence of the two. In statistical terms, we look for **correlations** between items; correlations can be positive, in that the co-occurrence is higher than would have been expected, or negative, in that the items co-occur less frequently than predicted. See a standard textbook on statistics for more information about correlations.

Another important class of data-mining applications is sequence associations (or correlations). Time-series data, such as stock prices on a sequence of days, form an example of sequence data. Stock-market analysts want to find associations among stock-market price sequences. An example of such a association is the following rule: "Whenever bond rates go up, the stock prices go down within 2 days." Discovering such association between sequences can help us to make intelligent investment decisions. See the bibliographical notes for references to research on this topic.

Deviations from temporal patterns are often interesting. For instance, if a company has been growing at a steady rate each year, a deviation from the usual growth rate is surprising. If sales of winter clothes go down in summer, it is not surprising, since we can predict it from past years; a deviation that we could not have predicted from past experience would be considered interesting. Mining techniques can find deviations from what one would have expected on the basis of past temporal/sequential patterns. See the bibliographical notes for references to research on this topic.

22.3.5 Clustering

Intuitively, clustering refers to the problem of finding clusters of points in the given data. The problem of **clustering** can be formalized from distance metrics in several ways. One way is to phrase it as the problem of grouping points into k sets (for a given k) so that the average distance of points from the *centroid* of their assigned cluster is minimized.[5] Another way is to group points so that the average distance between every pair of points in each cluster is minimized. There are other definitions too; see the bibliographical notes for details. But the intuition behind all these definitions is to group similar points together in a single set.

Another type of clustering appears in classification systems in biology. (Such classification systems do not attempt to *predict* classes, rather they attempt to cluster related items together.) For instance, leopards and humans are clustered under the class mammalia, while crocodiles and snakes are clustered under reptilia. Both mammalia and reptilia come under the common class chordata. The clustering of mammalia has further subclusters, such as carnivora and primates. We thus have **hierarchical clustering**. Given characteristics of different species, biologists have created a complex hierarchical clustering scheme grouping related species together at different levels of the hierarchy.

Hierarchical clustering is also useful in other domains—for clustering documents, for example. Internet directory systems (such as Yahoo's) cluster related documents in a hierarchical fashion (see Section 22.5.5). Hierarchical clustering algorithms can be classified as **agglomerative clustering** algorithms, which start by building small clusters and then creater higher levels, or **divisive clustering** algorithms, which first create higher levels of the hierarchical clustering, then refine each resulting cluster into lower level clusters.

The statistics community has studied clustering extensively. Database research has provided scalable clustering algorithms that can cluster very large data sets (that may not fit in memory). The Birch clustering algorithm is one such algorithm. Intuitively, data points are inserted into a multidimensional tree structure (based on R-trees, described in Section 23.3.5.3), and guided to appropriate leaf nodes based on nearness to representative points in the internal nodes of the tree. Nearby points are thus clustered together in leaf nodes, and summarized if there are more points than fit in memory. Some postprocessing after insertion of all points gives the desired overall clustering. See the bibliographical notes for references to the Birch algorithm, and other techniques for clustering, including algorithms for hierarchical clustering.

An interesting application of clustering is to predict what new movies (or books, or music) a person is likely to be interested in, on the basis of:

1. The person's past preferences in movies
2. Other people with similar past preferences
3. The preferences of such people for new movies

5. The centroid of a set of points is defined as a point whose coordinate on each dimension is the average of the coordinates of all the points of that set on that dimension. For example in two dimensions, the centroid of a set of points $\{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \}$ is given by $(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n})$

One approach to this problem is as follows. To find people with similar past preferences we create clusters of people based on their preferences for movies. The accuracy of clustering can be improved by previously clustering movies by their similarity, so even if people have not seen the same movies, if they have seen similar movies they would be clustered together. We can repeat the clustering, alternately clustering people, then movies, then people, and so on till we reach an equilibrium. Given a new user, we find a cluster of users most similar to that user, on the basis of the user's preferences for movies already seen. We then predict movies in movie clusters that are popular with that user's cluster as likely to be interesting to the new user. In fact, this problem is an instance of *collaborative filtering*, where users collaborate in the task of filtering information to find information of interest.

22.3.6 Other Types of Mining

Text mining applies data mining techniques to textual documents. For instance, there are tools that form clusters on pages that a user has visited; this helps users when they browse the history of their browsing to find pages they have visited earlier. The distance between pages can be based, for instance, on common words in the pages (see Section 22.5.1.3). Another application is to classify pages into a Web directory automatically, according to their similarity with other pages (see Section 22.5.5).

Data-visualization systems help users to examine large volumes of data, and to detect patterns visually. Visual displays of data—such as maps, charts, and other graphical representations—allow data to be presented compactly to users. A single graphical screen can encode as much information as a far larger number of text screens. For example, if the user wants to find out whether production problems at plants are correlated to the locations of the plants, the problem locations can be encoded in a special color—say, red—on a map. The user can then quickly discover locations where problems are occurring. The user may then form hypotheses about why problems are occurring in those locations, and may verify the hypotheses quantitatively against the database.

As another example, information about values can be encoded as a color, and can be displayed with as little as one pixel of screen area. To detect associations between pairs of items, we can use a two-dimensional pixel matrix, with each row and each column representing an item. The percentage of transactions that buy both items can be encoded by the color intensity of the pixel. Items with high association will show up as bright pixels in the screen—easy to detect against the darker background.

Data visualization systems do not automatically detect patterns, but provide system support for users to detect patterns. Since humans are very good at detecting visual patterns, data visualization is an important component of data mining.

22.4 Data Warehousing

Large companies have presences in many places, each of which may generate a large volume of data. For instance, large retail chains have hundreds or thousands of stores, whereas insurance companies may have data from thousands of local branches. Further, large organizations have a complex internal organization structure, and there-

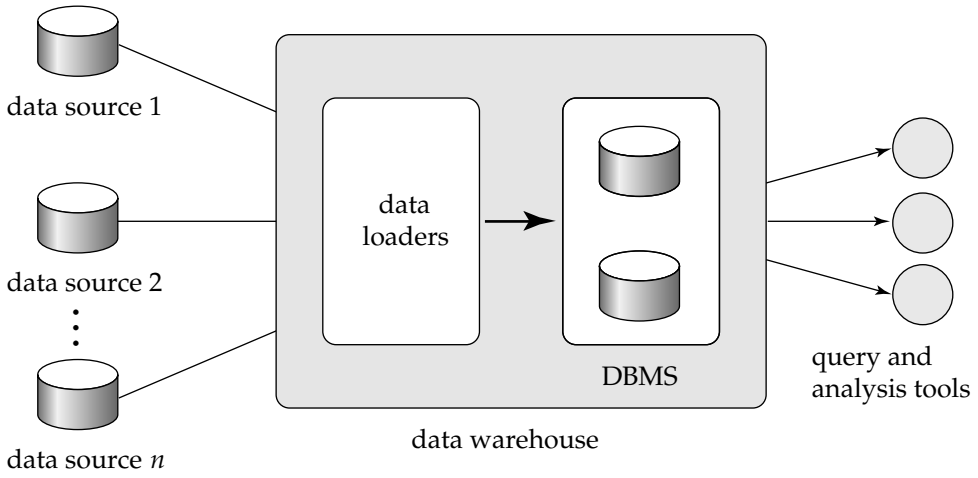


Figure 22.8 Data-warehouse architecture.

fore different data may be present in different locations, or on different operational systems, or under different schemas. For instance, manufacturing-problem data and customer-complaint data may be stored on different database systems. Corporate decision makers require access to information from all such sources. Setting up queries on individual sources is both cumbersome and inefficient. Moreover, the sources of data may store only current data, whereas decision makers may need access to past data as well; for instance, information about how purchase patterns have changed in the past year could be of great importance. Data warehouses provide a solution to these problems.

A **data warehouse** is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write. Moreover, by accessing information for decision support from a data warehouse, the decision maker ensures that online transaction-processing systems are not affected by the decision-support workload.

22.4.1 Components of a Data Warehouse

Figure 22.8 shows the architecture of a typical data warehouse, and illustrates the gathering of data, the storage of data, and the querying and data-analysis support. Among the issues to be addressed in building a warehouse are the following:

- **When and how to gather data.** In a **source-driven architecture** for gathering data, the data sources transmit new information, either continually (as transaction processing takes place), or periodically (nightly, for example). In a **destination-driven architecture**, the data warehouse periodically sends requests for new data to the sources.

Unless updates at the sources are replicated at the warehouse via two-phase commit, the warehouse will never be quite up to date with the sources. Two-phase commit is usually far too expensive to be an option, so data warehouses typically have slightly out-of-date data. That, however, is usually not a problem for decision-support systems.

- **What schema to use.** Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration, and to convert data to the integrated schema before they are stored. As a result, the data stored in the warehouse are not just a copy of the data at the sources. Instead, they can be thought of as a materialized view of the data at the sources.
- **Data cleansing.** The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, that can be corrected. For example, names are often misspelled, and addresses may have street/area/city names misspelled, or zip codes entered incorrectly. These can be corrected to a reasonable extent by consulting a database of street names and zip codes in each city. Address lists collected from multiple sources may have duplicates that need to be eliminated in a **merge–purge operation**. Records for multiple individuals in a house may be grouped together so only one mailing is sent to each house; this operation is called **householding**.
- **How to propagate updates**. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward. If they are not, the problem of propagating updates is basically the *view-maintenance* problem, which was discussed in Section 14.5.
- **What data to summarize.** The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation. For example, instead of storing data about every sale of clothing, we can store total sales of clothing by itemname and category.

  Suppose that a relation r has been replaced by a summary relation s. Users may still be permitted to pose queries as though the relation r were available online. If the query requires only summary data, it may be possible to transform it into an equivalent one using s instead; see Section 14.5.

22.4.2 Warehouse Schemas

Data warehouses typically have schemas that are designed for data analysis, using tools such as OLAP tools. Thus, the data are usually multidimensional data, with dimension attributes and measure attributes. Tables containing multidimensional data are called **fact tables** and are usually very large. A table recording sales information

for a retail store, with one tuple for each item that is sold, is a typical example of a fact table. The dimensions of the *sales* table would include what the item is (usually an item identifier such as that used in bar codes), the date when the item is sold, which location (store) the item was sold from, which customer bought the item, and so on. The measure attributes may include the number of items sold and the price of the items.

To minimize storage requirements, dimension attributes are usually short identifiers that are foreign keys into other other tables called **dimension tables**. For instance, a fact table *sales* would have attributes *item-id*, *store-id*, *customer-id*, and *date*, and measure attributes *number* and *price*. The attribute *store-id* is a foreign key into a dimension table *store*, which has other attributes such as store location (city, state, country). The *item-id* attribute of the *sales* table would be a foreign key into a dimension table *item-info*, which would contain information such as the name of the item, the category to which the item belongs, and other item details such as color and size. The *customer-id* attribute would be a foreign key into a *customer* table containing attributes such as name and address of the customer. We can also view the *date* attribute as a foreign key into a *date-info* table giving the month, quarter, and year of each date.

The resultant schema appears in Figure 22.9. Such a schema, with a fact table, multiple dimension tables, and foreign keys from the fact table to the dimension tables, is called a **star schema**. More complex data warehouse designs may have multiple levels of dimension tables; for instance, the *item-info* table may have an attribute *manufacturer-id* that is a foreign key into another table giving details of the manufacturer. Such schemas are called *snowflake schema*s. Complex data warehouse designs may also have more than one fact table.

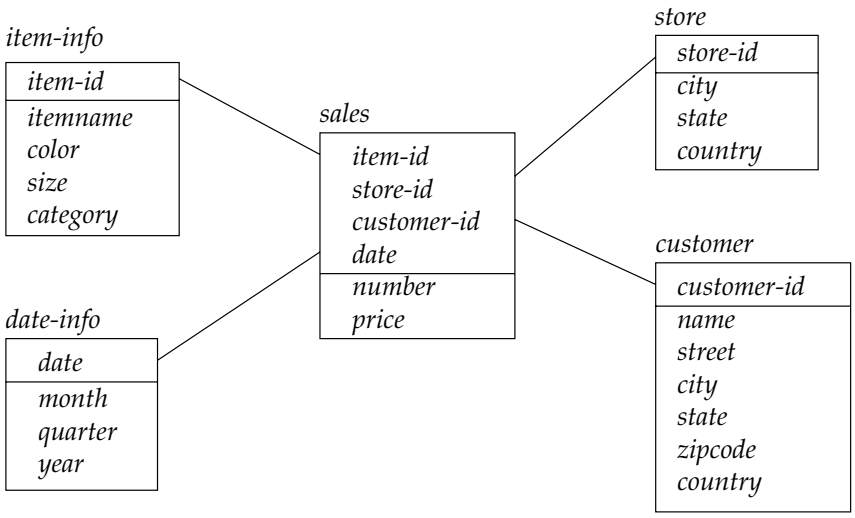


Figure 22.9 Star schema for a data warehouse.

22.5 Information-Retrieval Systems

The field of **information retrieval** has developed in parallel with the field of databases. In the traditional model used in the field of information retrieval, information is organized into documents, and it is assumed that there is a large number of documents. Data contained in documents is unstructured, without any associated schema. The process of information retrieval consists of locating relevant documents, on the basis of user input, such as keywords or example documents.

The Web provides a convenient way to get to, and to interact with, information sources across the Internet. However, a persistent problem facing the Web is the explosion of stored information, with little guidance to help the user to locate what is interesting. Information retrieval has played a critical role in making the Web a productive and useful tool, especially for researchers.

Traditional examples of information-retrieval systems are online library catalogs and online document-management systems such as those that store newspaper articles. The data in such systems are organized as a collection of *documents*; a newspaper article or a catalog entry (in a library catalog) are examples of documents. In the context of the Web, usually each HTML page is considered to be a document.

A user of such a system may want to retrieve a particular document or a particular class of documents. The intended documents are typically described by a set of **keywords**—for example, the keywords "database system" may be used to locate books on database systems, and the keywords "stock" and "scandal" may be used to locate articles about stock-market scandals. Documents have associated with them a set of keywords, and documents whose keywords contain those supplied by the user are retrieved.

Keyword-based information retrieval can be used not only for retrieving textual data, but also for retrieving other types of data, such as video or audio data, that have descriptive keywords associated with them. For instance, a video movie may have associated with it keywords such as its title, director, actors, type, and so on.

There are several differences between this model and the models used in traditional database systems.

- Database systems deal with several operations that are not addressed in information-retrieval systems. For instance, database systems deal with updates and with the associated transactional requirements of concurrency control and durability. These matters are viewed as less important in information systems. Similarly, database systems deal with structured information organized with relatively complex data models (such as the relational model or object-oriented data models), whereas information-retrieval systems traditionally have used a much simpler model, where the information in the database is organized simply as a collection of unstructured documents.
- Information-retrieval systems deal with several issues that have not been addressed adequately in database systems. For instance, the field of information retrieval has dealt with the problems of managing unstructured documents, such as approximate searching by keywords, and of ranking of documents on estimated degree of relevance of the documents to the query.

22.5.1 Keyword Search

Information-retrieval systems typically allow query expressions formed using keywords and the logical connectives *and*, *or*, and *not*. For example, a user could ask for all documents that contain the keywords "motorcycle *and* maintenance," or documents that contain the keywords "computer *or* microprocessor," or even documents that contain the keyword "computer *but not* database." A query containing keywords without any of the above connectives is assumed to have *and*s implicitly connecting the keywords.

In **full text** retrieval, all the words in each document are considered to be keywords. For unstructured documents, full text retrieval is essential since there may be no information about what words in the document are keywords. We shall use the word **term** to refer to the words in a document, since all words are keywords.

In its simplest form an information retrieval system locates and returns all documents that contain all the keywords in the query, if the query has no connectives; connectives are handled as you would expect. More sophisticated systems estimate relevance of documents to a query so that the documents can be shown in order of estimated relevance. They use information about term occurrences, as well as hyperlink information, to estimate relevance; Section 22.5.1.1 and 22.5.1.2 outline how to do so. Section 22.5.1.3 outlines how to define similarity of documents, and use similarity for searching. Some systems also attempt to provide a better set of answers by using the meanings of terms, rather than just the syntactic occurrence of terms, as outlined in Section 22.5.1.4.

22.5.1.1 Relevance Ranking Using Terms

The set of all documents that satisfy a query expression may be very large; in particular, there are billions of documents on the Web, and most keyword queries on a Web search engine find hundreds of thousands of documents containing the keywords. Full text retrieval makes this problem worse: Each document may contain many terms, and even terms that are only mentioned in passing are treated equivalently with documents where the term is indeed relevant. Irrelevant documents may get retrieved as a result.

Information retrieval systems therefore estimate relevance of documents to a query, and return only highly ranked documents as answers. Relevance ranking is not an exact science, but there are some well-accepted approaches.

The first question to address is, given a particular term t, how relevant is a particular document d to the term. One approach is to use the the number of occurrences of the term in the document as a measure of its relevance, on the assumption that relevant terms are likely to be mentioned many times in a document. Just counting the number of occurrences of a term is usually not a good indicator: First, the number of occurrences depends on the length of the document, and second, a document containing 10 occurrences of a term may not be 10 times as relevant as a document containing one occurrence.

One way of measuring $r(d, t)$, the relevance of a document d to a term t, is

$$r(d, t) = \log\left(1 + \frac{n(d, t)}{n(d)}\right)$$

where $n(d)$ denotes the number of terms in the document and $n(d, t)$ denotes the number of occurrences of term t in the document d. Observe that this metric takes the length of the document into account. The relevance grows with more occurrences of a term in the document, although it is not directly proportional to the number of occurrences.

Many systems refine the above metric by using other information. For instance, if the term occurs in the title, or the author list, or the abstract, the document would be considered more relevant to the term. Similarly, if the first occurrence of a term is late in the document, the document may be considered less relevant than if the first occurrence is early in the document. The above notions can be formalized by extensions of the formula we have shown for $r(d, t)$. In the information retrieval community, the relevance of a document to a term is referred to as **term frequency**, regardless of the exact formula used.

A query Q may contain multiple keywords. The relevance of a document to a query with two or more keywords is estimated by combining the relevance measures of the document to each keyword. A simple way of combining the measures is to add them up. However, not all terms used as keywords are equal. Suppose a query uses two terms, one of which occurs frequently, such as "web," and another that is less frequent, such as "Silberschatz." A document containing "Silberschatz" but not "web" should be ranked higher than a document containing the term "web" but not "Silberschatz."

To fix the above problem, weights are assigned to terms using the **inverse document frequency**, defined as $1/n(t)$, where $n(t)$ denotes the number of documents (among those indexed by the system) that contain the term t. The **relevance** of a document d to a set of terms Q is then defined as

$$r(d, Q) = \sum_{t \in Q} \frac{r(d, t)}{n(t)}$$

This measure can be further refined if the user is permitted to specify weights $w(t)$ for terms in the query, in which case the user-specified weights are also taken into account by using $w(t)/n(t)$ in place of $1/n(t)$.

Almost all text documents (in English) contain words such as "and," "or," "a," and so on, and hence these words are useless for querying purposes since their inverse document frequency is extremely low. Information-retrieval systems define a set of words, called **stop words**, containing 100 or so of the most common words, and remove this set from the document when indexing; such words are not used as keywords, and are discarded if present in the keywords supplied by the user.

Another factor taken into account when a query contains multiple terms is the **proximity** of the term in the document. If the terms occur close to each other in the document, the document would be ranked higher than if they occur far apart. The formula for $r(d, Q)$ can be modified to take proximity into account.

Given a query Q, the job of an information retrieval system is to return documents in descending order of their relevance to Q. Since there may be a very large number of documents that are relevant, information retrieval systems typically return only the first few documents with the highest degree of estimated relevance, and permit users to interactively request further documents.

22.5.1.2 Relevance Using Hyperlinks

Early Web search engines ranked documents by using only relevance measures similar to those described in Section 22.5.1.1. However, researchers soon realized that Web documents have information that plain text documents do not have, namely hyperlinks. And in fact, the relevance ranking of a document is affected more by hyperlinks that point *to* the document, than by hyperlinks going out of the document.

The basic idea of site ranking is to find sites that are popular, and to rank pages from such sites higher than pages from other sites. A site is identified by the internet address part of the URL, such as www.bell-labs.com in a URL http://www.bell-labs.com/topic/books/db-book. A site usually contains multiple Web pages. Since most searches are intended to find information from popular sites, ranking pages from popular sites higher is generally a good idea. For instance, the term "google" may occur in vast numbers of pages, but the site google.com is the most popular among the sites with pages that contain the term "google". Documents from google.com containing the term "google" would therefore be ranked as the most relevant to the term "google".

This raises the question of how to define the popularity of a site. One way would be to find how many times a site is accessed. However, getting such information is impossible without the cooperation of the site, and is infeasible for a Web search engine to implement. A very effective alternative uses hyperlinks; it defines $p(s)$, the **popularity of a site** s, as the number of sites that contain at least one page with a link to site s.

Traditional measures of relevance of the page (which we saw in Section 22.5.1.2) can be combined with the popularity of the site containing the page to get an overall measure of the relevance of the page. Pages with high overall relevance value are returned as answers to a query, as before.

Note also that we used the popularity of a *site* as a measure of relevance of individual pages at the site, not the popularity of individual *pages*. There are at least two reasons for this. First, most sites contain only links to root pages of other sites, so all other pages would appear to have almost zero popularity, when in fact they may be accessed quite frequently by following links from the root page. Second, there are far fewer sites than pages, so computing and using popularity of sites is cheaper than computing and using popularity of pages.

There are more refined notions of popularity of sites. For instance, a link from a popular site to another site s may be considered to be a better indication of the popularity of s than a link to s from a less popular site.[6] This notion of popularity

6. This is similar in some sense to giving extra weight to endorsements of products by celebrities (such as film stars), so its significance is open to question!

is in fact circular, since the popularity of a site is defined by the popularity of other sites, and there may be cycles of links between sites. However, the popularity of sites can be defined by a system of simultaneous linear equations, which can be solved by matrix manipulation techniques. The linear equations are defined in such a way that they have a unique and well-defined solution.

The popular Web search engine google.com uses the referring-site popularity idea in its definition **page rank**, which is a measure of popularity of a page. This approach of ranking of pages gave results so much better than previously used ranking techniques, that google.com became a widely used search engine, in a rather short period of time.

There is another, somewhat similar, approach, derived interestingly from a theory of social networking developed by sociologists in the 1950s. In the social networking context, the goal was to define the prestige of people. For example, the president of the United States has high prestige since a large number of people know him. If someone is known by multiple prestigious people, then she also has high prestige, even if she is not known by as large a number of people.

The above idea was developed into a notion of *hubs* and *authorities* that takes into account the presence of directories that link to pages containing useful information. A **hub** is a page that stores links to many pages; it does not in itself contain actual information on a topic, but points to pages that contain actual information. In contrast, an **authority** is a page that contains actual information on a topic, although it may not be directly pointed to by many pages. Each page then gets a prestige value as a hub (*hub-prestige*), and another prestige value as an authority (*authority-prestige*). The definitions of prestige, as before, are cyclic and are defined by a set of simultaneous linear equations. A page gets higher hub-prestige if it points to many pages with high authority-prestige, while a page gets higher authority-prestige if it is pointed to by many pages with high hub-prestige. Given a query, pages with highest authority-prestige are ranked higher than other pages. See the bibliographical notes for references giving further details.

22.5.1.3 Similarity-Based Retrieval

Certain information-retrieval systems permit **similarity-based retrieval**. Here, the user can give the system document A, and ask the system to retrieve documents that are "similar" to A. The similarity of a document to another may be defined, for example, on the basis of common terms. One approach is to find k terms in A with highest values of $r(d, t)$, and to use these k terms as a query to find relevance of other documents. The terms in the query are themselves weighted by $r(d, t)$.

If the set of documents similar to A is large, the system may present the user a few of the similar documents, allow him to choose the most relevant few, and start a new search based on similarity to A *and* to the chosen documents. The resultant set of documents is likely to be what the user intended to find.

The same idea is also used to help users who find many documents that appear to be relevant on the basis of the keywords, but are not. In such a situation, instead of adding further keywords to the query, users may be allowed to identify one or a few of the returned documents as relevant; the system then uses the identified documents

to find other similar ones. The resultant set of documents is likely to be what the user intended to find.

22.5.1.4 Synonyms and Homonyms

Consider the problem of locating documents about motorcycle maintenance for the keywords "motorcycle" and "maintenance." Suppose that the keywords for each document are the words in the title and the names of the authors. The document titled *Motorcycle Repair* would not be retrieved, since the word "maintenance" does not occur in its title.

We can solve that problem by making use of **synonyms**. Each word can have a set of synonyms defined, and the occurrence of a word can be replaced by the *or* of all its synonyms (including the word itself). Thus, the query "motorcycle *and* repair" can be replaced by "motorcycle *and* (repair *or* maintenance)." This query would find the desired document.

Keyword-based queries also suffer from the opposite problem, of **homonyms**, that is single words with multiple meanings. For instance, the word object has different meanings as a noun and as a verb. The word table may refer to a dinner table, or to a relational table. Some keyword query systems attempt to disambiguate the meaning of words in documents, and when a user poses a query, they find out the intended meaning by asking the user. The returned documents are those that use the term in the intended meaning of the user. However, disambiguating meanings of words in documents is not an easy task, so not many systems implement this idea.

In fact, a danger even with using synonyms to extend queries is that the synonyms may themselves have different meanings. Documents that use the synonyms with an alternative intended meaning would be retrieved. The user is then left wondering why the system thought that a particular retrieved document is relevant, if it contains neither the keywords the user specified, nor words whose intended meaning in the document is synonymous with specified keywords! It is therefore advisable to verify synonyms with the user, before using them to extend a query submitted by the user.

22.5.2 Indexing of Documents

An effective index structure is important for efficient processing of queries in an information-retrieval system. Documents that contain a specified keyword can be efficiently located by using an **inverted index**, which maps each keyword K_i to the set S_i of (identifiers of) the documents that contain K_i. To support relevance ranking based on proximity of keywords, such an index may provide not just identifiers of documents, but also a list of locations in the document where the keyword appears. Since such indices must be stored on disk, the index organization also attempts to minimize the number of I/O operations to retrieve the set of (identifiers of) documents that contain a keyword. Thus, the system may attempt to keep the set of documents for a keyword in consecutive disk pages.

The *and* operation finds documents that contain all of a specified set of keywords $K_1, K_2, \ldots, K_n$. We implement the *and* operation by first retrieving the sets of document identifiers $S_1, S_2, \ldots, S_n$ of all documents that contain the respective keywords.

The intersection, $S_1 \cap S_2 \cap \cdots \cap S_n$, of the sets gives the document identifiers of the desired set of documents. The *or* operation gives the set of all documents that contain at least one of the keywords $K_1, K_2, \ldots, K_n$. We implement the *or* operation by computing the union, $S_1 \cup S_2 \cup \cdots \cup S_n$, of the sets. The *not* operation finds documents that do not contain a specified keyword K_i. Given a set of document identifiers S, we can eliminate documents that contain the specified keyword K_i by taking the difference $S - S_i$, where S_i is the set of identifiers of documents that contain the keyword K_i.

Given a set of keywords in a query, many information retrieval systems do not insist that the retrieved documents contain all the keywords (unless an *and* operation is explicitly used). In this case, all documents containing at least one of the words are retrieved (as in the *or* operation), but are ranked by their relevance measure.

To use term frequency for ranking, the index structure should additionally maintain the number of times terms occur in each document. To reduce this effort, they may use a compressed representation with only a few bits, which approximates the term frequency. The index should also store the document frequency of each term (that is, the number of documents in which the term appears).

22.5.3 Measuring Retrieval Effectiveness

Each keyword may be contained in a large number of documents; hence, a compact representation is critical to keep space usage of the index low. Thus, the sets of documents for a keyword are maintained in a compressed form. So that storage space is saved, the index is sometimes stored such that the retrieval is approximate; a few relevant documents may not be retrieved (called a **false drop** or **false negative**), or a few irrelevant documents may be retrieved (called a **false positive**). A good index structure will not have *any* false drops, but may permit a few false positives; the system can filter them away later by looking at the keywords that they actually contain. In Web indexing, false positives are not desirable either, since the actual document may not be quickly accessible for filtering.

Two metrics are used to measure how well an information-retrieval system is able to answer queries. The first, **precision**, measures what percentage of the retrieved documents are actually relevant to the query. The second, **recall**, measures what percentage of the documents relevant to the query were retrieved. Ideally both should be 100 percent.

Precision and recall are also important measures for understanding how well a particular document ranking strategy performs. Ranking strategies can result in false negatives and false positives, but in a more subtle sense.

- False negatives may occur when documents are ranked, because relevant documents get low rankings; if we fetched all documents down to documents with very low ranking there would be very few false negatives. However, humans would rarely look beyond the first few tens of returned documents, and may thus miss relevant documents because they are not ranked among the top few. Exactly what is a false negative depends on how many documents are examined.

  Therefore instead of having a single number as the measure of recall, we can measure the recall as a function of the number of documents fetched.

- False positives may occur because irrelevant documents get higher rankings than relevant documents. This too depends on how many documents are examined. One option is to measure precision as a function of number of documents fetched.

A better and more intuitive alternative for measuring precision is to measure it as a function of recall. With this combined measure, both precision and recall can be computed as a function of number of documents, if required.

For instance, we can say that with a recall of 50 percent the precision was 75 percent, whereas at a recall of 75 percent the precision dropped to 60 percent. In general, we can draw a graph relating precision to recall. These measures can be computed for individual queries, then averaged out across a suite of queries in a query benchmark.

Yet another problem with measuring precision and recall lies in how to define which documents are really relevant and which are not. In fact, it requires understanding of natural language, and understanding of the intent of the query, to decide if a document is relevant or not. Researchers therefore have created collections of documents and queries, and have manually tagged documents as relevant or irrelevant to the queries. Different ranking systems can be run on these collections to measure their average precision and recall across multiple queries.

22.5.4 Web Search Engines

Web crawlers are programs that locate and gather information on the Web. They recursively follow hyperlinks present in known documents to find other documents. A crawler retrieves the documents and adds information found in the documents to a combined index; the document is generally not stored, although some search engines do cache a copy of the document to give clients faster access to the documents.

Since the number of documents on the Web is very large, it is not possible to crawl the whole Web in a short period of time; and in fact, all search engines cover only some portions of the Web, not all of it, and their crawlers may take weeks or months to perform a single crawl of all the pages they cover. There are usually many processes, running on multiple machines, involved in crawling. A database stores a set of links (or sites) to be crawled; it assigns links from this set to each crawler process. New links found during a crawl are added to the database, and may be crawled later if they are not crawled immediately. Pages found during a crawl are also handed over to an indexing system, which may be running on a different machine. Pages have to be refetched (that is, links recrawled) periodically to obtain updated information, and to discard sites that no longer exist, so that the information in the search index is kept reasonably up to date.

The indexing system itself runs on multiple machines in parallel. It is not a good idea to add pages to the same index that is being used for queries, since doing so would require concurrency control on the index, and affect query and update performance. Instead, one copy of the index is used to answer queries while another copy is updated with newly crawled pages. At periodic intervals the copies switch over, with the old one being updated while the new copy is being used for queries.

To support very high query rates, the indices may be kept in main memory, and there are multiple machines; the system selectively routes queries to the machines to balance the load among them.

22.5.5 Directories

A typical library user may use a catalog to locate a book for which she is looking. When she retrieves the book from the shelf, however, she is likely to *browse* through other books that are located nearby. Libraries organize books in such a way that related books are kept close together. Hence, a book that is physically near the desired book may be of interest as well, making it worthwhile for users to browse through such books.

To keep related books close together, libraries use a **classification hierarchy**. Books on science are classified together. Within this set of books, there is a finer classification, with computer-science books organized together, mathematics books organized together, and so on. Since there is a relation between mathematics and computer science, relevant sets of books are stored close to each other physically. At yet another level in the classification hierarchy, computer-science books are broken down into subareas, such as operating systems, languages, and algorithms. Figure 22.10 illustrates a classification hierarchy that may be used by a library. Because books can be kept at only one place, each book in a library is classified into exactly one spot in the classification hierarchy.

In an information retrieval system, there is no need to store related documents close together. However, such systems need to *organize documents logically* so as to permit browsing. Thus, such a system could use a classification hierarchy similar to

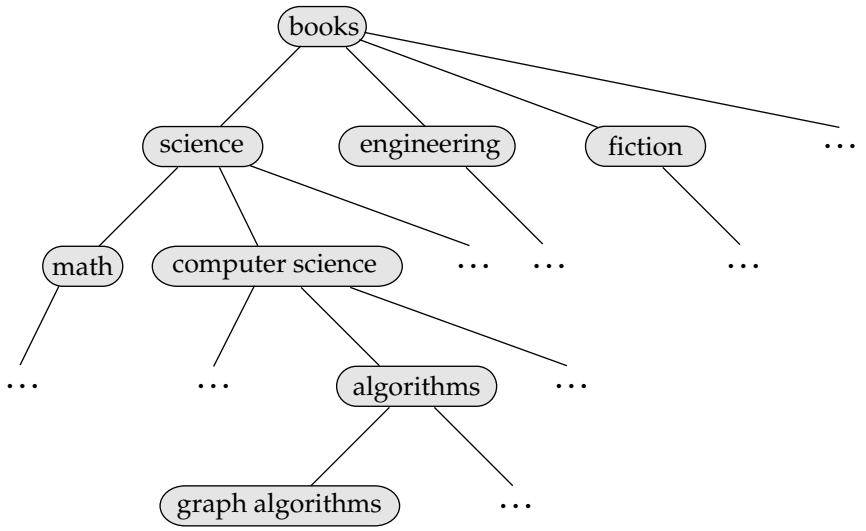


Figure 22.10 A classification hierarchy for a library system.

one that libraries use, and, when it displays a particular document, it can also display a brief description of documents that are close in the hierarchy.

In an information retrieval system, there is no need to keep a document in a single spot in the hierarchy. A document that talks of mathematics for computer scientists could be classified under mathematics as well as under computer science. All that is stored at each spot is an identifier of the document (that is, a pointer to the document), and it is easy to fetch the contents of the document by using the identifier.

As a result of this flexibility, not only can a document be classified under two locations, but also a subarea in the classification hierarchy can itself occur under two areas. The class of "graph algorithm" document can appear both under mathematics and under computer science. Thus, the classification hierarchy is now a directed acyclic graph (DAG), as shown in Figure 22.11. A graph-algorithm document may appear in a single location in the DAG, but can be reached via multiple paths.

A **directory** is simply a classification DAG structure. Each leaf of the directory stores links to documents on the topic represented by the leaf. Internal nodes may also contain links, for example to documents that cannot be classified under any of the child nodes.

To find information on a topic, a user would start at the root of the directory and follow paths down the DAG until reaching a node representing the desired topic. While browsing down the directory, the user can find not only documents on the topic he is interested in, but also find related documents and related classes in the classification hierarchy. The user may learn new information by browsing through documents (or subclasses) within the related classes.

Organizing the enormous amount of information available on the Web into a directory structure is a daunting task.

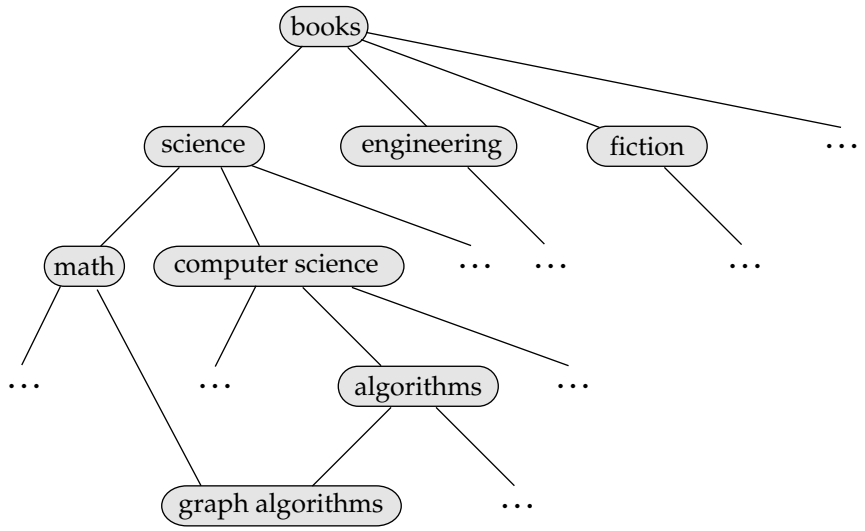


Figure 22.11 A classification DAG for a library information retrieval system.

- The first problem is determining what exactly the directory hierarchy should be.
- The second problem is, given a document, deciding which nodes of the directory are categories relevant to the document.

To tackle the first problem, portals such as Yahoo have teams of "internet librarians" who come up with the classification hierarchy and continually refine it. The *Open Directory Project* is a large collaborative effort, with different volunteers being responsible for organizing different branches of the directory.

The second problem can also be tackled manually by librarians, or Web site maintainers may be responsible for deciding where their sites should lie in the hierarchy. There are also techniques for automatically deciding the location of documents based on computing their similarity to documents that have already been classified.

22.6 Summary

- Decision-support systems analyze online data collected by transaction-processing systems, to help people make business decisions. Since most organizations are extensively computerized today, a very large body of information is available for decision support. Decision-support systems come in various forms, including OLAP systems and data mining systems.
- Online analytical processing (OLAP) tools help analysts view data summarized in different ways, so that they can gain insight into the functioning of an organization.
  - OLAP tools work on multidimensional data, characterized by dimension attributes and measure attributes.
  - The data cube consists of multidimensional data summarized in different ways. Precomputing the data cube helps speed up queries on summaries of data.
  - Cross-tab displays permit users to view two dimensions of multidimensional data at a time, along with summaries of the data.
  - Drill down, rollup, slicing, and dicing are among the operations that users perform with OLAP tools.
- The OLAP component of the SQL:1999 standard provides a variety of new functionality for data analysis, including new aggregate functions, cube and rollup operations, ranking functions, windowing functions, which support summarization on moving windows, and partitioning, with windowing and ranking applied inside each partition.
- Data mining is the process of semiautomatically analyzing large databases to find useful patterns. There are a number of applications of data mining, such as prediction of values based on past examples, finding of associations between purchases, and automatic clustering of people and movies.

- Classification deals with predicting the class of test instances, by using attributes of the test instances, based on attributes of training instances, and the actual class of training instances. Classification can be used, for instance, to predict credit-worthiness levels of new applicants, or to predict the performance of applicants to a university.

  There are several types of classifiers, such as

  - Decision-tree classifiers. These perform classification by constructing a tree based on training instances with leaves having class labels. The tree is traversed for each test instance to find a leaf, and the class of the leaf is the predicted class.

    Several techniques are available to construct decision trees, most of them based on greedy heuristics.
  - Bayesian classifiers are simpler to construct than decision-tree classifiers, and work better in the case of missing/null attribute values.

- Association rules identify items that co-occur frequently, for instance, items that tend to be bought by the same customer. Correlations look for deviations from expected levels of association.

- Other types of data mining include clustering, text mining, and data visualization.

- Data warehouses help gather and archive important operational data. Warehouses are used for decision support and analysis on historical data, for instance to predict trends. Data cleansing from input data sources is often a major task in data warehousing. Warehouse schemas tend to be multidimensional, involving one or a few very large fact tables and several much smaller dimension tables.

- Information retrieval systems are used to store and query textual data such as documents. They use a simpler data model than do database systems, but provide more powerful querying capabilities within the restricted model.

  Queries attempt to locate documents that are of interest by specifying, for example, sets of keywords. The query that a user has in mind usually cannot be stated precisely; hence, information-retrieval systems order answers on the basis of potential relevance.

- Relevance ranking makes use of several types of information, such as:
  - Term frequency: how important each term is to each document.
  - Inverse document frequency.
  - Site popularity. Page rank and hub/authority rank are two ways to assign importance to sites on the basis of links to the site.

- Similarity of documents is used to retrieve documents similar to an example document. Synonyms and homonyms complicate the task of information retrieval.

- Precision and recall are two measures of the effectiveness of an information retrieval system.
- Directory structures are used to classify documents with other similar documents.

Review Terms

- Decision-support systems
- Statistical analysis
- Multidimensional data
  - Measure attributes
  - Dimension attributes
- Cross-tabulation
- Data cube
- Online analytical processing (OLAP)
  - Pivoting
  - Slicing and dicing
  - Rollup and drill down
- Multidimensional OLAP (MOLAP)
- Relational OLAP (ROLAP)
- Hybrid OLAP (HOLAP)
- Extended aggregation
  - Variance
  - Standard deviation
  - Correlation
  - Regression
- Ranking functions
  - Rank
  - Dense rank
  - Partition by
- Windowing
- Data mining
- Prediction
- Associations
- Classification
  - Training data
  - Test data
- Decision-tree classifiers
  - Partitioning attribute
  - Partitioning condition
  - Purity
    - Gini measure
    - Entropy measure
  - Information gain
  - Information content
  - Information gain ratio
  - Continuous-valued attribute
  - Categorical attribute
  - Binary split
  - Multiway split
  - Overfitting
- Bayesian classifiers
  - Bayes theorem
  - Naive Bayesian classifiers
- Regression
  - Linear regression
  - Curve fitting
- Association rules
  - Population
  - Support
  - Confidence
  - Large itemsets
- Other types of associations
- Clustering
  - Hierarchical clustering
  - Agglomerative clustering
  - Divisive clustering
- Text mining
- Data visualization
- Data warehousing
  - Gathering data
  - Source-driven architecture

  - ☐ Destination-driven architecture
  - ☐ Data cleansing
    - — Merge–purge
    - — Householding
- Warehouse schemas
  - ☐ Fact table
  - ☐ Dimension tables
  - ☐ Star schema
- Information retrieval systems
- Keyword search
- Full text retrieval
- Term
- Relevance ranking
  - ☐ Term frequency
  - ☐ Inverse document frequency
  - ☐ Relevance
  - ☐ Proximity
- Stop words
- Relevance using hyperlinks
  - ☐ Site popularity
  - ☐ Page rank
  - ☐ Hub/authority ranking
- Similarity-based retrieval
- Synonyms
- Homonyms
- Inverted index
- False drop
- False negative
- False positive
- Precision
- Recall
- Web crawlers
- Directories
- Classification hierarchy

Exercises

22.1 For each of the SQL aggregate functions **sum, count, min** and **max**, show how to compute the aggregate value on a multiset $S_1 \cup S_2$, given the aggregate values on multisets S_1 and S_2.

Based on the above, give expressions to compute aggregate values with grouping on a subset S of the attributes of a relation $r(A, B, C, D, E)$, given aggregate values for grouping on attributes $T \supseteq S$, for the following aggregate functions:

a. **sum, count, min** and **max**
b. **avg**
c. standard deviation

22.2 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.

22.3 Give an example of a pair of groupings that cannot be expressed by using a single **group by** clause with **cube** and **rollup**.

22.4 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.

22.5 Given relation $r(a, b, d, d)$, Show how to use the extended SQL features to generate a histogram of d versus a, dividing a into 20 equal-sized partitions (that is, where each partition contains 5 percent of the tuples in r, sorted by a).

22.6 Write a query to find cumulative balances, equivalent to that shown in Section 22.2.5, but without using the extended SQL windowing constructs.

22.7 Consider the *balance* attribute of the *account* relation. Write an SQL query to compute a histogram of *balance* values, dividing the range 0 to the maximum account balance present, into three equal ranges.

22.8 Consider the *sales* relation from Section 22.2. Write an SQL query to compute the cube operation on the relation, giving the relation in Figure 22.2. Do not use the **with cube** construct.

22.9 Construct a decision tree classifier with binary splits at each node, using tuples in relation $r(A, B, C)$ shown below as training data; attribute C denotes the class. Show the final tree, and with each node show the best split for each attribute along with its information gain value.

$$(1, 2, a), (2, 1, a), (2, 5, b), (3, 3, b), (3, 6, b), (4, 5, b), (5, 5, c), (6, 3, b), (6, 7, c)$$

22.10 Suppose there are two classification rules, one that says that people with salaries between \$10,000 and \$20,000 have a credit rating of *good*, and another that says that people with salaries between \$20,000 and \$30,000 have a credit rating of *good*. Under what conditions can the rules be replaced, without any loss of information, by a single rule that says people with salaries between \$10,000 and \$30,000 have a credit rating of *good*.

22.11 Suppose half of all the transactions in a clothes shop purchase jeans, and one third of all transactions in the shop purchase T-shirts. Suppose also that half of the transactions that purchase jeans also purchase T-shirts. Write down all the (nontrivial) association rules you can deduce from the above information, giving support and confidence of each rule.

22.12 Consider the problem of finding large itemsets.

a. Describe how to find the support for a given collection of itemsets by using a single scan of the data. Assume that the itemsets and associated information, such as counts, will fit in memory.

b. Suppose an itemset has support less than j. Show that no superset of this itemset can have support greater than or equal to j.

22.13 Describe benefits and drawbacks of a source-driven architecture for gathering of data at a data-warehouse, as compared to a destination-driven architecture.

22.14 Consider the schema depicted in Figure 22.9. Give an SQL:1999 query to summarize sales numbers and price by store and date, along with the hierarchies on store and date.

22.15 Compute the relevance (using appropriate definitions of term frequency and inverse document frequency) of each of the questions in this chapter to the query "SQL relation."

22.16 What is the difference between a false positive and a false drop? If it is essential that no relevant information be missed by an information retrieval query, is it acceptable to have either false positives or false drops? Why?

22.17 Suppose you want to find documents that contain at least k of a given set of n keywords. Suppose also you have a keyword index that gives you a (sorted) list of identifiers of documents that contain a specified keyword. Give an efficient algorithm to find the desired set of documents.

Bibliographical Notes

Gray et al. [1995] and Gray et al. [1997] describe the data-cube operator. Efficient algorithms for computing data cubes are described by Agarwal et al. [1996], Harinarayan et al. [1996] and Ross and Srivastava [1997]. Descriptions of extended aggregation support in SQL:1999 can be found in the product manuals of database systems such as Oracle and IBM DB2. Definitions of statistical functions can be found in standard statistics textbooks such as Bulmer [1979] and Ross [1999].

Witten and Frank [1999] and Han and Kamber [2000] provide textbook coverage of data mining. Mitchell [1997] is a classic textbook on machine learning, and covers classification techniques in detail. Fayyad et al. [1995] presents an extensive collection of articles on knowledge discovery and data mining. Kohavi and Provost [2001] presents a collection of articles on applications of data mining to electronic commerce.

Agrawal et al. [1993] provides an early overview of data mining in databases. Algorithms for computing classifiers with large training sets are described by Agrawal et al. [1992] and Shafer et al. [1996]; the decision tree construction algorithm described in this chapter is based on the SPRINT algorithm of Shafer et al. [1996]. Agrawal and Srikant [1994] was an early paper on association rule mining. Algorithms for mining of different forms of association rules are described by Srikant and Agrawal [1996a] and Srikant and Agrawal [1996b]. Chakrabarti et al. [1998] describes techniques for mining surprising temporal patterns.

Clustering has long been studied in the area of statistics, and Jain and Dubes [1988] provides textbook coverage of clustering. Ng and Han [1994] describes spatial clustering techniques. Clustering techniques for large datasets are described by Zhang et al. [1996]. Breese et al. [1998] provides an empirical analysis of different algorithms for collaborative filtering. Techniques for collaborative filtering of news articles are described by Konstan et al. [1997].

Chakrabarti [2000] provides a survey of hypertext mining techniques such as hypertext classification and clustering. Chakrabarti [1999] provides a survey of Web resource discovery. Techniques for integrating data cubes with data mining are described by Sarawagi [2000].

Poe [1995] and Mattison [1996] provide textbook coverage of data warehousing. Zhuge et al. [1995] describes view maintenance in a data-warehousing environment.

Witten et al. [1999], Grossman and Frieder [1998], and Baeza-Yates and Ribeiro-Neto [1999] provide textbook descriptions of information retrieval. Indexing of documents is covered in detail by Witten et al. [1999]. Jones and Willet [1997] is a collection of articles on information retrieval. Salton [1989] is an early textbook on information-

retrieval systems. The TREC benchmark (trec.nist.gov) is a benchmark for measuring retrieval effectiveness.

Brin and Page [1998] describes the anatomy of the Google search engine, including the PageRank technique, while a hubs and authorities based ranking technique called HITS is described by Kleinberg [1999]. Bharat and Henzinger [1998] presents a refinement of the HITS ranking technique. A point worth noting is that the PageRank of a page is computed independent of any query, and as a result a highly ranked page which just happens to contain some irrelevant keywords would figure among the top answers for a query on the irrelevant keywords. In contrast, the HITS algorithm takes the query keywords into account when computing prestige, but has a higher cost for answering queries.

Tools

A variety of tools are available for each of the applications we have studied in this chapter. Most database vendors provide OLAP tools as part of their database system, or as add-on applications. These include OLAP tools from Microsoft Corp., Oracle Express, Informix Metacube. The Arbor Essbase OLAP tool is from an independent software vendor. The site www.databeacon.com provides an online demo of the databeacon OLAP tools for use on Web and text file data sources. Many companies also provide analysis tools specialized for specific applications, such as customer relationship management.

There is also a wide variety of general purpose data mining tools, including mining tools from the SAS Institute, IBM Intelligent Miner, and SGI Mineset. A good deal of expertise is required to apply general purpose mining tools for specific applications. As a result a large number of mining tools have been developed to address specialized applications. The Web site www.kdnuggets.com provides an extensive directory of mining software, solutions, publications, and so on.

Major database vendors also offer data warehousing products coupled with their database systems. These provide support functionality for data modeling, cleansing, loading, and querying. The Web site www.dwinfocenter.org provides information datawarehousing products.

Google (www.google.com) is a popular search engine. Yahoo (www.yahoo.com) and the Open Directory Project (dmoz.org) provide classification hierarchies for Web sites.