12 *Data Organization and Databases*

12.1 Introduction

One of the features that distinguishes data mining from other types of data analytic tasks is the quantity of data. In many data mining applications (such as Web log analysis for example) there may be millions of rows and thousands of columns in the standard form data matrix, so that questions of efficiency of data analysis algorithms are very important. An algorithm whose running time scales exponentially in the number n of rows may be unusable for all but the smallest data sets. Examples of operations that can be carried out in time $O(n)$ or $O(n \log n)$ are counting simple frequencies from the data, finding the mode of a discrete variable or attribute, or sorting the data. Generally, such computations are feasible even for large data sets. However, even a linear time algorithm can be prohibitively costly to use if multiple passes through a data set are required.

If the number of rows n of a data set influences algorithm complexity, so also can the number of variables p. For some applications p is very small (less than 10, for example), but in others, like market basket analysis or analysis of text documents, we can encounter data sets with 10^5 or even 10^6 variables. In such situations we cannot use methods that involve, for example, operations as the $O(p^2)$ computation of pairwise measures of association for all pairs of attributes.

In any data analysis project it is useful to distinguish between two phases. The first is actually getting the data to the analysis algorithm, and the second is running the analysis method itself. The first phase might seem trivial, but it can often become the bottleneck. For example, in analyzing a set of data it may be necessary to apply an algorithm to many different subsets of the data. This means we have to be able to search and identify the members

of each subset rapidly, and also to load that subset into main memory. Tree algorithms provide an obvious illustration of this, where the data set is progressively split into smaller subsets, each of which has to be identified before the tree can be extended. The purpose of *data organization* is to find methods for storing the data so that accessing subgroups of data is as fast as possible. Even in cases when all the data fit into main memory, data organization is important.

In addition to supporting efficient access to data for data mining algorithms, data organization plays an important role in the iterative and interactive nature of the overall data mining process. The aim of this chapter is to discuss briefly the memory hierarchy of modern computer and then present some index structures that database systems use to speed up the evaluation of queries. We then move to a discussion on relational databases and their query languages, as well as some special purpose database systems.

12.2 Memory Hierarchy

The memory of a computer is divided into several layers. These layers have different access times (where access time is the average time to retrieve a randomly selected byte of memory). Indeed, if disk storage were as fast as on-board cache, there would be no need to develop any sophisticated methods for data organization.

A general categorization of different memory structures is the following:

1. Registers of the processor. Typically there are fewer than 100 of these, and the processor can access data in the registers directly; that is, there is no slowdown associated with accessing a register.

2. On-processor or on-board cache. This is fast semiconductor memory implemented on the same chip as the processor or residing on the motherboard. Typical size is 16–1,000 kilobytes and access time is about 20 ns.

3. Main memory. Normal semiconductor memory, with sizes from 16 megabytes to several gigabytes, and access time about 70 ns.

4. Disk cache. Semiconductor memory implemented as an intermediate storage between main memory and disks.

5. Disk memory. Sizes vary from 1 gigabyte to hundreds or thousands of gigabytes for large arrays of disks. Typical access time is around 10 ms.

6. Magnetic tape. A magnetic tape can hold up to several gigabytes of data. Access time varies, but can be minutes.

The differences between the access times are truly large: in the 10 milliseconds needed for accessing a disk, we could perform up to a million accesses to fast cache. Another way to think about this is to pretend that access time is linearly proportional to actual distance. Thus, if we imagine main memory to be an effective distance of 1 meter away (within reach of your hand), the equivalent distance for disk memory is order of 10^5 times greater, i.e., 100 km!

Another major difference between main memory and disk is that individual bytes of main memory can be accessed, whereas for disk, whenever we access a byte, actually the whole disk page, about 4 kilobytes, containing that byte will be loaded to main memory. So if that page happens to contain information that can be used later, it will already be in fast memory. As an example, if we want to retrieve 1,000 integers, each taking 4 bytes to store, this can take between 1 and 1,000 disk accesses, depending on whether the integers are all stored in the same disk page or each on a page of their own.

The physical properties of the memory hierarchy lead to the following rules of thumb:

- If possible, data should be in main memory.
- In main memory, data items that are used together should be logically close to each other (that is, we should quickly be able to find the next element of a subset).
- On disk, data items that are used together should be also physically close to each other (that is, on the same disk page, if possible).

In practice, the user of a system typically has little control over the details of the way the data are placed in caches, or over the actual physical layout of data on disk. Normally, the systems try to load as much data as possible into main memory, and decide on their own how to deal the data objects onto disk pages. The user can influence the kinds of auxiliary structures that are created to access subgroups of the data. The next section describes in brief some of the data structures used for accessing large masses of data.

12.3 Index Structures

A primary goal of data organization is to find ways of quickly locating all the data points that satisfy a given selection condition. Usually the selection condition is a conjunction of conditions on individual attributes, such as "Age $\leq$ 40" and "Income $\leq$ 20,000." We consider first data structures that are especially applicable to situations in which there is only one conjunct.

An *index* on an attribute A is a data structure that makes it possible to locate points with a given value of A more efficiently than by a sequential scan of the whole data set. Indices are typically built either by the use of B*-trees or by the use of hash functions.

12.3.1 B-trees

A *search tree* is probably the simplest index structure. Suppose we have a set S of data vectors $\{\mathbf{x}(1), \ldots, \mathbf{x}(n)\}$, and that we want to find all points having a particular value of an ordinal attribute (variable) A as quickly as we can. A search tree is a binary tree structure such that each node has a value of A stored into it, and each leaf has a pointer to an element of S. Moreover, the tree is structured so that all elements of S pointed to by leaves from the left subtree of a node u containing value a will have values of A which are less than or equal to a. Likewise, all elements of S pointed to by leaves in the right subtree of u have values for A that are greater than a.

Given a binary search tree for an attribute A, it is easy to find the data points from S that have a given value b for A. We simply start from the root of the tree, selecting the left or the right subtree by comparing b against the values stored in the nodes. When we get to a leaf, either we find a pointer to the record(s) with $A = b$, or we find that no such pointer exists.

It is also easy to find all the points from S that satisfy the condition $b \leq A \leq c$, a so-called "interval query." Simply locate the leaf where b should be (as above), locate the leaf where c should be, and the desired records are pointed to by the leaves between these two positions.

The time needed for finding the records with a given value for attribute A is proportional to the height of the tree plus the number of such records. In the worst case, the height of the tree is n, the number of points in the set S, but there are ways of ensuring that the height of the tree will be $O(\log n)$ (although they are beyond the scope of this text). In practice, binary search trees are relatively seldom used, since B*-trees, discussed below, are clearly superior for accessing data on a disk.

The basic idea for B*-trees is the same as for search trees: the pointers to the data objects are in the leaves of the tree, and interior nodes contain values of the attribute A that indicate where certain pointers are to be found. However, instead of having two children and one value for A per interior node, a B*-tree typically has hundreds of children and values.

In more detail, a B*-tree of degree M for set of values is a tree where

- all leaves are at the same depth;
- each leaf contains between $M/2$ and M keys (possible target values);
- each interior node (except possibly the root) has K children $C_1, \ldots, C_K$, where $M/2 \leq K \leq M$ and $K-1$ values $a_1, \ldots, a_{K-1}$; for all i, all the key values stored in the leaves of subtree C_i are larger than a_{i-1} and at most as large as a_i.

Searching from a B*-tree is carried out in the same way as from a binary search tree: for each interior node of the tree, the values a_i are used to select the correct subtree.

A B*-tree differs from the basic binary search tree in that the height is guaranteed to be $O(\log n)$, since all leaves are on the same depth. Actually, the depth of the tree is bounded by $\log_{M/2} n$. Typically, the value of M is selected so that each node of the tree fits into a single disk page. If M is 100, then $(M/2)^5$ is over 300 million, and we find that for most realistic values of n, the number of elements in the set, the tree will have at most five levels: This means that finding a data point from 300 million points on the basis of the value of a single attribute can be done in three disk accesses, as the root node and the second level of the tree can be held in main memory. Most database management systems use B*-tree structures as one of their index structures.

12.3.2 Hash Indices

Suppose again that we have a set S of data points, and that we want to find all points such that attribute A has value a. If the set of possible values of A is small, we can do the following: for each possible value, construct a list of pointers to the data points with that value for A. Then, given the query "Find the points with $A = a$," we need only to access the list for a.

This method is not feasible, however, if there is a large number of potential values for A: we cannot maintain a list for each of the possible 2^{32} integers

which can be represented by 32 bits, for example. What we can do is to apply a transformation to the A-values so as to reduce the range of possible values.

In more detail, let $Dom(A)$ be the set of possible values of A. A *hash function* is a function h from $Dom(A)$ to $\{1, \ldots, M\}$, where M is the size of the hash table r. For each $j \in \{1, \ldots, M\}$ we store into $r[j]$ a list of pointers to those records $\mathbf{x}_i$ in S whose A value a_i satisfies $h(a_i) = j$. When we want to find all the data points with $A = a$, we simply compute $h(a)$, go to location $r[h(a)]$ and traverse the list of data points, for each of them checking whether the value of A really was a, or whether it was another value b with the property that $h(b) = h(a)$ (this is called a *collision*).

A typical hash function is a mod M, when M is chosen to be suitable prime larger than n, the number of data points. If the hash function is well chosen and the hash table is sufficiently large, collisions are rare, and searching for the points with a given A value can be done in time essentially proportional to the number of such points. Hash indices, however, do not directly support interval queries.

12.4 Multidimensional Indexing

Traditional index structures such as hashing and B*-trees provide fast access to rows of tables on the basis of values of a given attribute or collection of attributes. In some applications, however, it is necessary to express selection conditions on the basis of several attributes, and normal index structures do not help. Consider, for example, the case of storing geographic information about cities. Suppose, for example, we wish to find all the cities with latitude between 30 N and 40 N, longitude between 60 W and 70 W, and population at least 1,000. Such a query is called a *rectangular range query*. Suppose the cities table is large, containing millions of city names. How should the query be evaluated? A B*-tree index on the latitude attribute makes it possible to find the cities that satisfy the conditions for that attribute, but for finding the rows that satisfy the conditions on longitude among these, we have to resort to a sequential scan. Similarly, an index on longitude does not help much. What is needed is an index structure that makes it possible to use directly the conditions on both attributes.

Multidimensional indexing refers to techniques for finding rows of tables on the basis of conditions on multiple attributes. One of the widely used methods is the R*-tree. Each node in the tree corresponds to a region in the underlying space, and the node represents the points within that region. For

dimensions up to about 10, the multidimensional index structures speed up searches on large databases. Fast evaluation of range queries for data sets with larger numbers of dimensions (e.g., in the 100s) is still an open problem.

12.5 Relational Databases

In data mining we often need to access a particular subset of the data and compute a function from the values of certain attributes on that subset. We have discussed some data structures that can help in finding the relevant data points quickly. Relational databases provide a unified mechanism for fast access to selected parts of the data.

In database terminology, a *data model* is a set of constructs that can be used to describe the structure of data, plus a set of operations for manipulating the data. (Note that this use of the word *model* is rather different from that given earlier in the book. Here it is a structure imposed on the data by design, rather than a structure discovered existing within the data. The dual use of the word *model* is perhaps unfortunate, and arises because of the different disciplines that have contributed to data mining; in this case, statistics and database theory. Fortunately, confusion seldom arises; which of the two meanings is intended will generally be clear from the context). The *relational data model* is based on the idea of representing data in tabular form. A table header (*schema*) consists of the table name and a set of named columns; the column names are also called *attributes*. The actual *table* (an instance of the schema), also called a *relation*, is a named set of rows. Each table entry in the column for attribute A is a value from the domain $Dom(A)$ of A. Note that when the attributes are defined, the domain of each must also be specified. An attribute can be of any data type: categorical, numeric, etc. The order of the row and columns in a table is not significant.

We can put this more formally. A *relation schema* R is a set of attributes $\{A_1, \ldots, A_p\}$, where each attribute A_j has an associated domain $Dom(A_j)$. A row over the schema R is a mapping $t : R \rightarrow \cup_i Dom(A_j)$ where $t(A_j) \in Dom(A_j)$. A table or relation over the schema R is a collection of rows over R. A relational database schema $\mathbf{R}$ is a collection $\{R_1, \ldots, R_k\}$ of relation schemas (with possibly some constraints on the relation instances), and a relational database $\mathbf{r}$ over the schema $\mathbf{R}$ consists of a relation over R_i, for each $i = 1, \ldots, k$.

Example 12.1 Consider a retail outlet with barcode readers, or a Web site where we log each purchase by a customer. For each *customer transaction*, also called

transactions

basket-id	chips	mustard	sausage	Pepsi	Coca-Cola	Miller	Bud
t_1	1	0	0	0	0	1	0
t_2	2	1	3	5	0	1	0
t_3	1	0	1	0	1	0	0
t_4	0	0	2	0	0	6	0
t_5	0	1	1	1	0	0	2
t_6	1	1	1	0	0	1	0
t_7	4	0	2	4	0	1	0
t_8	0	1	1	0	4	0	1
t_9	1	0	0	1	0	0	1
t_{10}	0	1	2	0	4	1	1

Figure 12.1 Representing market basket data as a table with an attribute for each product.

here a *basket*, we can collect information about which products the customer bought, and how many of each product. In principle, these data could be represented as a table, where there is an attribute for each product and a row for each transaction. For row t and attribute A the entry $t(A)$ in the matrix indicates how many As the customer bought. That is, for each attribute A the domain $Dom(A)$ is the set of nonnegative integers. See figure 12.1 for an example table, here called *transactions*.

As the product selection probably changes rapidly, encoding the names of products into attributes may not be a very good idea. An alternative representation would be to use a table such as the one called *baskets*, shown in figure 12.2, where the product names are represented as entries. This table has three attributes, basket-id, product, and quantity, and the domain of product is the set of all strings, while the domain of quantity is the set of nonnegative numbers. Note that there is no unique way of representing a given set of data as a relational database: both the transactions and baskets tables represent the same data set.

In addition to the data about the transactions, the retailer maintains information about the prices of individual products. This could be represented as a table such as the products table shown in figure 12.3.

The product data can be too detailed for useful summaries. Therefore, the retailer could use a classification of various products into larger product categories. An example is shown in figure 12.4.

baskets

basket-id	product	quantity
t_1	chips	1
t_1	Miller	1
t_2	chips	2
t_2	mustard	1
t_2	sausage	3
t_2	Pepsi	5
t_2	Miller	1
	...	

Figure 12.2 A more realistic representation of market basket data.

products

product	price	supplier	category
chips	1.00	ABC	food
Miller	0.55	ABC	drink
mustard	1.25	DEF	spices
sausage	2.00	DEF	food
Pepsi	0.75	ABC	drink
Coke	0.75	DEF	drink
	...		

Figure 12.3 Representing prices of products.

The table describes a hierarchy, in saying that Pepsi and Coke are soft drinks, and that soft drinks and beers are drinks.

The schemas of the tables in this example can be described succinctly by listing just the names of the tables and their attributes:

```
baskets(basket-id,product,quantity)
products(product,price)
product-hierarchy(product,category)
```

Thus the relational data model is based on the idea of tabular representation. The values in the cells may be arbitrary atomic values, such as real

product-hierarchy

product	category
Pepsi	soft drink
Coke	soft drink
Budweiser	beer
Miller	beer
soft drink	drink
beer	drink
...	

Figure 12.4 Representing the hierarchy of products as a table.

numbers, integers, or strings; sets or lists of values are not allowed. This means that, if, for example, we want to represent information about people, their ages, and phone numbers, we cannot store multiple phone numbers in one attribute. If restricted in this way, the model is said to have *first normal form*.

The relational model is widely used in data management, and virtually all major database systems are based on it. Some systems provide additional functionality, such as the possibility of using object-oriented data modeling methods.

Even in relatively small organizations, relational databases can have hundreds of tables and thousands of attributes. Managing the schema of the database can, therefore, be a complicated task. Sometimes it is claimed that for data analysis purposes it suffices to combine all the tables into a massive observation matrix, or "universal table," and that therefore in data mining one does not have to care about the fact that the data are in a database. However, an examination of simple examples shows that this is not feasible: the universal table would be so large that operations on it would be prohibitively costly.

Example 12.2 Consider the example of products in a supermarket, and see what it would look like in a more realistic setting. Instead of having a table with attributes Product and Price only, we probably would have a table with at least attributes Product, Supplier, and Price, and an additional table about suppliers with attributes Supplier, Address, Phone Number, etc. If we wanted to combine the tables into one table, this table would have to include attributes Transaction ID, Product, Number, Supplier Address, Phone Number, Product

Price, etc. Furthermore, if each product belongs on the average to K different product groups, including the information from the Product-Hierarchy table would increase the size of the representation by a factor of K. For even a moderately sized database, this combining process would lead to a table that would be far too large to be stored explicitly.

12.6 Manipulating Tables

Being able to describe the structure of data and to store data using this structure is not sufficient in itself for data management: we also must be able to retrieve data from the database. We briefly describe two languages for manipulating collections of tables (that is, relational databases): relational algebra, in this section, and the Structured Query Language (SQL), in the next. Relational algebra is based on set-theoretic notation and is quite handy for theoretical purposes, while SQL is widely used in practice.

In the examples, we use r, s, etc. to refer to tables, and R, S, etc. to refer to the sets of attributes for those tables.

Relational algebra contains a set of basic operations for manipulating data given in tabular form, and several derived operations (operations that can be expressed as a sequence of basic operations) are also used. The operations include the three set operations—union, intersection, and difference—and the projection operation for removing columns, the selection operation for selecting rows, and the join and Cartesian product operations for combining rows from two tables.

Example 12.3 The operations of relational algebra are formally defined as follows: Assume r and s are tables over the set R of attributes,

Union $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$.

Intersection $r \cap s = \{t \mid t \in r \text{ and } t \in s\}$.

Difference $r \setminus s = \{t \mid t \in r \text{ and } t \notin s\}$.

Projection Given $X \subseteq R$, then $r[X] = \{t[X] \mid t \in r\}$, where $t[X]$ is the row obtained from row t by leaving only the values in the columns of X.

Selection Given a condition F on rows of table r,

$$\sigma_F(r) = \{t \in r \mid t \text{ satisfies } F\}.$$

Join $r \bowtie s = \{tu \mid t \in r, u \in s, t[A] = u[A] \text{ for all } A \in R \cap S\}$, where tu is the row obtained by pasting t and u together.

Set Operations

Tables are sets of rows, and all operations in the relational algebra are set-oriented: they take sets as arguments and produce a set as their result. This makes it possible to compose relational queries: the results of a query are relations, as are the arguments.

Conventional set operations are useful for manipulating tables. We shall include *union*, *intersection*, and *difference* (denoted by $r \cup s$, $r \cap s$, and $r \setminus s$, respectively) as the basic operations in relational algebra. The *union* operation combines two tables over the same set of attributes: the result $r \cup s$ contains all the rows that occur in r or s. The *intersection* operation $r \cap s$ results in the table containing those rows that occur in r and in s. The *difference* operation $r \setminus s$ gives the rows that occur in r but not in s. These operations all assume that r and s are tables over the same set of attributes.

As an example, suppose r is a table representing the prices of all soft drinks, and s is a table representing the prices of all products costing at most \$2.00. Then $r \cup s$ is the table of all soft drinks and products costing less than \$2.00, $r \cap s$ is the table of all soft drinks costing less than \$2.00, and $r \setminus s$ contains one row for each soft drink that does not cost less than \$2.00, i.e. that costs at least \$2.00. The intersection operation could, of course, be defined using the union and difference operations: $r \cap s = (r \cup s) \setminus ((r \setminus s) \cup (s \setminus r))$.

Care must be taken to ensure that the resulting set is a table, in the sense that it has a schema. Therefore $r \cup s$, $r \cap s$ and $r \setminus s$ are defined only if r and s are tables over the same schema—that is, over the same set of attributes.

Intersection queries can be used in construction of rule sets, for example. (Algorithms for rule learning are discussed in chapter 13.) Suppose, we have computed a table r corresponding to the observations that satisfy a condition F, and similarly another table s that corresponds to the observations satisfying condition G. The intersection $r \cap s$ corresponds to those observations that satisfy both conditions; the cardinality of the intersection tells what the overlap between the conditions are. If r and s are computed from the same base table of observations, we can also achieve the same effect by using the conjunction $F \wedge G$ as the selection condition in the query. Intersection queries occur most naturally in situations in which we need to check whether the same value occurs in two tables.

Projection

The purpose of the projection operation is to trim a table so that only the data in specific columns of interest remain. Given a table r with attributes R, and $X \subseteq R$, the projection of r on X is obtained by removing from the table all the columns outside X. A side effect of projecting a table is that the number of rows, as well as the number of columns, may decrease. If the argument table over R is projected on a set of attributes X, and if table r over R contains two rows that agree on the X attributes, but differ on some attribute in $R \setminus X$, the projected rows would be identical. Such identical rows are commonly called *duplicates*. Since tables are sets, they cannot contain duplicates, and only one representative of each duplicate is retained. Because this feature is implicit in the concept of a set, it does not show up in the definition of the projection operation.

Commercial database systems often differ from the pure relational model on this point. In real implementations, tables are stored as files. Files, of course, *can* contain several identical records. Checking the uniqueness of records could take a lot of time. It is therefore customary that tables in commercial database management systems can contain duplicates.

The projection operation in relational databases is related to but not identical to the projection encountered in vector spaces. Both operations take points (called rows in databases) and produce points in a lower-dimensional space (rows with fewer attributes). In relational databases, we can project only to subspaces defined directly by the attributes; for vector spaces, projection can be defined for any subspace (that is, any linear combination of basis vectors (here attributes)).

Selection

The selection operation is used to select rows from a table. Given a Boolean condition F on the rows of a table r, the selection operation σ_F applied to r yields the table $\sigma_F(r)$ consisting of those rows of r that satisfy the condition.

Selection is probably the most frequently used operation of the relational algebra: each time we want to focus on a particular row or subset of rows in a table, we need to use selection. Selection occurs often in the implementation of data mining algorithms. For example, in building a decision tree we want a list of the observations that belong to a particular node of the tree. This set of observations is exactly the answer to a selection query, where the selection condition is the conjunction of the conditions appearing in the nodes from

the root of the tree to the node in question. Similarly, if we want to implement association rule algorithms using the relational algebra, one has to execute several selection queries, each one that looks at the subset of observations satisfying the condition that each variable in a candidate frequent set has value 1.

In pure relational algebra, selections are based on exact equalities or inequalities. For data mining, we often need concepts of inexact or approximate matching. If a predicate `match` for approximate matching between attribute values is available, we can (at least in some database systems) use that directly in database operations to select rows that satisfy the approximate matching condition. (Chapter 14 discusses approximate matching in more detail.)

Cartesian Product and Join

Both projection and selection are used for removing data from a table. The *join* and *Cartesian product* operations are used for connecting data that are stored in two different tables. Given tables r and s with attributes R and S, respectively, and assuming that R and S are disjoint (that is, that no attribute name occurs in both) then the *Cartesian product* $r \times s$ of r and s is a table over the attributes $R \cup S$, and it contains all rows that can be obtained by pasting together a row from r and a row from s. Thus $r \times s$ will have $|r||s|$ rows, where $|r|$ is the number of rows in r.

The Cartesian product is needed for combining rows from different tables. It is seldom used by itself; more often, we use the *join* operation. Given a selection condition F, the *join* $r \bowtie_F s$ of r and s is obtained by selecting the rows satisfying F from $r \times s$. For example, we might compute the join of tables `baskets` and `products`, using the equality `baskets.product = products.product` as the join condition. The result of this operation is a table that has columns for the basket id, for the product name, quantity, and price. (To be precise, the result has two columns for the product name, one from each of the original tables; we might want to project one of them away.)

A typical application of the join in data mining algorithms is to combine different sources of information. If for example, we have data about customer demographics and customer purchase behavior, such data are usually stored in different tables. To combine the relevant pieces of data, we need to do a join operation.

12.7 The Structured Query Language (SQL)

Relational algebra is a useful and compact notation. In database management systems, SQL is the standard adopted by most database management system vendors. SQL implements a superset of the relational algebra. Here we introduce only the basic structure of SQL programs.

The basic statement of SQL is the "select-from-where" expression or query, which has the form

select $A_1, A_2, \ldots, A_p$
from $r_1, r_2, \ldots, r_k$
where list of conditions

Here each r_i is a table, and each A_j is an attribute. The intuitive meaning is that for each possible choice of rows $t_1, \ldots, t_k$ from the tables $r_1, \ldots, r_k$, we test whether the conditions are true. If they are, a row consisting of the values of the attributes A_j is output.

The second line of the query, the *from clause*, specifies the tables to which the SQL statement is applied. The third line, the *where clause*, specifies the conditions that the rows in those tables must satisfy to be accepted into the result of the statement. The first line, the *select clause*, then specifies which attributes of the participating tables should appear in the result. It corresponds to the projection operation of relational algebra (*not* the selection operation). The "where" clause is used for representing the selection conditions occurring in the selection and the join operations. For a selection operation, the selection conditions are simply listed in the list of conditions of the where clause, separated by the keywords **and**, **or**, and **not**.

Example 12.4 All products that cost more than 2.00 can be found by the query

select product
from products
where price $>$ 2.00

Finding all transactions that included at least one product that cost more than 2.00 is achieved by

select basket-id, product, price
from baskets, products
where baskets.product = products.product and price $>$ 2.00

If some tables in the "from" clause have common attributes, the attribute names must be prefixed by a dot and the name of the table when they appear in the "select" clause or "where" clause. If all attributes of participating tables should appear in the result, the list of attributes in the "select" clause can be replaced by a star.

Aggregation in database queries refers to the combination of several values into one, by the sum or maximum operators, for example. Relational algebra does not have operations for aggregation, but SQL does. An *aggregate* is in general a quantity computed from the database whose value depends on several rows of the database.

Example 12.5 The following queries show how aggregate queries relating to supermarket purchases can be described in SQL. First, we find for each product how many exemplars of it have been sold. To do this, we use the **group by** construct of SQL. This operation groups the rows of the input relation by the values of a certain attribute; the other operations in the SQL statement are performed separately for each clause.

```
select item, sum(quantity)
from baskets
group by item
```

The execution of this statement would proceed by first grouping the rows of the baskets relation according to the item attribute, and then for each group outputting the item name and the sum of the quantities for that group.

The next query finds the total sales for each product.

```
select item, sum(quantity)*price
from baskets, products
where item=product
group by item
```

Next we find total sales for each product belonging to soft drinks.

```
select item, sum(quantity)*price
from baskets, products, product-hierarchy
where item=product and products.product=product-hierarchy.product
      and class = "soft drink"
group by item
```

SQL was developed for traditional database applications such as generating reports and concurrent access and updating of transaction data by many users in real-time. Thus, it is not a big surprise that the language as such does not provide a very good platform for implementing data mining algorithms. There are two reasons for this: lack of suitable primitives and the need for efficiency.

Regarding the primitives, in SQL it is quite easy to do counting and aggregation. Therefore, for example, the operations needed for association rule algorithms are straightforward to implement by accessing the data using SQL. For building decision trees we need to be able to count the number of observations that fulfill the conditions occurring in the tree nodes from the root to the node in question. This is possible to do by selection and count queries. Where the primitives of SQL fail is in common statistical operations, such as matrix inversion, singular value decomposition (SVD), and so forth. Such operations would be extremely cumbersome to implement using SQL. This means that fitting complicated models is usually carried out outside the database system.

Even in cases when the SQL primitives are sufficient for expressing the operations in the data mining algorithm, there are reasons to implement the algorithm in a loosely-coupled manner, i.e., by downloading the relevant data to the algorithm. The reason is that the connection between a database management system and an application program typically enforces a large overhead for each query. Thus, while it is quite elegant to express the basic operations of association rule algorithms (for example) using SQL, such an implementation would typically be fairly slow. An additional cause for performance problems is that in association rule algorithms (for example) we must compute the frequency of a large number of candidate frequent sets. In a specialized implementation it is easy to do many of these counting operations in one pass through the data, whereas in an implementation based on using an SQL database management system, each candidate frequent set would cause a separate query to be issued.

12.8 Query Execution and Optimization

A query can be evaluated in various different ways. Consider, for example, the query

```
select t.product
from baskets t, baskets u
```

```
where t.transaction = u.transaction
      and u.product = "beer"
```

Here the notation baskets t, baskets u means that, in the query, t and u refer to rows of the baskets table. The notation is needed because we want to be able to refer to two different rows of the same table. The query finds all the products that have been bought in a transaction that also included beer.

The trivial method for evaluating such a query would be to try all possible pairs of rows from the baskets table, to check whether they agree on the basket-id attribute, and to test that the second row has "beer" in the product attribute. This would require n^2 operations on rows, where n is the size of the baskets table.

A more efficient method is to first locate the rows from the baskets table that have "beer" in the product attribute and sort the basket-ids of those rows into a list L. Then we can sort the baskets table using the basket-id attribute as the sort key and extract the products from the rows whose basket-id appears in the list L. Assuming that L is a relatively short list, this approach requires $O(n)$ operations for finding the rows with beer, $O(n \log n)$ operations for sorting the rows, and $O(n)$ operations for scanning the sorted list and selecting the correct values; i.e., altogether $O(n \log n)$ operations are needed. This is a clear improvement over the $O(n^2)$ operations needed for the naive method.

Query optimization is the task of finding the best possible evaluation method for a given query. Typically, query optimizers translate the SQL query into an expression tree, where the leaves represent tables and the internal nodes represent operations on the children of the nodes. Next, algebraic equalities between operations can be used to transform the tree into an equivalent form that is faster to evaluate. In the previous example, we have used the equation $\sigma_F(r \bowtie s) = \sigma_F(r) \bowtie s$, where F is a selection condition that concerns only the attributes of r. After a suitable expression tree is found, evaluation methods for each of the operations are selected. For example, a join operation can be evaluated in several different ways: by nested loops (as in the trivial method above), by sorting, or by using indices. The efficiency of each method depends on the sizes of the tables and the distribution of the values in the tables. Thus, query optimizers keep information about such changing quantities to find a good evaluation method. Theoretically, finding the best evaluation strategy for a given query is an NP-hard problem, so that finding the best method is not feasible. However, good query optimizers can be surprisingly effective.

Database management systems strive to provide good performance for a wide variety of queries. Thus, while for a single query it might be possible to write a program that computes the result more efficiently than a database management system would compute it, the strength of databases is that they provide fast execution for *most* of the queries. In data mining applications this is useful, as the queries are typically not known in advance (for example, in decision tree construction).

12.9 Data Warehousing and Online Analytical Processing (OLAP)

A retail database, with information about customers, transactions, products, prices, etc., is a typical example of an *operational database*: the database is used to conduct the daily operations of the organization, and the operations can rely quite heavily on it. Other examples of operational databases include airline reservation systems, bank account databases, etc. *Strategic databases* are databases that are used in decision making in the organization. The decision support viewpoint is quite closely aligned with the goal of data mining. Indeed one could say that a major goal of data mining is decision support.

Typically, an organization has several different operational databases. For example, a retail outlet might have a database about market baskets, a warehouse system, a customer database (or several), a payroll database, a database about suppliers, etc. Indeed, a diversified service company might even have several customer databases. Altogether, large organizations can have tens or hundreds of different operational databases. For decision support purposes one needs to combine information from various operational databases to find out overall patterns of activity within the company and with its customers. Building decision support applications that directly access the operational databases can be quite difficult.

Operational databases such as our hypothetical retail database, any customer database, or the reservation system of an airline, are most often used to answer well-defined and repetitive queries such as "What is the total price of the products in this basket," "What is the address of customer Smith," or "What is the balance of account 123456?" Such databases have to support a large number of transactions consisting of simple queries and updates on the contents of the data. This type of database usage is called *online transaction processing* (*OLTP*).

Decision support tasks require different types of queries: aggregation is

far more important. A typical decision support query might be "Find the sales of all products by region and by month, and the difference compared to last year." The term *online analytical processing* (*OLAP*) refers to the use of databases for obtaining summaries of the data, with aggregation as the principal mechanism.

Example 12.6 The tables of the database of the retailer could have the following form:

```
baskets(basket-id, item, quantity)
products(product, price, supplier, category)
product-hierarchy(product,category)
basket-stores(basket-id,store,day)
stores(store's name,city,country)
```

Here we have added the table basket-stores that tells in which store and on what date a certain basket was produced. For decision support purposes a more useful representation of the data might be using the table

```
sales(product,store,date,amount)
```

for representing the amount of a product sold at a given store on a given date. We can add rows to this table by SQL statements

```
insert into sales(product,store,date,amount)
select item, store, date, sum(quantity)*price
from baskets, basket-stores, products
where baskets.basket-id = basket-stores.basket-id and item = product
group by item, store, date
```

After this, we can find the total dollar sales of all product categories by countries by giving the following query:

```
select products.product, store.country, sum(amount)
from sales, stores, dates, products
where dates.year ≥ 1997
      and sales.product=products.product
      and sales.store=stores.store
      and sales.date=dates.date
group by products.category, store.country
```

OLTP and OLAP pose different requirements on the database management system. OLTP requires that the data are completely up to date, allows the queries to modify the database, allow several transactions to execute concurrently without interfering with each other, requires that responses be fast, and so forth. However, the OLTP queries and updates themselves are relatively simple. In contrast, in OLAP the queries can be quite complex, but normally only one of them executes at a given time. OLAP queries do not modify the data, and in finding out facts about last year's sales it is not crucial to have today's sale information. The requirements are so different that it makes sense to use different types of storage organizations for handling the two applications.

A *data warehouse* is a database system used to store information from various operational databases for decision support purposes. A data warehouse for a retailer might include information from a market basket database, a supplier database, customer databases, etc. The data in the payroll database might not be in the data warehouse if they are not considered to be crucial in decision support. A data warehouse is not created just by dumping the data from various databases to a single disk. Several integration tasks have to be carried out, such as resolving possible inconsistencies between attribute names and usages, finding out the semantics of attributes and values, and so on. Building data warehouses is often an expensive operation, as it requires much manual intervention and a detailed understanding of the operational databases.

The difference between OLTP, OLAP, and data mining is not always clear cut. We can in fact see a continuum of queries: find the address of a customer; find the sales of this product in the last month; find the sales of all products by region and month; find the trends in the sales; find what products have similar sales patterns; find rules that predict the sale of a certain product customer segmentation/clustering. The first query is typically carried out by using an OLTP query, the second is a typical OLAP query, and the last two might be called data mining queries. But it is difficult to define exactly where data mining starts and OLAP ends.

12.10 Data Structures for OLAP

OLAP requires the computation of various aggregates from large base tables. Since many aggregates will be needed over and over again, it makes sense to store some of them. The *data cube* is a clever technique for viewing the results

of various aggregations in a tabular way.

The previous example showed the sales table with the schema

sales(product,store,date,amount).

A possible row from this table might be

sales(red wine, store 1, August 25, 17.25),

indicating that the sales of red wine at store number 1 on August 25 were $17.25. Inventing a new value **all** to stand for any product, we might consider rows like

sales(**all**, store 1, August 25, 14214.70),

with the intended meaning that the total sales of all products in store 1 on August 25 were $14,214.70. In statistical terms, this gives us the marginal of the table, summing over values of the first attribute.

The *data cube* for the sales table contains all rows

sales(a, b, c, d),

where a, b, and c, are either values from the domains of the corresponding attributes or the specific value **all**, and d is the corresponding sum. That is, the *data cube* consists of the raw table and all marginal tables: the one-dimensional ones, the two-dimensional ones, and so on up to those obtained by summing over each attribute individually.

12.11 String Databases

Interest in text and string-oriented databases has increased dramatically in recent years. Molecular biology is one of the reasons: modern biotechnology generates huge amounts of protein and DNA data sets that are often recorded as strings. Even more important has been the rise of the Web: search engines require efficient methods for finding documents that include a given set of terms. Relational databases are fine for storing data in a tabular form, but they are not well suited for representing and accessing large volumes of text. Recently, several commercial database systems have added support for the efficient querying of large text data fields.

Given a large collection of text, a typical query might be "find all occurrences of the word *mining* in the text." More generally, the problem is to find occurrences of a pattern P in a text T. The pattern P might be a simple string,

a string with wildcards, or even a regular expression. The occurrence of P in T might be defined as an exact match or an approximate match, where errors are allowed.

The occurrences of the pattern P in text T can obviously be found by sequentially scanning the text and for each position testing whether P matches or not. Much more efficient solutions exist, however. For example, using the *suffix tree* data structure we can find the list of all occurrences of pattern p in time that is proportional to the length of p (and not dependent on the size of the text), and outputting the occurrences of p can be done in time $O(|p| + L)$, where L is the number of occurrences of p in the text. The suffix tree can be constructed in linear time in the size of the original text, and it is fast also in practice.

Schematically, a Web search engine might have two data structures: a relational table pages(page-address, page-text) and a suffix tree containing all the text of all the documents loaded into the system. When a user issues a query such as "find all documents containing the words *data* and *mining*," the suffix tree is used to find two lists pages: those containing the word *data* and those containing *mining*. Assuming the lists are sorted, it is straightforward to find the documents containing both words. Note, however, that the number of documents containing both *data* and *mining* is probably much less than the number containing one of the terms.

12.12 Massive Data Sets, Data Management, and Data Mining

So far in this chapter we have focused on database technology in a general sense. An important question remains as to how data mining and database technology interact. Our discussion of this interaction will be relatively brief, since there is no consensus to date among researchers and practitioners as to any "best" approach in terms of handling the interaction between data mining algorithms and database technology. At issue is the following: many massive data sets are either already stored in relational databases or could be more effectively managed and accessed during a data mining project if they were converted into relational database form. On the other hand, most data mining algorithms focus on the modeling and optimization aspects of the problem and effectively assume the data reside in a flat file in main memory. If the data to be mined are primarily on disk, and/or stored in a relational

format (perhaps with an SQL interface), how then should we approach the question of interfacing our data mining algorithm to the data?

This is the issue of *data management*, which, as we briefly discussed in chapter 5, is typically not addressed explicitly in most descriptions of data mining algorithms. And perhaps this is indeed the most flexible approach, since the solutions we adopt in practice will be a function of various application factors, such as the amount of data, the amount of available main memory, how often the algorithm will need to be rerun, and so forth. Nonetheless, we can identify a few general approaches to this problem, which we discuss below.

12.12.1 Force the Data into Main Memory

The most obvious approach, and one that practitioners have used for years, is to see whether the data can in fact be stored in main memory and (subsequently) accessed efficiently by the data mining algorithm. As main memory technology allows random access memory sizes to grow into the gigabyte range, this approach can be quite practical for many "medium-sized" data analysis applications. Of course there are other applications, e.g., those with hundreds of millions of complex transactions, where we cannot hope to ever load the data into main memory in the forseeable future. In such cases we can hope to subselect parts of the data, perhaps by generating a random sample of records so that we have n' transactions instead of n to deal with (where n' is much smaller than n).

We could also select subsets of features in some manner. For example, one of the authors worked on a predictive modeling application involving on the order of 1,000 variables and 200,000 customers. Decision trees were built on random samples of 5,000 customers, and the union of variables from the resulting trees was then used to build models (using trees, nonlinear regression, and other techniques) on the entire set of 200,000 records. This is of course an entirely heuristic procedure, and an important variable might have been omitted from the trees as a result of the multiple random sampling during model building. Nonetheless, this is a fairly typical example of the type of "data engineering" that is often required in practice to obtain meaningful results in a reasonable amount of time. Note also that generating a random sample from a relational database can itself be a nontrivial process. There are, of course, numerous refinements to the basic idea of random sampling, e.g., taking an initial small sample to get a general idea of the "data landscape," then further refining this sample in some automated manner, and so forth.

Of course even if the data fit in main memory, we still must be careful. It may well be that we have to subsample the data even further to get our data mining algorithm to run in reasonable time. Furthermore, naive implementations of algorithms may create large internal data structures when they run (e.g., unnecessary copies of data matrices), which in turn may cause available memory to be exceeded. Thus, it goes without saying that efficient implementation from a memory and time viewpoint is still important, even when the data all reside in main memory.

12.12.2 Scalable Versions of Data Mining Algorithms

The term *scalable* is somewhat loosely used in the data mining literature, but we can think of it as referring to data mining algorithms that scale gracefully and predictably (e.g., linearly) as the number of records n and/or the number of variables p grow. For example, naive implementation of a decision tree algorithm will exhibit a dramatic slowdown in run-time performance once n becomes large enough that the algorithm needs to frequently access data on disk. In practice, research on scalability focuses more on the large n problem than on the large p problem: large p is inherently more difficult than large n.

One line of investigation in scalable data mining algorithms is to develop special-purpose scalable implementations of existing well-known algorithms that are guaranteed to return the same result as the original (naive) implementation, but that typically will run much faster on large data sets. An example of this general approach is that of Gehrke et al. (1999), who propose a family of algorithms called BOAT (Bootstrapped Optimistic Algorithm for Tree Construction). The BOAT approach uses two scans through the entire data set. In the first scan an "optimistic tree" is constructed using a small random sample from the full data (and that can fit in main memory). The second scan then takes care of any differences between the initial tree and the tree that would have been built using all of the data. The resulting tree is then the same tree that the naive algorithm would have constructed (in a potentially inefficient manner). The method involves various clever data structures to keep track of tree-node statistics. Gehrke et al. (1999) report fitting classification trees to nine-dimensional synthetically generated data sets with 10 million data vectors in about 200 seconds.

A related strategy is to derive new approximate algorithms that inherently have desirable scaling performance by virtue of relying on various heuristics based on a relatively small number of linear scans of the data. These algorithms typically return "good" solutions but are not necessarily in agreement

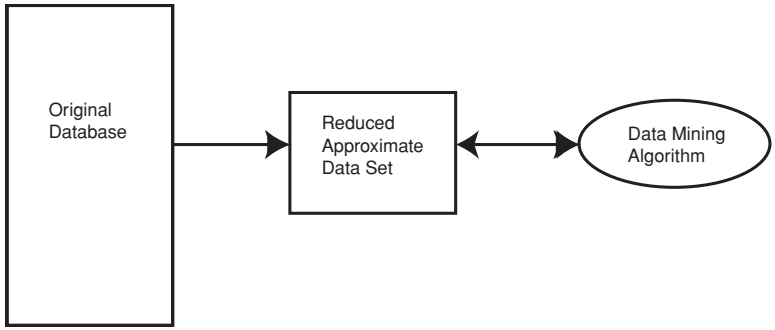

Figure 12.5 The concept of data mining algorithms which operate on an approximate version of the full data set.

with the original "nonscalable" version of the algorithm. For example, scalable clustering algorithms of this nature are described by Bradley, Fayyad, and Reina (1998) and Zhang, Ramakrishnan, and Livny (1997).

12.12.3 Special-Purpose Algorithms for Disk Access

Yet another approach to the problem of dealing with data on disk has been the development of new algorithms that are closely coupled with relational databases and transaction data. The best example in this context is that of association rule algorithms, which we have mentioned in chapter 5 and will discuss in more detail in chapter 13. The search component of association rule algorithms takes advantage of the typical sparsity of transaction data sets (i.e., most customers purchase relatively few items per transaction). At a high level, the algorithms typically involve breadth-first search strategies, where each level of the tree involves a single scan of the data that can be executed relatively easily. Agrawal et al. (1996) report results on synthetic data involving 1,000 items and up to 10 million transactions. They empirically demonstrate that the run-time of their algorithm scales up linearly on these data sets as a function of the number of transactions. Similar results have since been reported on a wide range of sparse transaction data sets and many variations of the basic algorithm have been developed (see chapter 13).

12.12.4 Pseudo Data Sets and Sufficient Statistics

Figure 12.5 illustrates another general idea that can be thought of as a generalization of random sampling. An approximate (and typically much smaller) data set is created that can then be accessed (e.g., in main memory) by the data mining algorithm instead of dealing with the full data (on disk). This general approach can, of course, only approximate the results we would have obtained had the algorithm been run on the full data. However, if the approximate data set is constructed in a clever enough manner, we can often get almost the same results on only a fraction of the data. It is often the case in practice that as part of the overall data mining process we will run our data mining algorithm many times, with different models, different variables, and so forth, in an exploratory manner, before finally settling on a final model. The use of an approximate data set for such exploratory modeling can be particularly useful (rather than having to deal with the full data set).

In this general context Du Mouchel et al. (1999) propose a statistically motivated methodology for "data-squashing" which amounts to creating a set of n' weighted "pseudo" data points, where n' is much smaller than the original number n, and where the pseudo data points are automatically chosen by the algorithm to mimic the statistical structure of the original larger data set. The general idea is to approximate the structure of the likelihood function as closely as possible, even without the functional form of the model being used in the data mining algorithm being specified. The method was empirically demonstrated to provide significant reduction in prediction error on a logistic regression problem compared to simple random sampling of a data set (Du Mouchel et al. (1999)).

On a related theme, for some data sets it may be sufficient simply to store the original data via a more efficient data structure than as a flat file or multiple tables in a relational database. The AD-Tree data structure proposed by Moore and Lee (1998) provides an efficient mechanism for storing multivariate categorical data (i.e., counts). Data mining algorithms can then quickly access counts and related statistics from the AD-Tree much more quickly than if the algorithm had to access the original data. Computational speedups of 50 to 5,000-fold on various classification algorithms (compared to naive implementation of the algorithms) have been reported (Moore (1999)).

In conclusion, we see that many different techniques can be used to implement data mining algorithms that are efficient in both time and space when we deal with very large data sets. Indeed there are several other approaches we have not even mentioned here, including the use of online algorithms that

see each data point only once (useful for applications where data are arriving rapidly in a continuous stream over time) and more hardware-oriented solutions such as parallel processing implementations of algorithms (in cases when both the algorithm and the data permit efficient parallel approaches). Choice of a particular technique often depends on quite practical aspects of the data mining application—i.e., how quickly must the data mining algorithm produce an answer? Does the model need to be continually updated? and so forth. Research on scalable data mining algorithms is likely to continue for some time, and we can expect more developments in this area. The reader should be cautioned to be aware that, as in everything else, there is no free lunch! In other words, there are typically trade-offs involving model accuracy, algorithm speed and memory, and so forth. Informed judgment on which type of algorithm and data structures best suit your problem will require careful consideration of both algorithmic issues and application details about how the algorithm and model will be used in practice.

12.13 Further Reading

There are several high-quality yearly database conferences, such as ACM's SIGMOD Conference on Management of Data (SIGMOD), and the SIGACT-SIGMOD-SIGART Symposium on Principles of Database and Knowledge-base Systems (PODS), the Very Large Database Conference (VLDB), and the International Conference on Data Engineering (ICDE).

There are several fine database textbooks, including Ullman (1988), Abiteboul, Hull, and Vianu (1995), and Ramakrishnan and Gehrke (1999). A recent survey of query optimization is Chaudhuri (1998). The data cube is presented in Gray et al. (1996) and Gray et al. (1997). A good introduction to OLAP is Chaudhuri and Dayal (1997). Implementation of database management systems is described in detail in Garcia-Molina et al. (1999). A nice discussion of OLAP and statistical databases is given by Shoshani (1997). Issues in using database management systems to implement mining algorithms are considered in Sarawagi et al. (2000) and Holsheimer et al. (1995).

Madigan et al. (in press) discuss various extensions of the the original squashing approach. Provost and Kolluri (1999) provide an overview of different techniques for scaling up data mining algorithms to handle very large data sets. Provost, Jensen, and Oates (1999), and Domingos and Hulten (2000) give examples of sampling problems with very large databases in data mining.