

3 Data Preprocessing

Today's real-world databases are highly susceptible to noisy, missing, and inconsistent data due to their typically huge size (often several gigabytes or more) and their likely origin from multiple, heterogeneous sources. Low-quality data will lead to low-quality mining results. *“How can the data be preprocessed in order to help improve the quality of the data and, consequently, of the mining results? How can the data be preprocessed so as to improve the efficiency and ease of the mining process?”*

There are several data preprocessing techniques. *Data cleaning* can be applied to remove noise and correct inconsistencies in data. *Data integration* merges data from multiple sources into a coherent data store such as a data warehouse. *Data reduction* can reduce data size by, for instance, aggregating, eliminating redundant features, or clustering. *Data transformations* (e.g., normalization) may be applied, where data are scaled to fall within a smaller range like 0.0 to 1.0. This can improve the accuracy and efficiency of mining algorithms involving distance measurements. These techniques are not mutually exclusive; they may work together. For example, data cleaning can involve transformations to correct wrong data, such as by transforming all entries for a *date* field to a common format.

In Chapter 2, we learned about the different attribute types and how to use basic statistical descriptions to study data characteristics. These can help identify erroneous values and outliers, which will be useful in the data cleaning and integration steps. Data processing techniques, when applied before mining, can substantially improve the overall quality of the patterns mined and/or the time required for the actual mining.

In this chapter, we introduce the basic concepts of data preprocessing in Section 3.1. The methods for data preprocessing are organized into the following categories: data cleaning (Section 3.2), data integration (Section 3.3), data reduction (Section 3.4), and data transformation (Section 3.5).

3.1 Data Preprocessing: An Overview

This section presents an overview of data preprocessing. Section 3.1.1 illustrates the many elements defining data quality. This provides the incentive behind data preprocessing. Section 3.1.2 outlines the major tasks in data preprocessing.

3.1.1 Data Quality: Why Preprocess the Data?

Data have quality if they satisfy the requirements of the intended use. There are many factors comprising **data quality**, including *accuracy*, *completeness*, *consistency*, *timeliness*, *believability*, and *interpretability*.

Imagine that you are a manager at *AllElectronics* and have been charged with analyzing the company's data with respect to your branch's sales. You immediately set out to perform this task. You carefully inspect the company's database and data warehouse, identifying and selecting the attributes or dimensions (e.g., *item*, *price*, and *units_sold*) to be included in your analysis. Alas! You notice that several of the attributes for various tuples have no recorded value. For your analysis, you would like to include information as to whether each item purchased was advertised as on sale, yet you discover that this information has not been recorded. Furthermore, users of your database system have reported errors, unusual values, and inconsistencies in the data recorded for some transactions. In other words, the data you wish to analyze by data mining techniques are *incomplete* (lacking attribute values or certain attributes of interest, or containing only aggregate data); *inaccurate* or *noisy* (containing errors, or values that deviate from the expected); and *inconsistent* (e.g., containing discrepancies in the department codes used to categorize items). Welcome to the real world!

This scenario illustrates three of the elements defining data quality: **accuracy**, **completeness**, and **consistency**. Inaccurate, incomplete, and inconsistent data are commonplace properties of large real-world databases and data warehouses. There are many possible reasons for inaccurate data (i.e., having incorrect attribute values). The data collection instruments used may be faulty. There may have been human or computer errors occurring at data entry. Users may purposely submit incorrect data values for mandatory fields when they do not wish to submit personal information (e.g., by choosing the default value "January 1" displayed for birthday). This is known as *disguised missing data*. Errors in data transmission can also occur. There may be technology limitations such as limited buffer size for coordinating synchronized data transfer and consumption. Incorrect data may also result from inconsistencies in naming conventions or data codes, or inconsistent formats for input fields (e.g., *date*). Duplicate tuples also require data cleaning.

Incomplete data can occur for a number of reasons. Attributes of interest may not always be available, such as customer information for sales transaction data. Other data may not be included simply because they were not considered important at the time of entry. Relevant data may not be recorded due to a misunderstanding or because of equipment malfunctions. Data that were inconsistent with other recorded data may

have been deleted. Furthermore, the recording of the data history or modifications may have been overlooked. Missing data, particularly for tuples with missing values for some attributes, may need to be inferred.

Recall that data quality depends on the intended use of the data. Two different users may have very different assessments of the quality of a given database. For example, a marketing analyst may need to access the database mentioned before for a list of customer addresses. Some of the addresses are outdated or incorrect, yet overall, 80% of the addresses are accurate. The marketing analyst considers this to be a large customer database for target marketing purposes and is pleased with the database's accuracy, although, as sales manager, you found the data inaccurate.

Timeliness also affects data quality. Suppose that you are overseeing the distribution of monthly sales bonuses to the top sales representatives at *AllElectronics*. Several sales representatives, however, fail to submit their sales records on time at the end of the month. There are also a number of corrections and adjustments that flow in after the month's end. For a period of time following each month, the data stored in the database are incomplete. However, once all of the data are received, it is correct. The fact that the month-end data are not updated in a timely fashion has a negative impact on the data quality.

Two other factors affecting data quality are believability and interpretability. **Believability** reflects how much the data are trusted by users, while **interpretability** reflects how easy the data are understood. Suppose that a database, at one point, had several errors, all of which have since been corrected. The past errors, however, had caused many problems for sales department users, and so they no longer trust the data. The data also use many accounting codes, which the sales department does not know how to interpret. Even though the database is now accurate, complete, consistent, and timely, sales department users may regard it as of low quality due to poor believability and interpretability.

3.1.2 Major Tasks in Data Preprocessing

In this section, we look at the major steps involved in data preprocessing, namely, data cleaning, data integration, data reduction, and data transformation.

Data cleaning routines work to “clean” the data by filling in missing values, smoothing noisy data, identifying or removing outliers, and resolving inconsistencies. If users believe the data are dirty, they are unlikely to trust the results of any data mining that has been applied. Furthermore, dirty data can cause confusion for the mining procedure, resulting in unreliable output. Although most mining routines have some procedures for dealing with incomplete or noisy data, they are not always robust. Instead, they may concentrate on avoiding overfitting the data to the function being modeled. Therefore, a useful preprocessing step is to run your data through some data cleaning routines. Section 3.2 discusses methods for data cleaning.

Getting back to your task at *AllElectronics*, suppose that you would like to include data from multiple sources in your analysis. This would involve integrating multiple databases, data cubes, or files (i.e., **data integration**). Yet some attributes representing a

given concept may have different names in different databases, causing inconsistencies and redundancies. For example, the attribute for customer identification may be referred to as *customer_id* in one data store and *cust_id* in another. Naming inconsistencies may also occur for attribute values. For example, the same first name could be registered as “Bill” in one database, “William” in another, and “B.” in a third. Furthermore, you suspect that some attributes may be inferred from others (e.g., annual revenue). Having a large amount of redundant data may slow down or confuse the knowledge discovery process. Clearly, in addition to data cleaning, steps must be taken to help avoid redundancies during data integration. Typically, data cleaning and data integration are performed as a preprocessing step when preparing data for a data warehouse. Additional data cleaning can be performed to detect and remove redundancies that may have resulted from data integration.

“Hmmm,” you wonder, as you consider your data even further. “*The data set I have selected for analysis is HUGE, which is sure to slow down the mining process. Is there a way I can reduce the size of my data set without jeopardizing the data mining results?*”

Data reduction obtains a reduced representation of the data set that is much smaller in volume, yet produces the same (or almost the same) analytical results. Data reduction strategies include *dimensionality reduction* and *numerosity reduction*.

In **dimensionality reduction**, data encoding schemes are applied so as to obtain a reduced or “compressed” representation of the original data. Examples include data compression techniques (e.g., *wavelet transforms* and *principal components analysis*), *attribute subset selection* (e.g., removing irrelevant attributes), and *attribute construction* (e.g., where a small set of more useful attributes is derived from the original set).

In **numerosity reduction**, the data are replaced by alternative, smaller representations using parametric models (e.g., *regression* or *log-linear models*) or nonparametric models (e.g., *histograms*, *clusters*, *sampling*, or *data aggregation*). Data reduction is the topic of Section 3.4.

Getting back to your data, you have decided, say, that you would like to use a distance-based mining algorithm for your analysis, such as neural networks, nearest-neighbor classifiers, or clustering.¹ Such methods provide better results if the data to be analyzed have been *normalized*, that is, scaled to a smaller range such as [0.0, 1.0]. Your customer data, for example, contain the attributes *age* and *annual salary*. The *annual salary* attribute usually takes much larger values than *age*. Therefore, if the attributes are left unnormalized, the distance measurements taken on *annual salary* will generally outweigh distance measurements taken on *age*. *Discretization* and *concept hierarchy generation* can also be useful, where raw data values for attributes are replaced by ranges or higher conceptual levels. For example, raw values for *age* may be replaced by higher-level concepts, such as *youth*, *adult*, or *senior*.

Discretization and concept hierarchy generation are powerful tools for data mining in that they allow data mining at multiple abstraction levels. Normalization, data

¹Neural networks and nearest-neighbor classifiers are described in Chapter 9, and clustering is discussed in Chapters 10 and 11.

discretization, and concept hierarchy generation are forms of **data transformation**. You soon realize such data transformation operations are additional data preprocessing procedures that would contribute toward the success of the mining process. Data integration and data discretization are discussed in Sections 3.5.

Figure 3.1 summarizes the data preprocessing steps described here. Note that the previous categorization is not mutually exclusive. For example, the removal of redundant data may be seen as a form of data cleaning, as well as data reduction.

In summary, real-world data tend to be dirty, incomplete, and inconsistent. Data preprocessing techniques can improve data quality, thereby helping to improve the accuracy and efficiency of the subsequent mining process. Data preprocessing is an important step in the knowledge discovery process, because quality decisions must be based on quality data. Detecting data anomalies, rectifying them early, and reducing the data to be analyzed can lead to huge payoffs for decision making.

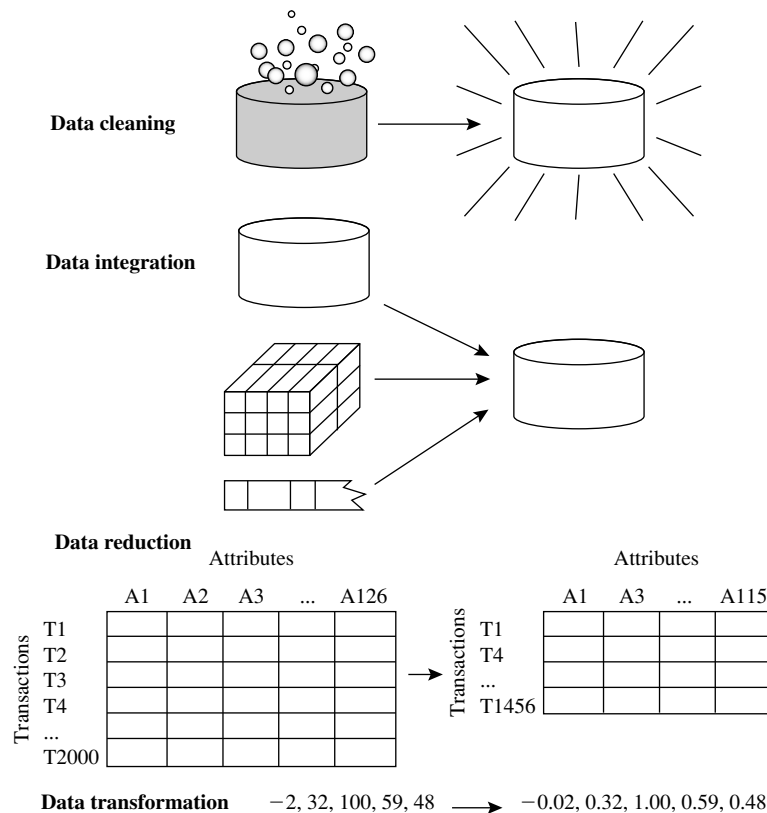


Figure 3.1 Forms of data preprocessing.

3.2 Data Cleaning

Real-world data tend to be incomplete, noisy, and inconsistent. *Data cleaning* (or *data cleansing*) routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data. In this section, you will study basic methods for data cleaning. Section 3.2.1 looks at ways of handling missing values. Section 3.2.2 explains data smoothing techniques. Section 3.2.3 discusses approaches to data cleaning as a process.

3.2.1 Missing Values

Imagine that you need to analyze *AllElectronics* sales and customer data. You note that many tuples have no recorded value for several attributes such as customer *income*. How can you go about filling in the missing values for this attribute? Let's look at the following methods.

1. **Ignore the tuple:** This is usually done when the class label is missing (assuming the mining task involves classification). This method is not very effective, unless the tuple contains several attributes with missing values. It is especially poor when the percentage of missing values per attribute varies considerably. By ignoring the tuple, we do not make use of the remaining attributes' values in the tuple. Such data could have been useful to the task at hand.
2. **Fill in the missing value manually:** In general, this approach is time consuming and may not be feasible given a large data set with many missing values.
3. **Use a global constant to fill in the missing value:** Replace all missing attribute values by the same constant such as a label like "*Unknown*" or $-\infty$. If missing values are replaced by, say, "*Unknown*," then the mining program may mistakenly think that they form an interesting concept, since they all have a value in common—that of "*Unknown*." Hence, although this method is simple, it is not foolproof.
4. **Use a measure of central tendency for the attribute (e.g., the mean or median) to fill in the missing value:** Chapter 2 discussed measures of central tendency, which indicate the "middle" value of a data distribution. For normal (symmetric) data distributions, the mean can be used, while skewed data distribution should employ the median (Section 2.2). For example, suppose that the data distribution regarding the income of *AllElectronics* customers is symmetric and that the mean income is \$56,000. Use this value to replace the missing value for *income*.
5. **Use the attribute mean or median for all samples belonging to the same class as the given tuple:** For example, if classifying customers according to *credit_risk*, we may replace the missing value with the mean *income* value for customers in the same credit risk category as that of the given tuple. If the data distribution for a given class is skewed, the median value is a better choice.
6. **Use the most probable value to fill in the missing value:** This may be determined with regression, inference-based tools using a Bayesian formalism, or decision tree

induction. For example, using the other customer attributes in your data set, you may construct a decision tree to predict the missing values for *income*. Decision trees and Bayesian inference are described in detail in Chapters 8 and 9, respectively, while regression is introduced in Section 3.4.5.

Methods 3 through 6 bias the data—the filled-in value may not be correct. Method 6, however, is a popular strategy. In comparison to the other methods, it uses the most information from the present data to predict missing values. By considering the other attributes' values in its estimation of the missing value for *income*, there is a greater chance that the relationships between *income* and the other attributes are preserved.

It is important to note that, in some cases, a missing value may not imply an error in the data! For example, when applying for a credit card, candidates may be asked to supply their driver's license number. Candidates who do not have a driver's license may naturally leave this field blank. Forms should allow respondents to specify values such as “not applicable.” Software routines may also be used to uncover other null values (e.g., “don't know,” “?” or “none”). Ideally, each attribute should have one or more rules regarding the *null* condition. The rules may specify whether or not nulls are allowed and/or how such values should be handled or transformed. Fields may also be intentionally left blank if they are to be provided in a later step of the business process. Hence, although we can try our best to clean the data after it is seized, good database and data entry procedure design should help minimize the number of missing values or errors in the first place.

3.2.2 Noisy Data

“*What is noise?*” **Noise** is a random error or variance in a measured variable. In Chapter 2, we saw how some basic statistical description techniques (e.g., boxplots and scatter plots), and methods of data visualization can be used to identify outliers, which may represent noise. Given a numeric attribute such as, say, *price*, how can we “smooth” out the data to remove the noise? Let's look at the following data smoothing techniques.

Binning: Binning methods smooth a sorted data value by consulting its “neighborhood,” that is, the values around it. The sorted values are distributed into a number of “buckets,” or *bins*. Because binning methods consult the neighborhood of values, they perform *local* smoothing. Figure 3.2 illustrates some binning techniques. In this example, the data for *price* are first sorted and then partitioned into *equal-frequency* bins of size 3 (i.e., each bin contains three values). In **smoothing by bin means**, each value in a bin is replaced by the mean value of the bin. For example, the mean of the values 4, 8, and 15 in Bin 1 is 9. Therefore, each original value in this bin is replaced by the value 9.

Similarly, **smoothing by bin medians** can be employed, in which each bin value is replaced by the bin median. In **smoothing by bin boundaries**, the minimum and maximum values in a given bin are identified as the *bin boundaries*. Each bin value is then replaced by the closest boundary value. In general, the larger the width, the

Sorted data for *price* (in dollars): 4, 8, 15, 21, 21, 24, 25, 28, 34

Partition into (equal-frequency) bins:

Bin 1: 4, 8, 15
 Bin 2: 21, 21, 24
 Bin 3: 25, 28, 34

Smoothing by bin means:

Bin 1: 9, 9, 9
 Bin 2: 22, 22, 22
 Bin 3: 29, 29, 29

Smoothing by bin boundaries:

Bin 1: 4, 4, 15
 Bin 2: 21, 21, 24
 Bin 3: 25, 25, 34

Figure 3.2 Binning methods for data smoothing.

greater the effect of the smoothing. Alternatively, bins may be *equal width*, where the interval range of values in each bin is constant. Binning is also used as a discretization technique and is further discussed in Section 3.5.

Regression: Data smoothing can also be done by regression, a technique that conforms data values to a function. *Linear regression* involves finding the “best” line to fit two attributes (or variables) so that one attribute can be used to predict the other. *Multiple linear regression* is an extension of linear regression, where more than two attributes are involved and the data are fit to a multidimensional surface. Regression is further described in Section 3.4.5.

Outlier analysis: Outliers may be detected by clustering, for example, where similar values are organized into groups, or “clusters.” Intuitively, values that fall outside of the set of clusters may be considered outliers (Figure 3.3). Chapter 12 is dedicated to the topic of outlier analysis.

Many data smoothing methods are also used for data discretization (a form of data transformation) and data reduction. For example, the binning techniques described before reduce the number of distinct values per attribute. This acts as a form of data reduction for logic-based data mining methods, such as decision tree induction, which repeatedly makes value comparisons on sorted data. Concept hierarchies are a form of data discretization that can also be used for data smoothing. A concept hierarchy for *price*, for example, may map real *price* values into *inexpensive*, *moderately-priced*, and *expensive*, thereby reducing the number of data values to be handled by the mining

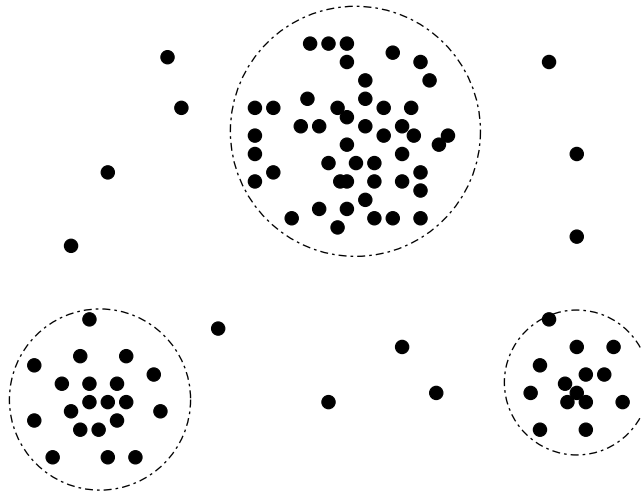


Figure 3.3 A 2-D customer data plot with respect to customer locations in a city, showing three data clusters. Outliers may be detected as values that fall outside of the cluster sets.

process. Data discretization is discussed in Section 3.5. Some methods of classification (e.g., neural networks) have built-in data smoothing mechanisms. Classification is the topic of Chapters 8 and 9.

3.2.3 Data Cleaning as a Process

Missing values, noise, and inconsistencies contribute to inaccurate data. So far, we have looked at techniques for handling missing data and for smoothing data. *“But data cleaning is a big job. What about data cleaning as a process? How exactly does one proceed in tackling this task? Are there any tools out there to help?”*

The first step in data cleaning as a process is *discrepancy detection*. Discrepancies can be caused by several factors, including poorly designed data entry forms that have many optional fields, human error in data entry, deliberate errors (e.g., respondents not wanting to divulge information about themselves), and data decay (e.g., outdated addresses). Discrepancies may also arise from inconsistent data representations and inconsistent use of codes. Other sources of discrepancies include errors in instrumentation devices that record data and system errors. Errors can also occur when the data are (inadequately) used for purposes other than originally intended. There may also be inconsistencies due to data integration (e.g., where a given attribute can have different names in different databases).²

²Data integration and the removal of redundant data that can result from such integration are further described in Section 3.3.

“So, how can we proceed with discrepancy detection?” As a starting point, use any knowledge you may already have regarding properties of the data. Such knowledge or “data about data” is referred to as **metadata**. This is where we can make use of the knowledge we gained about our data in Chapter 2. For example, what are the data type and domain of each attribute? What are the acceptable values for each attribute? The basic statistical data descriptions discussed in Section 2.2 are useful here to grasp data trends and identify anomalies. For example, find the mean, median, and mode values. Are the data symmetric or skewed? What is the range of values? Do all values fall within the expected range? What is the standard deviation of each attribute? Values that are more than two standard deviations away from the mean for a given attribute may be flagged as potential outliers. Are there any known dependencies between attributes? In this step, you may write your own scripts and/or use some of the tools that we discuss further later. From this, you may find noise, outliers, and unusual values that need investigation.

As a data analyst, you should be on the lookout for the inconsistent use of codes and any inconsistent data representations (e.g., “2010/12/25” and “25/12/2010” for *date*). **Field overloading** is another error source that typically results when developers squeeze new attribute definitions into unused (bit) portions of already defined attributes (e.g., an unused bit of an attribute that has a value range that uses only, say, 31 out of 32 bits).

The data should also be examined regarding unique rules, consecutive rules, and null rules. A **unique rule** says that each value of the given attribute must be different from all other values for that attribute. A **consecutive rule** says that there can be no missing values between the lowest and highest values for the attribute, and that all values must also be unique (e.g., as in check numbers). A **null rule** specifies the use of blanks, question marks, special characters, or other strings that may indicate the null condition (e.g., where a value for a given attribute is not available), and how such values should be handled. As mentioned in Section 3.2.1, reasons for missing values may include (1) the person originally asked to provide a value for the attribute refuses and/or finds that the information requested is not applicable (e.g., a *license.number* attribute left blank by nondrivers); (2) the data entry person does not know the correct value; or (3) the value is to be provided by a later step of the process. The null rule should specify how to record the null condition, for example, such as to store zero for numeric attributes, a blank for character attributes, or any other conventions that may be in use (e.g., entries like “don’t know” or “?” should be transformed to blank).

There are a number of different commercial tools that can aid in the discrepancy detection step. **Data scrubbing tools** use simple domain knowledge (e.g., knowledge of postal addresses and spell-checking) to detect errors and make corrections in the data. These tools rely on parsing and fuzzy matching techniques when cleaning data from multiple sources. **Data auditing tools** find discrepancies by analyzing the data to discover rules and relationships, and detecting data that violate such conditions. They are variants of data mining tools. For example, they may employ statistical analysis to find correlations, or clustering to identify outliers. They may also use the basic statistical data descriptions presented in Section 2.2.

Some data inconsistencies may be corrected manually using external references. For example, errors made at data entry may be corrected by performing a paper

trace. Most errors, however, will require *data transformations*. That is, once we find discrepancies, we typically need to define and apply (a series of) transformations to correct them.

Commercial tools can assist in the data transformation step. **Data migration tools** allow simple transformations to be specified such as to replace the string “gender” by “sex.” **ETL (extraction/transformation/loading) tools** allow users to specify transforms through a graphical user interface (GUI). These tools typically support only a restricted set of transforms so that, often, we may also choose to write custom scripts for this step of the data cleaning process.

The two-step process of discrepancy detection and data transformation (to correct discrepancies) iterates. This process, however, is error-prone and time consuming. Some transformations may introduce more discrepancies. Some *nested discrepancies* may only be detected after others have been fixed. For example, a typo such as “20010” in a year field may only surface once all date values have been converted to a uniform format. Transformations are often done as a batch process while the user waits without feedback. Only after the transformation is complete can the user go back and check that no new anomalies have been mistakenly created. Typically, numerous iterations are required before the user is satisfied. Any tuples that cannot be automatically handled by a given transformation are typically written to a file without any explanation regarding the reasoning behind their failure. As a result, the entire data cleaning process also suffers from a lack of interactivity.

New approaches to data cleaning emphasize increased interactivity. Potter’s Wheel, for example, is a publicly available data cleaning tool that integrates discrepancy detection and transformation. Users gradually build a series of transformations by composing and debugging individual transformations, one step at a time, on a spreadsheet-like interface. The transformations can be specified graphically or by providing examples. Results are shown immediately on the records that are visible on the screen. The user can choose to undo the transformations, so that transformations that introduced additional errors can be “erased.” The tool automatically performs discrepancy checking in the background on the latest transformed view of the data. Users can gradually develop and refine transformations as discrepancies are found, leading to more effective and efficient data cleaning.

Another approach to increased interactivity in data cleaning is the development of declarative languages for the specification of data transformation operators. Such work focuses on defining powerful extensions to SQL and algorithms that enable users to express data cleaning specifications efficiently.

As we discover more about the data, it is important to keep updating the metadata to reflect this knowledge. This will help speed up data cleaning on future versions of the same data store.

3.3 Data Integration

Data mining often requires data integration—the merging of data from multiple data stores. Careful integration can help reduce and avoid redundancies and inconsistencies

in the resulting data set. This can help improve the accuracy and speed of the subsequent data mining process.

The semantic heterogeneity and structure of data pose great challenges in data integration. How can we match schema and objects from different sources? This is the essence of the *entity identification problem*, described in Section 3.3.1. Are any attributes correlated? Section 3.3.2 presents correlation tests for numeric and nominal data. Tuple duplication is described in Section 3.3.3. Finally, Section 3.3.4 touches on the detection and resolution of data value conflicts.

3.3.1 Entity Identification Problem

It is likely that your data analysis task will involve *data integration*, which combines data from multiple sources into a coherent data store, as in data warehousing. These sources may include multiple databases, data cubes, or flat files.

There are a number of issues to consider during data integration. *Schema integration* and *object matching* can be tricky. How can equivalent real-world entities from multiple data sources be matched up? This is referred to as the **entity identification problem**. For example, how can the data analyst or the computer be sure that *customer_id* in one database and *cust_number* in another refer to the same attribute? Examples of metadata for each attribute include the name, meaning, data type, and range of values permitted for the attribute, and null rules for handling blank, zero, or null values (Section 3.2). Such metadata can be used to help avoid errors in schema integration. The metadata may also be used to help transform the data (e.g., where data codes for *pay_type* in one database may be “H” and “S” but 1 and 2 in another). Hence, this step also relates to data cleaning, as described earlier.

When matching attributes from one database to another during integration, special attention must be paid to the *structure* of the data. This is to ensure that any attribute functional dependencies and referential constraints in the source system match those in the target system. For example, in one system, a *discount* may be applied to the order, whereas in another system it is applied to each individual line item within the order. If this is not caught before integration, items in the target system may be improperly discounted.

3.3.2 Redundancy and Correlation Analysis

Redundancy is another important issue in data integration. An attribute (such as *annual revenue*, for instance) may be redundant if it can be “derived” from another attribute or set of attributes. Inconsistencies in attribute or dimension naming can also cause redundancies in the resulting data set.

Some redundancies can be detected by **correlation analysis**. Given two attributes, such analysis can measure how strongly one attribute implies the other, based on the available data. For nominal data, we use the χ^2 (*chi-square*) test. For numeric attributes, we can use the *correlation coefficient* and *covariance*, both of which access how one attribute’s values vary from those of another.

χ^2 Correlation Test for Nominal Data

For nominal data, a correlation relationship between two attributes, A and B , can be discovered by a χ^2 (**chi-square**) test. Suppose A has c distinct values, namely a_1, a_2, \dots, a_c . B has r distinct values, namely b_1, b_2, \dots, b_r . The data tuples described by A and B can be shown as a **contingency table**, with the c values of A making up the columns and the r values of B making up the rows. Let (A_i, B_j) denote the joint event that attribute A takes on value a_i and attribute B takes on value b_j , that is, where $(A = a_i, B = b_j)$. Each and every possible (A_i, B_j) joint event has its own cell (or slot) in the table. The χ^2 value (also known as the *Pearson χ^2 statistic*) is computed as

$$\chi^2 = \sum_{i=1}^c \sum_{j=1}^r \frac{(o_{ij} - e_{ij})^2}{e_{ij}}, \quad (3.1)$$

where o_{ij} is the *observed frequency* (i.e., actual count) of the joint event (A_i, B_j) and e_{ij} is the *expected frequency* of (A_i, B_j) , which can be computed as

$$e_{ij} = \frac{\text{count}(A = a_i) \times \text{count}(B = b_j)}{n}, \quad (3.2)$$

where n is the number of data tuples, $\text{count}(A = a_i)$ is the number of tuples having value a_i for A , and $\text{count}(B = b_j)$ is the number of tuples having value b_j for B . The sum in Eq. (3.1) is computed over all of the $r \times c$ cells. Note that the cells that contribute the most to the χ^2 value are those for which the actual count is very different from that expected.

The χ^2 statistic tests the hypothesis that A and B are *independent*, that is, there is no correlation between them. The test is based on a significance level, with $(r - 1) \times (c - 1)$ degrees of freedom. We illustrate the use of this statistic in Example 3.1. If the hypothesis can be rejected, then we say that A and B are statistically correlated.

Example 3.1 Correlation analysis of nominal attributes using χ^2 . Suppose that a group of 1500 people was surveyed. The gender of each person was noted. Each person was polled as to whether his or her preferred type of reading material was fiction or nonfiction. Thus, we have two attributes, *gender* and *preferred_reading*. The observed frequency (or count) of each possible joint event is summarized in the contingency table shown in Table 3.1, where the numbers in parentheses are the expected frequencies. The expected frequencies are calculated based on the data distribution for both attributes using Eq. (3.2).

Using Eq. (3.2), we can verify the expected frequencies for each cell. For example, the expected frequency for the cell (*male, fiction*) is

$$e_{11} = \frac{\text{count}(\text{male}) \times \text{count}(\text{fiction})}{n} = \frac{300 \times 450}{1500} = 90,$$

and so on. Notice that in any row, the sum of the expected frequencies must equal the total observed frequency for that row, and the sum of the expected frequencies in any column must also equal the total observed frequency for that column.

Table 3.1 Example 2.1's 2×2 Contingency Table Data

	<i>male</i>	<i>female</i>	<i>Total</i>
<i>fiction</i>	250 (90)	200 (360)	450
<i>non_fiction</i>	50 (210)	1000 (840)	1050
Total	300	1200	1500

Note: Are *gender* and *preferred_reading* correlated?

Using Eq. (3.1) for χ^2 computation, we get

$$\begin{aligned}\chi^2 &= \frac{(250 - 90)^2}{90} + \frac{(50 - 210)^2}{210} + \frac{(200 - 360)^2}{360} + \frac{(1000 - 840)^2}{840} \\ &= 284.44 + 121.90 + 71.11 + 30.48 = 507.93.\end{aligned}$$

For this 2×2 table, the degrees of freedom are $(2 - 1)(2 - 1) = 1$. For 1 degree of freedom, the χ^2 value needed to reject the hypothesis at the 0.001 significance level is 10.828 (taken from the table of upper percentage points of the χ^2 distribution, typically available from any textbook on statistics). Since our computed value is above this, we can reject the hypothesis that *gender* and *preferred_reading* are independent and conclude that the two attributes are (strongly) correlated for the given group of people. ■

Correlation Coefficient for Numeric Data

For numeric attributes, we can evaluate the correlation between two attributes, *A* and *B*, by computing the **correlation coefficient** (also known as **Pearson's product moment coefficient**, named after its inventor, Karl Pearson). This is

$$r_{A,B} = \frac{\sum_{i=1}^n (a_i - \bar{A})(b_i - \bar{B})}{n\sigma_A\sigma_B} = \frac{\sum_{i=1}^n (a_i b_i) - n\bar{A}\bar{B}}{n\sigma_A\sigma_B}, \quad (3.3)$$

where n is the number of tuples, a_i and b_i are the respective values of *A* and *B* in tuple i , \bar{A} and \bar{B} are the respective mean values of *A* and *B*, σ_A and σ_B are the respective standard deviations of *A* and *B* (as defined in Section 2.2.2), and $\sum (a_i b_i)$ is the sum of the *AB* cross-product (i.e., for each tuple, the value for *A* is multiplied by the value for *B* in that tuple). Note that $-1 \leq r_{A,B} \leq +1$. If $r_{A,B}$ is greater than 0, then *A* and *B* are *positively correlated*, meaning that the values of *A* increase as the values of *B* increase. The higher the value, the stronger the correlation (i.e., the more each attribute implies the other). Hence, a higher value may indicate that *A* (or *B*) may be removed as a redundancy.

If the resulting value is equal to 0, then *A* and *B* are *independent* and there is no correlation between them. If the resulting value is less than 0, then *A* and *B* are *negatively correlated*, where the values of one attribute increase as the values of the other attribute decrease. This means that each attribute discourages the other. Scatter plots can also be used to view correlations between attributes (Section 2.2.3). For example, Figure 2.8's

scatter plots respectively show positively correlated data and negatively correlated data, while Figure 2.9 displays uncorrelated data.

Note that correlation does not imply causality. That is, if A and B are correlated, this does not necessarily imply that A causes B or that B causes A . For example, in analyzing a demographic database, we may find that attributes representing the number of hospitals and the number of car thefts in a region are correlated. This does not mean that one causes the other. Both are actually causally linked to a third attribute, namely, *population*.

Covariance of Numeric Data

In probability theory and statistics, correlation and covariance are two similar measures for assessing how much two attributes change together. Consider two numeric attributes A and B , and a set of n observations $\{(a_1, b_1), \dots, (a_n, b_n)\}$. The mean values of A and B , respectively, are also known as the **expected values** on A and B , that is,

$$E(A) = \bar{A} = \frac{\sum_{i=1}^n a_i}{n}$$

and

$$E(B) = \bar{B} = \frac{\sum_{i=1}^n b_i}{n}.$$

The **covariance** between A and B is defined as

$$\text{Cov}(A, B) = E((A - \bar{A})(B - \bar{B})) = \frac{\sum_{i=1}^n (a_i - \bar{A})(b_i - \bar{B})}{n}. \quad (3.4)$$

If we compare Eq. (3.3) for $r_{A,B}$ (correlation coefficient) with Eq. (3.4) for covariance, we see that

$$r_{A,B} = \frac{\text{Cov}(A, B)}{\sigma_A \sigma_B}, \quad (3.5)$$

where σ_A and σ_B are the standard deviations of A and B , respectively. It can also be shown that

$$\text{Cov}(A, B) = E(A \cdot B) - \bar{A}\bar{B}. \quad (3.6)$$

This equation may simplify calculations.

For two attributes A and B that tend to change together, if A is larger than \bar{A} (the expected value of A), then B is likely to be larger than \bar{B} (the expected value of B). Therefore, the covariance between A and B is *positive*. On the other hand, if one of the attributes tends to be above its expected value when the other attribute is below its expected value, then the covariance of A and B is *negative*.

If A and B are *independent* (i.e., they do not have correlation), then $E(A \cdot B) = E(A) \cdot E(B)$. Therefore, the covariance is $\text{Cov}(A, B) = E(A \cdot B) - \bar{A}\bar{B} = E(A) \cdot E(B) - \bar{A}\bar{B} = 0$. However, the converse is not true. Some pairs of random variables (attributes) may have a covariance of 0 but are not independent. Only under some additional assumptions

Table 3.2 Stock Prices for *AllElectronics* and *HighTech*

Time point	<i>AllElectronics</i>	<i>HighTech</i>
t1	6	20
t2	5	10
t3	4	14
t4	3	5
t5	2	5

(e.g., the data follow multivariate normal distributions) does a covariance of 0 imply independence.

Example 3.2 Covariance analysis of numeric attributes. Consider Table 3.2, which presents a simplified example of stock prices observed at five time points for *AllElectronics* and *HighTech*, a high-tech company. If the stocks are affected by the same industry trends, will their prices rise or fall together?

$$E(\text{AllElectronics}) = \frac{6 + 5 + 4 + 3 + 2}{5} = \frac{20}{5} = \$4$$

and

$$E(\text{HighTech}) = \frac{20 + 10 + 14 + 5 + 5}{5} = \frac{54}{5} = \$10.80.$$

Thus, using Eq. (3.4), we compute

$$\begin{aligned} \text{Cov}(\text{AllElectronics}, \text{HighTech}) &= \frac{6 \times 20 + 5 \times 10 + 4 \times 14 + 3 \times 5 + 2 \times 5}{5} - 4 \times 10.80 \\ &= 50.2 - 43.2 = 7. \end{aligned}$$

Therefore, given the positive covariance we can say that stock prices for both companies rise together. ■

Variance is a special case of covariance, where the two attributes are identical (i.e., the covariance of an attribute with itself). Variance was discussed in Chapter 2.

3.3.3 Tuple Duplication

In addition to detecting redundancies between attributes, duplication should also be detected at the tuple level (e.g., where there are two or more identical tuples for a given unique data entry case). The use of denormalized tables (often done to improve performance by avoiding joins) is another source of data redundancy. Inconsistencies often arise between various duplicates, due to inaccurate data entry or updating some but not all data occurrences. For example, if a purchase order database contains attributes for

the purchaser's name and address instead of a key to this information in a purchaser database, discrepancies can occur, such as the same purchaser's name appearing with different addresses within the purchase order database.

3.3.4 Data Value Conflict Detection and Resolution

Data integration also involves the *detection and resolution of data value conflicts*. For example, for the same real-world entity, attribute values from different sources may differ. This may be due to differences in representation, scaling, or encoding. For instance, a *weight* attribute may be stored in metric units in one system and British imperial units in another. For a hotel chain, the *price* of rooms in different cities may involve not only different currencies but also different services (e.g., free breakfast) and taxes. When exchanging information between schools, for example, each school may have its own curriculum and grading scheme. One university may adopt a quarter system, offer three courses on database systems, and assign grades from A+ to F, whereas another may adopt a semester system, offer two courses on databases, and assign grades from 1 to 10. It is difficult to work out precise course-to-grade transformation rules between the two universities, making information exchange difficult.

Attributes may also differ on the abstraction level, where an attribute in one system is recorded at, say, a lower abstraction level than the “same” attribute in another. For example, the *total_sales* in one database may refer to one branch of *All_Electronics*, while an attribute of the same name in another database may refer to the total sales for *All_Electronics* stores in a given region. The topic of discrepancy detection is further described in Section 3.2.3 on data cleaning as a process.

3.4 Data Reduction

Imagine that you have selected data from the *AllElectronics* data warehouse for analysis. The data set will likely be huge! Complex data analysis and mining on huge amounts of data can take a long time, making such analysis impractical or infeasible.

Data reduction techniques can be applied to obtain a reduced representation of the data set that is much smaller in volume, yet closely maintains the integrity of the original data. That is, mining on the reduced data set should be more efficient yet produce the same (or almost the same) analytical results. In this section, we first present an overview of data reduction strategies, followed by a closer look at individual techniques.

3.4.1 Overview of Data Reduction Strategies

Data reduction strategies include *dimensionality reduction*, *numerosity reduction*, and *data compression*.

Dimensionality reduction is the process of reducing the number of random variables or attributes under consideration. Dimensionality reduction methods include *wavelet*

transforms (Section 3.4.2) and *principal components analysis* (Section 3.4.3), which transform or project the original data onto a smaller space. *Attribute subset selection* is a method of dimensionality reduction in which irrelevant, weakly relevant, or redundant attributes or dimensions are detected and removed (Section 3.4.4).

Numerosity reduction techniques replace the original data volume by alternative, smaller forms of data representation. These techniques may be parametric or non-parametric. For *parametric methods*, a model is used to estimate the data, so that typically only the data parameters need to be stored, instead of the actual data. (Outliers may also be stored.) Regression and log-linear models (Section 3.4.5) are examples. *Nonparametric methods* for storing reduced representations of the data include *histograms* (Section 3.4.6), *clustering* (Section 3.4.7), *sampling* (Section 3.4.8), and *data cube aggregation* (Section 3.4.9).

In **data compression**, transformations are applied so as to obtain a reduced or “compressed” representation of the original data. If the original data can be *reconstructed* from the compressed data without any information loss, the data reduction is called **lossless**. If, instead, we can reconstruct only an approximation of the original data, then the data reduction is called **lossy**. There are several lossless algorithms for string compression; however, they typically allow only limited data manipulation. Dimensionality reduction and numerosity reduction techniques can also be considered forms of data compression.

There are many other ways of organizing methods of data reduction. The computational time spent on data reduction should not outweigh or “erase” the time saved by mining on a reduced data set size.

3.4.2 Wavelet Transforms

The **discrete wavelet transform (DWT)** is a linear signal processing technique that, when applied to a data vector \mathbf{X} , transforms it to a numerically different vector, \mathbf{X}' , of **wavelet coefficients**. The two vectors are of the same length. When applying this technique to data reduction, we consider each tuple as an n -dimensional data vector, that is, $\mathbf{X} = (x_1, x_2, \dots, x_n)$, depicting n measurements made on the tuple from n database attributes.³

“How can this technique be useful for data reduction if the wavelet transformed data are of the same length as the original data?” The usefulness lies in the fact that the wavelet transformed data can be truncated. A compressed approximation of the data can be retained by storing only a small fraction of the strongest of the wavelet coefficients. For example, all wavelet coefficients larger than some user-specified threshold can be retained. All other coefficients are set to 0. The resulting data representation is therefore very sparse, so that operations that can take advantage of data sparsity are computationally very fast if performed in wavelet space. The technique also works to remove noise without smoothing out the main features of the data, making it effective for data

³In our notation, any variable representing a vector is shown in bold italic font; measurements depicting the vector are shown in italic font.

cleaning as well. Given a set of coefficients, an approximation of the original data can be constructed by applying the *inverse* of the DWT used.

The DWT is closely related to the *discrete Fourier transform (DFT)*, a signal processing technique involving sines and cosines. In general, however, the DWT achieves better lossy compression. That is, if the same number of coefficients is retained for a DWT and a DFT of a given data vector, the DWT version will provide a more accurate approximation of the original data. Hence, for an equivalent approximation, the DWT requires less space than the DFT. Unlike the DFT, wavelets are quite localized in space, contributing to the conservation of local detail.

There is only one DFT, yet there are several families of DWTs. Figure 3.4 shows some wavelet families. Popular wavelet transforms include the Haar-2, Daubechies-4, and Daubechies-6. The general procedure for applying a discrete wavelet transform uses a hierarchical *pyramid algorithm* that halves the data at each iteration, resulting in fast computational speed. The method is as follows:

1. The length, L , of the input data vector must be an integer power of 2. This condition can be met by padding the data vector with zeros as necessary ($L \geq n$).
2. Each transform involves applying two functions. The first applies some data smoothing, such as a sum or weighted average. The second performs a weighted difference, which acts to bring out the detailed features of the data.
3. The two functions are applied to pairs of data points in \mathbf{X} , that is, to all pairs of measurements (x_{2i}, x_{2i+1}) . This results in two data sets of length $L/2$. In general, these represent a smoothed or low-frequency version of the input data and the high-frequency content of it, respectively.
4. The two functions are recursively applied to the data sets obtained in the previous loop, until the resulting data sets obtained are of length 2.
5. Selected values from the data sets obtained in the previous iterations are designated the wavelet coefficients of the transformed data.

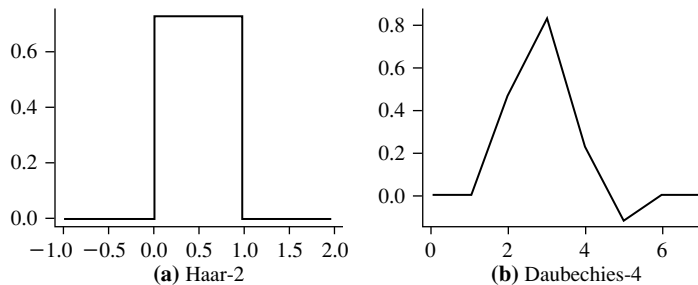


Figure 3.4 Examples of wavelet families. The number next to a wavelet name is the number of *vanishing moments* of the wavelet. This is a set of mathematical relationships that the coefficients must satisfy and is related to the number of coefficients.

Equivalently, a matrix multiplication can be applied to the input data in order to obtain the wavelet coefficients, where the matrix used depends on the given DWT. The matrix must be **orthonormal**, meaning that the columns are unit vectors and are mutually orthogonal, so that the matrix inverse is just its transpose. Although we do not have room to discuss it here, this property allows the reconstruction of the data from the smooth and smooth-difference data sets. By factoring the matrix used into a product of a few sparse matrices, the resulting “fast DWT” algorithm has a complexity of $O(n)$ for an input vector of length n .

Wavelet transforms can be applied to multidimensional data such as a data cube. This is done by first applying the transform to the first dimension, then to the second, and so on. The computational complexity involved is linear with respect to the number of cells in the cube. Wavelet transforms give good results on sparse or skewed data and on data with ordered attributes. Lossy compression by wavelets is reportedly better than JPEG compression, the current commercial standard. Wavelet transforms have many real-world applications, including the compression of fingerprint images, computer vision, analysis of time-series data, and data cleaning.

3.4.3 Principal Components Analysis

In this subsection we provide an intuitive introduction to principal components analysis as a method of dimensionality reduction. A detailed theoretical explanation is beyond the scope of this book. For additional references, please see the bibliographic notes (Section 3.8) at the end of this chapter.

Suppose that the data to be reduced consist of tuples or data vectors described by n attributes or dimensions. **Principal components analysis (PCA)**; also called the Karhunen-Loeve, or K-L, method) searches for k n -dimensional orthogonal vectors that can best be used to represent the data, where $k \leq n$. The original data are thus projected onto a much smaller space, resulting in dimensionality reduction. Unlike attribute subset selection (Section 3.4.4), which reduces the attribute set size by retaining a subset of the initial set of attributes, PCA “combines” the essence of attributes by creating an alternative, smaller set of variables. The initial data can then be projected onto this smaller set. PCA often reveals relationships that were not previously suspected and thereby allows interpretations that would not ordinarily result.

The basic procedure is as follows:

1. The input data are normalized, so that each attribute falls within the same range. This step helps ensure that attributes with large domains will not dominate attributes with smaller domains.
2. PCA computes k orthonormal vectors that provide a basis for the normalized input data. These are unit vectors that each point in a direction perpendicular to the others. These vectors are referred to as the *principal components*. The input data are a linear combination of the principal components.
3. The principal components are sorted in order of decreasing “significance” or strength. The principal components essentially serve as a new set of axes for the data,

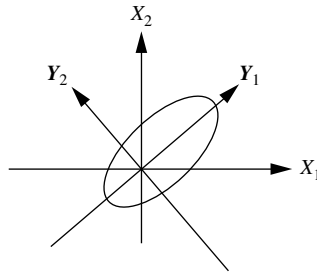


Figure 3.5 Principal components analysis. Y_1 and Y_2 are the first two principal components for the given data.

providing important information about variance. That is, the sorted axes are such that the first axis shows the most variance among the data, the second axis shows the next highest variance, and so on. For example, Figure 3.5 shows the first two principal components, Y_1 and Y_2 , for the given set of data originally mapped to the axes X_1 and X_2 . This information helps identify groups or patterns within the data.

4. Because the components are sorted in decreasing order of “significance,” the data size can be reduced by eliminating the weaker components, that is, those with low variance. Using the strongest principal components, it should be possible to reconstruct a good approximation of the original data.

PCA can be applied to ordered and unordered attributes, and can handle sparse data and skewed data. Multidimensional data of more than two dimensions can be handled by reducing the problem to two dimensions. Principal components may be used as inputs to multiple regression and cluster analysis. In comparison with wavelet transforms, PCA tends to be better at handling sparse data, whereas wavelet transforms are more suitable for data of high dimensionality.

3.4.4 Attribute Subset Selection

Data sets for analysis may contain hundreds of attributes, many of which may be irrelevant to the mining task or redundant. For example, if the task is to classify customers based on whether or not they are likely to purchase a popular new CD at *AllElectronics* when notified of a sale, attributes such as the customer’s telephone number are likely to be irrelevant, unlike attributes such as *age* or *music.taste*. Although it may be possible for a domain expert to pick out some of the useful attributes, this can be a difficult and time-consuming task, especially when the data’s behavior is not well known. (Hence, a reason behind its analysis!) Leaving out relevant attributes or keeping irrelevant attributes may be detrimental, causing confusion for the mining algorithm employed. This can result in discovered patterns of poor quality. In addition, the added volume of irrelevant or redundant attributes can slow down the mining process.

Attribute subset selection⁴ reduces the data set size by removing irrelevant or redundant attributes (or dimensions). The goal of attribute subset selection is to find a minimum set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Mining on a reduced set of attributes has an additional benefit: It reduces the number of attributes appearing in the discovered patterns, helping to make the patterns easier to understand.

“How can we find a ‘good’ subset of the original attributes?” For n attributes, there are 2^n possible subsets. An exhaustive search for the optimal subset of attributes can be prohibitively expensive, especially as n and the number of data classes increase. Therefore, heuristic methods that explore a reduced search space are commonly used for attribute subset selection. These methods are typically **greedy** in that, while searching through attribute space, they always make what looks to be the best choice at the time. Their strategy is to make a locally optimal choice in the hope that this will lead to a globally optimal solution. Such greedy methods are effective in practice and may come close to estimating an optimal solution.

The “best” (and “worst”) attributes are typically determined using tests of statistical significance, which assume that the attributes are independent of one another. Many other attribute evaluation measures can be used such as the *information gain* measure used in building decision trees for classification.⁵

Basic heuristic methods of attribute subset selection include the techniques that follow, some of which are illustrated in Figure 3.6.

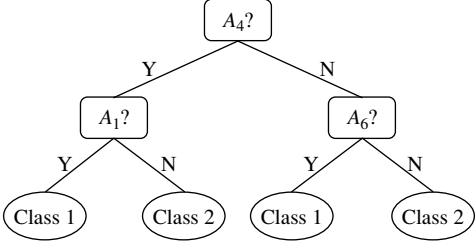
Forward selection	Backward elimination	Decision tree induction
<p>Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$</p> <p>Initial reduced set: $\{\}$ $\Rightarrow \{A_1\}$ $\Rightarrow \{A_1, A_4\}$ \Rightarrow Reduced attribute set: $\{A_1, A_4, A_6\}$</p>	<p>Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$</p> <p>$\Rightarrow \{A_1, A_3, A_4, A_5, A_6\}$ $\Rightarrow \{A_1, A_4, A_5, A_6\}$ \Rightarrow Reduced attribute set: $\{A_1, A_4, A_6\}$</p>	<p>Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$</p>  <p>\Rightarrow Reduced attribute set: $\{A_1, A_4, A_6\}$</p>

Figure 3.6 Greedy (heuristic) methods for attribute subset selection.

⁴In machine learning, attribute subset selection is known as *feature subset selection*.

⁵The information gain measure is described in detail in Chapter 8.

1. **Stepwise forward selection:** The procedure starts with an empty set of attributes as the reduced set. The best of the original attributes is determined and added to the reduced set. At each subsequent iteration or step, the best of the remaining original attributes is added to the set.
2. **Stepwise backward elimination:** The procedure starts with the full set of attributes. At each step, it removes the worst attribute remaining in the set.
3. **Combination of forward selection and backward elimination:** The stepwise forward selection and backward elimination methods can be combined so that, at each step, the procedure selects the best attribute and removes the worst from among the remaining attributes.
4. **Decision tree induction:** Decision tree algorithms (e.g., ID3, C4.5, and CART) were originally intended for classification. Decision tree induction constructs a flowchart-like structure where each internal (nonleaf) node denotes a test on an attribute, each branch corresponds to an outcome of the test, and each external (leaf) node denotes a class prediction. At each node, the algorithm chooses the “best” attribute to partition the data into individual classes.

When decision tree induction is used for attribute subset selection, a tree is constructed from the given data. All attributes that do not appear in the tree are assumed to be irrelevant. The set of attributes appearing in the tree form the reduced subset of attributes.

The stopping criteria for the methods may vary. The procedure may employ a threshold on the measure used to determine when to stop the attribute selection process.

In some cases, we may want to create new attributes based on others. Such **attribute construction**⁶ can help improve accuracy and understanding of structure in high-dimensional data. For example, we may wish to add the attribute *area* based on the attributes *height* and *width*. By combining attributes, attribute construction can discover missing information about the relationships between data attributes that can be useful for knowledge discovery.

3.4.5 Regression and Log-Linear Models: Parametric Data Reduction

Regression and log-linear models can be used to approximate the given data. In (simple) **linear regression**, the data are modeled to fit a straight line. For example, a random variable, y (called a *response variable*), can be modeled as a linear function of another random variable, x (called a *predictor variable*), with the equation

$$y = wx + b, \quad (3.7)$$

where the variance of y is assumed to be constant. In the context of data mining, x and y are numeric database attributes. The coefficients, w and b (called *regression coefficients*),

⁶In the machine learning literature, attribute construction is known as *feature construction*.

specify the slope of the line and the y -intercept, respectively. These coefficients can be solved for by the *method of least squares*, which minimizes the error between the actual line separating the data and the estimate of the line. **Multiple linear regression** is an extension of (simple) linear regression, which allows a response variable, y , to be modeled as a linear function of two or more predictor variables.

Log-linear models approximate discrete multidimensional probability distributions. Given a set of tuples in n dimensions (e.g., described by n attributes), we can consider each tuple as a point in an n -dimensional space. Log-linear models can be used to estimate the probability of each point in a multidimensional space for a set of discretized attributes, based on a smaller subset of dimensional combinations. This allows a higher-dimensional data space to be constructed from lower-dimensional spaces. Log-linear models are therefore also useful for dimensionality reduction (since the lower-dimensional points together typically occupy less space than the original data points) and data smoothing (since aggregate estimates in the lower-dimensional space are less subject to sampling variations than the estimates in the higher-dimensional space).

Regression and log-linear models can both be used on sparse data, although their application may be limited. While both methods can handle skewed data, regression does exceptionally well. Regression can be computationally intensive when applied to high-dimensional data, whereas log-linear models show good scalability for up to 10 or so dimensions.

Several software packages exist to solve regression problems. Examples include SAS (www.sas.com), SPSS (www.spss.com), and S-Plus (www.insightful.com). Another useful resource is the book *Numerical Recipes in C*, by Press, Teukolsky, Vetterling, and Flannery [PTVF07], and its associated source code.

3.4.6 Histograms

Histograms use binning to approximate data distributions and are a popular form of data reduction. Histograms were introduced in Section 2.2.3. A **histogram** for an attribute, A , partitions the data distribution of A into disjoint subsets, referred to as *buckets* or *bins*. If each bucket represents only a single attribute–value/frequency pair, the buckets are called *singleton buckets*. Often, buckets instead represent continuous ranges for the given attribute.

Example 3.3 Histograms. The following data are a list of *AllElectronics* prices for commonly sold items (rounded to the nearest dollar). The numbers have been sorted: 1, 1, 5, 5, 5, 5, 8, 8, 10, 10, 10, 10, 12, 14, 14, 14, 15, 15, 15, 15, 15, 15, 18, 18, 18, 18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 25, 25, 25, 25, 25, 28, 28, 30, 30, 30.

Figure 3.7 shows a histogram for the data using singleton buckets. To further reduce the data, it is common to have each bucket denote a continuous value range for the given attribute. In Figure 3.8, each bucket represents a different \$10 range for *price*. ■

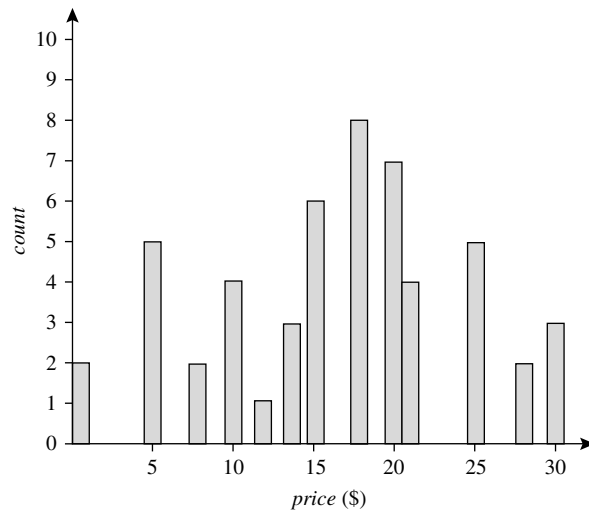


Figure 3.7 A histogram for *price* using singleton buckets—each bucket represents one price–value/frequency pair.

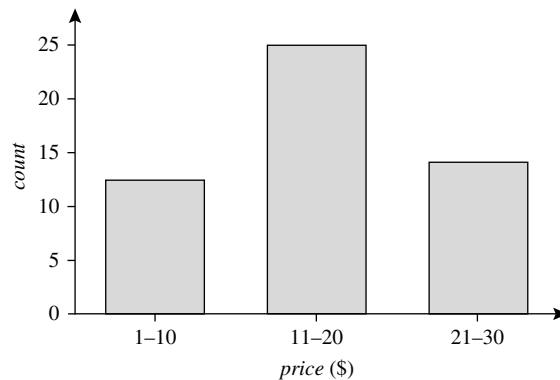


Figure 3.8 An equal-width histogram for *price*, where values are aggregated so that each bucket has a uniform width of \$10.

“How are the buckets determined and the attribute values partitioned?” There are several partitioning rules, including the following:

- **Equal-width:** In an equal-width histogram, the width of each bucket range is uniform (e.g., the width of \$10 for the buckets in Figure 3.8).
- **Equal-frequency** (or equal-depth): In an equal-frequency histogram, the buckets are created so that, roughly, the frequency of each bucket is constant (i.e., each bucket contains roughly the same number of contiguous data samples).

Histograms are highly effective at approximating both sparse and dense data, as well as highly skewed and uniform data. The histograms described before for single attributes can be extended for multiple attributes. *Multidimensional histograms* can capture dependencies between attributes. These histograms have been found effective in approximating data with up to five attributes. More studies are needed regarding the effectiveness of multidimensional histograms for high dimensionalities.

Singleton buckets are useful for storing high-frequency outliers.

3.4.7 Clustering

Clustering techniques consider data tuples as objects. They partition the objects into groups, or *clusters*, so that objects within a cluster are “similar” to one another and “dis-similar” to objects in other clusters. Similarity is commonly defined in terms of how “close” the objects are in space, based on a distance function. The “quality” of a cluster may be represented by its *diameter*, the maximum distance between any two objects in the cluster. **Centroid distance** is an alternative measure of cluster quality and is defined as the average distance of each cluster object from the cluster centroid (denoting the “average object,” or average point in space for the cluster). Figure 3.3 showed a 2-D plot of customer data with respect to customer locations in a city. Three data clusters are visible.

In data reduction, the cluster representations of the data are used to replace the actual data. The effectiveness of this technique depends on the data’s nature. It is much more effective for data that can be organized into distinct clusters than for smeared data.

There are many measures for defining clusters and cluster quality. Clustering methods are further described in Chapters 10 and 11.

3.4.8 Sampling

Sampling can be used as a data reduction technique because it allows a large data set to be represented by a much smaller random data sample (or subset). Suppose that a large data set, D , contains N tuples. Let’s look at the most common ways that we could sample D for data reduction, as illustrated in Figure 3.9.

- **Simple random sample without replacement (SRSWOR) of size s :** This is created by drawing s of the N tuples from D ($s < N$), where the probability of drawing any tuple in D is $1/N$, that is, all tuples are equally likely to be sampled.
- **Simple random sample with replacement (SRSWR) of size s :** This is similar to SRSWOR, except that each time a tuple is drawn from D , it is recorded and then *replaced*. That is, after a tuple is drawn, it is placed back in D so that it may be drawn again.
- **Cluster sample:** If the tuples in D are grouped into M mutually disjoint “clusters,” then an SRS of s clusters can be obtained, where $s < M$. For example, tuples in a database are usually retrieved a page at a time, so that each page can be considered

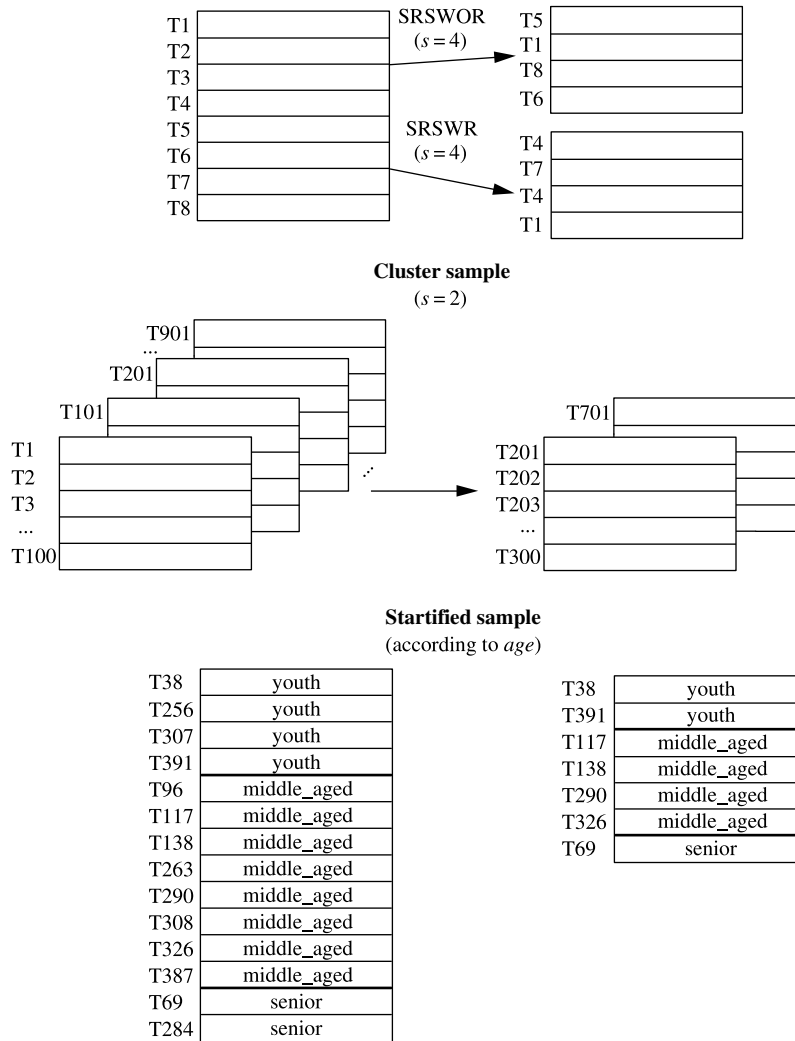


Figure 3.9 Sampling can be used for data reduction.

a cluster. A reduced data representation can be obtained by applying, say, SRSWOR to the pages, resulting in a cluster sample of the tuples. Other clustering criteria conveying rich semantics can also be explored. For example, in a spatial database, we may choose to define clusters geographically based on how closely different areas are located.

- **Stratified sample:** If D is divided into mutually disjoint parts called *strata*, a stratified sample of D is generated by obtaining an SRS at each stratum. This helps ensure a

representative sample, especially when the data are skewed. For example, a stratified sample may be obtained from customer data, where a stratum is created for each customer age group. In this way, the age group having the smallest number of customers will be sure to be represented.

An advantage of sampling for data reduction is that the cost of obtaining a sample is *proportional to the size of the sample, s* , as opposed to N , the data set size. Hence, sampling complexity is potentially *sublinear* to the size of the data. Other data reduction techniques can require at least one complete pass through D . For a fixed sample size, sampling complexity increases only linearly as the number of data dimensions, n , increases, whereas techniques using histograms, for example, increase exponentially in n .

When applied to data reduction, sampling is most commonly used to estimate the answer to an aggregate query. It is possible (using the central limit theorem) to determine a sufficient sample size for estimating a given function within a specified degree of error. This sample size, s , may be extremely small in comparison to N . Sampling is a natural choice for the progressive refinement of a reduced data set. Such a set can be further refined by simply increasing the sample size.

3.4.9 Data Cube Aggregation

Imagine that you have collected the data for your analysis. These data consist of the *AllElectronics* sales per quarter, for the years 2008 to 2010. You are, however, interested in the annual sales (total per year), rather than the total per quarter. Thus, the data can be *aggregated* so that the resulting data summarize the total sales per year instead of per quarter. This aggregation is illustrated in Figure 3.10. The resulting data set is smaller in volume, without loss of information necessary for the analysis task.

Data cubes are discussed in detail in Chapter 4 on data warehousing and Chapter 5 on data cube technology. We briefly introduce some concepts here. Data cubes store

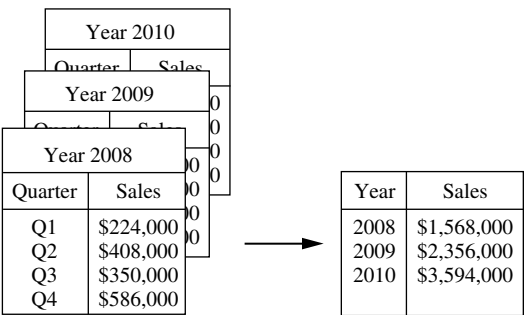


Figure 3.10 Sales data for a given branch of *AllElectronics* for the years 2008 through 2010. On the *left*, the sales are shown per quarter. On the *right*, the data are aggregated to provide the annual sales.

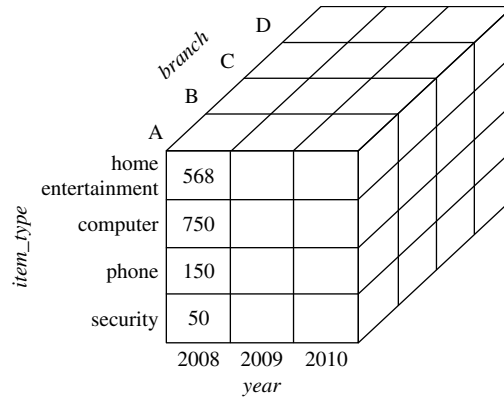


Figure 3.11 A data cube for sales at *AllElectronics*.

multidimensional aggregated information. For example, Figure 3.11 shows a data cube for multidimensional analysis of sales data with respect to annual sales per item type for each *AllElectronics* branch. Each cell holds an aggregate data value, corresponding to the data point in multidimensional space. (For readability, only some cell values are shown.) *Concept hierarchies* may exist for each attribute, allowing the analysis of data at multiple abstraction levels. For example, a hierarchy for *branch* could allow branches to be grouped into regions, based on their address. Data cubes provide fast access to precomputed, summarized data, thereby benefiting online analytical processing as well as data mining.

The cube created at the lowest abstraction level is referred to as the **base cuboid**. The base cuboid should correspond to an individual entity of interest such as *sales* or *customer*. In other words, the lowest level should be usable, or useful for the analysis. A cube at the highest level of abstraction is the **apex cuboid**. For the sales data in Figure 3.11, the apex cuboid would give one total—the total *sales* for all three years, for all item types, and for all branches. Data cubes created for varying levels of abstraction are often referred to as *cuboids*, so that a data cube may instead refer to a *lattice of cuboids*. Each higher abstraction level further reduces the resulting data size. When replying to data mining requests, the *smallest* available cuboid relevant to the given task should be used. This issue is also addressed in Chapter 4.

3.5 Data Transformation and Data Discretization

This section presents methods of data transformation. In this preprocessing step, the data are transformed or consolidated so that the resulting mining process may be more efficient, and the patterns found may be easier to understand. Data discretization, a form of data transformation, is also discussed.

3.5.1 Data Transformation Strategies Overview

In *data transformation*, the data are transformed or consolidated into forms appropriate for mining. Strategies for data transformation include the following:

1. **Smoothing**, which works to remove noise from the data. Techniques include binning, regression, and clustering.
2. **Attribute construction** (or *feature construction*), where new attributes are constructed and added from the given set of attributes to help the mining process.
3. **Aggregation**, where summary or aggregation operations are applied to the data. For example, the daily sales data may be aggregated so as to compute monthly and annual total amounts. This step is typically used in constructing a data cube for data analysis at multiple abstraction levels.
4. **Normalization**, where the attribute data are scaled so as to fall within a smaller range, such as -1.0 to 1.0 , or 0.0 to 1.0 .
5. **Discretization**, where the raw values of a numeric attribute (e.g., *age*) are replaced by interval labels (e.g., $0-10$, $11-20$, etc.) or conceptual labels (e.g., *youth*, *adult*, *senior*). The labels, in turn, can be recursively organized into higher-level concepts, resulting in a *concept hierarchy* for the numeric attribute. Figure 3.12 shows a concept hierarchy for the attribute *price*. More than one concept hierarchy can be defined for the same attribute to accommodate the needs of various users.
6. **Concept hierarchy generation for nominal data**, where attributes such as *street* can be generalized to higher-level concepts, like *city* or *country*. Many hierarchies for nominal attributes are implicit within the database schema and can be automatically defined at the schema definition level.

Recall that there is much overlap between the major data preprocessing tasks. The first three of these strategies were discussed earlier in this chapter. Smoothing is a form of

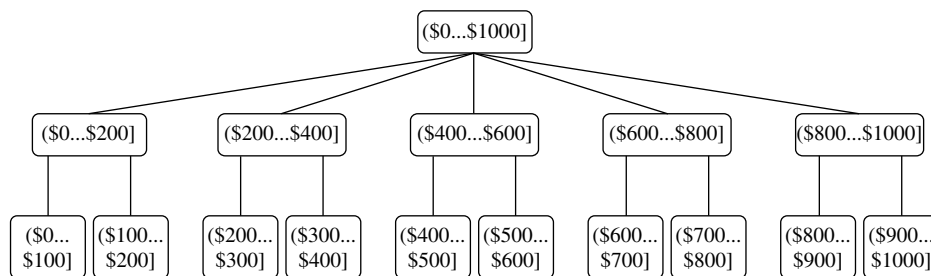


Figure 3.12 A concept hierarchy for the attribute *price*, where an interval $(\$X \dots \$Y]$ denotes the range from $\$X$ (exclusive) to $\$Y$ (inclusive).

data cleaning and was addressed in Section 3.2.2. Section 3.2.3 on the data cleaning process also discussed ETL tools, where users specify transformations to correct data inconsistencies. Attribute construction and aggregation were discussed in Section 3.4 on data reduction. In this section, we therefore concentrate on the latter three strategies.

Discretization techniques can be categorized based on how the discretization is performed, such as whether it uses class information or which direction it proceeds (i.e., top-down vs. bottom-up). If the discretization process uses class information, then we say it is *supervised discretization*. Otherwise, it is *unsupervised*. If the process starts by first finding one or a few points (called *split points* or *cut points*) to split the entire attribute range, and then repeats this recursively on the resulting intervals, it is called *top-down discretization* or *splitting*. This contrasts with *bottom-up discretization* or *merging*, which starts by considering all of the continuous values as potential split-points, removes some by merging neighborhood values to form intervals, and then recursively applies this process to the resulting intervals.

Data discretization and concept hierarchy generation are also forms of data reduction. The raw data are replaced by a smaller number of interval or concept labels. This simplifies the original data and makes the mining more efficient. The resulting patterns mined are typically easier to understand. Concept hierarchies are also useful for mining at multiple abstraction levels.

The rest of this section is organized as follows. First, normalization techniques are presented in Section 3.5.2. We then describe several techniques for data discretization, each of which can be used to generate concept hierarchies for numeric attributes. The techniques include *binning* (Section 3.5.3) and *histogram analysis* (Section 3.5.4), as well as *cluster analysis*, *decision tree analysis*, and *correlation analysis* (Section 3.5.5). Finally, Section 3.5.6 describes the automatic generation of concept hierarchies for nominal data.

3.5.2 Data Transformation by Normalization

The measurement unit used can affect the data analysis. For example, changing measurement units from meters to inches for *height*, or from kilograms to pounds for *weight*, may lead to very different results. In general, expressing an attribute in smaller units will lead to a larger range for that attribute, and thus tend to give such an attribute greater effect or “weight.” To help avoid dependence on the choice of measurement units, the data should be *normalized* or *standardized*. This involves transforming the data to fall within a smaller or common range such as $[-1, 1]$ or $[0.0, 1.0]$. (The terms *standardize* and *normalize* are used interchangeably in data preprocessing, although in statistics, the latter term also has other connotations.)

Normalizing the data attempts to give all attributes an equal weight. Normalization is particularly useful for classification algorithms involving neural networks or distance measurements such as nearest-neighbor classification and clustering. If using the neural network backpropagation algorithm for classification mining (Chapter 9), normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. For distance-based methods, normalization helps prevent

attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). It is also useful when given no prior knowledge of the data.

There are many methods for data normalization. We study *min-max normalization*, *z-score normalization*, and *normalization by decimal scaling*. For our discussion, let A be a numeric attribute with n observed values, v_1, v_2, \dots, v_n .

Min-max normalization performs a linear transformation on the original data. Suppose that \min_A and \max_A are the minimum and maximum values of an attribute, A . Min-max normalization maps a value, v_i , of A to v'_i in the range $[\text{new_min}_A, \text{new_max}_A]$ by computing

$$v'_i = \frac{v_i - \min_A}{\max_A - \min_A} (\text{new_max}_A - \text{new_min}_A) + \text{new_min}_A. \quad (3.8)$$

Min-max normalization preserves the relationships among the original data values. It will encounter an “out-of-bounds” error if a future input case for normalization falls outside of the original data range for A .

Example 3.4 Min-max normalization. Suppose that the minimum and maximum values for the attribute *income* are \$12,000 and \$98,000, respectively. We would like to map *income* to the range $[0.0, 1.0]$. By min-max normalization, a value of \$73,600 for *income* is transformed to $\frac{73,600 - 12,000}{98,000 - 12,000} (1.0 - 0) + 0 = 0.716$. ■

In **z-score normalization** (or *zero-mean normalization*), the values for an attribute, A , are normalized based on the mean (i.e., average) and standard deviation of A . A value, v_i , of A is normalized to v'_i by computing

$$v'_i = \frac{v_i - \bar{A}}{\sigma_A}, \quad (3.9)$$

where \bar{A} and σ_A are the mean and standard deviation, respectively, of attribute A . The mean and standard deviation were discussed in Section 2.2, where $\bar{A} = \frac{1}{n}(v_1 + v_2 + \dots + v_n)$ and σ_A is computed as the square root of the variance of A (see Eq. (2.6)). This method of normalization is useful when the actual minimum and maximum of attribute A are unknown, or when there are outliers that dominate the min-max normalization.

Example 3.5 z-score normalization. Suppose that the mean and standard deviation of the values for the attribute *income* are \$54,000 and \$16,000, respectively. With z-score normalization, a value of \$73,600 for *income* is transformed to $\frac{73,600 - 54,000}{16,000} = 1.225$. ■

A variation of this z-score normalization replaces the standard deviation of Eq. (3.9) by the *mean absolute deviation* of A . The *mean absolute deviation* of A , denoted s_A , is

$$s_A = \frac{1}{n} (|v_1 - \bar{A}| + |v_2 - \bar{A}| + \dots + |v_n - \bar{A}|). \quad (3.10)$$

Thus, z-score normalization using the mean absolute deviation is

$$v'_i = \frac{v_i - \bar{A}}{s_A}. \quad (3.11)$$

The mean absolute deviation, s_A , is more robust to outliers than the standard deviation, σ_A . When computing the mean absolute deviation, the deviations from the mean (i.e., $|x_i - \bar{x}|$) are not squared; hence, the effect of outliers is somewhat reduced.

Normalization by decimal scaling normalizes by moving the decimal point of values of attribute A . The number of decimal points moved depends on the maximum absolute value of A . A value, v_i , of A is normalized to v'_i by computing

$$v'_i = \frac{v_i}{10^j}, \quad (3.12)$$

where j is the smallest integer such that $\max(|v'_i|) < 1$.

Example 3.6 Decimal scaling. Suppose that the recorded values of A range from -986 to 917 . The maximum absolute value of A is 986 . To normalize by decimal scaling, we therefore divide each value by 1000 (i.e., $j = 3$) so that -986 normalizes to -0.986 and 917 normalizes to 0.917 . ■

Note that normalization can change the original data quite a bit, especially when using z-score normalization or decimal scaling. It is also necessary to save the normalization parameters (e.g., the mean and standard deviation if using z-score normalization) so that future data can be normalized in a uniform manner.

3.5.3 Discretization by Binning

Binning is a top-down splitting technique based on a specified number of bins. Section 3.2.2 discussed binning methods for data smoothing. These methods are also used as discretization methods for data reduction and concept hierarchy generation. For example, attribute values can be discretized by applying equal-width or equal-frequency binning, and then replacing each bin value by the bin mean or median, as in *smoothing by bin means* or *smoothing by bin medians*, respectively. These techniques can be applied recursively to the resulting partitions to generate concept hierarchies.

Binning does not use class information and is therefore an unsupervised discretization technique. It is sensitive to the user-specified number of bins, as well as the presence of outliers.

3.5.4 Discretization by Histogram Analysis

Like binning, histogram analysis is an unsupervised discretization technique because it does not use class information. Histograms were introduced in Section 2.2.3. A histogram partitions the values of an attribute, A , into disjoint ranges called *buckets* or *bins*.

Various partitioning rules can be used to define histograms (Section 3.4.6). In an *equal-width* histogram, for example, the values are partitioned into equal-size partitions or ranges (e.g., earlier in Figure 3.8 for *price*, where each bucket has a width of \$10). With an *equal-frequency* histogram, the values are partitioned so that, ideally, each partition contains the same number of data tuples. The histogram analysis algorithm can be applied recursively to each partition in order to automatically generate a multilevel concept hierarchy, with the procedure terminating once a prespecified number of concept levels has been reached. A *minimum interval size* can also be used per level to control the recursive procedure. This specifies the minimum width of a partition, or the minimum number of values for each partition at each level. Histograms can also be partitioned based on cluster analysis of the data distribution, as described next.

3.5.5 Discretization by Cluster, Decision Tree, and Correlation Analyses

Clustering, decision tree analysis, and correlation analysis can be used for data discretization. We briefly study each of these approaches.

Cluster analysis is a popular data discretization method. A clustering algorithm can be applied to discretize a numeric attribute, *A*, by partitioning the values of *A* into clusters or groups. Clustering takes the distribution of *A* into consideration, as well as the closeness of data points, and therefore is able to produce high-quality discretization results.

Clustering can be used to generate a concept hierarchy for *A* by following either a top-down splitting strategy or a bottom-up merging strategy, where each cluster forms a node of the concept hierarchy. In the former, each initial cluster or partition may be further decomposed into several subclusters, forming a lower level of the hierarchy. In the latter, clusters are formed by repeatedly grouping neighboring clusters in order to form higher-level concepts. Clustering methods for data mining are studied in Chapters 10 and 11.

Techniques to generate decision trees for classification (Chapter 8) can be applied to discretization. Such techniques employ a top-down splitting approach. Unlike the other methods mentioned so far, decision tree approaches to discretization are supervised, that is, they make use of class label information. For example, we may have a data set of patient symptoms (the attributes) where each patient has an associated *diagnosis* class label. Class distribution information is used in the calculation and determination of split-points (data values for partitioning an attribute range). Intuitively, the main idea is to select split-points so that a given resulting partition contains as many tuples of the same class as possible. *Entropy* is the most commonly used measure for this purpose. To discretize a numeric attribute, *A*, the method selects the value of *A* that has the minimum entropy as a split-point, and recursively partitions the resulting intervals to arrive at a hierarchical discretization. Such discretization forms a concept hierarchy for *A*.

Because decision tree-based discretization uses class information, it is more likely that the interval boundaries (split-points) are defined to occur in places that may help improve classification accuracy. Decision trees and the entropy measure are described in greater detail in Section 8.2.2.

Measures of correlation can be used for discretization. *ChiMerge* is a χ^2 -based discretization method. The discretization methods that we have studied up to this point have all employed a top-down, splitting strategy. This contrasts with *ChiMerge*, which employs a bottom-up approach by finding the best neighboring intervals and then merging them to form larger intervals, recursively. As with decision tree analysis, *ChiMerge* is supervised in that it uses class information. The basic notion is that for accurate discretization, the relative class frequencies should be fairly consistent within an interval. Therefore, if two adjacent intervals have a very similar distribution of classes, then the intervals can be merged. Otherwise, they should remain separate.

ChiMerge proceeds as follows. Initially, each distinct value of a numeric attribute *A* is considered to be one interval. χ^2 tests are performed for every pair of adjacent intervals. Adjacent intervals with the least χ^2 values are merged together, because low χ^2 values for a pair indicate similar class distributions. This merging process proceeds recursively until a predefined stopping criterion is met.

3.5.6 Concept Hierarchy Generation for Nominal Data

We now look at data transformation for nominal data. In particular, we study concept hierarchy generation for nominal attributes. Nominal attributes have a finite (but possibly large) number of distinct values, with no ordering among the values. Examples include *geographic_location*, *job_category*, and *item_type*.

Manual definition of concept hierarchies can be a tedious and time-consuming task for a user or a domain expert. Fortunately, many hierarchies are implicit within the database schema and can be automatically defined at the schema definition level. The concept hierarchies can be used to transform the data into multiple levels of granularity. For example, data mining patterns regarding sales may be found relating to specific regions or countries, in addition to individual branch locations.

We study four methods for the generation of concept hierarchies for nominal data, as follows.

1. **Specification of a partial ordering of attributes explicitly at the schema level by users or experts:** Concept hierarchies for nominal attributes or dimensions typically involve a group of attributes. A user or expert can easily define a concept hierarchy by specifying a partial or total ordering of the attributes at the schema level. For example, suppose that a relational database contains the following group of attributes: *street*, *city*, *province_or_state*, and *country*. Similarly, a data warehouse *location* dimension may contain the same attributes. A hierarchy can be defined by specifying the total ordering among these attributes at the schema level such as *street* < *city* < *province_or_state* < *country*.
2. **Specification of a portion of a hierarchy by explicit data grouping:** This is essentially the manual definition of a portion of a concept hierarchy. In a large database, it is unrealistic to define an entire concept hierarchy by explicit value enumeration. On the contrary, we can easily specify explicit groupings for a small portion of intermediate-level data. For example, after specifying that *province* and *country*

form a hierarchy at the schema level, a user could define some intermediate levels manually, such as “{*Alberta, Saskatchewan, Manitoba*} \subset *prairies.Canada*” and “{*British Columbia, prairies.Canada*} \subset *Western.Canada*.”

3. **Specification of a set of attributes, but not of their partial ordering:** A user may specify a set of attributes forming a concept hierarchy, but omit to explicitly state their partial ordering. The system can then try to automatically generate the attribute ordering so as to construct a meaningful concept hierarchy.

“Without knowledge of data semantics, how can a hierarchical ordering for an arbitrary set of nominal attributes be found?” Consider the observation that since higher-level concepts generally cover several subordinate lower-level concepts, an attribute defining a high concept level (e.g., *country*) will usually contain a smaller number of distinct values than an attribute defining a lower concept level (e.g., *street*). Based on this observation, a concept hierarchy can be automatically generated based on the number of distinct values per attribute in the given attribute set. The attribute with the most distinct values is placed at the lowest hierarchy level. The lower the number of distinct values an attribute has, the higher it is in the generated concept hierarchy. This heuristic rule works well in many cases. Some local-level swapping or adjustments may be applied by users or experts, when necessary, after examination of the generated hierarchy.

Let’s examine an example of this third method.

Example 3.7 Concept hierarchy generation based on the number of distinct values per attribute.

Suppose a user selects a set of location-oriented attributes—*street*, *country*, *province_or_state*, and *city*—from the *AllElectronics* database, but does not specify the hierarchical ordering among the attributes.

A concept hierarchy for *location* can be generated automatically, as illustrated in Figure 3.13. First, sort the attributes in ascending order based on the number of distinct values in each attribute. This results in the following (where the number of distinct values per attribute is shown in parentheses): *country* (15), *province_or_state* (365), *city* (3567), and *street* (674,339). Second, generate the hierarchy from the top down according to the sorted order, with the first attribute at the top level and the last attribute at the bottom level. Finally, the user can examine the generated hierarchy, and when necessary, modify it to reflect desired semantic relationships among the attributes. In this example, it is obvious that there is no need to modify the generated hierarchy. ■

Note that this heuristic rule is not foolproof. For example, a time dimension in a database may contain 20 distinct years, 12 distinct months, and 7 distinct days of the week. However, this does not suggest that the time hierarchy should be “*year* < *month* < *days_of_the_week*,” with *days_of_the_week* at the top of the hierarchy.

4. **Specification of only a partial set of attributes:** Sometimes a user can be careless when defining a hierarchy, or have only a vague idea about what should be included in a hierarchy. Consequently, the user may have included only a small subset of the

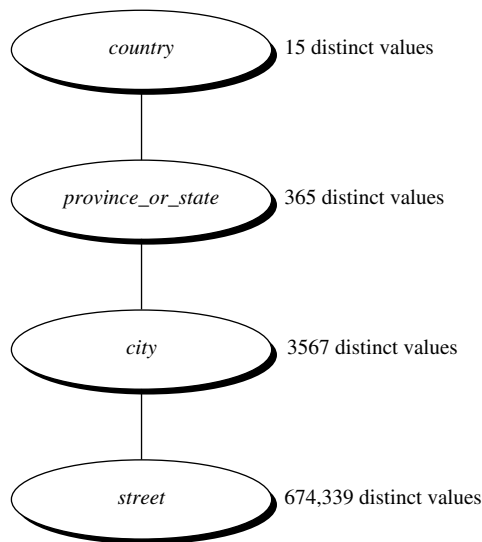


Figure 3.13 Automatic generation of a schema concept hierarchy based on the number of distinct attribute values.

relevant attributes in the hierarchy specification. For example, instead of including all of the hierarchically relevant attributes for *location*, the user may have specified only *street* and *city*. To handle such partially specified hierarchies, it is important to embed data semantics in the database schema so that attributes with tight semantic connections can be pinned together. In this way, the specification of one attribute may trigger a whole group of semantically tightly linked attributes to be “dragged in” to form a complete hierarchy. Users, however, should have the option to override this feature, as necessary.

Example 3.8 Concept hierarchy generation using prespecified semantic connections. Suppose that a data mining expert (serving as an administrator) has pinned together the five attributes *number*, *street*, *city*, *province_or_state*, and *country*, because they are closely linked semantically regarding the notion of *location*. If a user were to specify only the attribute *city* for a hierarchy defining *location*, the system can automatically drag in all five semantically related attributes to form a hierarchy. The user may choose to drop any of these attributes (e.g., *number* and *street*) from the hierarchy, keeping *city* as the lowest conceptual level. ■

In summary, information at the schema level and on attribute–value counts can be used to generate concept hierarchies for nominal data. Transforming nominal data with the use of concept hierarchies allows higher-level knowledge patterns to be found. It allows mining at multiple levels of abstraction, which is a common requirement for data mining applications.

3.6 Summary

- **Data quality** is defined in terms of *accuracy*, *completeness*, *consistency*, *timeliness*, *believability*, and *interpretability*. These qualities are assessed based on the intended use of the data.
- **Data cleaning** routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data. Data cleaning is usually performed as an iterative two-step process consisting of discrepancy detection and data transformation.
- **Data integration** combines data from multiple sources to form a coherent data store. The resolution of semantic heterogeneity, metadata, correlation analysis, tuple duplication detection, and data conflict detection contribute to smooth data integration.
- **Data reduction** techniques obtain a reduced representation of the data while minimizing the loss of information content. These include methods of *dimensionality reduction*, *numerosity reduction*, and *data compression*. **Dimensionality reduction** reduces the number of random variables or attributes under consideration. Methods include *wavelet transforms*, *principal components analysis*, *attribute subset selection*, and *attribute creation*. **Numerosity reduction** methods use parametric or nonparametric models to obtain smaller representations of the original data. Parametric models store only the model parameters instead of the actual data. Examples include regression and log-linear models. Nonparametric methods include histograms, clustering, sampling, and data cube aggregation. **Data compression** methods apply transformations to obtain a reduced or “compressed” representation of the original data. The data reduction is *lossless* if the original data can be reconstructed from the compressed data without any loss of information; otherwise, it is *lossy*.
- **Data transformation** routines convert the data into appropriate forms for mining. For example, in **normalization**, attribute data are scaled so as to fall within a small range such as 0.0 to 1.0. Other examples are **data discretization** and **concept hierarchy generation**.
- **Data discretization** transforms numeric data by mapping values to interval or concept labels. Such methods can be used to automatically generate *concept hierarchies* for the data, which allows for mining at multiple levels of granularity. Discretization techniques include binning, histogram analysis, cluster analysis, decision tree analysis, and correlation analysis. For nominal data, **concept hierarchies** may be generated based on schema definitions as well as the number of distinct values per attribute.
- Although numerous methods of data preprocessing have been developed, data preprocessing remains an active area of research, due to the huge amount of inconsistent or dirty data and the complexity of the problem.

3.7 Exercises

- 3.1 *Data quality* can be assessed in terms of several issues, including accuracy, completeness, and consistency. For each of the above three issues, discuss how data quality assessment can depend on the *intended use* of the data, giving examples. Propose two other dimensions of data quality.
- 3.2 In real-world data, tuples with *missing values* for some attributes are a common occurrence. Describe various methods for handling this problem.
- 3.3 Exercise 2.2 gave the following data (in increasing order) for the attribute *age*: 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70.
 - (a) Use *smoothing by bin means* to smooth these data, using a bin depth of 3. Illustrate your steps. Comment on the effect of this technique for the given data.
 - (b) How might you determine *outliers* in the data?
 - (c) What other methods are there for *data smoothing*?
- 3.4 Discuss issues to consider during *data integration*.
- 3.5 What are the value ranges of the following *normalization methods*?
 - (a) min-max normalization
 - (b) z-score normalization
 - (c) z-score normalization using the mean absolute deviation instead of standard deviation
 - (d) normalization by decimal scaling
- 3.6 Use these methods to *normalize* the following group of data:
200, 300, 400, 600, 1000
 - (a) min-max normalization by setting $min = 0$ and $max = 1$
 - (b) z-score normalization
 - (c) z-score normalization using the mean absolute deviation instead of standard deviation
 - (d) normalization by decimal scaling
- 3.7 Using the data for *age* given in Exercise 3.3, answer the following:
 - (a) Use min-max normalization to transform the value 35 for *age* onto the range $[0.0, 1.0]$.
 - (b) Use z-score normalization to transform the value 35 for *age*, where the standard deviation of *age* is 12.94 years.
 - (c) Use normalization by decimal scaling to transform the value 35 for *age*.
 - (d) Comment on which method you would prefer to use for the given data, giving reasons as to why.

3.8 Using the data for *age* and *body fat* given in Exercise 2.4, answer the following:

- (a) Normalize the two attributes based on *z-score normalization*.
- (b) Calculate the *correlation coefficient* (Pearson's product moment coefficient). Are these two attributes positively or negatively correlated? Compute their covariance.

3.9 Suppose a group of 12 *sales price* records has been sorted as follows:

5, 10, 11, 13, 15, 35, 50, 55, 72, 92, 204, 215.

Partition them into three bins by each of the following methods:

- (a) equal-frequency (equal-depth) partitioning
- (b) equal-width partitioning
- (c) clustering

3.10 Use a flowchart to summarize the following procedures for *attribute subset selection*:

- (a) stepwise forward selection
- (b) stepwise backward elimination
- (c) a combination of forward selection and backward elimination

3.11 Using the data for *age* given in Exercise 3.3,

- (a) Plot an equal-width histogram of width 10.
- (b) Sketch examples of each of the following sampling techniques: SRSWOR, SRSWR, cluster sampling, and stratified sampling. Use samples of size 5 and the strata "youth," "middle-aged," and "senior."

3.12 ChiMerge [Ker92] is a supervised, bottom-up (i.e., merge-based) *data discretization* method. It relies on χ^2 analysis: Adjacent intervals with the least χ^2 values are merged together until the chosen stopping criterion satisfies.

- (a) Briefly describe how ChiMerge works.
- (b) Take the IRIS data set, obtained from the University of California–Irvine Machine Learning Data Repository (www.ics.uci.edu/~mlearn/MLRepository.html), as a data set to be discretized. Perform data discretization for each of the four numeric attributes using the ChiMerge method. (Let the stopping criteria be: *max-interval* = 6). You need to write a small program to do this to avoid clumsy numerical computation. Submit your simple analysis and your test results: split-points, final intervals, and the documented source program.

3.13 Propose an algorithm, in pseudocode or in your favorite programming language, for the following:

- (a) The automatic generation of a concept hierarchy for nominal data based on the number of distinct values of attributes in the given schema.
- (b) The automatic generation of a concept hierarchy for numeric data based on the *equal-width* partitioning rule.

- (c) The automatic generation of a concept hierarchy for numeric data based on the *equal-frequency* partitioning rule.
- 3.14 Robust data loading poses a challenge in database systems because the input data are often dirty. In many cases, an input record may miss multiple values; some records could be *contaminated*, with some data values out of range or of a different data type than expected. Work out an automated *data cleaning and loading* algorithm so that the erroneous data will be marked and contaminated data will not be mistakenly inserted into the database during data loading.

3.8 Bibliographic Notes

Data preprocessing is discussed in a number of textbooks, including English [Eng99], Pyle [Pyl99], Loshin [Los01], Redman [Red01], and Dasu and Johnson [DJ03]. More specific references to individual preprocessing techniques are given later.

For discussion regarding data quality, see Redman [Red92]; Wang, Storey, and Firth [WSF95]; Wand and Wang [WW96]; Ballou and Tayi [BT99]; and Olson [Ols03]. Potter's Wheel (control.cx.berkeley.edu/abc), the interactive data cleaning tool described in Section 3.2.3, is presented in Raman and Hellerstein [RH01]. An example of the development of declarative languages for the specification of data transformation operators is given in Galhardas et al. [GFS⁺01]. The handling of missing attribute values is discussed in Friedman [Fri77]; Breiman, Friedman, Olshen, and Stone [BFOS84]; and Quinlan [Qui89]. Hua and Pei [HP07] presented a heuristic approach to cleaning *disguised missing data*, where such data are captured when users falsely select default values on forms (e.g., “January 1” for *birthdate*) when they do not want to disclose personal information.

A method for the detection of outlier or “garbage” patterns in a handwritten character database is given in Guyon, Matic, and Vapnik [GMV96]. Binning and data normalization are treated in many texts, including Kennedy et al. [KLV⁺98], Weiss and Indurkha [WI98], and Pyle [Pyl99]. Systems that include attribute (or feature) construction include BACON by Langley, Simon, Bradshaw, and Zytkow [LSBZ87]; Stagger by Schlimmer [Sch86]; FRINGE by Pagallo [Pag89]; and AQ17-DCI by Bloedorn and Michalski [BM98]. Attribute construction is also described in Liu and Motoda [LM98a, LM98b]. Dasu et al. built a BELLMAN system and proposed a set of interesting methods for building a data quality browser by mining database structures [DJMS02].

A good survey of data reduction techniques can be found in Barbará et al. [BDF⁺97]. For algorithms on data cubes and their precomputation, see Sarawagi and Stonebraker [SS94]; Agarwal et al. [AAD⁺96]; Harinarayan, Rajaraman, and Ullman [HRU96]; Ross and Srivastava [RS97]; and Zhao, Deshpande, and Naughton [ZDN97]. Attribute subset selection (or *feature subset selection*) is described in many texts such as Neter, Kutner, Nachtsheim, and Wasserman [NKNW96]; Dash and Liu [DL97]; and Liu and Motoda [LM98a, LM98b]. A combination forward selection and backward elimination method

was proposed in Siedlecki and Sklansky [SS88]. A wrapper approach to attribute selection is described in Kohavi and John [KJ97]. Unsupervised attribute subset selection is described in Dash, Liu, and Yao [DLY97].

For a description of wavelets for dimensionality reduction, see Press, Teukolosky, Vetterling, and Flannery [PTVF07]. A general account of wavelets can be found in Hubbard [Hub96]. For a list of wavelet software packages, see Bruce, Donoho, and Gao [BDG96]. Daubechies transforms are described in Daubechies [Dau92]. The book by Press et al. [PTVF07] includes an introduction to singular value decomposition for principal components analysis. Routines for PCA are included in most statistical software packages such as SAS (www.sas.com/SASHome.html).

An introduction to regression and log-linear models can be found in several textbooks such as James [Jam85]; Dobson [Dob90]; Johnson and Wichern [JW92]; Devore [Dev95]; and Neter, Kutner, Nachtsheim, and Wasserman [NKNW96]. For log-linear models (known as *multiplicative models* in the computer science literature), see Pearl [Pea88]. For a general introduction to histograms, see Barbará et al. [BDF⁺97] and Devore and Peck [DP97]. For extensions of single-attribute histograms to multiple attributes, see Muralikrishna and DeWitt [MD88] and Poosala and Ioannidis [PI97]. Several references to clustering algorithms are given in Chapters 10 and 11 of this book, which are devoted to the topic.

A survey of multidimensional indexing structures is given in Gaede and Günther [GG98]. The use of multidimensional index trees for data aggregation is discussed in Aoki [Aok98]. Index trees include R-trees (Guttman [Gut84]), quad-trees (Finkel and Bentley [FB74]), and their variations. For discussion on sampling and data mining, see Kivinen and Mannila [KM94] and John and Langley [JL96].

There are many methods for assessing attribute relevance. Each has its own bias. The information gain measure is biased toward attributes with many values. Many alternatives have been proposed, such as gain ratio (Quinlan [Qui93]), which considers the probability of each attribute value. Other relevance measures include the Gini index (Breiman, Friedman, Olshen, and Stone [BFOS84]), the χ^2 contingency table statistic, and the uncertainty coefficient (Johnson and Wichern [JW92]). For a comparison of attribute selection measures for decision tree induction, see Buntine and Niblett [BN92]. For additional methods, see Liu and Motoda [LM98a], Dash and Liu [DL97], and Almuallim and Dietterich [AD91].

Liu et al. [LHTD02] performed a comprehensive survey of data discretization methods. Entropy-based discretization with the C4.5 algorithm is described in Quinlan [Qui93]. In Catlett [Cat91], the D-2 system binarizes a numeric feature recursively. ChiMerge by Kerber [Ker92] and Chi2 by Liu and Setiono [LS95] are methods for the automatic discretization of numeric attributes that both employ the χ^2 statistic. Fayyad and Irani [FI93] apply the minimum description length principle to determine the number of intervals for numeric discretization. Concept hierarchies and their automatic generation from categorical data are described in Han and Fu [HF94].

Data Warehousing and Online Analytical Processing

Data warehouses generalize and consolidate data in multidimensional space. The construction of data warehouses involves data cleaning, data integration, and data transformation, and can be viewed as an important preprocessing step for data mining. Moreover, data warehouses provide *online analytical processing (OLAP)* tools for the interactive analysis of multidimensional data of varied granularities, which facilitates effective data generalization and data mining. Many other data mining functions, such as association, classification, prediction, and clustering, can be integrated with OLAP operations to enhance interactive mining of knowledge at multiple levels of abstraction. Hence, the data warehouse has become an increasingly important platform for data analysis and OLAP and will provide an effective platform for data mining. Therefore, data warehousing and OLAP form an essential step in the knowledge discovery process. This chapter presents an overview of data warehouse and OLAP technology. This overview is essential for understanding the overall data mining and knowledge discovery process.

In this chapter, we study a well-accepted definition of the data warehouse and see why more and more organizations are building data warehouses for the analysis of their data (Section 4.1). In particular, we study the *data cube*, a multidimensional data model for data warehouses and OLAP, as well as OLAP operations such as roll-up, drill-down, slicing, and dicing (Section 4.2). We also look at data warehouse design and usage (Section 4.3). In addition, we discuss *multidimensional data mining*, a powerful paradigm that integrates data warehouse and OLAP technology with that of data mining. An overview of data warehouse implementation examines general strategies for efficient data cube computation, OLAP data indexing, and OLAP query processing (Section 4.4). Finally, we study data generalization by attribute-oriented induction (Section 4.5). This method uses concept hierarchies to generalize data to multiple levels of abstraction.

4.1 Data Warehouse: Basic Concepts

This section gives an introduction to data warehouses. We begin with a definition of the data warehouse (Section 4.1.1). We outline the differences between operational database

systems and data warehouses (Section 4.1.2), then explain the need for using data warehouses for data analysis, rather than performing the analysis directly on traditional databases (Section 4.1.3). This is followed by a presentation of data warehouse architecture (Section 4.1.4). Next, we study three data warehouse models—an enterprise model, a data mart, and a virtual warehouse (Section 4.1.5). Section 4.1.6 describes back-end utilities for data warehousing, such as extraction, transformation, and loading. Finally, Section 4.1.7 presents the metadata repository, which stores data about data.

4.1.1 What Is a Data Warehouse?

Data warehousing provides architectures and tools for business executives to systematically organize, understand, and use their data to make strategic decisions. Data warehouse systems are valuable tools in today's competitive, fast-evolving world. In the last several years, many firms have spent millions of dollars in building enterprise-wide data warehouses. Many people feel that with competition mounting in every industry, data warehousing is the latest must-have marketing weapon—a way to retain customers by learning more about their needs.

“Then, what exactly is a data warehouse?” Data warehouses have been defined in many ways, making it difficult to formulate a rigorous definition. Loosely speaking, a data warehouse refers to a data repository that is maintained separately from an organization's operational databases. Data warehouse systems allow for integration of a variety of application systems. They support information processing by providing a solid platform of consolidated historic data for analysis.

According to William H. Inmon, a leading architect in the construction of data warehouse systems, “A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision making process” [Inm96]. This short but comprehensive definition presents the major features of a data warehouse. The four keywords—*subject-oriented*, *integrated*, *time-variant*, and *nonvolatile*—distinguish data warehouses from other data repository systems, such as relational database systems, transaction processing systems, and file systems.

Let's take a closer look at each of these key features.

- **Subject-oriented:** A data warehouse is organized around major subjects such as customer, supplier, product, and sales. Rather than concentrating on the day-to-day operations and transaction processing of an organization, a data warehouse focuses on the modeling and analysis of data for decision makers. Hence, data warehouses typically provide a simple and concise view of particular subject issues by excluding data that are not useful in the decision support process.
- **Integrated:** A data warehouse is usually constructed by integrating multiple heterogeneous sources, such as relational databases, flat files, and online transaction records. Data cleaning and data integration techniques are applied to ensure consistency in naming conventions, encoding structures, attribute measures, and so on.

- **Time-variant:** Data are stored to provide information from an historic perspective (e.g., the past 5–10 years). Every key structure in the data warehouse contains, either implicitly or explicitly, a time element.
- **Nonvolatile:** A data warehouse is always a physically separate store of data transformed from the application data found in the operational environment. Due to this separation, a data warehouse does not require transaction processing, recovery, and concurrency control mechanisms. It usually requires only two operations in data accessing: *initial loading of data* and *access of data*.

In sum, a data warehouse is a semantically consistent data store that serves as a physical implementation of a decision support data model. It stores the information an enterprise needs to make strategic decisions. A data warehouse is also often viewed as an architecture, constructed by integrating data from multiple heterogeneous sources to support structured and/or ad hoc queries, analytical reporting, and decision making.

Based on this information, we view **data warehousing** as the process of constructing and using data warehouses. The construction of a data warehouse requires data cleaning, data integration, and data consolidation. The utilization of a data warehouse often necessitates a collection of *decision support* technologies. This allows “knowledge workers” (e.g., managers, analysts, and executives) to use the warehouse to quickly and conveniently obtain an overview of the data, and to make sound decisions based on information in the warehouse. Some authors use the term *data warehousing* to refer only to the process of data warehouse *construction*, while the term *warehouse DBMS* is used to refer to the *management and utilization* of data warehouses. We will not make this distinction here.

“How are organizations using the information from data warehouses?” Many organizations use this information to support business decision-making activities, including (1) increasing customer focus, which includes the analysis of customer buying patterns (such as buying preference, buying time, budget cycles, and appetites for spending); (2) repositioning products and managing product portfolios by comparing the performance of sales by quarter, by year, and by geographic regions in order to fine-tune production strategies; (3) analyzing operations and looking for sources of profit; and (4) managing customer relationships, making environmental corrections, and managing the cost of corporate assets.

Data warehousing is also very useful from the point of view of *heterogeneous database integration*. Organizations typically collect diverse kinds of data and maintain large databases from multiple, heterogeneous, autonomous, and distributed information sources. It is highly desirable, yet challenging, to integrate such data and provide easy and efficient access to it. Much effort has been spent in the database industry and research community toward achieving this goal.

The traditional database approach to heterogeneous database integration is to build **wrappers** and **integrators** (or **mediators**) on top of multiple, heterogeneous databases. When a query is posed to a client site, a metadata dictionary is used to translate the query into queries appropriate for the individual heterogeneous sites involved. These

queries are then mapped and sent to local query processors. The results returned from the different sites are integrated into a global answer set. This **query-driven approach** requires complex information filtering and integration processes, and competes with local sites for processing resources. It is inefficient and potentially expensive for frequent queries, especially queries requiring aggregations.

Data warehousing provides an interesting alternative to this traditional approach. Rather than using a query-driven approach, data warehousing employs an **update-driven** approach in which information from multiple, heterogeneous sources is integrated in advance and stored in a warehouse for direct querying and analysis. Unlike online transaction processing databases, data warehouses do not contain the most current information. However, a data warehouse brings high performance to the integrated heterogeneous database system because data are copied, preprocessed, integrated, annotated, summarized, and restructured into one semantic data store. Furthermore, query processing in data warehouses does not interfere with the processing at local sources. Moreover, data warehouses can store and integrate historic information and support complex multidimensional queries. As a result, data warehousing has become popular in industry.

4.1.2 Differences between Operational Database Systems and Data Warehouses

Because most people are familiar with commercial relational database systems, it is easy to understand what a data warehouse is by comparing these two kinds of systems.

The major task of online operational database systems is to perform online transaction and query processing. These systems are called **online transaction processing (OLTP)** systems. They cover most of the day-to-day operations of an organization such as purchasing, inventory, manufacturing, banking, payroll, registration, and accounting. Data warehouse systems, on the other hand, serve users or knowledge workers in the role of data analysis and decision making. Such systems can organize and present data in various formats in order to accommodate the diverse needs of different users. These systems are known as **online analytical processing (OLAP)** systems.

The major distinguishing features of OLTP and OLAP are summarized as follows:

- **Users and system orientation:** An OLTP system is *customer-oriented* and is used for transaction and query processing by clerks, clients, and information technology professionals. An OLAP system is *market-oriented* and is used for data analysis by knowledge workers, including managers, executives, and analysts.
- **Data contents:** An OLTP system manages current data that, typically, are too detailed to be easily used for decision making. An OLAP system manages large amounts of historic data, provides facilities for summarization and aggregation, and stores and manages information at different levels of granularity. These features make the data easier to use for informed decision making.

- **Database design:** An OLTP system usually adopts an entity-relationship (ER) data model and an application-oriented database design. An OLAP system typically adopts either a *star* or a *snowflake* model (see Section 4.2.2) and a subject-oriented database design.
- **View:** An OLTP system focuses mainly on the current data within an enterprise or department, without referring to historic data or data in different organizations. In contrast, an OLAP system often spans multiple versions of a database schema, due to the evolutionary process of an organization. OLAP systems also deal with information that originates from different organizations, integrating information from many data stores. Because of their huge volume, OLAP data are stored on multiple storage media.
- **Access patterns:** The access patterns of an OLTP system consist mainly of short, atomic transactions. Such a system requires concurrency control and recovery mechanisms. However, accesses to OLAP systems are mostly read-only operations (because most data warehouses store historic rather than up-to-date information), although many could be complex queries.

Other features that distinguish between OLTP and OLAP systems include database size, frequency of operations, and performance metrics. These are summarized in Table 4.1.

4.1.3 But, Why Have a Separate Data Warehouse?

Because operational databases store huge amounts of data, you may wonder, “*Why not perform online analytical processing directly on such databases instead of spending additional time and resources to construct a separate data warehouse?*” A major reason for such a separation is to help promote the *high performance of both systems*. An operational database is designed and tuned from known tasks and workloads like indexing and hashing using primary keys, searching for particular records, and optimizing “canned” queries. On the other hand, data warehouse queries are often complex. They involve the computation of large data groups at summarized levels, and may require the use of special data organization, access, and implementation methods based on multidimensional views. Processing OLAP queries in operational databases would substantially degrade the performance of operational tasks.

Moreover, an operational database supports the concurrent processing of multiple transactions. Concurrency control and recovery mechanisms (e.g., locking and logging) are required to ensure the consistency and robustness of transactions. An OLAP query often needs read-only access of data records for summarization and aggregation. Concurrency control and recovery mechanisms, if applied for such OLAP operations, may jeopardize the execution of concurrent transactions and thus substantially reduce the throughput of an OLTP system.

Finally, the separation of operational databases from data warehouses is based on the different structures, contents, and uses of the data in these two systems. Decision

Table 4.1 Comparison of OLTP and OLAP Systems

<i>Feature</i>	<i>OLTP</i>	<i>OLAP</i>
Characteristic	operational processing	informational processing
Orientation	transaction	analysis
User	clerk, DBA, database professional	knowledge worker (e.g., manager, executive, analyst)
Function	day-to-day operations	long-term informational requirements decision support
DB design	ER-based, application-oriented	star/snowflake, subject-oriented
Data	current, guaranteed up-to-date	historic, accuracy maintained over time
Summarization	primitive, highly detailed	summarized, consolidated
View	detailed, flat relational	summarized, multidimensional
Unit of work	short, simple transaction	complex query
Access	read/write	mostly read
Focus	data in	information out
Operations	index/hash on primary key	lots of scans
Number of records accessed	tens	millions
Number of users	thousands	hundreds
DB size	GB to high-order GB	≥ TB
Priority	high performance, high availability	high flexibility, end-user autonomy
Metric	transaction throughput	query throughput, response time

Note: Table is partially based on Chaudhuri and Dayal [CD97].

support requires historic data, whereas operational databases do not typically maintain historic data. In this context, the data in operational databases, though abundant, are usually far from complete for decision making. Decision support requires consolidation (e.g., aggregation and summarization) of data from heterogeneous sources, resulting in high-quality, clean, integrated data. In contrast, operational databases contain only detailed raw data, such as transactions, which need to be consolidated before analysis. Because the two systems provide quite different functionalities and require different kinds of data, it is presently necessary to maintain separate databases. However, many vendors of operational relational database management systems are beginning to optimize such systems to support OLAP queries. As this trend continues, the separation between OLTP and OLAP systems is expected to decrease.

4.1.4 Data Warehousing: A Multitiered Architecture

Data warehouses often adopt a three-tier architecture, as presented in Figure 4.1.

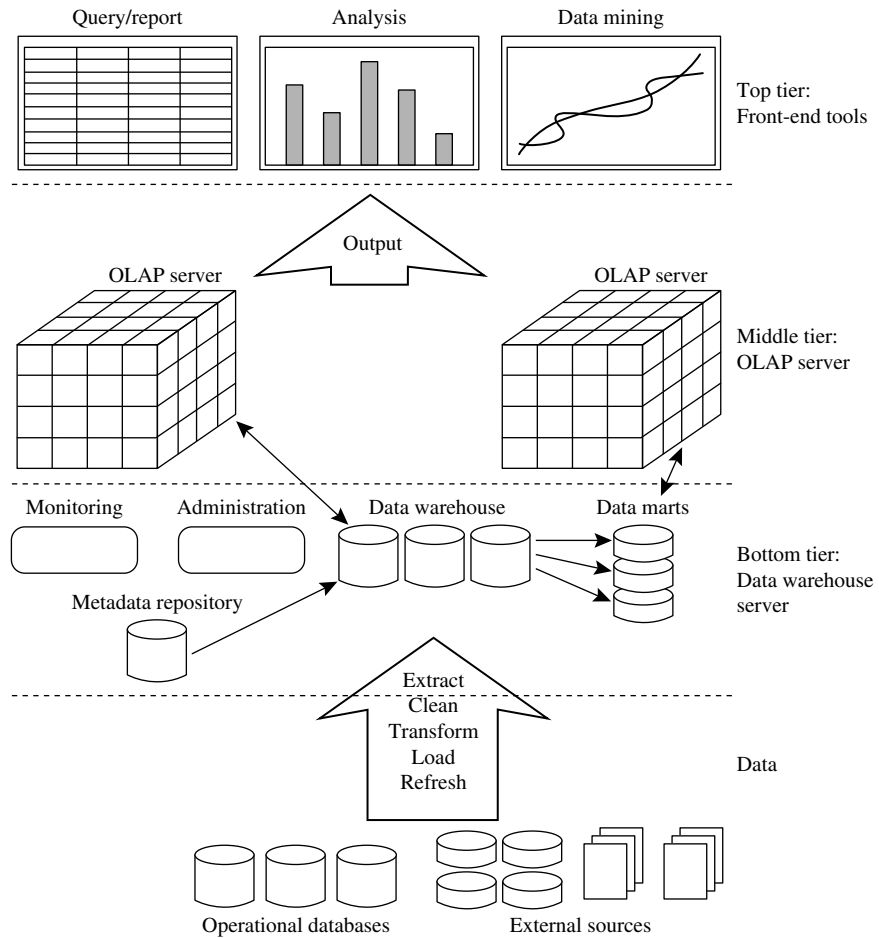


Figure 4.1 A three-tier data warehousing architecture.

1. The bottom tier is a **warehouse database server** that is almost always a relational database system. Back-end tools and utilities are used to feed data into the bottom tier from operational databases or other external sources (e.g., customer profile information provided by external consultants). These tools and utilities perform data extraction, cleaning, and transformation (e.g., to merge similar data from different sources into a unified format), as well as load and refresh functions to update the data warehouse (see Section 4.1.6). The data are extracted using application program interfaces known as **gateways**. A gateway is supported by the underlying DBMS and allows client programs to generate SQL code to be executed at a server. Examples of gateways include ODBC (Open Database Connection) and OLEDB (Object

Linking and Embedding Database) by Microsoft and JDBC (Java Database Connection). This tier also contains a metadata repository, which stores information about the data warehouse and its contents. The metadata repository is further described in Section 4.1.7.

2. The middle tier is an **OLAP server** that is typically implemented using either (1) a **relational OLAP (ROLAP)** model (i.e., an extended relational DBMS that maps operations on multidimensional data to standard relational operations); or (2) a **multi-dimensional OLAP (MOLAP)** model (i.e., a special-purpose server that directly implements multidimensional data and operations). OLAP servers are discussed in Section 4.4.4.
3. The top tier is a **front-end client layer**, which contains query and reporting tools, analysis tools, and/or data mining tools (e.g., trend analysis, prediction, and so on).

4.1.5 Data Warehouse Models: Enterprise Warehouse, Data Mart, and Virtual Warehouse

From the architecture point of view, there are three data warehouse models: the *enterprise warehouse*, the *data mart*, and the *virtual warehouse*.

Enterprise warehouse: An enterprise warehouse collects all of the information about subjects spanning the entire organization. It provides corporate-wide data integration, usually from one or more operational systems or external information providers, and is cross-functional in scope. It typically contains detailed data as well as summarized data, and can range in size from a few gigabytes to hundreds of gigabytes, terabytes, or beyond. An enterprise data warehouse may be implemented on traditional mainframes, computer superservers, or parallel architecture platforms. It requires extensive business modeling and may take years to design and build.

Data mart: A data mart contains a subset of corporate-wide data that is of value to a specific group of users. The scope is confined to specific selected subjects. For example, a marketing data mart may confine its subjects to customer, item, and sales. The data contained in data marts tend to be summarized.

Data marts are usually implemented on low-cost departmental servers that are Unix/Linux or Windows based. The implementation cycle of a data mart is more likely to be measured in weeks rather than months or years. However, it may involve complex integration in the long run if its design and planning were not enterprise-wide.

Depending on the source of data, data marts can be categorized as independent or dependent. *Independent* data marts are sourced from data captured from one or more operational systems or external information providers, or from data generated locally within a particular department or geographic area. *Dependent* data marts are sourced directly from enterprise data warehouses.

Virtual warehouse: A virtual warehouse is a set of views over operational databases. For efficient query processing, only some of the possible summary views may be materialized. A virtual warehouse is easy to build but requires excess capacity on operational database servers.

“What are the pros and cons of the top-down and bottom-up approaches to data warehouse development?” The top-down development of an enterprise warehouse serves as a systematic solution and minimizes integration problems. However, it is expensive, takes a long time to develop, and lacks flexibility due to the difficulty in achieving consistency and consensus for a common data model for the entire organization. The bottom-up approach to the design, development, and deployment of independent data marts provides flexibility, low cost, and rapid return of investment. It, however, can lead to problems when integrating various disparate data marts into a consistent enterprise data warehouse.

A recommended method for the development of data warehouse systems is to implement the warehouse in an incremental and evolutionary manner, as shown in Figure 4.2. First, a high-level corporate data model is defined within a reasonably short period (such as one or two months) that provides a corporate-wide, consistent, integrated view of data among different subjects and potential usages. This high-level model, although it will need to be refined in the further development of enterprise data warehouses and departmental data marts, will greatly reduce future integration problems. Second, independent data marts can be implemented in parallel with the enterprise

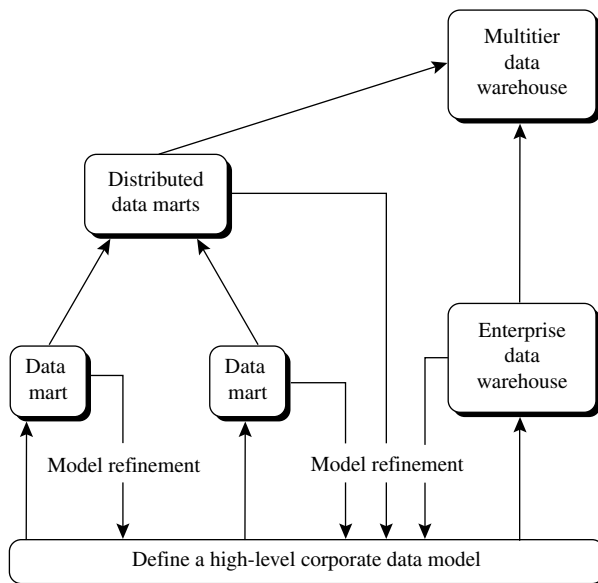


Figure 4.2 A recommended approach for data warehouse development.

warehouse based on the same corporate data model set noted before. Third, distributed data marts can be constructed to integrate different data marts via hub servers. Finally, a **multitier data warehouse** is constructed where the enterprise warehouse is the sole custodian of all warehouse data, which is then distributed to the various dependent data marts.

4.1.6 Extraction, Transformation, and Loading

Data warehouse systems use back-end tools and utilities to populate and refresh their data (Figure 4.1). These tools and utilities include the following functions:

- **Data extraction**, which typically gathers data from multiple, heterogeneous, and external sources.
- **Data cleaning**, which detects errors in the data and rectifies them when possible.
- **Data transformation**, which converts data from legacy or host format to warehouse format.
- **Load**, which sorts, summarizes, consolidates, computes views, checks integrity, and builds indices and partitions.
- **Refresh**, which propagates the updates from the data sources to the warehouse.

Besides cleaning, loading, refreshing, and metadata definition tools, data warehouse systems usually provide a good set of data warehouse management tools.

Data cleaning and data transformation are important steps in improving the data quality and, subsequently, the data mining results (see Chapter 3). Because we are mostly interested in the aspects of data warehousing technology related to data mining, we will not get into the details of the remaining tools, and recommend interested readers to consult books dedicated to data warehousing technology.

4.1.7 Metadata Repository

Metadata are data about data. When used in a data warehouse, metadata are the data that define warehouse objects. Figure 4.1 showed a metadata repository within the bottom tier of the data warehousing architecture. Metadata are created for the data names and definitions of the given warehouse. Additional metadata are created and captured for timestamping any extracted data, the source of the extracted data, and missing fields that have been added by data cleaning or integration processes.

A metadata repository should contain the following:

- A description of the *data warehouse structure*, which includes the warehouse schema, view, dimensions, hierarchies, and derived data definitions, as well as data mart locations and contents.

- *Operational metadata*, which include data lineage (history of migrated data and the sequence of transformations applied to it), currency of data (active, archived, or purged), and monitoring information (warehouse usage statistics, error reports, and audit trails).
- *The algorithms used for summarization*, which include measure and dimension definition algorithms, data on granularity, partitions, subject areas, aggregation, summarization, and predefined queries and reports.
- *Mapping from the operational environment to the data warehouse*, which includes source databases and their contents, gateway descriptions, data partitions, data extraction, cleaning, transformation rules and defaults, data refresh and purging rules, and security (user authorization and access control).
- *Data related to system performance*, which include indices and profiles that improve data access and retrieval performance, in addition to rules for the timing and scheduling of refresh, update, and replication cycles.
- *Business metadata*, which include business terms and definitions, data ownership information, and charging policies.

A data warehouse contains different levels of summarization, of which metadata is one. Other types include current detailed data (which are almost always on disk), older detailed data (which are usually on tertiary storage), lightly summarized data, and highly summarized data (which may or may not be physically housed).

Metadata play a very different role than other data warehouse data and are important for many reasons. For example, metadata are used as a directory to help the decision support system analyst locate the contents of the data warehouse, and as a guide to the data mapping when data are transformed from the operational environment to the data warehouse environment. Metadata also serve as a guide to the algorithms used for summarization between the current detailed data and the lightly summarized data, and between the lightly summarized data and the highly summarized data. Metadata should be stored and managed persistently (i.e., on disk).

4.2 Data Warehouse Modeling: Data Cube and OLAP

Data warehouses and OLAP tools are based on a **multidimensional data model**. This model views data in the form of a *data cube*. In this section, you will learn how data cubes model n -dimensional data (Section 4.2.1). In Section 4.2.2, various multidimensional models are shown: star schema, snowflake schema, and fact constellation. You will also learn about concept hierarchies (Section 4.2.3) and measures (Section 4.2.4) and how they can be used in basic OLAP operations to allow interactive mining at multiple levels of abstraction. Typical OLAP operations such as drill-down and roll-up are illustrated

(Section 4.2.5). Finally, the starnet model for querying multidimensional databases is presented (Section 4.2.6).

4.2.1 Data Cube: A Multidimensional Data Model

“What is a data cube?” A **data cube** allows data to be modeled and viewed in multiple dimensions. It is defined by dimensions and facts.

In general terms, **dimensions** are the perspectives or entities with respect to which an organization wants to keep records. For example, *AllElectronics* may create a *sales* data warehouse in order to keep records of the store’s sales with respect to the dimensions *time*, *item*, *branch*, and *location*. These dimensions allow the store to keep track of things like monthly sales of items and the branches and locations at which the items were sold. Each dimension may have a table associated with it, called a **dimension table**, which further describes the dimension. For example, a dimension table for *item* may contain the attributes *item_name*, *brand*, and *type*. Dimension tables can be specified by users or experts, or automatically generated and adjusted based on data distributions.

A multidimensional data model is typically organized around a central theme, such as *sales*. This theme is represented by a fact table. **Facts** are numeric measures. Think of them as the quantities by which we want to analyze relationships between dimensions. Examples of facts for a sales data warehouse include *dollars_sold* (sales amount in dollars), *units_sold* (number of units sold), and *amount_budgeted*. The **fact table** contains the names of the *facts*, or measures, as well as keys to each of the related dimension tables. You will soon get a clearer picture of how this works when we look at multidimensional schemas.

Although we usually think of cubes as 3-D geometric structures, in data warehousing the data cube is *n*-dimensional. To gain a better understanding of data cubes and the multidimensional data model, let’s start by looking at a simple 2-D data cube that is, in fact, a table or spreadsheet for sales data from *AllElectronics*. In particular, we will look at the *AllElectronics* sales data for items sold per quarter in the city of Vancouver. These data are shown in Table 4.2. In this 2-D representation, the sales for Vancouver are shown with respect to the *time* dimension (organized in quarters) and the *item* dimension (organized according to the types of items sold). The fact or measure displayed is *dollars_sold* (in thousands).

Now, suppose that we would like to view the sales data with a third dimension. For instance, suppose we would like to view the data according to *time* and *item*, as well as *location*, for the cities Chicago, New York, Toronto, and Vancouver. These 3-D data are shown in Table 4.3. The 3-D data in the table are represented as a series of 2-D tables. Conceptually, we may also represent the same data in the form of a 3-D data cube, as in Figure 4.3.

Suppose that we would now like to view our sales data with an additional fourth dimension such as *supplier*. Viewing things in 4-D becomes tricky. However, we can think of a 4-D cube as being a series of 3-D cubes, as shown in Figure 4.4. If we continue

Table 4.2 2-D View of Sales Data for *AllElectronics* According to *time* and *item*

<i>location</i> = "Vancouver"				
<i>time</i> (quarter)	<i>item</i> (type)			
	<i>home</i> <i>entertainment</i>	<i>computer</i>	<i>phone</i>	<i>security</i>
Q1	605	825	14	400
Q2	680	952	31	512
Q3	812	1023	30	501
Q4	927	1038	38	580

Note: The sales are from branches located in the city of Vancouver. The measure displayed is *dollars_sold* (in thousands).

Table 4.3 3-D View of Sales Data for *AllElectronics* According to *time*, *item*, and *location*

<i>location</i> = "Chicago"					<i>location</i> = "New York"				<i>location</i> = "Toronto"				<i>location</i> = "Vancouver"			
<i>time</i>	<i>item</i>				<i>item</i>				<i>item</i>				<i>item</i>			
	<i>home</i> <i>ent.</i>	<i>comp.</i>	<i>phone</i>	<i>sec.</i>	<i>home</i> <i>ent.</i>	<i>comp.</i>	<i>phone</i>	<i>sec.</i>	<i>home</i> <i>ent.</i>	<i>comp.</i>	<i>phone</i>	<i>sec.</i>	<i>home</i> <i>ent.</i>	<i>comp.</i>	<i>phone</i>	<i>sec.</i>
Q1	854	882	89	623	1087	968	38	872	818	746	43	591	605	825	14	400
Q2	943	890	64	698	1130	1024	41	925	894	769	52	682	680	952	31	512
Q3	1032	924	59	789	1034	1048	45	1002	940	795	58	728	812	1023	30	501
Q4	1129	992	63	870	1142	1091	54	984	978	864	59	784	927	1038	38	580

Note: The measure displayed is *dollars_sold* (in thousands).

in this way, we may display any n -dimensional data as a series of $(n - 1)$ -dimensional "cubes." The data cube is a metaphor for multidimensional data storage. The actual physical storage of such data may differ from its logical representation. The important thing to remember is that data cubes are n -dimensional and do not confine data to 3-D.

Tables 4.2 and 4.3 show the data at different degrees of summarization. In the data warehousing research literature, a data cube like those shown in Figures 4.3 and 4.4 is often referred to as a **cuboid**. Given a set of dimensions, we can generate a cuboid for each of the possible subsets of the given dimensions. The result would form a *lattice* of cuboids, each showing the data at a different level of summarization, or *group-by*. The lattice of cuboids is then referred to as a data cube. Figure 4.5 shows a lattice of cuboids forming a data cube for the dimensions *time*, *item*, *location*, and *supplier*.

The cuboid that holds the lowest level of summarization is called the **base cuboid**. For example, the 4-D cuboid in Figure 4.4 is the base cuboid for the given *time*, *item*, *location*, and *supplier* dimensions. Figure 4.3 is a 3-D (nonbase) cuboid for *time*, *item*,

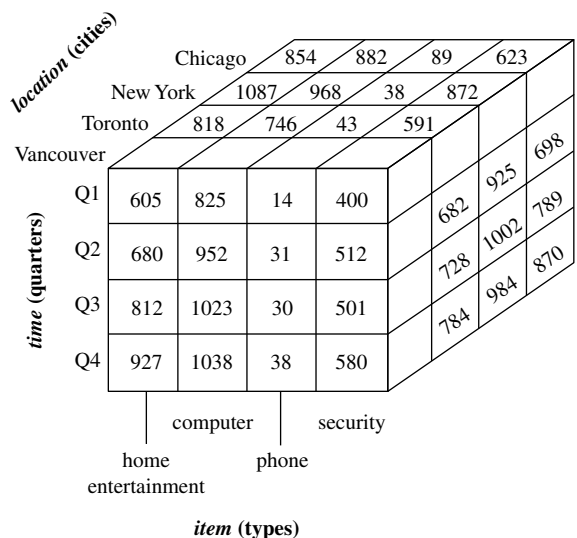


Figure 4.3 A 3-D data cube representation of the data in Table 4.3, according to *time*, *item*, and *location*. The measure displayed is *dollars_sold* (in thousands).

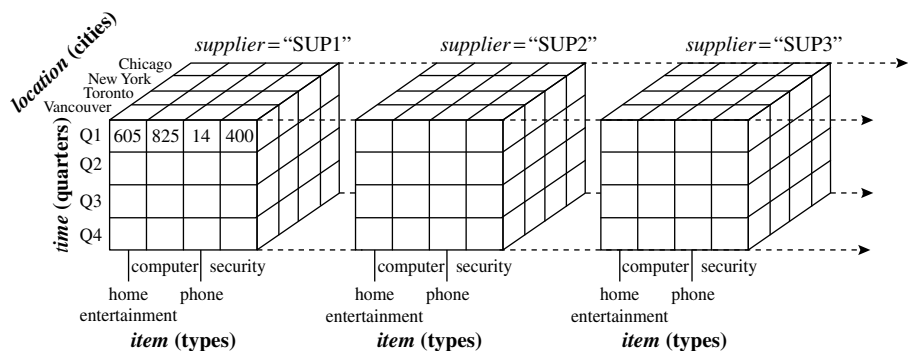


Figure 4.4 A 4-D data cube representation of sales data, according to *time*, *item*, *location*, and *supplier*. The measure displayed is *dollars_sold* (in thousands). For improved readability, only some of the cube values are shown.

and *location*, summarized for all suppliers. The 0-D cuboid, which holds the highest level of summarization, is called the **apex cuboid**. In our example, this is the total sales, or *dollars_sold*, summarized over all four dimensions. The apex cuboid is typically denoted by all.

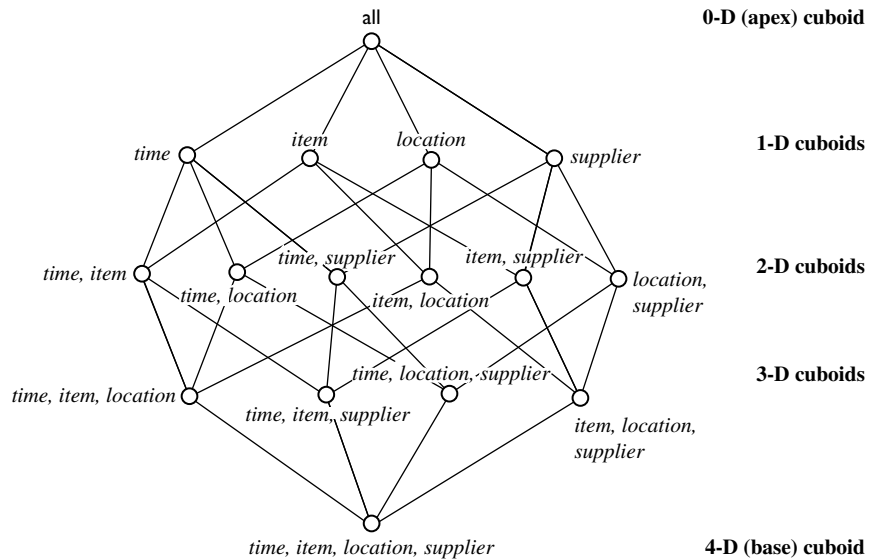


Figure 4.5 Lattice of cuboids, making up a 4-D data cube for *time*, *item*, *location*, and *supplier*. Each cuboid represents a different degree of summarization.

4.2.2 Stars, Snowflakes, and Fact Constellations: Schemas for Multidimensional Data Models

The entity-relationship data model is commonly used in the design of relational databases, where a database schema consists of a set of entities and the relationships between them. Such a data model is appropriate for online transaction processing. A data warehouse, however, requires a concise, subject-oriented schema that facilitates online data analysis.

The most popular data model for a data warehouse is a **multidimensional model**, which can exist in the form of a **star schema**, a **snowflake schema**, or a **fact constellation schema**. Let's look at each of these.

Star schema: The most common modeling paradigm is the star schema, in which the data warehouse contains (1) a large central table (**fact table**) containing the bulk of the data, with no redundancy, and (2) a set of smaller attendant tables (**dimension tables**), one for each dimension. The schema graph resembles a starburst, with the dimension tables displayed in a radial pattern around the central fact table.

Example 4.1 Star schema. A star schema for *AllElectronics* sales is shown in Figure 4.6. Sales are considered along four dimensions: *time*, *item*, *branch*, and *location*. The schema contains a central fact table for *sales* that contains keys to each of the four dimensions, along

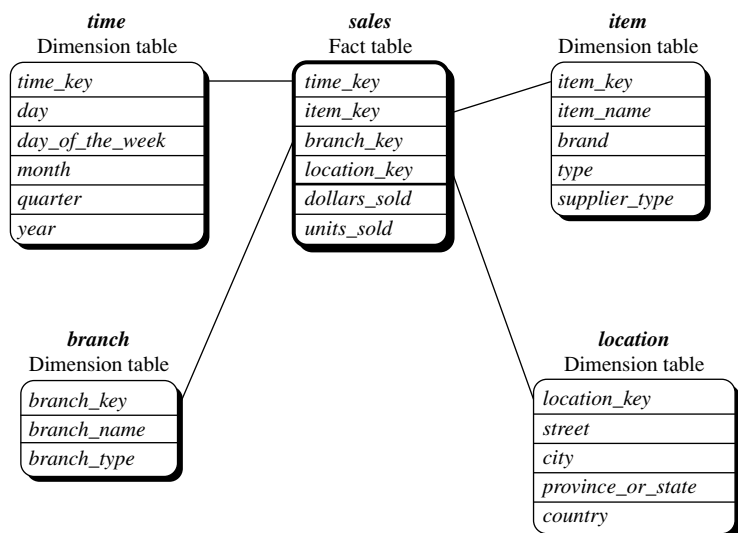


Figure 4.6 Star schema of *sales* data warehouse.

with two measures: *dollars_sold* and *units_sold*. To minimize the size of the fact table, dimension identifiers (e.g., *time_key* and *item_key*) are system-generated identifiers. ■

Notice that in the star schema, each dimension is represented by only one table, and each table contains a set of attributes. For example, the *location* dimension table contains the attribute set {*location_key*, *street*, *city*, *province_or_state*, *country*}. This constraint may introduce some redundancy. For example, “Urbana” and “Chicago” are both cities in the state of Illinois, USA. Entries for such cities in the *location* dimension table will create redundancy among the attributes *province_or_state* and *country*; that is, (... , Urbana, IL, USA) and (... , Chicago, IL, USA). Moreover, the attributes within a dimension table may form either a hierarchy (total order) or a lattice (partial order).

Snowflake schema: The snowflake schema is a variant of the star schema model, where some dimension tables are *normalized*, thereby further splitting the data into additional tables. The resulting schema graph forms a shape similar to a snowflake.

The major difference between the snowflake and star schema models is that the dimension tables of the snowflake model may be kept in normalized form to reduce redundancies. Such a table is easy to maintain and saves storage space. However, this space savings is negligible in comparison to the typical magnitude of the fact table. Furthermore, the snowflake structure can reduce the effectiveness of browsing, since more joins will be needed to execute a query. Consequently, the system performance may be adversely impacted. Hence, although the snowflake schema reduces redundancy, it is not as popular as the star schema in data warehouse design.

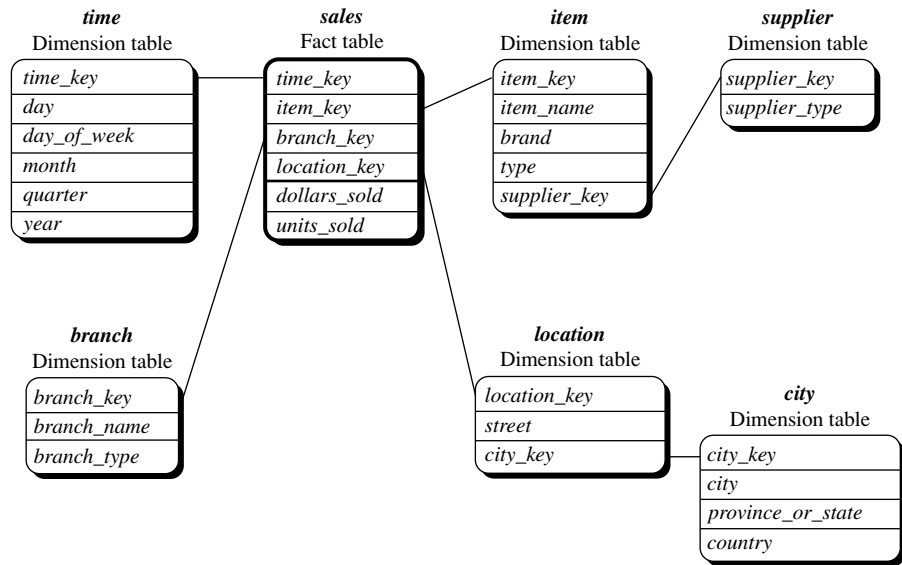


Figure 4.7 Snowflake schema of a *sales* data warehouse.

Example 4.2 Snowflake schema. A snowflake schema for *Allelectronics* sales is given in Figure 4.7. Here, the *sales* fact table is identical to that of the star schema in Figure 4.6. The main difference between the two schemas is in the definition of dimension tables. The single dimension table for *item* in the star schema is normalized in the snowflake schema, resulting in new *item* and *supplier* tables. For example, the *item* dimension table now contains the attributes *item_key*, *item_name*, *brand*, *type*, and *supplier_key*, where *supplier_key* is linked to the *supplier* dimension table, containing *supplier_key* and *supplier_type* information. Similarly, the single dimension table for *location* in the star schema can be normalized into two new tables: *location* and *city*. The *city_key* in the new *location* table links to the *city* dimension. Notice that, when desirable, further normalization can be performed on *province_or_state* and *country* in the snowflake schema shown in Figure 4.7. ■

Fact constellation: Sophisticated applications may require multiple fact tables to *share* dimension tables. This kind of schema can be viewed as a collection of stars, and hence is called a **galaxy schema** or a **fact constellation**.

Example 4.3 Fact constellation. A fact constellation schema is shown in Figure 4.8. This schema specifies two fact tables, *sales* and *shipping*. The *sales* table definition is identical to that of the star schema (Figure 4.6). The *shipping* table has five dimensions, or keys—*item_key*, *time_key*, *shipper_key*, *from_location*, and *to_location*—and two measures—*dollars_cost*

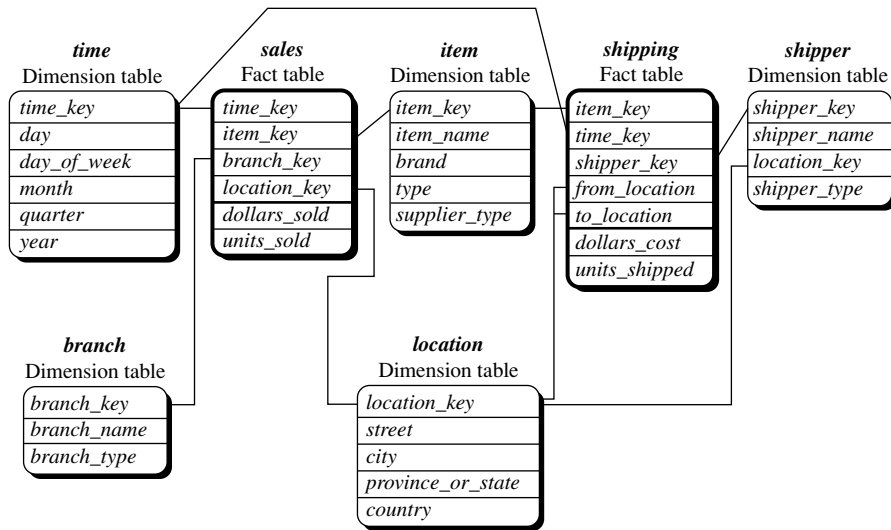


Figure 4.8 Fact constellation schema of a sales and shipping data warehouse.

and *units_shipped*. A fact constellation schema allows dimension tables to be shared between fact tables. For example, the dimensions tables for *time*, *item*, and *location* are shared between the *sales* and *shipping* fact tables. ■

In data warehousing, there is a distinction between a data warehouse and a data mart. A data warehouse collects information about subjects that span the *entire organization*, such as *customers*, *items*, *sales*, *assets*, and *personnel*, and thus its scope is *enterprise-wide*. For data warehouses, the fact constellation schema is commonly used, since it can model multiple, interrelated subjects. A **data mart**, on the other hand, is a department subset of the data warehouse that focuses on selected subjects, and thus its scope is *department-wide*. For data marts, the *star* or *snowflake* schema is commonly used, since both are geared toward modeling single subjects, although the star schema is more popular and efficient.

4.2.3 Dimensions: The Role of Concept Hierarchies

A **concept hierarchy** defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts. Consider a concept hierarchy for the dimension *location*. City values for *location* include Vancouver, Toronto, New York, and Chicago. Each city, however, can be mapped to the province or state to which it belongs. For example, Vancouver can be mapped to British Columbia, and Chicago to Illinois. The provinces and states can in turn be mapped to the country (e.g., Canada or the United States) to which they belong. These mappings form a concept hierarchy for the

dimension *location*, mapping a set of low-level concepts (i.e., cities) to higher-level, more general concepts (i.e., countries). This concept hierarchy is illustrated in Figure 4.9.

Many concept hierarchies are implicit within the database schema. For example, suppose that the dimension *location* is described by the attributes *number*, *street*, *city*, *province_or_state*, *zip_code*, and *country*. These attributes are related by a total order, forming a concept hierarchy such as “*street* < *city* < *province_or_state* < *country*.” This hierarchy is shown in Figure 4.10(a). Alternatively, the attributes of a dimension may

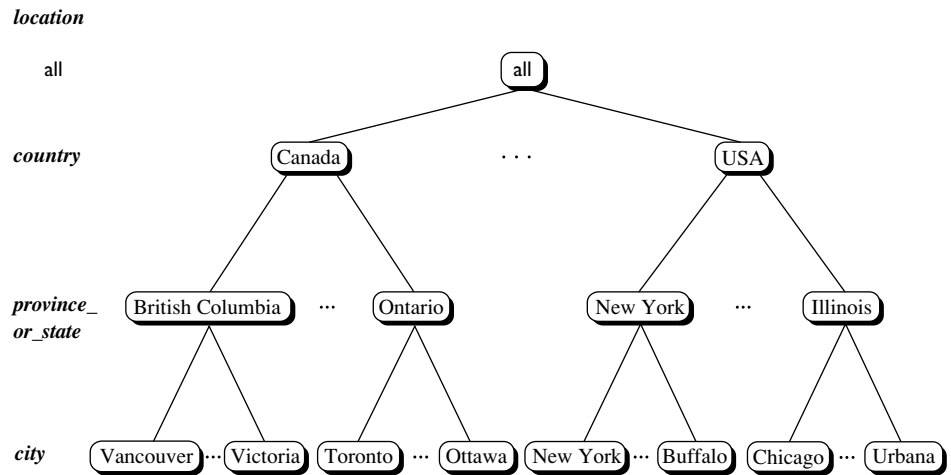


Figure 4.9 A concept hierarchy for *location*. Due to space limitations, not all of the hierarchy nodes are shown, indicated by ellipses between nodes.

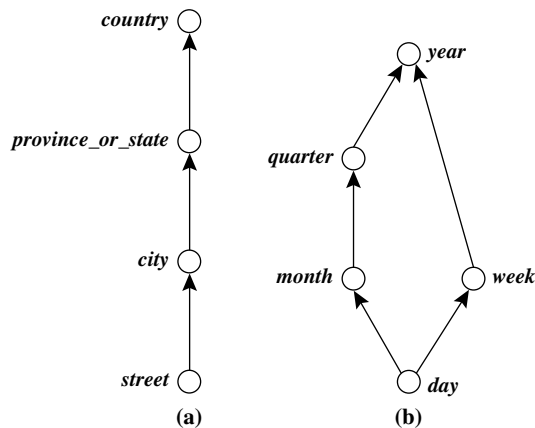


Figure 4.10 Hierarchical and lattice structures of attributes in warehouse dimensions: (a) a hierarchy for *location* and (b) a lattice for *time*.

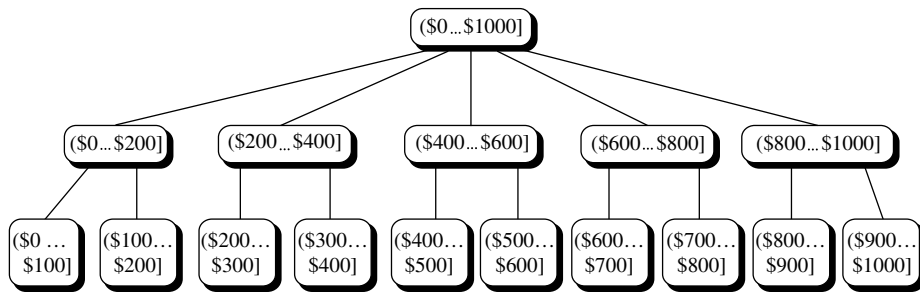


Figure 4.11 A concept hierarchy for *price*.

be organized in a partial order, forming a lattice. An example of a partial order for the *time* dimension based on the attributes *day*, *week*, *month*, *quarter*, and *year* is “*day* < {*month* < *quarter*; *week*} < *year*.”¹ This lattice structure is shown in Figure 4.10(b). A concept hierarchy that is a total or partial order among attributes in a database schema is called a **schema hierarchy**. Concept hierarchies that are common to many applications (e.g., for *time*) may be predefined in the data mining system. Data mining systems should provide users with the flexibility to tailor predefined hierarchies according to their particular needs. For example, users may want to define a fiscal year starting on April 1 or an academic year starting on September 1.

Concept hierarchies may also be defined by discretizing or grouping values for a given dimension or attribute, resulting in a **set-grouping hierarchy**. A total or partial order can be defined among groups of values. An example of a set-grouping hierarchy is shown in Figure 4.11 for the dimension *price*, where an interval $(\$X \dots \$Y]$ denotes the range from $\$X$ (exclusive) to $\$Y$ (inclusive).

There may be more than one concept hierarchy for a given attribute or dimension, based on different user viewpoints. For instance, a user may prefer to organize *price* by defining ranges for *inexpensive*, *moderately-priced*, and *expensive*.

Concept hierarchies may be provided manually by system users, domain experts, or knowledge engineers, or may be automatically generated based on statistical analysis of the data distribution. The automatic generation of concept hierarchies is discussed in Chapter 3 as a preprocessing step in preparation for data mining.

Concept hierarchies allow data to be handled at varying levels of abstraction, as we will see in Section 4.2.4.

4.2.4 Measures: Their Categorization and Computation

“How are measures computed?” To answer this question, we first study how measures can be categorized. Note that a *multidimensional point* in the data cube space can be defined

¹ Since a *week* often crosses the boundary of two consecutive months, it is usually not treated as a lower abstraction of *month*. Instead, it is often treated as a lower abstraction of *year*, since a year contains approximately 52 weeks.

by a set of dimension–value pairs; for example, (*time* = “Q1”, *location* = “Vancouver”, *item* = “computer”). A data cube **measure** is a numeric function that can be evaluated at each point in the data cube space. A measure value is computed for a given point by aggregating the data corresponding to the respective dimension–value pairs defining the given point. We will look at concrete examples of this shortly.

Measures can be organized into three categories—distributive, algebraic, and holistic—based on the kind of aggregate functions used.

Distributive: An aggregate function is *distributive* if it can be computed in a distributed manner as follows. Suppose the data are partitioned into n sets. We apply the function to each partition, resulting in n aggregate values. If the result derived by applying the function to the n aggregate values is the same as that derived by applying the function to the entire data set (without partitioning), the function can be computed in a distributed manner. For example, `sum()` can be computed for a data cube by first partitioning the cube into a set of subcubes, computing `sum()` for each subcube, and then summing up the counts obtained for each subcube. Hence, `sum()` is a distributive aggregate function.

For the same reason, `count()`, `min()`, and `max()` are distributive aggregate functions. By treating the count value of each nonempty base cell as 1 by default, `count()` of any cell in a cube can be viewed as the sum of the count values of all of its corresponding child cells in its subcube. Thus, `count()` is distributive. A measure is *distributive* if it is obtained by applying a distributive aggregate function. Distributive measures can be computed efficiently because of the way the computation can be partitioned.

Algebraic: An aggregate function is *algebraic* if it can be computed by an algebraic function with M arguments (where M is a bounded positive integer), each of which is obtained by applying a distributive aggregate function. For example, `avg()` (average) can be computed by `sum()/count()`, where both `sum()` and `count()` are distributive aggregate functions. Similarly, it can be shown that `min.N()` and `max.N()` (which find the N minimum and N maximum values, respectively, in a given set) and `standard_deviation()` are algebraic aggregate functions. A measure is *algebraic* if it is obtained by applying an algebraic aggregate function.

Holistic: An aggregate function is *holistic* if there is no constant bound on the storage size needed to describe a subaggregate. That is, there does not exist an algebraic function with M arguments (where M is a constant) that characterizes the computation. Common examples of holistic functions include `median()`, `mode()`, and `rank()`. A measure is *holistic* if it is obtained by applying a holistic aggregate function.

Most large data cube applications require efficient computation of distributive and algebraic measures. Many efficient techniques for this exist. In contrast, it is difficult to compute holistic measures efficiently. Efficient techniques to *approximate* the computation of some holistic measures, however, do exist. For example, rather than computing the exact `median()`, Equation (2.3) of Chapter 2 can be used to estimate the approximate median value for a large data set. In many cases, such techniques are sufficient to overcome the difficulties of efficient computation of holistic measures.

Various methods for computing different measures in data cube construction are discussed in depth in Chapter 5. Notice that most of the current data cube technology confines the measures of multidimensional databases to *numeric data*. However, measures can also be applied to other kinds of data, such as spatial, multimedia, or text data.

4.2.5 Typical OLAP Operations

“How are concept hierarchies useful in OLAP?” In the multidimensional model, data are organized into multiple dimensions, and each dimension contains multiple levels of abstraction defined by concept hierarchies. This organization provides users with the flexibility to view data from different perspectives. A number of OLAP data cube operations exist to materialize these different views, allowing interactive querying and analysis of the data at hand. Hence, OLAP provides a user-friendly environment for interactive data analysis.

Example 4.4 OLAP operations. Let’s look at some typical OLAP operations for multidimensional data. Each of the following operations described is illustrated in Figure 4.12. At the center of the figure is a data cube for *Allelectronics* sales. The cube contains the dimensions *location*, *time*, and *item*, where *location* is aggregated with respect to city values, *time* is aggregated with respect to quarters, and *item* is aggregated with respect to item types. To aid in our explanation, we refer to this cube as the central cube. The measure displayed is *dollars_sold* (in thousands). (For improved readability, only some of the cubes’ cell values are shown.) The data examined are for the cities Chicago, New York, Toronto, and Vancouver.

Roll-up: The roll-up operation (also called the *drill-up* operation by some vendors) performs aggregation on a data cube, either by *climbing up a concept hierarchy* for a dimension or by *dimension reduction*. Figure 4.12 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for *location* given in Figure 4.9. This hierarchy was defined as the total order “*street* < *city* < *province_or_state* < *country*.” The roll-up operation shown aggregates the data by ascending the *location* hierarchy from the level of *city* to the level of *country*. In other words, rather than grouping the data by city, the resulting cube groups the data by country.

When roll-up is performed by dimension reduction, one or more dimensions are removed from the given cube. For example, consider a sales data cube containing only the *location* and *time* dimensions. Roll-up may be performed by removing, say, the *time* dimension, resulting in an aggregation of the total sales by location, rather than by location and by time.

Drill-down: Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be realized by either *stepping down a concept hierarchy* for a dimension or *introducing additional dimensions*. Figure 4.12 shows the result of a drill-down operation performed on the central cube by stepping down a

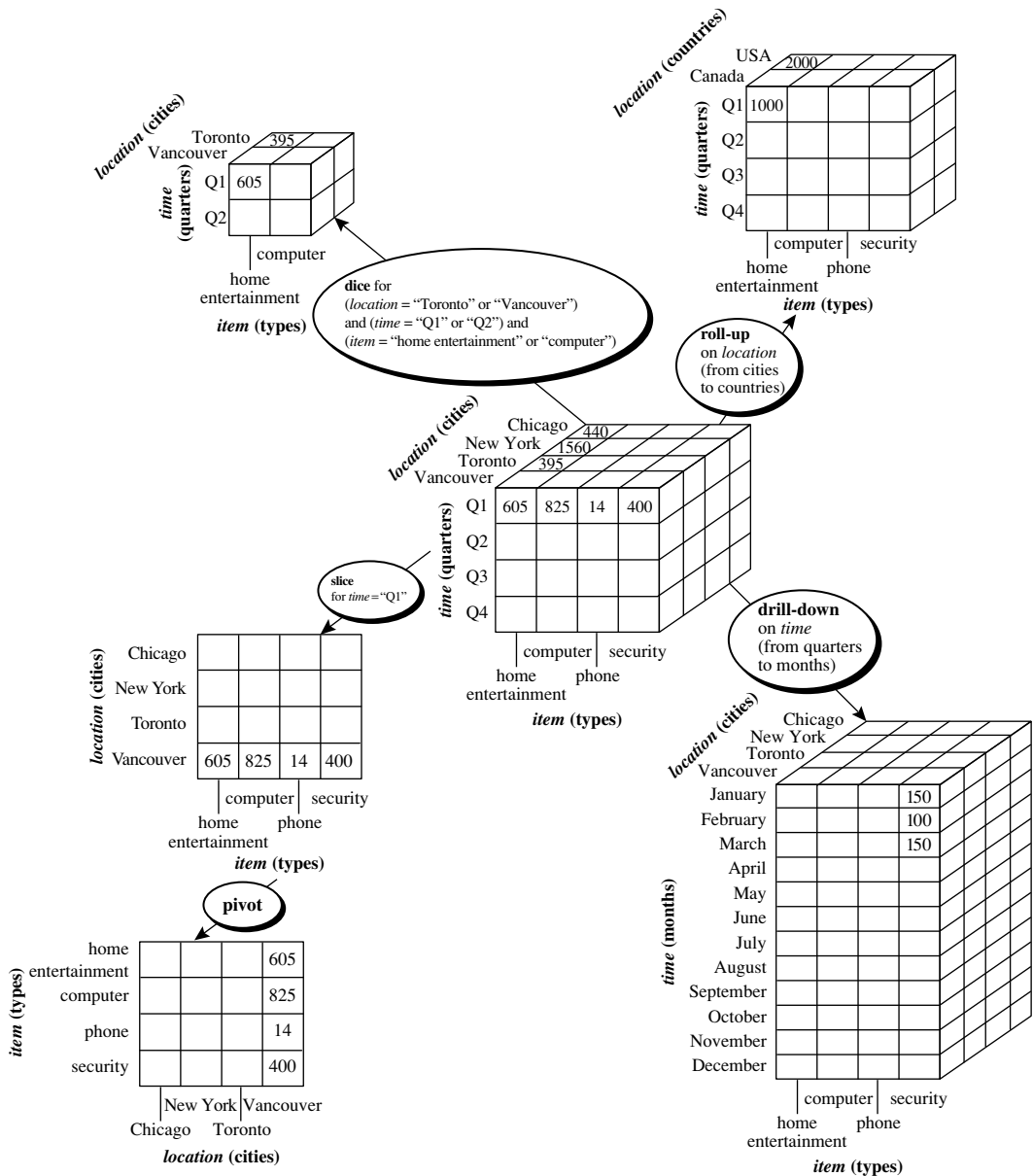


Figure 4.12 Examples of typical OLAP operations on multidimensional data.

concept hierarchy for *time* defined as “*day < month < quarter < year*.” Drill-down occurs by descending the *time* hierarchy from the level of *quarter* to the more detailed level of *month*. The resulting data cube details the total sales per month rather than summarizing them by quarter.

Because a drill-down adds more detail to the given data, it can also be performed by adding new dimensions to a cube. For example, a drill-down on the central cube of Figure 4.12 can occur by introducing an additional dimension, such as *customer_group*.

Slice and dice: The *slice* operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 4.12 shows a slice operation where the sales data are selected from the central cube for the dimension *time* using the criterion *time* = “Q1.” The *dice* operation defines a subcube by performing a selection on two or more dimensions. Figure 4.12 shows a dice operation on the central cube based on the following selection criteria that involve three dimensions: (*location* = “Toronto” or “Vancouver”) and (*time* = “Q1” or “Q2”) and (*item* = “home entertainment” or “computer”).

Pivot (rotate): *Pivot* (also called *rotate*) is a visualization operation that rotates the data axes in view to provide an alternative data presentation. Figure 4.12 shows a pivot operation where the *item* and *location* axes in a 2-D slice are rotated. Other examples include rotating the axes in a 3-D cube, or transforming a 3-D cube into a series of 2-D planes.

Other OLAP operations: Some OLAP systems offer additional drilling operations. For example, **drill-across** executes queries involving (i.e., across) more than one fact table. The **drill-through** operation uses relational SQL facilities to drill through the bottom level of a data cube down to its back-end relational tables.

Other OLAP operations may include ranking the top *N* or bottom *N* items in lists, as well as computing moving averages, growth rates, interests, internal return rates, depreciation, currency conversions, and statistical functions. ■

OLAP offers analytical modeling capabilities, including a calculation engine for deriving ratios, variance, and so on, and for computing measures across multiple dimensions. It can generate summarizations, aggregations, and hierarchies at each granularity level and at every dimension intersection. OLAP also supports functional models for forecasting, trend analysis, and statistical analysis. In this context, an OLAP engine is a powerful data analysis tool.

OLAP Systems versus Statistical Databases

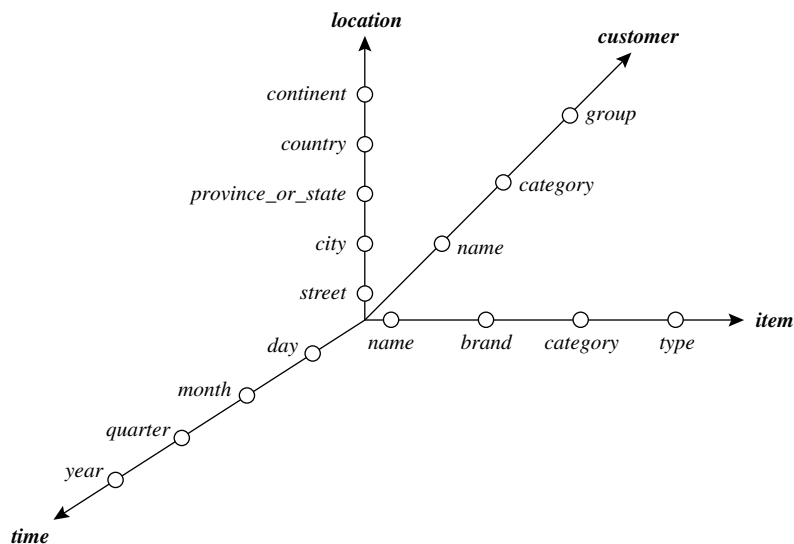
Many OLAP systems’ characteristics (e.g., the use of a multidimensional data model and concept hierarchies, the association of measures with dimensions, and the notions of roll-up and drill-down) also exist in earlier work on statistical databases (SDBs). A **statistical database** is a database system that is designed to support statistical applications. Similarities between the two types of systems are rarely discussed, mainly due to differences in terminology and application domains.

OLAP and SDB systems, however, have distinguishing differences. While SDBs tend to focus on socioeconomic applications, OLAP has been targeted for business applications. Privacy issues regarding concept hierarchies are a major concern for SDBs. For example, given summarized socioeconomic data, it is controversial to allow users to view the corresponding low-level data. Finally, unlike SDBs, OLAP systems are designed for efficiently handling huge amounts of data.

4.2.6 A Starnet Query Model for Querying Multidimensional Databases

The querying of multidimensional databases can be based on a **starnet model**, which consists of radial lines emanating from a central point, where each line represents a concept hierarchy for a dimension. Each abstraction level in the hierarchy is called a **footprint**. These represent the granularities available for use by OLAP operations such as drill-down and roll-up.

Example 4.5 Starnet. A starnet query model for the *AllElectronics* data warehouse is shown in Figure 4.13. This starnet consists of four radial lines, representing concept hierarchies for the dimensions *location*, *customer*, *item*, and *time*, respectively. Each line consists of footprints representing abstraction levels of the dimension. For example, the *time* line has four footprints: “day,” “month,” “quarter,” and “year.” A concept hierarchy may involve a single attribute (e.g., *date* for the *time* hierarchy) or several attributes (e.g., the concept hierarchy for *location* involves the attributes *street*, *city*, *province_or_state*, and *country*). In order to examine the item sales at *AllElectronics*, users can roll up along the



time dimension from *month* to *quarter*, or, say, drill down along the *location* dimension from *country* to *city*.

Concept hierarchies can be used to **generalize** data by replacing low-level values (such as “day” for the *time* dimension) by higher-level abstractions (such as “year”), or to **specialize** data by replacing higher-level abstractions with lower-level values. ■

4.3 Data Warehouse Design and Usage

“What goes into a data warehouse design? How are data warehouses used? How do data warehousing and OLAP relate to data mining?” This section tackles these questions. We study the design and usage of data warehousing for information processing, analytical processing, and data mining. We begin by presenting a business analysis framework for data warehouse design (Section 4.3.1). Section 4.3.2 looks at the design process, while Section 4.3.3 studies data warehouse usage. Finally, Section 4.3.4 describes *multi-dimensional data mining*, a powerful paradigm that integrates OLAP with data mining technology.

4.3.1 A Business Analysis Framework for Data Warehouse Design

“What can business analysts gain from having a data warehouse?” First, having a data warehouse may provide a *competitive advantage* by presenting relevant information from which to measure performance and make critical adjustments to help win over competitors. Second, a data warehouse can enhance business *productivity* because it is able to quickly and efficiently gather information that accurately describes the organization. Third, a data warehouse facilitates *customer relationship management* because it provides a consistent view of customers and items across all lines of business, all departments, and all markets. Finally, a data warehouse may bring about *cost reduction* by tracking trends, patterns, and exceptions over long periods in a consistent and reliable manner.

To design an effective data warehouse we need to understand and analyze business needs and construct a *business analysis framework*. The construction of a large and complex information system can be viewed as the construction of a large and complex building, for which the owner, architect, and builder have different views. These views are combined to form a complex framework that represents the top-down, business-driven, or owner’s perspective, as well as the bottom-up, builder-driven, or implementor’s view of the information system.

Four different views regarding a data warehouse design must be considered: the *top-down view*, the *data source view*, the *data warehouse view*, and the *business query view*.

- The **top-down view** allows the selection of the relevant information necessary for the data warehouse. This information matches current and future business needs.
- The **data source view** exposes the information being captured, stored, and managed by operational systems. This information may be documented at various levels of detail and accuracy, from individual data source tables to integrated data source tables. Data sources are often modeled by traditional data modeling techniques, such as the entity-relationship model or CASE (computer-aided software engineering) tools.
- The **data warehouse view** includes fact tables and dimension tables. It represents the information that is stored inside the data warehouse, including precalculated totals and counts, as well as information regarding the source, date, and time of origin, added to provide historical context.
- Finally, the **business query view** is the data perspective in the data warehouse from the end-user's viewpoint.

Building and using a data warehouse is a complex task because it requires *business skills*, *technology skills*, and *program management skills*. Regarding *business skills*, building a data warehouse involves understanding how systems store and manage their data, how to build **extractors** that transfer data from the operational system to the data warehouse, and how to build **warehouse refresh software** that keeps the data warehouse reasonably up-to-date with the operational system's data. Using a data warehouse involves understanding the significance of the data it contains, as well as understanding and translating the business requirements into queries that can be satisfied by the data warehouse.

Regarding *technology skills*, data analysts are required to understand how to make assessments from quantitative information and derive facts based on conclusions from historic information in the data warehouse. These skills include the ability to discover patterns and trends, to extrapolate trends based on history and look for anomalies or paradigm shifts, and to present coherent managerial recommendations based on such analysis. Finally, *program management skills* involve the need to interface with many technologies, vendors, and end-users in order to deliver results in a timely and cost-effective manner.

4.3.2 Data Warehouse Design Process

Let's look at various approaches to the data warehouse design process and the steps involved.

A data warehouse can be built using a *top-down approach*, a *bottom-up approach*, or a *combination of both*. The **top-down approach** starts with overall design and planning. It is useful in cases where the technology is mature and well known, and where the business problems that must be solved are clear and well understood. The **bottom-up approach** starts with experiments and prototypes. This is useful in the early stage of business modeling and technology development. It allows an organization to move

forward at considerably less expense and to evaluate the technological benefits before making significant commitments. In the **combined approach**, an organization can exploit the planned and strategic nature of the top-down approach while retaining the rapid implementation and opportunistic application of the bottom-up approach.

From the software engineering point of view, the design and construction of a data warehouse may consist of the following steps: *planning, requirements study, problem analysis, warehouse design, data integration and testing*, and finally *deployment of the data warehouse*. Large software systems can be developed using one of two methodologies: the *waterfall method* or the *spiral method*. The **waterfall method** performs a structured and systematic analysis at each step before proceeding to the next, which is like a waterfall, falling from one step to the next. The **spiral method** involves the rapid generation of increasingly functional systems, with short intervals between successive releases. This is considered a good choice for data warehouse development, especially for data marts, because the turnaround time is short, modifications can be done quickly, and new designs and technologies can be adapted in a timely manner.

In general, the warehouse design process consists of the following steps:

1. Choose a *business process* to model (e.g., orders, invoices, shipments, inventory, account administration, sales, or the general ledger). If the business process is organizational and involves multiple complex object collections, a data warehouse model should be followed. However, if the process is departmental and focuses on the analysis of one kind of business process, a data mart model should be chosen.
2. Choose the business process *grain*, which is the fundamental, atomic level of data to be represented in the fact table for this process (e.g., individual transactions, individual daily snapshots, and so on).
3. Choose the *dimensions* that will apply to each fact table record. Typical dimensions are time, item, customer, supplier, warehouse, transaction type, and status.
4. Choose the *measures* that will populate each fact table record. Typical measures are numeric additive quantities like *dollars_sold* and *units_sold*.

Because data warehouse construction is a difficult and long-term task, its implementation scope should be clearly defined. The goals of an initial data warehouse implementation should be *specific, achievable, and measurable*. This involves determining the time and budget allocations, the subset of the organization that is to be modeled, the number of data sources selected, and the number and types of departments to be served.

Once a data warehouse is designed and constructed, the initial deployment of the warehouse includes initial installation, roll-out planning, training, and orientation. Platform upgrades and maintenance must also be considered. Data warehouse administration includes data refreshment, data source synchronization, planning for disaster recovery, managing access control and security, managing data growth, managing database performance, and data warehouse enhancement and extension. Scope

management includes controlling the number and range of queries, dimensions, and reports; limiting the data warehouse's size; or limiting the schedule, budget, or resources.

Various kinds of data warehouse design tools are available. **Data warehouse development tools** provide functions to define and edit metadata repository contents (e.g., schemas, scripts, or rules), answer queries, output reports, and ship metadata to and from relational database system catalogs. **Planning and analysis tools** study the impact of schema changes and of refresh performance when changing refresh rates or time windows.

4.3.3 Data Warehouse Usage for Information Processing

Data warehouses and data marts are used in a wide range of applications. Business executives use the data in data warehouses and data marts to perform data analysis and make strategic decisions. In many firms, data warehouses are used as an integral part of a *plan-execute-assess* “closed-loop” feedback system for enterprise management. Data warehouses are used extensively in banking and financial services, consumer goods and retail distribution sectors, and controlled manufacturing such as demand-based production.

Typically, the longer a data warehouse has been in use, the more it will have evolved. This evolution takes place throughout a number of phases. Initially, the data warehouse is mainly used for generating reports and answering predefined queries. Progressively, it is used to analyze summarized and detailed data, where the results are presented in the form of reports and charts. Later, the data warehouse is used for strategic purposes, performing multidimensional analysis and sophisticated slice-and-dice operations. Finally, the data warehouse may be employed for knowledge discovery and strategic decision making using data mining tools. In this context, the tools for data warehousing can be categorized into *access and retrieval tools*, *database reporting tools*, *data analysis tools*, and *data mining tools*.

Business users need to have the means to know what exists in the data warehouse (through metadata), how to access the contents of the data warehouse, how to examine the contents using analysis tools, and how to present the results of such analysis.

There are three kinds of data warehouse applications: *information processing*, *analytical processing*, and *data mining*.

- **Information processing** supports querying, basic statistical analysis, and reporting using crosstabs, tables, charts, or graphs. A current trend in data warehouse information processing is to construct low-cost web-based accessing tools that are then integrated with web browsers.
- **Analytical processing** supports basic OLAP operations, including slice-and-dice, drill-down, roll-up, and pivoting. It generally operates on historic data in both summarized and detailed forms. The major strength of online analytical processing over information processing is the multidimensional data analysis of data warehouse data.

- **Data mining** supports knowledge discovery by finding hidden patterns and associations, constructing analytical models, performing classification and prediction, and presenting the mining results using visualization tools.

“How does data mining relate to information processing and online analytical processing?” Information processing, based on queries, can find useful information. However, answers to such queries reflect the information directly stored in databases or computable by aggregate functions. They do not reflect sophisticated patterns or regularities buried in the database. Therefore, information processing is not data mining.

Online analytical processing comes a step closer to data mining because it can derive information summarized at multiple granularities from user-specified subsets of a data warehouse. Such descriptions are equivalent to the class/concept descriptions discussed in Chapter 1. Because data mining systems can also mine generalized class/concept descriptions, this raises some interesting questions: *“Do OLAP systems perform data mining? Are OLAP systems actually data mining systems?”*

The functionalities of OLAP and data mining can be viewed as disjoint: OLAP is a data summarization/aggregation *tool* that helps simplify data analysis, while data mining allows the *automated discovery* of implicit patterns and interesting knowledge hidden in large amounts of data. OLAP tools are targeted toward simplifying and supporting interactive data analysis, whereas the goal of data mining tools is to automate as much of the process as possible, while still allowing users to guide the process. In this sense, data mining goes one step beyond traditional online analytical processing.

An alternative and broader view of data mining may be adopted in which data mining covers both data description and data modeling. Because OLAP systems can present general descriptions of data from data warehouses, OLAP functions are essentially for user-directed data summarization and comparison (by drilling, pivoting, slicing, dicing, and other operations). These are, though limited, data mining functionalities. Yet according to this view, data mining covers a much broader spectrum than simple OLAP operations, because it performs not only data summarization and comparison but also association, classification, prediction, clustering, time-series analysis, and other data analysis tasks.

Data mining is not confined to the analysis of data stored in data warehouses. It may analyze data existing at more detailed granularities than the summarized data provided in a data warehouse. It may also analyze transactional, spatial, textual, and multimedia data that are difficult to model with current multidimensional database technology. In this context, data mining covers a broader spectrum than OLAP with respect to data mining functionality and the complexity of the data handled.

Because data mining involves more automated and deeper analysis than OLAP, it is expected to have broader applications. Data mining can help business managers find and reach more suitable customers, as well as gain critical business insights that may help drive market share and raise profits. In addition, data mining can help managers understand customer group characteristics and develop optimal pricing strategies accordingly. It can correct item bundling based not on intuition but on actual item groups derived from customer purchase patterns, reduce promotional spending, and at the same time increase the overall net effectiveness of promotions.

4.3.4 From Online Analytical Processing to Multidimensional Data Mining

The data mining field has conducted substantial research regarding mining on various data types, including relational data, data from data warehouses, transaction data, time-series data, spatial data, text data, and flat files. **Multidimensional data mining** (also known as *exploratory multidimensional data mining*, **online analytical mining**, or **OLAM**) integrates OLAP with data mining to uncover knowledge in multidimensional databases. Among the many different paradigms and architectures of data mining systems, multidimensional data mining is particularly important for the following reasons:

- **High quality of data in data warehouses:** Most data mining tools need to work on integrated, consistent, and cleaned data, which requires costly data cleaning, data integration, and data transformation as preprocessing steps. A data warehouse constructed by such preprocessing serves as a valuable source of high-quality data for OLAP as well as for data mining. Notice that data mining may serve as a valuable tool for data cleaning and data integration as well.
- **Available information processing infrastructure surrounding data warehouses:** Comprehensive information processing and data analysis infrastructures have been or will be systematically constructed surrounding data warehouses, which include accessing, integration, consolidation, and transformation of multiple heterogeneous databases, ODBC/OLEDB connections, Web accessing and service facilities, and reporting and OLAP analysis tools. It is prudent to make the best use of the available infrastructures rather than constructing everything from scratch.
- **OLAP-based exploration of multidimensional data:** Effective data mining needs exploratory data analysis. A user will often want to traverse through a database, select portions of relevant data, analyze them at different granularities, and present knowledge/results in different forms. Multidimensional data mining provides facilities for mining on different subsets of data and at varying levels of abstraction—by drilling, pivoting, filtering, dicing, and slicing on a data cube and/or intermediate data mining results. This, together with data/knowledge visualization tools, greatly enhances the power and flexibility of data mining.
- **Online selection of data mining functions:** Users may not always know the specific kinds of knowledge they want to mine. By integrating OLAP with various data mining functions, multidimensional data mining provides users with the flexibility to select desired data mining functions and swap data mining tasks dynamically.

Chapter 5 describes data warehouses on a finer level by exploring implementation issues such as data cube computation, OLAP query answering strategies, and multidimensional data mining. The chapters following it are devoted to the study of data mining techniques. As we have seen, the introduction to data warehousing and OLAP technology presented in this chapter is essential to our study of data mining. This is because data warehousing provides users with large amounts of clean, organized,

and summarized data, which greatly facilitates data mining. For example, rather than storing the details of each sales transaction, a data warehouse may store a summary of the transactions per item type for each branch or, summarized to a higher level, for each country. The capability of OLAP to provide multiple and dynamic views of summarized data in a data warehouse sets a solid foundation for successful data mining.

Moreover, we also believe that data mining should be a human-centered process. Rather than asking a data mining system to generate patterns and knowledge automatically, a user will often need to interact with the system to perform exploratory data analysis. OLAP sets a good example for interactive data analysis and provides the necessary preparations for exploratory data mining. Consider the discovery of association patterns, for example. Instead of mining associations at a primitive (i.e., low) data level among transactions, users should be allowed to specify roll-up operations along any dimension.

For example, a user may want to roll up on the *item* dimension to go from viewing the data for particular TV sets that were purchased to viewing the brands of these TVs (e.g., SONY or Toshiba). Users may also navigate from the transaction level to the customer or customer-type level in the search for interesting associations. Such an OLAP data mining style is characteristic of multidimensional data mining. In our study of the principles of data mining in this book, we place particular emphasis on multidimensional data mining, that is, on the *integration of data mining and OLAP technology*.

4.4 Data Warehouse Implementation

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data warehouse systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of methods for the efficient implementation of data warehouse systems. Section 4.4.1 explores how to compute data cubes efficiently. Section 4.4.2 shows how OLAP data can be indexed, using either bitmap or join indices. Next, we study how OLAP queries are processed (Section 4.4.3). Finally, Section 4.4.4 presents various types of warehouse servers for OLAP processing.

4.4.1 Efficient Data Cube Computation: An Overview

At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions. In SQL terms, these aggregations are referred to as *group-by*'s. Each *group-by* can be represented by a *cuboid*, where the set of *group-by*'s forms a lattice of cuboids defining a data cube. In this subsection, we explore issues relating to the efficient computation of data cubes.

The compute cube Operator and the Curse of Dimensionality

One approach to cube computation extends SQL so as to include a **compute cube** operator. The **compute cube** operator computes aggregates over all subsets of the dimensions specified in the operation. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

Example 4.6 A data cube is a lattice of cuboids. Suppose that you want to create a data cube for *AllElectronics* sales that contains the following: *city*, *item*, *year*, and *sales_in_dollars*. You want to be able to analyze the data, with queries such as the following:

- “Compute the sum of sales, grouping by city and item.”
- “Compute the sum of sales, grouping by city.”
- “Compute the sum of sales, grouping by item.”

What is the total number of cuboids, or group-by’s, that can be computed for this data cube? Taking the three attributes, *city*, *item*, and *year*, as the dimensions for the data cube, and *sales_in_dollars* as the measure, the total number of cuboids, or group-by’s, that can be computed for this data cube is $2^3 = 8$. The possible group-by’s are the following: $\{(city, item, year), (city, item), (city, year), (item, year), (city), (item), (year), ()\}$, where $()$ means that the group-by is empty (i.e., the dimensions are not grouped). These group-by’s form a lattice of cuboids for the data cube, as shown in Figure 4.14.

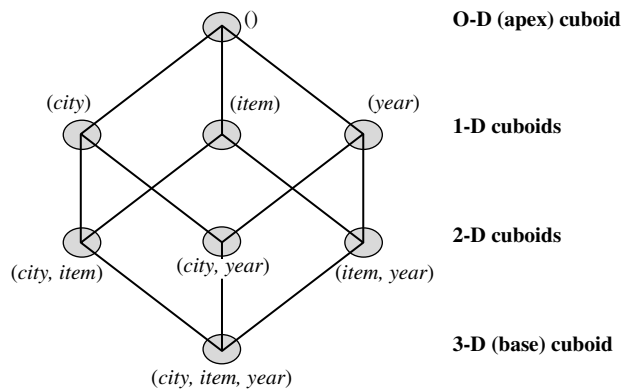


Figure 4.14 Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains *city*, *item*, and *year* dimensions.

The **base cuboid** contains all three dimensions, *city*, *item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as all. If we start at the apex cuboid and explore downward in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upward, this is akin to rolling up. ■

An SQL query containing no group-by (e.g., “*compute the sum of total sales*”) is a *zero-dimensional operation*. An SQL query containing one group-by (e.g., “*compute the sum of sales, group-by city*”) is a *one-dimensional operation*. A cube operator on n dimensions is equivalent to a collection of group-by statements, one for each subset of the n dimensions. Therefore, the cube operator is the n -dimensional generalization of the group-by operator.

Similar to the SQL syntax, the data cube in Example 4.1 could be defined as

```
define cube sales_cube [city, item, year]: sum(sales.in.dollars)
```

For a cube with n dimensions, there are a total of 2^n cuboids, including the base cuboid. A statement such as

```
compute cube sales_cube
```

would explicitly instruct the system to compute the sales aggregate cuboids for all eight subsets of the set $\{city, item, year\}$, including the empty subset. A cube computation operator was first proposed and studied by Gray et al. [GCB⁺97].

Online analytical processing may need to access different cuboids for different queries. Therefore, it may seem like a good idea to compute in advance all or at least some of the cuboids in a data cube. Precomputation leads to fast response time and avoids some redundant computation. Most, if not all, OLAP products resort to some degree of precomputation of multidimensional aggregates.

A major challenge related to this precomputation, however, is that the required storage space may explode if all the cuboids in a data cube are precomputed, especially when the cube has many dimensions. The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels. This problem is referred to as the **curse of dimensionality**. The extent of the curse of dimensionality is illustrated here.

“*How many cuboids are there in an n -dimensional data cube?*” If there were no hierarchies associated with each dimension, then the total number of cuboids for an n -dimensional data cube, as we have seen, is 2^n . However, in practice, many dimensions do have hierarchies. For example, *time* is usually explored not at only one conceptual level (e.g., *year*), but rather at multiple conceptual levels such as in the hierarchy “*day < month < quarter < year*.” For an n -dimensional data cube, the total number of cuboids

that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$\text{Total number of cuboids} = \prod_{i=1}^n (L_i + 1), \quad (4.1)$$

where L_i is the number of levels associated with dimension i . One is added to L_i in Eq. (4.1) to include the *virtual* top level, all. (Note that generalizing to all is equivalent to the removal of the dimension.)

This formula is based on the fact that, at most, one abstraction level in each dimension will appear in a cuboid. For example, the time dimension as specified before has four conceptual levels, or five if we include the virtual level all. If the cube has 10 dimensions and each dimension has five levels (including all), the total number of cuboids that can be generated is $5^{10} \approx 9.8 \times 10^6$. The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if the *All-Electronics* branch in each city sold every item, there would be $|city| \times |item|$ tuples in the *city_item* group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (fixed) size of the input relation.

By now, you probably realize that it is unrealistic to precompute and materialize all of the cuboids that can possibly be generated for a data cube (i.e., from a base cuboid). If there are many cuboids, and these cuboids are large in size, a more reasonable option is *partial materialization*; that is, to materialize only *some* of the possible cuboids that can be generated.

Partial Materialization: Selected Computation of Cuboids

There are three choices for data cube materialization given a base cuboid:

1. **No materialization:** Do not precompute any of the “nonbase” cuboids. This leads to computing expensive multidimensional aggregates on-the-fly, which can be extremely slow.
2. **Full materialization:** Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube*. This choice typically requires huge amounts of memory space in order to store all of the precomputed cuboids.
3. **Partial materialization:** Selectively compute a proper subset of the whole set of possible cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where the tuple count of each cell is above some threshold. We will use the term *subcube* to refer to the latter case, where only some of the cells may be precomputed for various cuboids. Partial materialization represents an interesting trade-off between storage space and response time.

The partial materialization of cuboids or subcubes should consider three factors: (1) identify the subset of cuboids or subcubes to materialize; (2) exploit the materialized cuboids or subcubes during query processing; and (3) efficiently update the materialized cuboids or subcubes during load and refresh.

The selection of the subset of cuboids or subcubes to materialize should take into account the queries in the workload, their frequencies, and their accessing costs. In addition, it should consider workload characteristics, the cost for incremental updates, and the total storage requirements. The selection must also consider the broad context of physical database design such as the generation and selection of indices. Several OLAP products have adopted heuristic approaches for cuboid and subcube selection. A popular approach is to materialize the cuboids set on which other frequently referenced cuboids are based. Alternatively, we can compute an **iceberg cube**, which is a data cube that stores only those cube cells with an aggregate value (e.g., *count*) that is above some minimum support threshold.

Another common strategy is to materialize a *shell cube*. This involves precomputing the cuboids for only a small number of dimensions (e.g., three to five) of a data cube. Queries on additional combinations of the dimensions can be computed on-the-fly. Because our aim in this chapter is to provide a solid introduction and overview of data warehousing for data mining, we defer our detailed discussion of cuboid selection and computation to Chapter 5, which studies various data cube computation methods in greater depth.

Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves several issues, such as how to determine the relevant cuboid(s) from among the candidate materialized cuboids, how to use available index structures on the materialized cuboids, and how to transform the OLAP operations onto the selected cuboid(s). These issues are discussed in Section 4.4.3 as well as in Chapter 5.

Finally, during load and refresh, the materialized cuboids should be updated efficiently. Parallelism and incremental update techniques for this operation should be explored.

4.4.2 Indexing OLAP Data: Bitmap Index and Join Index

To facilitate efficient data accessing, most data warehouse systems support index structures and materialized views (using cuboids). General methods to select cuboids for materialization were discussed in Section 4.4.1. In this subsection, we examine how to index OLAP data by *bitmap indexing* and *join indexing*.

The **bitmap indexing** method is popular in OLAP products because it allows quick searching in data cubes. The bitmap index is an alternative representation of the *record.ID* (*RID*) list. In the bitmap index for a given attribute, there is a distinct bit vector, B_v , for each value v in the attribute's domain. If a given attribute's domain consists of n values, then n bits are needed for each entry in the bitmap index (i.e., there are n bit vectors). If the attribute has the value v for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

Example 4.7 Bitmap indexing. In the *AllElectronics* data warehouse, suppose the dimension *item* at the top level has four values (representing item types): “home entertainment,” “computer,” “phone,” and “security.” Each value (e.g., “computer”) is represented by a bit vector in the *item* bitmap index table. Suppose that the cube is stored as a relation table with 100,000 rows. Because the domain of *item* consists of four values, the bitmap index table requires four bit vectors (or lists), each with 100,000 bits. Figure 4.15 shows a base (data) table containing the dimensions *item* and *city*, and its mapping to bitmap index tables for each of the dimensions. ■

Base table			<i>item</i> bitmap index table					<i>city</i> bitmap index table		
<i>RID</i>	<i>item</i>	<i>city</i>	<i>RID</i>	H	C	P	S	<i>RID</i>	V	T
R1	H	V	R1	1	0	0	0	R1	1	0
R2	C	V	R2	0	1	0	0	R2	1	0
R3	P	V	R3	0	0	1	0	R3	1	0
R4	S	V	R4	0	0	0	1	R4	1	0
R5	H	T	R5	1	0	0	0	R5	0	1
R6	C	T	R6	0	1	0	0	R6	0	1
R7	P	T	R7	0	0	1	0	R7	0	1
R8	S	T	R8	0	0	0	1	R8	0	1

Note: H for “home entertainment,” C for “computer,” P for “phone,” S for “security,” V for “Vancouver,” T for “Toronto.”

Figure 4.15 Indexing OLAP data using bitmap indices.

Bitmap indexing is advantageous compared to hash and tree indices. It is especially useful for low-cardinality domains because comparison, join, and aggregation operations are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and input/output (I/O) since a string of characters can be represented by a single bit. For higher-cardinality domains, the method can be adapted using compression techniques.

The **join indexing** method gained popularity from its use in relational database query processing. Traditional indexing maps the value in a given column to a list of rows having that value. In contrast, join indexing registers the joinable rows of two relations from a relational database. For example, if two relations $R(RID, A)$ and $S(B, SID)$ join on the attributes A and B , then the join index record contains the pair (RID, SID) , where RID and SID are record identifiers from the R and S relations, respectively. Hence, the join index records can identify joinable tuples without performing costly join operations. Join indexing is especially useful for maintaining the relationship between a foreign key² and its matching primary keys, from the joinable relation.

The star schema model of data warehouses makes join indexing attractive for cross-table search, because the linkage between a fact table and its corresponding dimension tables comprises the fact table’s foreign key and the dimension table’s primary key. Join

²A set of attributes in a relation schema that forms a primary key for another relation schema is called a **foreign key**.

indexing maintains relationships between attribute values of a dimension (e.g., within a dimension table) and the corresponding rows in the fact table. Join indices may span multiple dimensions to form **composite join indices**. We can use join indices to identify subcubes that are of interest.

Example 4.8 Join indexing. In Example 3.4, we defined a star schema for *AllElectronics* of the form “*sales_star* [*time*, *item*, *branch*, *location*]: *dollars_sold* = sum (*sales_in_dollars*).” An example of a join index relationship between the *sales* fact table and the *location* and *item* dimension tables is shown in Figure 4.16. For example, the “Main Street” value in the *location* dimension table joins with tuples T57, T238, and T884 of the *sales* fact table. Similarly, the “Sony-TV” value in the *item* dimension table joins with tuples T57 and T459 of the *sales* fact table. The corresponding join index tables are shown in Figure 4.17.

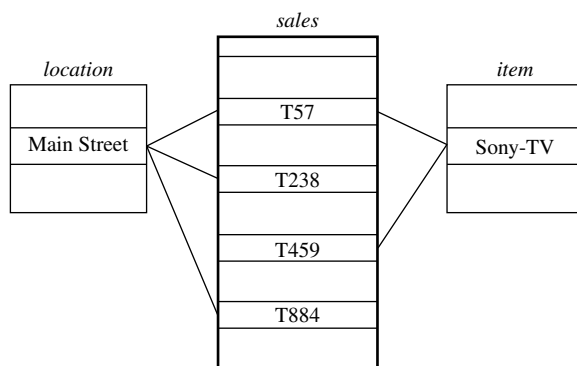


Figure 4.16 Linkages between a *sales* fact table and *location* and *item* dimension tables.

Join index table for <i>location/sales</i>		Join index table for <i>item/sales</i>	
<i>location</i>	<i>sales_key</i>	<i>item</i>	<i>sales_key</i>
...
Main Street	T57	Sony-TV	T57
Main Street	T238	Sony-TV	T459
Main Street	T884
...	...		

Join index table linking <i>location</i> and <i>item</i> to <i>sales</i>		
<i>location</i>	<i>item</i>	<i>sales_key</i>
...
Main Street	Sony-TV	T57
...

Figure 4.17 Join index tables based on the linkages between the *sales* fact table and the *location* and *item* dimension tables shown in Figure 4.16.

Suppose that there are 360 time values, 100 items, 50 branches, 30 locations, and 10 million sales tuples in the *sales_star* data cube. If the *sales* fact table has recorded sales for only 30 items, the remaining 70 items will obviously not participate in joins. If join indices are not used, additional I/Os have to be performed to bring the joining portions of the fact table and the dimension tables together. ■

To further speed up query processing, the join indexing and the bitmap indexing methods can be integrated to form **bitmapped join indices**.

4.4.3 Efficient Processing of OLAP Queries

The purpose of materializing cuboids and constructing OLAP index structures is to speed up query processing in data cubes. Given materialized views, query processing should proceed as follows:

1. **Determine which operations should be performed on the available cuboids:** This involves transforming any selection, projection, roll-up (group-by), and drill-down operations specified in the query into corresponding SQL and/or OLAP operations. For example, slicing and dicing a data cube may correspond to selection and/or projection operations on a materialized cuboid.
2. **Determine to which materialized cuboid(s) the relevant operations should be applied:** This involves identifying all of the materialized cuboids that may potentially be used to answer the query, pruning the set using knowledge of “dominance” relationships among the cuboids, estimating the costs of using the remaining materialized cuboids, and selecting the cuboid with the least cost.

Example 4.9 OLAP query processing. Suppose that we define a data cube for *AllElectronics* of the form “*sales_cube* [*time*, *item*, *location*]: *sum(sales_in_dollars)*.” The dimension hierarchies used are “*day* < *month* < *quarter* < *year*” for *time*; “*item_name* < *brand* < *type*” for *item*; and “*street* < *city* < *province_or_state* < *country*” for *location*.

Suppose that the query to be processed is on {*brand*, *province_or_state*}, with the selection constant “*year* = 2010.” Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year*, *item_name*, *city*}
- cuboid 2: {*year*, *brand*, *country*}
- cuboid 3: {*year*, *brand*, *province_or_state*}
- cuboid 4: {*item_name*, *province_or_state*}, where *year* = 2010

“Which of these four cuboids should be selected to process the query?” Finer-granularity data cannot be generated from coarser-granularity data. Therefore, cuboid 2 cannot be used because *country* is a more general concept than *province_or_state*. Cuboids 1, 3, and 4 can be used to process the query because (1) they have the same set or a superset of the

dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *location* dimensions in these cuboids are at a finer level than *brand* and *province_or_state*, respectively.

“How would the costs of each cuboid compare if used to process the query?” It is likely that using cuboid 1 would cost the most because both *item_name* and *city* are at a lower level than the *brand* and *province_or_state* concepts specified in the query. If there are not many *year* values associated with *items* in the cube, but there are several *item_names* for each *brand*, then cuboid 3 will be smaller than cuboid 4, and thus cuboid 3 should be chosen to process the query. However, if efficient indices are available for cuboid 4, then cuboid 4 may be a better choice. Therefore, some cost-based estimation is required to decide which set of cuboids should be selected for query processing. ■

4.4.4 OLAP Server Architectures: ROLAP versus MOLAP versus HOLAP

Logically, OLAP servers present business users with multidimensional data from data warehouses or data marts, without concerns regarding how or where the data are stored. However, the physical architecture and implementation of OLAP servers must consider data storage issues. Implementations of a warehouse server for OLAP processing include the following:

Relational OLAP (ROLAP) servers: These are the intermediate servers that stand in between a relational back-end server and client front-end tools. They use a *relational* or *extended-relational DBMS* to store and manage warehouse data, and OLAP middleware to support missing pieces. ROLAP servers include optimization for each DBMS back end, implementation of aggregation navigation logic, and additional tools and services. ROLAP technology tends to have greater scalability than MOLAP technology. The DSS server of Microstrategy, for example, adopts the ROLAP approach.

Multidimensional OLAP (MOLAP) servers: These servers support multidimensional data views through *array-based multidimensional storage engines*. They map multidimensional views directly to data cube array structures. The advantage of using a data cube is that it allows fast indexing to precomputed summarized data. Notice that with multidimensional data stores, the storage utilization may be low if the data set is sparse. In such cases, sparse matrix compression techniques should be explored (Chapter 5).

Many MOLAP servers adopt a two-level storage representation to handle dense and sparse data sets: Denser subcubes are identified and stored as array structures, whereas sparse subcubes employ compression technology for efficient storage utilization.

Hybrid OLAP (HOLAP) servers: The hybrid OLAP approach combines ROLAP and MOLAP technology, benefiting from the greater scalability of ROLAP and the faster computation of MOLAP. For example, a HOLAP server may allow large volumes

of detailed data to be stored in a relational database, while aggregations are kept in a separate MOLAP store. The Microsoft SQL Server 2000 supports a hybrid OLAP server.

Specialized SQL servers: To meet the growing demand of OLAP processing in relational databases, some database system vendors implement specialized SQL servers that provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

“How are data actually stored in ROLAP and MOLAP architectures?” Let’s first look at ROLAP. As its name implies, ROLAP uses relational tables to store data for online analytical processing. Recall that the fact table associated with a base cuboid is referred to as a *base fact table*. The base fact table stores data at the abstraction level indicated by the join keys in the schema for the given data cube. Aggregated data can also be stored in fact tables, referred to as **summary fact tables**. Some summary fact tables store both base fact table data and aggregated data (see Example 3.10). Alternatively, separate summary fact tables can be used for each abstraction level to store only aggregated data.

Example 4.10 A ROLAP data store. Table 4.4 shows a summary fact table that contains both base fact data and aggregated data. The schema is “{*record_identifier (RID)*, *item*, . . . , *day*, *month*, *quarter*, *year*, *dollars_sold*},” where *day*, *month*, *quarter*, and *year* define the sales date, and *dollars_sold* is the sales amount. Consider the tuples with an *RID* of 1001 and 1002, respectively. The data of these tuples are at the base fact level, where the sales dates are October 15, 2010, and October 23, 2010, respectively. Consider the tuple with an *RID* of 5001. This tuple is at a more general level of abstraction than the tuples 1001 and 1002. The *day* value has been generalized to all, so that the corresponding *time* value is October 2010. That is, the *dollars_sold* amount shown is an aggregation representing the entire month of October 2010, rather than just October 15 or 23, 2010. The special value all is used to represent subtotals in summarized data. ■

MOLAP uses multidimensional array structures to store data for online analytical processing. This structure is discussed in greater detail in Chapter 5.

Most data warehouse systems adopt a client-server architecture. A relational data store always resides at the data warehouse/data mart server site. A multidimensional data store can reside at either the database server site or the client site.

Table 4.4 Single Table for Base and Summary Facts

<i>RID</i>	<i>item</i>	...	<i>day</i>	<i>month</i>	<i>quarter</i>	<i>year</i>	<i>dollars_sold</i>
1001	TV	...	15	10	Q4	2010	250.60
1002	TV	...	23	10	Q4	2010	175.00
...
5001	TV	...	all	10	Q4	2010	45,786.08
...

4.5 Data Generalization by Attribute-Oriented Induction

Conceptually, the data cube can be viewed as a kind of multidimensional data generalization. In general, *data generalization* summarizes data by replacing relatively low-level values (e.g., numeric values for an attribute *age*) with higher-level concepts (e.g., *young*, *middle-aged*, and *senior*), or by reducing the number of dimensions to summarize data in concept space involving fewer dimensions (e.g., removing *birth_date* and *telephone number* when summarizing the behavior of a group of students). Given the large amount of data stored in databases, it is useful to be able to describe concepts in concise and succinct terms at generalized (rather than low) levels of abstraction. Allowing data sets to be generalized at multiple levels of abstraction facilitates users in examining the general behavior of the data. Given the *AllElectronics* database, for example, instead of examining individual customer transactions, sales managers may prefer to view the data generalized to higher levels, such as summarized by customer groups according to geographic regions, frequency of purchases per group, and customer income.

This leads us to the notion of *concept description*, which is a form of data generalization. A concept typically refers to a data collection such as *frequent_buyers*, *graduate_students*, and so on. As a data mining task, concept description is not a simple enumeration of the data. Instead, **concept description** generates descriptions for data *characterization* and *comparison*. It is sometimes called **class description** when the concept to be described refers to a class of objects. **Characterization** provides a concise and succinct summarization of the given data collection, while concept or class **comparison** (also known as **discrimination**) provides descriptions comparing two or more data collections.

Up to this point, we have studied data cube (or OLAP) approaches to concept description using multidimensional, multilevel data generalization in data warehouses. “Is data cube technology sufficient to accomplish all kinds of concept description tasks for large data sets?” Consider the following cases.

- **Complex data types and aggregation:** Data warehouses and OLAP tools are based on a multidimensional data model that views data in the form of a data cube, consisting of dimensions (or attributes) and measures (aggregate functions). However, many current OLAP systems confine dimensions to non-numeric data and measures to numeric data. In reality, the database can include attributes of various data types, including numeric, non-numeric, spatial, text, or image, which ideally should be included in the concept description.

Furthermore, the aggregation of attributes in a database may include sophisticated data types such as the collection of non-numeric data, the merging of spatial regions, the composition of images, the integration of texts, and the grouping of object pointers. Therefore, OLAP, with its restrictions on the possible dimension and measure types, represents a simplified model for data analysis. Concept description should handle complex data types of the attributes and their aggregations, as necessary.

- **User control versus automation:** Online analytical processing in data warehouses is a user-controlled process. The selection of dimensions and the application of OLAP operations (e.g., drill-down, roll-up, slicing, and dicing) are primarily directed and controlled by users. Although the control in most OLAP systems is quite user-friendly, users do require a good understanding of the role of each dimension. Furthermore, in order to find a satisfactory description of the data, users may need to specify a long sequence of OLAP operations. It is often desirable to have a more automated process that helps users determine which dimensions (or attributes) should be included in the analysis, and the degree to which the given data set should be generalized in order to produce an interesting summarization of the data.

This section presents an alternative method for concept description, called *attribute-oriented induction*, which works for complex data types and relies on a data-driven generalization process.

4.5.1 Attribute-Oriented Induction for Data Characterization

The **attribute-oriented induction (AOI)** approach to concept description was first proposed in 1989, a few years before the introduction of the data cube approach. The data cube approach is essentially based on *materialized views* of the data, which typically have been precomputed in a data warehouse. In general, it performs offline aggregation before an OLAP or data mining query is submitted for processing. On the other hand, the attribute-oriented induction approach is basically a *query-oriented*, generalization-based, online data analysis technique. Note that there is no inherent barrier distinguishing the two approaches based on online aggregation versus offline precomputation. Some aggregations in the data cube can be computed online, while offline precomputation of multidimensional space can speed up attribute-oriented induction as well.

The general idea of attribute-oriented induction is to first collect the task-relevant data using a database query and then perform generalization based on the examination of the number of each attribute's distinct values in the relevant data set. The generalization is performed by either *attribute removal* or *attribute generalization*. Aggregation is performed by merging identical generalized tuples and accumulating their respective counts. This reduces the size of the generalized data set. The resulting generalized relation can be mapped into different forms (e.g., charts or rules) for presentation to the user.

The following illustrates the process of attribute-oriented induction. We first discuss its use for characterization. The method is extended for the mining of class comparisons in Section 4.5.3.

Example 4.11 A data mining query for characterization. Suppose that a user wants to describe the general characteristics of graduate students in the *Big University* database, given the attributes *name*, *gender*, *major*, *birth_place*, *birth_date*, *residence*, *phone#* (telephone

number), and *gpa* (*grade_point_average*). A data mining query for this characterization can be expressed in the data mining query language, DMQL, as follows:

```
use Big_University_DB
mine characteristics as "Science_Students"
in relevance to name, gender, major, birth_place, birth_date, residence,
    phone#, gpa
from student
where status in "graduate"
```

We will see how this example of a typical data mining query can apply attribute-oriented induction to the mining of characteristic descriptions.

First, **data focusing** should be performed *before* attribute-oriented induction. This step corresponds to the specification of the task-relevant data (i.e., data for analysis). The data are collected based on the information provided in the data mining query. Because a data mining query is usually relevant to only a portion of the database, selecting the relevant data set not only makes mining more efficient, but also derives more meaningful results than mining the entire database.

Specifying the set of relevant attributes (i.e., attributes for mining, as indicated in DMQL with the *in relevance to* clause) may be difficult for the user. A user may select only a few attributes that he or she feels are important, while missing others that could also play a role in the description. For example, suppose that the dimension *birth_place* is defined by the attributes *city*, *province_or_state*, and *country*. Of these attributes, let's say that the user has only thought to specify *city*. In order to allow generalization on the *birth_place* dimension, the other attributes defining this dimension should also be included. In other words, having the system automatically include *province_or_state* and *country* as relevant attributes allows *city* to be generalized to these higher conceptual levels during the induction process.

At the other extreme, suppose that the user may have introduced too many attributes by specifying all of the possible attributes with the clause *in relevance to **. In this case, all of the attributes in the relation specified by the *from* clause would be included in the analysis. Many of these attributes are unlikely to contribute to an interesting description. A correlation-based analysis method (Section 3.3.2) can be used to perform attribute *relevance analysis* and filter out statistically irrelevant or weakly relevant attributes from the descriptive mining process. Other approaches such as attribute subset selection, are also described in Chapter 3.

Table 4.5 Initial Working Relation: A Collection of Task-Relevant Data

<i>name</i>	<i>gender</i>	<i>major</i>	<i>birth_place</i>	<i>birth_date</i>	<i>residence</i>	<i>phone#</i>	<i>gpa</i>
Jim Woodman	M	CS	Vancouver, BC, Canada	12-8-76	3511 Main St., Richmond	687-4598	3.67
Scott Lachance	M	CS	Montreal, Que, Canada	7-28-75	345 1st Ave., Richmond	253-9106	3.70
Laura Lee	F	Physics	Seattle, WA, USA	8-25-70	125 Austin Ave., Burnaby	420-5232	3.83
...

“What does the ‘where status in “graduate”’ clause mean?” The *where* clause implies that a concept hierarchy exists for the attribute *status*. Such a concept hierarchy organizes primitive-level data values for *status* (e.g., “M.Sc.,” “M.A.,” “M.B.A.,” “Ph.D.,” “B.Sc.,” and “B.A.”) into higher conceptual levels (e.g., “graduate” and “undergraduate”). This use of concept hierarchies does not appear in traditional relational query languages, yet is likely to become a common feature in data mining query languages.

The data mining query presented in Example 4.11 is transformed into the following relational query for the collection of the task-relevant data set:

```
use Big_University_DB
select name, gender, major, birth_place, birth_date, residence, phone#, gpa
from student
where status in {"M.Sc.," "M.A.," "M.B.A.," "Ph.D."}
```

The transformed query is executed against the relational database, *Big_University_DB*, and returns the data shown earlier in Table 4.5. This table is called the (task-relevant) **initial working relation**. It is the data on which induction will be performed. Note that each tuple is, in fact, a conjunction of attribute–value pairs. Hence, we can think of a tuple within a relation as a rule of conjuncts, and of induction on the relation as the generalization of these rules. ■

“Now that the data are ready for attribute-oriented induction, how is attribute-oriented induction performed?” The essential operation of attribute-oriented induction is *data generalization*, which can be performed in either of two ways on the initial working relation: *attribute removal* and *attribute generalization*.

Attribute removal is based on the following rule: *If there is a large set of distinct values for an attribute of the initial working relation, but either (case 1) there is no generalization operator on the attribute (e.g., there is no concept hierarchy defined for the attribute), or (case 2) its higher-level concepts are expressed in terms of other attributes, then the attribute should be removed from the working relation.*

Let’s examine the reasoning behind this rule. An attribute–value pair represents a conjunct in a generalized tuple, or rule. The removal of a conjunct eliminates a constraint and thus generalizes the rule. If, as in case 1, there is a large set of distinct values for an attribute but there is no generalization operator for it, the attribute should be removed because it cannot be generalized. Preserving it would imply keeping a large number of disjuncts, which contradicts the goal of generating concise rules. On the other hand, consider case 2, where the attribute’s higher-level concepts are expressed in terms of other attributes. For example, suppose that the attribute in question is *street*, with higher-level concepts that are represented by the attributes $\langle city, province_or_state, country \rangle$. The removal of *street* is equivalent to the application of a generalization operator. This rule corresponds to the generalization rule known as *dropping condition* in the machine learning literature on *learning from examples*.

Attribute generalization is based on the following rule: *If there is a large set of distinct values for an attribute in the initial working relation, and there exists a set of generalization operators on the attribute, then a generalization operator should be selected and applied*

to the attribute. This rule is based on the following reasoning. Use of a generalization operator to generalize an attribute value within a tuple, or rule, in the working relation will make the rule cover more of the original data tuples, thus generalizing the concept it represents. This corresponds to the generalization rule known as *climbing generalization trees* in *learning from examples*, or *concept tree ascension*.

Both rules—*attribute removal* and *attribute generalization*—claim that if there is a *large* set of distinct values for an attribute, further generalization should be applied. This raises the question: How large is “a large set of distinct values for an attribute” considered to be?

Depending on the attributes or application involved, a user may prefer some attributes to remain at a rather low abstraction level while others are generalized to higher levels. The control of how high an attribute should be generalized is typically quite subjective. The control of this process is called **attribute generalization control**. If the attribute is generalized “too high,” it may lead to overgeneralization, and the resulting rules may not be very informative.

On the other hand, if the attribute is not generalized to a “sufficiently high level,” then undergeneralization may result, where the rules obtained may not be informative either. Thus, a balance should be attained in attribute-oriented generalization. There are many possible ways to control a generalization process. We will describe two common approaches and illustrate how they work.

The first technique, called **attribute generalization threshold control**, either sets one generalization threshold for all of the attributes, or sets one threshold for each attribute. If the number of distinct values in an attribute is greater than the attribute threshold, further attribute removal or attribute generalization should be performed. Data mining systems typically have a default attribute threshold value generally ranging from 2 to 8 and should allow experts and users to modify the threshold values as well. If a user feels that the generalization reaches too high a level for a particular attribute, the threshold can be increased. This corresponds to drilling down along the attribute. Also, to further generalize a relation, the user can reduce an attribute’s threshold, which corresponds to rolling up along the attribute.

The second technique, called **generalized relation threshold control**, sets a threshold for the generalized relation. If the number of (distinct) tuples in the generalized relation is greater than the threshold, further generalization should be performed. Otherwise, no further generalization should be performed. Such a threshold may also be preset in the data mining system (usually within a range of 10 to 30), or set by an expert or user, and should be adjustable. For example, if a user feels that the generalized relation is too small, he or she can increase the threshold, which implies drilling down. Otherwise, to further generalize a relation, the threshold can be reduced, which implies rolling up.

These two techniques can be applied in sequence: First apply the attribute threshold control technique to generalize each attribute, and then apply relation threshold control to further reduce the size of the generalized relation. No matter which generalization control technique is applied, the user should be allowed to adjust the generalization thresholds in order to obtain interesting concept descriptions.

In many database-oriented induction processes, users are interested in obtaining quantitative or statistical information about the data at different abstraction levels.

Thus, it is important to accumulate count and other aggregate values in the induction process. Conceptually, this is performed as follows. The aggregate function, `count()`, is associated with each database tuple. Its value for each tuple in the initial working relation is initialized to 1. Through attribute removal and attribute generalization, tuples within the initial working relation may be generalized, resulting in groups of *identical tuples*. In this case, all of the identical tuples forming a group should be merged into one tuple.

The count of this new, generalized tuple is set to the total number of tuples from the initial working relation that are represented by (i.e., merged into) the new generalized tuple. For example, suppose that by attribute-oriented induction, 52 data tuples from the initial working relation are all generalized to the same tuple, T . That is, the generalization of these 52 tuples resulted in 52 identical instances of tuple T . These 52 identical tuples are merged to form one instance of T , with a count that is set to 52. Other popular aggregate functions that could also be associated with each tuple include `sum()` and `avg()`. For a given generalized tuple, `sum()` contains the sum of the values of a given numeric attribute for the initial working relation tuples making up the generalized tuple. Suppose that tuple T contained `sum(units_sold)` as an aggregate function. The sum value for tuple T would then be set to the total number of units sold for each of the 52 tuples. The aggregate `avg()` (average) is computed according to the formula $\text{avg}() = \text{sum}() / \text{count}()$.

Example 4.12 Attribute-oriented induction. Here we show how attribute-oriented induction is performed on the initial working relation of Table 4.5. For each attribute of the relation, the generalization proceeds as follows:

1. *name*: Since there are a large number of distinct values for *name* and there is no generalization operation defined on it, this attribute is removed.
2. *gender*: Since there are only two distinct values for *gender*, this attribute is retained and no generalization is performed on it.
3. *major*: Suppose that a concept hierarchy has been defined that allows the attribute *major* to be generalized to the values {arts&sciences, engineering, business}. Suppose also that the attribute generalization threshold is set to 5, and that there are more than 20 distinct values for *major* in the initial working relation. By attribute generalization and attribute generalization control, *major* is therefore generalized by climbing the given concept hierarchy.
4. *birth_place*: This attribute has a large number of distinct values; therefore, we would like to generalize it. Suppose that a concept hierarchy exists for *birth_place*, defined as “*city* < *province_or_state* < *country*.” If the number of distinct values for *country* in the initial working relation is greater than the attribute generalization threshold, then *birth_place* should be removed, because even though a generalization operator exists for it, the generalization threshold would not be satisfied. If, instead, the number of distinct values for *country* is less than the attribute generalization threshold, then *birth_place* should be generalized to *birth_country*.
5. *birth_date*: Suppose that a hierarchy exists that can generalize *birth_date* to *age* and *age* to *age_range*, and that the number of age ranges (or intervals) is small with

Table 4.6 Generalized Relation Obtained by Attribute-Oriented Induction on Table 4.5's Data

<i>gender</i>	<i>major</i>	<i>birth_country</i>	<i>age_range</i>	<i>residence_city</i>	<i>gpa</i>	<i>count</i>
M	Science	Canada	20–25	Richmond	very_good	16
F	Science	Foreign	25–30	Burnaby	excellent	22
...

respect to the attribute generalization threshold. Generalization of *birth_date* should therefore take place.

6. *residence*: Suppose that *residence* is defined by the attributes *number*, *street*, *residence_city*, *residence_province_or_state*, and *residence_country*. The number of distinct values for *number* and *street* will likely be very high, since these concepts are quite low level. The attributes *number* and *street* should therefore be removed so that *residence* is then generalized to *residence_city*, which contains fewer distinct values.
7. *phone#*: As with the *name* attribute, *phone#* contains too many distinct values and should therefore be removed in generalization.
8. *gpa*: Suppose that a concept hierarchy exists for *gpa* that groups values for grade point average into numeric intervals like {3.75–4.0, 3.5–3.75, ...}, which in turn are grouped into descriptive values such as {"excellent", "very-good", ...}. The attribute can therefore be generalized.

The generalization process will result in groups of identical tuples. For example, the first two tuples of Table 4.5 both generalize to the same identical tuple (namely, the first tuple shown in Table 4.6). Such identical tuples are then merged into one, with their counts accumulated. This process leads to the generalized relation shown in Table 4.6.

Based on the vocabulary used in OLAP, we may view *count()* as a *measure*, and the remaining attributes as *dimensions*. Note that aggregate functions, such as *sum()*, may be applied to numeric attributes (e.g., *salary* and *sales*). These attributes are referred to as *measure attributes*. ■

4.5.2 Efficient Implementation of Attribute-Oriented Induction

"How is attribute-oriented induction actually implemented?" Section 4.5.1 provided an introduction to attribute-oriented induction. The general procedure is summarized in Figure 4.18. The efficiency of this algorithm is analyzed as follows:

- Step 1 of the algorithm is essentially a relational query to collect the task-relevant data into the **working relation**, *W*. Its processing efficiency depends on the query processing methods used. Given the successful implementation and commercialization of database systems, this step is expected to have good performance.
- Step 2 collects statistics on the working relation. This requires scanning the relation at most once. The cost for computing the minimum desired level and determining the mapping pairs, (v, v') , for each attribute is dependent on the number of distinct

Algorithm: Attribute-oriented induction. Mining generalized characteristics in a relational database given a user's data mining request.

Input:

- DB , a relational database;
- $DMQuery$, a data mining query;
- a_list , a list of attributes (containing attributes, a_i);
- $Gen(a_i)$, a set of concept hierarchies or generalization operators on attributes, a_i ;
- $a_gen_thresh(a_i)$, attribute generalization thresholds for each a_i .

Output: P , a *Prime-generalized_relation*.

Method:

1. $W \leftarrow \text{get_task_relevant_data}(DMQuery, DB)$; // Let W , the working relation, hold the task-relevant data.
2. $\text{prepare_for_generalization}(W)$; // This is implemented as follows.
 - (a) Scan W and collect the distinct values for each attribute, a_i . (Note: If W is very large, this may be done by examining a sample of W .)
 - (b) For each attribute a_i , determine whether a_i should be removed. If not, compute its minimum desired level L_i based on its given or default attribute threshold, and determine the mapping pairs (v, v') , where v is a distinct value of a_i in W , and v' is its corresponding generalized value at level L_i .
3. $P \leftarrow \text{generalization}(W)$,

The *Prime-generalized_relation*, P , is derived by replacing each value v in W by its corresponding v' in the mapping while accumulating `count` and computing any other aggregate values.

This step can be implemented efficiently using either of the two following variations:

- (a) For each generalized tuple, insert the tuple into a sorted prime relation P by a binary search: if the tuple is already in P , simply increase its `count` and other aggregate values accordingly; otherwise, insert it into P .
- (b) Since in most cases the number of distinct values at the prime relation level is small, the prime relation can be coded as an m -dimensional array, where m is the number of attributes in P , and each dimension contains the corresponding generalized attribute values. Each array element holds the corresponding `count` and other aggregation values, if any. The insertion of a generalized tuple is performed by measure aggregation in the corresponding array element.

Figure 4.18 Basic algorithm for attribute-oriented induction.

values for each attribute and is smaller than $|W|$, the number of tuples in the working relation. Notice that it may not be necessary to scan the working relation once, since if the working relation is large, a sample of such a relation will be sufficient to get statistics and determine which attributes should be generalized to a certain high level and which attributes should be removed. Moreover, such statistics may also be obtained in the process of extracting and generating a working relation in Step 1.

- Step 3 derives the **prime relation**, P . This is performed by scanning each tuple in the working relation and inserting generalized tuples into P . There are a total of $|W|$ tuples in W and p tuples in P . For each tuple, t , in W , we substitute its attribute values based on the derived mapping pairs. This results in a generalized tuple, t' . If variation (a) in Figure 4.18 is adopted, each t' takes $O(\log p)$ to find the location for the count increment or tuple insertion. Thus, the total time complexity is $O(|W| \times \log p)$ for all of the generalized tuples. If variation (b) is adopted, each t' takes $O(1)$ to find the tuple for the count increment. Thus, the overall time complexity is $O(N)$ for all of the generalized tuples.

Many data analysis tasks need to examine a good number of dimensions or attributes. This may involve *dynamically* introducing and testing additional attributes rather than just those specified in the mining query. Moreover, a user with little knowledge of the *truly* relevant data set may simply specify “in relevance to $*$ ” in the mining query, which includes all of the attributes in the analysis. Therefore, an advanced-concept description mining process needs to perform attribute relevance analysis on large sets of attributes to select the most relevant ones. This analysis may employ correlation measures or tests of statistical significance, as described in Chapter 3 on data preprocessing.

Example 4.13 Presentation of generalization results. Suppose that attribute-oriented induction was performed on a *sales* relation of the *AllElectronics* database, resulting in the generalized description of Table 4.7 for sales last year. The description is shown in the form of a generalized relation. Table 4.6 is another generalized relation example.

Such generalized relations can also be presented in the form of cross-tabulation forms, various kinds of graphic presentation (e.g., pie charts and bar charts), and quantitative characteristics rules (i.e., showing how different value combinations are distributed in the generalized relation). ■

Table 4.7 Generalized Relation for Last Year’s Sales

<i>location</i>	<i>item</i>	<i>sales (in million dollars)</i>	<i>count (in thousands)</i>
Asia	TV	15	300
Europe	TV	12	250
North America	TV	28	450
Asia	computer	120	1000
Europe	computer	150	1200
North America	computer	200	1800

4.5.3 Attribute-Oriented Induction for Class Comparisons

In many applications, users may not be interested in having a single class (or concept) described or characterized, but prefer to mine a description that compares or distinguishes one class (or concept) from other comparable classes (or concepts). Class discrimination or comparison (hereafter referred to as **class comparison**) mines descriptions that distinguish a target class from its contrasting classes. Notice that the target and contrasting classes must be *comparable* in the sense that they share similar dimensions and attributes. For example, the three classes *person*, *address*, and *item* are not comparable. However, sales in the last three years are comparable classes, and so are, for example, computer science students versus physics students.

Our discussions on class characterization in the previous sections handle multilevel data summarization and characterization in a single class. The techniques developed can be extended to handle class comparison across several comparable classes. For example, the attribute generalization process described for class characterization can be modified so that the generalization is performed *synchronously* among all the classes compared. This allows the attributes in all of the classes to be generalized to the *same* abstraction levels.

Suppose, for instance, that we are given the *AlIElectronics* data for sales in 2009 and in 2010 and want to compare these two classes. Consider the dimension *location* with abstractions at the *city*, *province_or_state*, and *country* levels. Data in each class should be generalized to the same *location* level. That is, they are all synchronously generalized to either the *city* level, the *province_or_state* level, or the *country* level. Ideally, this is more useful than comparing, say, the sales in Vancouver in 2009 with the sales in the United States in 2010 (i.e., where each set of sales data is generalized to a different level). The users, however, should have the option to overwrite such an automated, synchronous comparison with their own choices, when preferred.

“How is class comparison performed?” In general, the procedure is as follows:

1. **Data collection:** The set of relevant data in the database is collected by query processing and is partitioned respectively into a *target class* and one or a set of *contrasting classes*.
2. **Dimension relevance analysis:** If there are many dimensions, then dimension relevance analysis should be performed on these classes to select only the highly relevant dimensions for further analysis. Correlation or entropy-based measures can be used for this step (Chapter 3).
3. **Synchronous generalization:** Generalization is performed on the target class to the level controlled by a user- or expert-specified dimension threshold, which results in a **prime target class relation**. The concepts in the contrasting class(es) are generalized to the same level as those in the prime target class relation, forming the **prime contrasting class(es) relation**.
4. **Presentation of the derived comparison:** The resulting class comparison description can be visualized in the form of tables, graphs, and rules. This presentation usually includes a “contrasting” measure such as *count%* (percentage count) that reflects the

comparison between the target and contrasting classes. The user can adjust the comparison description by applying drill-down, roll-up, and other OLAP operations to the target and contrasting classes, as desired.

The preceding discussion outlines a general algorithm for mining comparisons in databases. In comparison with characterization, the previous algorithm involves synchronous generalization of the target class with the contrasting classes, so that classes are simultaneously compared at the same abstraction levels.

Example 4.14 mines a class comparison describing the graduate and undergraduate students at *Big University*.

Example 4.14 Mining a class comparison. Suppose that you would like to compare the general properties of the graduate and undergraduate students at *Big University*, given the attributes *name*, *gender*, *major*, *birth_place*, *birth_date*, *residence*, *phone#*, and *gpa*.

This data mining task can be expressed in DMQL as follows:

```
use Big_University_DB
mine comparison as "grad_vs_undergrad_students"
in relevance to name, gender, major, birth_place, birth_date, residence,
    phone#, gpa
for "graduate_students"
where status in "graduate"
versus "undergraduate_students"
where status in "undergraduate"
analyze count%
from student
```

Let's see how this typical example of a data mining query for mining comparison descriptions can be processed.

First, the query is transformed into two relational queries that collect two sets of task-relevant data: one for the *initial target-class working relation* and the other for the *initial contrasting-class working relation*, as shown in Tables 4.8 and 4.9. This can also be viewed as the construction of a data cube, where the status {graduate, undergraduate} serves as one dimension, and the other attributes form the remaining dimensions.

Second, dimension relevance analysis can be performed, when necessary, on the two classes of data. After this analysis, irrelevant or weakly relevant dimensions (e.g., *name*, *gender*, *birth_place*, *residence*, and *phone#*) are removed from the resulting classes. Only the highly relevant attributes are included in the subsequent analysis.

Third, synchronous generalization is performed on the target class to the levels controlled by user- or expert-specified dimension thresholds, forming the *prime target class relation*. The contrasting class is generalized to the same levels as those in the prime target class relation, forming the *prime contrasting class(es) relation*, as presented in Tables 4.10 and 4.11. In comparison with undergraduate students, graduate students tend to be older and have a higher GPA in general.

Table 4.8 Initial Working Relations: The Target Class (Graduate Students)

<i>name</i>	<i>gender</i>	<i>major</i>	<i>birth_place</i>	<i>birth_date</i>	<i>residence</i>	<i>phone#</i>	<i>gpa</i>
Jim Woodman	M	CS	Vancouver, BC, Canada	12-8-76	3511 Main St., Richmond	687-4598	3.67
Scott Lachance	M	CS	Montreal, Que, Canada	7-28-75	345 1st Ave., Vancouver	253-9106	3.70
Laura Lee	F	Physics	Seattle, WA, USA	8-25-70	125 Austin Ave., Burnaby	420-5232	3.83
...

Table 4.9 Initial Working Relations: The Contrasting Class (Undergraduate Students)

<i>name</i>	<i>gender</i>	<i>major</i>	<i>birth_place</i>	<i>birth_date</i>	<i>residence</i>	<i>phone#</i>	<i>gpa</i>
Bob Schumann	M	Chemistry	Calgary, Alt, Canada	1-10-78	2642 Halifax St., Burnaby	294-4291	2.96
Amy Eau	F	Biology	Golden, BC, Canada	3-30-76	463 Sunset Cres., Vancouver	681-5417	3.52
...

Table 4.10 Prime Generalized Relation for the Target Class (Graduate Students)

<i>major</i>	<i>age_range</i>	<i>gpa</i>	<i>count%</i>
Science	21...25	good	5.53
Science	26...30	good	5.02
Science	over_30	very good	5.86
...
Business	over_30	excellent	4.68

Table 4.11 Prime Generalized Relation for the Contrasting Class (Undergraduate Students)

<i>major</i>	<i>age_range</i>	<i>gpa</i>	<i>count%</i>
Science	16...20	fair	5.53
Science	16...20	good	4.53
...
Science	26...30	good	2.32
...
Business	over_30	excellent	0.68

Finally, the resulting class comparison is presented in the form of tables, graphs, and/or rules. This visualization includes a contrasting measure (e.g., *count%*) that compares the target class and the contrasting class. For example, 5.02% of the graduate students majoring in science are between 26 and 30 years old and have a “good” GPA, while only 2.32% of undergraduates have these same characteristics. Drilling and other

OLAP operations may be performed on the target and contrasting classes as deemed necessary by the user in order to adjust the abstraction levels of the final description. ■

In summary, attribute-oriented induction for data characterization and generalization provides an alternative data generalization method in comparison to the data cube approach. It is not confined to relational data because such an induction can be performed on spatial, multimedia, sequence, and other kinds of data sets. In addition, there is no need to precompute a data cube because generalization can be performed online upon receiving a user's query.

Moreover, automated analysis can be added to such an induction process to automatically filter out irrelevant or unimportant attributes. However, because attribute-oriented induction automatically generalizes data to a higher level, it cannot efficiently support the process of drilling down to levels deeper than those provided in the generalized relation. The integration of data cube technology with attribute-oriented induction may provide a balance between precomputation and online computation. This would also support fast online computation when it is necessary to drill down to a level deeper than that provided in the generalized relation.

4.6 Summary

- A **data warehouse** is a *subject-oriented, integrated, time-variant, and nonvolatile* data collection organized in support of management decision making. Several factors distinguish data warehouses from operational databases. Because the two systems provide quite different functionalities and require different kinds of data, it is necessary to maintain data warehouses separately from operational databases.
- Data warehouses often adopt a **three-tier architecture**. The bottom tier is a *warehouse database server*, which is typically a relational database system. The middle tier is an *OLAP server*, and the top tier is a *client* that contains query and reporting tools.
- A data warehouse contains **back-end tools and utilities** for populating and refreshing the warehouse. These cover data extraction, data cleaning, data transformation, loading, refreshing, and warehouse management.
- Data warehouse **metadata** are data defining the warehouse objects. A metadata repository provides details regarding the warehouse structure, data history, the algorithms used for summarization, mappings from the source data to the warehouse form, system performance, and business terms and issues.
- A **multidimensional data model** is typically used for the design of corporate *data warehouses* and *departmental data marts*. Such a model can adopt a *star schema*, *snowflake schema*, or *fact constellation schema*. The core of the *multidimensional model* is the **data cube**, which consists of a large set of *facts* (or *measures*) and a number of *dimensions*. Dimensions are the entities or perspectives with respect to which an organization wants to keep records and are hierarchical in nature.

- A data cube consists of a **lattice of cuboids**, each corresponding to a different degree of summarization of the given multidimensional data.
- **Concept hierarchies** organize the values of attributes or dimensions into gradual abstraction levels. They are useful in mining at multiple abstraction levels.
- **Online analytical processing** can be performed in data warehouses/marts using the multidimensional data model. Typical OLAP operations include *roll-up*, and *drill-(down, across, through)*, *slice-and-dice*, and *pivot (rotate)*, as well as statistical operations such as ranking and computing moving averages and growth rates. OLAP operations can be implemented efficiently using the data cube structure.
- Data warehouses are used for *information processing* (querying and reporting), *analytical processing* (which allows users to navigate through summarized and detailed data by OLAP operations), and *data mining* (which supports knowledge discovery). OLAP-based data mining is referred to as **multidimensional data mining** (also known as exploratory multidimensional data mining, online analytical mining, or OLAM). It emphasizes the interactive and exploratory nature of data mining.
- OLAP servers may adopt a **relational OLAP (ROLAP)**, a **multidimensional OLAP (MOLAP)**, or a **hybrid OLAP (HOLAP)** implementation. A ROLAP server uses an extended relational DBMS that maps OLAP operations on multidimensional data to standard relational operations. A MOLAP server maps multidimensional data views directly to array structures. A HOLAP server combines ROLAP and MOLAP. For example, it may use ROLAP for historic data while maintaining frequently accessed data in a separate MOLAP store.
- **Full materialization** refers to the computation of all of the cuboids in the lattice defining a data cube. It typically requires an excessive amount of storage space, particularly as the number of dimensions and size of associated concept hierarchies grow. This problem is known as the **curse of dimensionality**. Alternatively, **partial materialization** is the selective computation of a subset of the cuboids or subcubes in the lattice. For example, an **iceberg cube** is a data cube that stores only those cube cells that have an aggregate value (e.g., *count*) above some minimum support threshold.
- OLAP query processing can be made more efficient with the use of indexing techniques. In **bitmap indexing**, each attribute has its own bitmap index table. Bitmap indexing reduces join, aggregation, and comparison operations to bit arithmetic. **Join indexing** registers the joinable rows of two or more relations from a relational database, reducing the overall cost of OLAP join operations. **Bitmapped join indexing**, which combines the bitmap and join index methods, can be used to further speed up OLAP query processing.
- **Data generalization** is a process that abstracts a large set of task-relevant data in a database from a relatively low conceptual level to higher conceptual levels. Data generalization approaches include data cube-based data aggregation and

attribute-oriented induction. **Concept description** is the most basic form of descriptive data mining. It describes a given set of task-relevant data in a concise and summarative manner, presenting interesting general properties of the data. Concept (or class) description consists of **characterization** and **comparison** (or **discrimination**). The former summarizes and describes a data collection, called the **target class**, whereas the latter summarizes and distinguishes one data collection, called the **target class**, from other data collection(s), collectively called the **contrasting class(es)**.

- **Concept characterization** can be implemented using **data cube (OLAP-based) approaches** and the **attribute-oriented induction approach**. These are attribute- or dimension-based generalization approaches. The **attribute-oriented induction approach** consists of the following techniques: *data focusing*, *data generalization by attribute removal or attribute generalization*, *count and aggregate value accumulation*, *attribute generalization control*, and *generalization data visualization*.
- **Concept comparison** can be performed using the attribute-oriented induction or data cube approaches in a manner similar to concept characterization. Generalized tuples from the target and contrasting classes can be quantitatively compared and contrasted.

4.7 Exercises

- 4.1 State why, for the integration of multiple heterogeneous information sources, many companies in industry prefer the *update-driven approach* (which constructs and uses data warehouses), rather than the *query-driven approach* (which applies wrappers and integrators). Describe situations where the query-driven approach is preferable to the update-driven approach.
- 4.2 Briefly compare the following concepts. You may use an example to explain your point(s).
 - (a) Snowflake schema, fact constellation, starlet query model
 - (b) Data cleaning, data transformation, refresh
 - (c) Discovery-driven cube, multifeature cube, virtual warehouse
- 4.3 Suppose that a data warehouse consists of the three dimensions *time*, *doctor*, and *patient*, and the two measures *count* and *charge*, where *charge* is the fee that a doctor charges a patient for a visit.
 - (a) Enumerate three classes of schemas that are popularly used for modeling data warehouses.
 - (b) Draw a schema diagram for the above data warehouse using one of the schema classes listed in (a).

- (c) Starting with the base cuboid [*day, doctor, patient*], what specific OLAP operations should be performed in order to list the total fee collected by each doctor in 2010?
 - (d) To obtain the same list, write an SQL query assuming the data are stored in a relational database with the schema *fee* (*day, month, year, doctor, hospital, patient, count, charge*).
- 4.4 Suppose that a data warehouse for *Big_University* consists of the four dimensions *student*, *course*, *semester*, and *instructor*, and two measures *count* and *avg_grade*. At the lowest conceptual level (e.g., for a given student, course, semester, and instructor combination), the *avg_grade* measure stores the actual course grade of the student. At higher conceptual levels, *avg_grade* stores the average grade for the given combination.
- (a) Draw a *snowflake schema* diagram for the data warehouse.
 - (b) Starting with the base cuboid [*student, course, semester, instructor*], what specific OLAP operations (e.g., roll-up from *semester* to *year*) should you perform in order to list the average grade of CS courses for each *Big_University* student.
 - (c) If each dimension has five levels (including all), such as “*student < major < status < university < all*”, how many cuboids will this cube contain (including the base and apex cuboids)?
- 4.5 Suppose that a data warehouse consists of the four dimensions *date*, *spectator*, *location*, and *game*, and the two measures *count* and *charge*, where *charge* is the fare that a spectator pays when watching a game on a given date. Spectators may be students, adults, or seniors, with each category having its own charge rate.
- (a) Draw a *star schema* diagram for the data warehouse.
 - (b) Starting with the base cuboid [*date, spectator, location, game*], what specific OLAP operations should you perform in order to list the total charge paid by student spectators at *GM_Place* in 2010?
 - (c) *Bitmap indexing* is useful in data warehousing. Taking this cube as an example, briefly discuss advantages and problems of using a bitmap index structure.
- 4.6 A data warehouse can be modeled by either a *star schema* or a *snowflake schema*. Briefly describe the similarities and the differences of the two models, and then analyze their advantages and disadvantages with regard to one another. Give your opinion of which might be more empirically useful and state the reasons behind your answer.
- 4.7 Design a data warehouse for a regional weather bureau. The weather bureau has about 1000 probes, which are scattered throughout various land and ocean locations in the region to collect basic weather data, including air pressure, temperature, and precipitation at each hour. All data are sent to the central station, which has collected such data for more than 10 years. Your design should facilitate efficient querying and online analytical processing, and derive general weather patterns in multidimensional space.
- 4.8 A popular data warehouse implementation is to construct a multidimensional database, known as a data cube. Unfortunately, this may often generate a huge, yet very sparse, multidimensional matrix.

- (a) Present an example illustrating such a huge and sparse data cube.
- (b) Design an implementation method that can elegantly overcome this sparse matrix problem. Note that you need to explain your data structures in detail and discuss the space needed, as well as how to retrieve data from your structures.
- (c) Modify your design in (b) to handle *incremental data updates*. Give the reasoning behind your new design.

4.9 Regarding the *computation of measures* in a data cube:

- (a) Enumerate three categories of measures, based on the kind of aggregate functions used in computing a data cube.
- (b) For a data cube with the three dimensions *time*, *location*, and *item*, which category does the function *variance* belong to? Describe how to compute it if the cube is partitioned into many chunks.
Hint: The formula for computing *variance* is $\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x}_i)^2$, where \bar{x}_i is the average of x_i s.
- (c) Suppose the function is “*top 10 sales*.” Discuss how to efficiently compute this measure in a data cube.

4.10 Suppose a company wants to design a data warehouse to facilitate the analysis of moving vehicles in an online analytical processing manner. The company registers huge amounts of auto movement data in the format of (*Auto_ID*, *location*, *speed*, *time*). Each *Auto_ID* represents a vehicle associated with information (e.g., *vehicle_category*, *driver_category*), and each location may be associated with a street in a city. Assume that a street map is available for the city.

- (a) Design such a data warehouse to facilitate effective online analytical processing in multidimensional space.
- (b) The movement data may contain noise. Discuss how you would develop a method to automatically discover data records that were likely erroneously registered in the data repository.
- (c) The movement data may be sparse. Discuss how you would develop a method that constructs a reliable data warehouse despite the sparsity of data.
- (d) If you want to drive from A to B starting at a particular time, discuss how a system may use the data in this warehouse to work out a fast route.

4.11 Radio-frequency identification is commonly used to trace object movement and perform inventory control. An RFID reader can successfully read an RFID tag from a limited distance at any scheduled time. Suppose a company wants to design a data warehouse to facilitate the analysis of objects with RFID tags in an online analytical processing manner. The company registers huge amounts of RFID data in the format of (*RFID*, *at_location*, *time*), and also has some information about the objects carrying the RFID tag, for example, (*RFID*, *product_name*, *product_category*, *producer*, *date_produced*, *price*).

- (a) Design a data warehouse to facilitate effective registration and online analytical processing of such data.

- (b) The RFID data may contain lots of redundant information. Discuss a method that maximally reduces redundancy during data registration in the RFID data warehouse.
 - (c) The RFID data may contain lots of noise such as missing registration and misread IDs. Discuss a method that effectively cleans up the noisy data in the RFID data warehouse.
 - (d) You may want to perform online analytical processing to determine how many TV sets were shipped from the LA seaport to BestBuy in Champaign, IL, by *month*, *brand*, and *price_range*. Outline how this could be done efficiently if you were to store such RFID data in the warehouse.
 - (e) If a customer returns a jug of milk and complains that it has spoiled before its expiration date, discuss how you can investigate such a case in the warehouse to find out what the problem is, either in shipping or in storage.
- 4.12 In many applications, new data sets are incrementally added to the existing large data sets. Thus, an important consideration is whether a measure can be computed efficiently in an incremental manner. Use *count*, *standard deviation*, and *median* as examples to show that a distributive or algebraic measure facilitates efficient incremental computation, whereas a holistic measure does not.
- 4.13 Suppose that we need to record three measures in a data cube: *min()*, *average()*, and *median()*. Design an efficient computation and storage method for each measure given that the cube allows data to be *deleted incrementally* (i.e., in small portions at a time) from the cube.
- 4.14 In data warehouse technology, a multiple dimensional view can be implemented by a relational database technique (*ROLAP*), by a multidimensional database technique (*MOLAP*), or by a hybrid database technique (*HOLAP*).
- (a) Briefly describe each implementation technique.
 - (b) For each technique, explain how each of the following functions may be implemented:
 - i. The generation of a data warehouse (including aggregation)
 - ii. Roll-up
 - iii. Drill-down
 - iv. Incremental updating
 - (c) Which implementation techniques do you prefer, and why?
- 4.15 Suppose that a data warehouse contains 20 dimensions, each with about five levels of granularity.
- (a) Users are mainly interested in four particular dimensions, each having three frequently accessed levels for rolling up and drilling down. How would you design a data cube structure to support this preference efficiently?
 - (b) At times, a user may want to *drill through* the cube to the raw data for one or two particular dimensions. How would you support this feature?

- 4.16 A data cube, C , has n dimensions, and each dimension has exactly p distinct values in the base cuboid. Assume that there are no concept hierarchies associated with the dimensions.
- (a) What is the *maximum number of cells* possible in the base cuboid?
 - (b) What is the *minimum number of cells* possible in the base cuboid?
 - (c) What is the *maximum number of cells* possible (including both base cells and aggregate cells) in the C data cube?
 - (d) What is the *minimum number of cells* possible in C ?
- 4.17 What are the differences between the three main types of data warehouse usage: *information processing*, *analytical processing*, and *data mining*? Discuss the motivation behind *OLAP mining (OLAM)*.

4.8 Bibliographic Notes

There are a good number of introductory-level textbooks on data warehousing and OLAP technology—for example, Kimball, Ross, Thornthwaite, et al. [KRTM08]; Imhoff, Galembo, and Geiger [IGG03]; and Inmon [Inm96]. Chaudhuri and Dayal [CD97] provide an early overview of data warehousing and OLAP technology. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [GM99].

The history of decision support systems can be traced back to the 1960s. However, the proposal to construct large data warehouses for multidimensional data analysis is credited to Codd [CCS93] who coined the term *OLAP* for *online analytical processing*. The OLAP Council was established in 1995. Widom [Wid95] identified several research problems in data warehousing. Kimball and Ross [KR02] provide an overview of the deficiencies of SQL regarding the ability to support comparisons that are common in the business world, and present a good set of application cases that require data warehousing and OLAP technology. For an overview of OLAP systems versus statistical databases, see Shoshani [Sho97].

Gray et al. [GCB⁺97] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and subtotals. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Data cube computation methods have been investigated by numerous studies such as Sarawagi and Stonebraker [SS94]; Agarwal et al. [AAD⁺96]; Zhao, Deshpande, and Naughton [ZDN97]; Ross and Srivastava [RS97]; Beyer and Ramakrishnan [BR99]; Han, Pei, Dong, and Wang [HPDW01]; and Xin, Han, Li, and Wah [XHLW03]. These methods are discussed in depth in Chapter 5.

The concept of iceberg queries was first introduced in Fang, Shivakumar, Garcia-Molina et al. [FSGM⁺98]. The use of join indices to speed up relational query processing was proposed by Valduriez [Val87]. O’Neil and Graefe [OG95] proposed a bitmapped