8 Mining Stream, Time-Series, and Sequence Data

Our previous chapters introduced the basic concepts and techniques of data mining. The techniques studied, however, were for simple and structured data sets, such as data in relational databases, transactional databases, and data warehouses. The growth of data in various *complex forms* (e.g., semi-structured and unstructured, spatial and temporal, hypertext and multimedia) has been explosive owing to the rapid progress of data collection and advanced database system technologies, and the World Wide Web. Therefore, an increasingly important task in data mining is to mine complex types of data. Furthermore, many data mining applications need to mine patterns that are more sophisticated than those discussed earlier, including sequential patterns, subgraph patterns, and features in interconnected networks. We treat such tasks as advanced topics in data mining.

In the following chapters, we examine how to further develop the essential data mining techniques (such as characterization, association, classification, and clustering) and how to develop new ones to cope with complex types of data. We start off, in this chapter, by discussing the mining of stream, time-series, and sequence data. Chapter 9 focuses on the mining of graphs, social networks, and multirelational data. Chapter 10 examines mining object, spatial, multimedia, text, and Web data. Research into such mining is fast evolving. Our discussion provides a broad introduction. We expect that many new books dedicated to the mining of complex kinds of data will become available in the future.

As this chapter focuses on the mining of stream data, time-series data, and sequence data, let's look at each of these areas.

Imagine a satellite-mounted remote sensor that is constantly generating data. The data are massive (e.g., terabytes in volume), temporally ordered, fast changing, and potentially infinite. This is an example of *stream data*. Other examples include telecommunications data, transaction data from the retail industry, and data from electric power grids. Traditional OLAP and data mining methods typically require multiple scans of the data and are therefore infeasible for stream data applications. In Section 8.1, we study advanced mining methods for the analysis of such constantly flowing data.

A *time-series database* consists of sequences of values or events obtained over repeated measurements of time. Suppose that you are given time-series data relating to stock market prices. How can the data be analyzed to identify trends? Given such data for

two different stocks, can we find any similarities between the two? These questions are explored in Section 8.2. Other applications involving time-series data include economic and sales forecasting, utility studies, and the observation of natural phenomena (such as atmosphere, temperature, and wind).

A *sequence database* consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. *Sequential pattern mining* is the discovery of frequently occurring ordered events or subsequences as patterns. An example of a sequential pattern is "*Customers who buy a Canon digital camera are likely to buy an HP color printer within a month*." Periodic patterns, which recur in regular periods or durations, are another kind of pattern related to sequences. Section 8.3 studies methods of sequential pattern mining.

Recent research in bioinformatics has resulted in the development of numerous methods for the analysis of biological sequences, such as DNA and protein sequences. Section 8.4 introduces several popular methods, including biological *sequence alignment algorithms* and the *hidden Markov model*.

8.1 Mining Data Streams

Tremendous and potentially infinite volumes of *data streams* are often generated by real-time surveillance systems, communication networks, Internet traffic, on-line transactions in the financial market or retail industry, electric power grids, industry production processes, scientific and engineering experiments, remote sensors, and other dynamic environments. Unlike traditional data sets, **stream data** flow in and out of a computer system *continuously* and with varying update rates. They are *temporally ordered, fast changing, massive, and potentially infinite*. It may be impossible to store an entire data stream or to scan through it multiple times due to its tremendous volume. Moreover, stream data tend to be of a rather low level of abstraction, whereas most analysts are interested in relatively high-level dynamic changes, such as trends and deviations. To discover knowledge or patterns from data streams, it is necessary to develop single-scan, on-line, multilevel, multidimensional stream processing and analysis methods.

Such single-scan, on-line data analysis methodology should not be confined to only stream data. It is also critically important for processing nonstream data that are massive. With data volumes mounting by terabytes or even petabytes, stream data nicely capture our data processing needs of today: even when the complete set of data is collected and can be stored in massive data storage devices, single scan (as in data stream systems) instead of random access (as in database systems) may still be the most realistic processing mode, because it is often too expensive to scan such a data set multiple times.

In this section, we introduce several on-line stream data analysis and mining methods. Section 8.1.1 introduces the basic methodologies for stream data processing and querying. Multidimensional analysis of stream data, encompassing stream data cubes and multiple granularities of time, is described in Section 8.1.2. Frequent-pattern mining and classification are presented in Sections 8.1.3 and 8.1.4, respectively. The clustering of dynamically evolving data streams is addressed in Section 8.1.5.

8.1.1 Methodologies for Stream Data Processing and Stream Data Systems

As seen from the previous discussion, it is impractical to scan through an entire data stream more than once. Sometimes we cannot even "look" at every element of a stream because the stream flows in so fast and changes so quickly. The gigantic size of such data sets also implies that we generally cannot store the entire stream data set in main memory or even on disk. The problem is not just that there is a lot of data, it is that the universes that we are keeping track of are relatively large, where a *universe* is the domain of possible values for an attribute. For example, if we were tracking the ages of millions of people, our universe would be relatively small, perhaps between zero and one hundred and twenty. We could easily maintain exact summaries of such data. In contrast, the universe corresponding to the set of all pairs of IP addresses on the Internet is very large, which makes exact storage intractable. A reasonable way of thinking about data streams is to actually think of a physical stream of water. Heraclitus once said that you can never step in the same stream twice,[1] and so it is with stream data.

For effective processing of stream data, new data structures, techniques, and algorithms are needed. Because we do not have an infinite amount of space to store stream data, we often trade off between accuracy and storage. That is, we generally are willing to settle for approximate rather than exact answers. **Synopses** allow for this by providing *summaries* of the data, which typically can be used to return approximate answers to queries. Synopses use *synopsis data structures*, which are any data structures that are substantially *smaller* than their base data set (in this case, the stream data). From the algorithmic point of view, we want our algorithms to be efficient in both space and time. Instead of storing all or most elements seen so far, using $O(N)$ space, we often want to use polylogarithmic space, $O(\log^k N)$, where N is the number of elements in the stream data. We may relax the requirement that our answers are exact, and ask for approximate answers within a small error range with high probability. That is, many data stream–based algorithms compute an approximate answer within a factor ε of the actual answer, with high probability. Generally, as the approximation factor $(1+\varepsilon)$ goes down, the space requirements go up. In this section, we examine some common synopsis data structures and techniques.

Random Sampling

Rather than deal with an entire data stream, we can think of *sampling* the stream at periodic intervals. "*To obtain an unbiased sampling of the data, we need to know the length of the stream in advance. But what can we do if we do not know this length in advance?*" In this case, we need to modify our approach.

[1] Plato citing Heraclitus: "Heraclitus somewhere says that all things are in process and nothing stays still, and likening existing things to the stream of a river he says you would not step twice into the same river."

A technique called **reservoir sampling** can be used to select an unbiased random sample of s elements without replacement. The idea behind reservoir sampling is relatively simple. We maintain a sample of size at least s, called the "reservoir," from which a random sample of size s can be generated. However, generating this sample from the reservoir can be costly, especially when the reservoir is large. To avoid this step, we maintain a set of s *candidates* in the reservoir, which form a true random sample of the elements seen so far in the stream. As the data stream flows, every new element has a certain probability of replacing an old element in the reservoir. Let's say we have seen N elements thus far in the stream. The probability that a new element replaces an old one, chosen at random, is then s/N. This maintains the invariant that the set of s candidates in our reservoir forms a random sample of the elements seen so far.

Sliding Windows

Instead of sampling the data stream randomly, we can use the **sliding window model** to analyze stream data. The basic idea is that rather than running computations on all of the data seen so far, or on some sample, we can make decisions based only on *recent data*. More formally, at every time t, a new data element arrives. This element "expires" at time $t + w$, where w is the window "size" or length. The sliding window model is useful for stocks or sensor networks, where only recent events may be important. It also reduces memory requirements because only a small window of data is stored.

Histograms

The histogram is a synopsis data structure that can be used to approximate the frequency distribution of element values in a data stream. A **histogram** partitions the data into a set of contiguous *buckets*. Depending on the partitioning rule used, the *width* (bucket value range) and *depth* (number of elements per bucket) can vary. The equal-width partitioning rule is a simple way to construct histograms, where the range of each bucket is the same. Although easy to implement, this may not sample the probability distribution function well. A better approach is to use V-Optimal histograms (see Section 2.5.4). Similar to clustering, V-Optimal histograms define bucket sizes that minimize the frequency variance within each bucket, which better captures the distribution of the data. These histograms can then be used to approximate query answers rather than using sampling techniques.

Multiresolution Methods

A common way to deal with a large amount of data is through the use of *data reduction* methods (see Section 2.5). A popular data reduction method is the use of divide-and-conquer strategies such as multiresolution data structures. These allow a program to trade off between accuracy and storage, but also offer the ability to understand a data stream at multiple levels of detail.

A concrete example is a *balanced binary tree*, where we try to maintain this balance as new data come in. Each level of the tree provides a different resolution. The farther away we are from the tree root, the more detailed is the level of resolution.

A more sophisticated way to form multiple resolutions is to use a clustering method to organize stream data into a hierarchical structure of trees. For example, we can use a typical hierarchical clustering data structure like CF-tree in BIRCH (see Section 7.5.2) to form a hierarchy of *microclusters*. With dynamic stream data flowing in and out, summary statistics of data streams can be incrementally updated over time in the hierarchy of microclusters. Information in such microclusters can be aggregated into larger *macroclusters* depending on the application requirements to derive general data statistics at multiresolution.

Wavelets (Section 2.5.3), a technique from signal processing, can be used to build a multiresolution hierarchy structure over an input signal, in this case, the stream data. Given an input signal, we would like to break it down or rewrite it in terms of simple, orthogonal basis functions. The simplest basis is the Haar wavelet. Using this basis corresponds to recursively performing averaging and differencing at multiple levels of resolution. Haar wavelets are easy to understand and implement. They are especially good at dealing with spatial and multimedia data. Wavelets have been used as approximations to histograms for query optimization. Moreover, wavelet-based histograms can be dynamically maintained over time. Thus, wavelets are a popular multiresolution method for data stream compression.

Sketches

Synopses techniques mainly differ by how exactly they trade off accuracy for storage. Sampling techniques and sliding window models focus on a small part of the data, whereas other synopses try to summarize the entire data, often at multiple levels of detail. Some techniques require multiple passes over the data, such as histograms and wavelets, whereas other methods, such as *sketches*, can operate in a single pass.

Suppose that, ideally, we would like to maintain the full histogram over the universe of objects or elements in a data stream, where the universe is $U = \{1, 2, \ldots, v\}$ and the stream is $A = \{a_1, a_2, \ldots, a_N\}$. That is, for each value i in the universe, we want to maintain the frequency or number of occurrences of i in the sequence A. If the universe is large, this structure can be quite large as well. Thus, we need a smaller representation instead.

Let's consider the **frequency moments** of A. These are the numbers, F_k, defined as

$$F_k = \sum_{i=1}^{v} m_i^k, \tag{8.1}$$

where v is the universe or domain size (as above), m_i is the frequency of i in the sequence, and $k \geq 0$. In particular, F_0 is the number of distinct elements in the sequence. F_1 is the length of the sequence (that is, N, here). F_2 is known as the *self-join size*, the repeat rate, or as Gini's index of homogeneity. The frequency moments of a data set provide useful information about the data for database applications, such as query answering. In addition, they indicate the degree of *skew* or asymmetry in the data (Section 2.2.1), which

is useful in parallel database applications for determining an appropriate partitioning algorithm for the data.

When the amount of memory available is smaller than v, we need to employ a synopsis. The estimation of the frequency moments can be done by synopses that are known as **sketches**. These build a small-space summary for a distribution vector (e.g., histogram) using randomized linear projections of the underlying data vectors. Sketches provide probabilistic guarantees on the quality of the approximate answer (e.g., the answer to the given query is 12 ± 1 with a probability of 0.90). Given N elements and a universe U of v values, such sketches can approximate F_0, F_1, and F_2 in $O(\log v + \log N)$ space. The basic idea is to hash every element uniformly at random to either $z_i \in \{-1, +1\}$, and then maintain a random variable, $X = \sum_i m_i z_i$. It can be shown that X^2 is a good estimate for F_2. To explain why this works, we can think of hashing elements to -1 or $+1$ as assigning each element value to an arbitrary side of a tug of war. When we sum up to get X, we can think of measuring the displacement of the rope from the center point. By squaring X, we square this displacement, capturing the data skew, F_2.

To get an even better estimate, we can maintain multiple random variables, X_i. Then by choosing the median value of the square of these variables, we can increase our confidence that the estimated value is close to F_2.

From a database perspective, **sketch partitioning** was developed to improve the performance of sketching on data stream query optimization. Sketch partitioning uses coarse statistical information on the base data to *intelligently* partition the domain of the underlying attributes in a way that provably tightens the error guarantees.

Randomized Algorithms

Randomized algorithms, in the form of random sampling and sketching, are often used to deal with massive, high-dimensional data streams. The use of randomization often leads to simpler and more efficient algorithms in comparison to known deterministic algorithms.

If a randomized algorithm always returns the right answer but the running times vary, it is known as a **Las Vegas** algorithm. In contrast, a **Monte Carlo** algorithm has bounds on the running time but may not return the correct result. We mainly consider Monte Carlo algorithms. One way to think of a randomized algorithm is simply as a probability distribution over a set of deterministic algorithms.

Given that a randomized algorithm returns a random variable as a result, we would like to have bounds on the tail probability of that random variable. This tells us that the probability that a random variable deviates from its expected value is small. One basic tool is **Chebyshev's Inequality**. Let X be a random variable with mean μ and standard deviation σ (variance σ^2). Chebyshev's inequality says that

$$P(|X - \mu| > k) \leq \frac{\sigma^2}{k^2} \tag{8.2}$$

for any given positive real number, k. This inequality can be used to bound the variance of a random variable.

In many cases, multiple random variables can be used to boost the confidence in our results. As long as these random variables are fully independent, **Chernoff bounds** can be used. Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials. In a Poisson trial, the probability of success varies from trial to trial. If X is the sum of X_1 to X_n, then a weaker version of the Chernoff bound tells us that

$$\Pr[X < (1+\delta)\mu] < e^{-\mu\delta^2/4} \tag{8.3}$$

where $\delta \in (0, 1]$. This shows that the probability decreases exponentially as we move from the mean, which makes poor estimates much more unlikely.

Data Stream Management Systems and Stream Queries

In traditional database systems, data are stored in finite and persistent databases. However, stream data are infinite and impossible to store fully in a database. In a **Data Stream Management System** (DSMS), there may be multiple data streams. They arrive on-line and are continuous, temporally ordered, and potentially infinite. Once an element from a data stream has been processed, it is discarded or archived, and it cannot be easily retrieved unless it is explicitly stored in memory.

A stream data query processing architecture includes three parts: *end user, query processor*, and *scratch space* (which may consist of main memory and disks). An end user issues a query to the DSMS, and the query processor takes the query, processes it using the information stored in the scratch space, and returns the results to the user.

Queries can be either *one-time queries* or *continuous queries*. A **one-time query** is evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. A **continuous query** is evaluated continuously as data streams continue to arrive. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. A continuous query can act as a watchdog, as in "*sound the alarm if the power consumption for Block 25 exceeds a certain threshold.*" Moreover, a query can be **predefined** (i.e., supplied to the data stream management system before any relevant data have arrived) or **ad hoc** (i.e., issued on-line after the data streams have already begun). A predefined query is generally a continuous query, whereas an ad hoc query can be either one-time or continuous.

Stream Query Processing

The special properties of stream data introduce new challenges in query processing. In particular, data streams may grow unboundedly, and it is possible that queries may require unbounded memory to produce an exact answer. How can we distinguish between queries that can be answered exactly using a given bounded amount of memory and queries that must be approximated? Actually, without knowing the size of the input data streams, it is impossible to place a limit on the memory requirements for most common queries, such as those involving joins, unless the domains of the attributes involved in the query are restricted. This is because without domain restrictions, an unbounded

number of attribute values must be remembered because they might turn out to join with tuples that arrive in the future.

Providing an exact answer to a query may require unbounded main memory; therefore a more realistic solution is to provide an approximate answer to the query. *Approximate query answering* relaxes the memory requirements and also helps in handling system load, because streams can come in too fast to process exactly. In addition, ad hoc queries need approximate history to return an answer. We have already discussed common synopses that are useful for approximate query answering, such as random sampling, sliding windows, histograms, and sketches.

As this chapter focuses on stream data mining, we will not go into any further details of stream query processing methods. For additional discussion, interested readers may consult the literature recommended in the bibliographic notes of this chapter.

8.1.2 Stream OLAP and Stream Data Cubes

Stream data are generated continuously in a dynamic environment, with huge volume, infinite flow, and fast-changing behavior. It is impossible to store such data streams completely in a data warehouse. Most stream data represent low-level information, consisting of various kinds of detailed temporal and other features. To find interesting or unusual patterns, it is essential to perform *multidimensional analysis* on aggregate measures (such as sum and average). This would facilitate the discovery of critical changes in the data at higher levels of abstraction, from which users can drill down to examine more detailed levels, when needed. Thus multidimensional OLAP analysis is still needed in stream data analysis, but how can we implement it?

Consider the following motivating example.

Example 8.1 **Multidimensional analysis for power supply stream data.** A power supply station generates infinite streams of power usage data. Suppose *individual_user, street_address*, and *second* are the attributes at the lowest level of granularity. Given a large number of users, it is only realistic to analyze the fluctuation of power usage at certain high levels, such as by city or street district and by quarter (of an hour), making timely power supply adjustments and handling unusual situations.

Conceptually, for multidimensional analysis, we can view such stream data as a *virtual* data cube, consisting of one or a few measures and a set of dimensions, including one *time* dimension, and a few other dimensions, such as *location*, *user-category*, and so on. However, in practice, it is impossible to materialize such a data cube, because the materialization requires a huge amount of data to be computed and stored. Some efficient methods must be developed for systematic analysis of such data. ■

Data warehouse and OLAP technology is based on the integration and consolidation of data in multidimensional space to facilitate powerful and fast on-line data analysis. A fundamental difference in the analysis of stream data from that of relational and warehouse data is that the stream data are generated in huge volume, flowing in and out dynamically and changing rapidly. Due to limited memory, disk space, and processing

power, it is impossible to register completely the detailed level of data and compute a fully materialized cube. A realistic design is to explore several data compression techniques, including (1) *tilted time frame* on the time dimension, (2) storing data only at some *critical layers*, and (3) exploring efficient computation of a *very partially materialized data cube*. The (partial) stream data cubes so constructed are much smaller than those constructed from the raw stream data but will still be effective for multidimensional stream data analysis. We examine such a design in more detail.

Time Dimension with Compressed Time Scale: Tilted Time Frame

In stream data analysis, people are usually interested in recent changes at a fine scale but in long-term changes at a coarse scale. Naturally, we can register time at different levels of granularity. The most recent time is registered at the finest granularity; the more distant time is registered at a coarser granularity; and the level of coarseness depends on the application requirements and on how old the time point is (from the current time). Such a time dimension model is called a **tilted time frame**. This model is sufficient for many analysis tasks and also ensures that the total amount of data to retain in memory or to be stored on disk is small.

There are many possible ways to design a titled time frame. Here we introduce three models, as illustrated in Figure 8.1: (1) *natural tilted time frame model*, (2) *logarithmic tilted time frame model*, and (3) *progressive logarithmic tilted time frame model*.

A **natural tilted time frame model** is shown in Figure 8.1(a), where the time frame (or window) is structured in multiple granularities based on the "natural" or usual time scale: the most recent 4 quarters (15 minutes), followed by the last 24 hours, then 31 days, and then 12 months (the actual scale used is determined by the application). Based on this model, we can compute frequent itemsets in the last hour with the precision of a quarter of an hour, or in the last day with the precision of an hour, and

(a) A natural tilted time frame model.

(b) A logarithmic tilted time frame model.

Frame no.	Snapshots (by clock time)
0	69 67 65
1	66 62 58
2	68 60 52
3	56 40 24
4	48 16
5	64 32

(c) A progressive logarithmic tilted time frame table.

Figure 8.1 Three models for tilted time frames.

so on until the whole year with the precision of a month.[2] This model registers only $4 + 24 + 31 + 12 = 71$ units of time for a year instead of $365 \times 24 \times 4 = 35{,}040$ units, with an acceptable trade-off of the grain of granularity at a distant time.

The second model is the **logarithmic tilted time frame model**, as shown in Figure 8.1(b), where the time frame is structured in multiple granularities according to a logarithmic scale. Suppose that the most recent slot holds the transactions of the current quarter. The remaining slots are for the last quarter, the next two quarters (ago), 4 quarters, 8 quarters, 16 quarters, and so on, growing at an exponential rate. According to this model, with one year of data and the finest precision at a quarter, we would need $\log_2(365 \times 24 \times 4) + 1 = 16.1$ units of time instead of $365 \times 24 \times 4 = 35{,}040$ units. That is, we would just need 17 time frames to store the compressed information.

The third method is the **progressive logarithmic tilted time frame model**, where snapshots are stored at differing levels of granularity depending on the recency. Let T be the clock time elapsed since the beginning of the stream. Snapshots are classified into different *frame numbers*, which can vary from 0 to *max_frame*, where $\log_2(T) - max\_capacity \leq max\_frame \leq \log_2(T)$, and *max_capacity* is the maximal number of snapshots held in each frame.

Each snapshot is represented by its timestamp. The rules for insertion of a snapshot t (at time t) into the snapshot frame table are defined as follows: (1) if $(t \bmod 2^i) = 0$ but $(t \bmod 2^{i+1}) \neq 0$, t is inserted into *frame_number* i if $i \leq max\_frame$; otherwise (i.e., $i > max\_frame$), t is inserted into *max_frame*; and (2) each slot has a *max_capacity*. At the insertion of t into *frame_number* i, if the slot already reaches its *max_capacity*, the oldest snapshot in this frame is removed and the new snapshot inserted.

Example 8.2 **Progressive logarithmic tilted time frame.** Consider the snapshot frame table of Figure 8.1(c), where *max_frame* is 5 and *max_capacity* is 3. Let's look at how timestamp 64 was inserted into the table. We know $(64 \bmod 2^6) = 0$ but $(64 \bmod 2^7) \neq 0$, that is, $i = 6$. However, since this value of i exceeds *max_frame*, 64 was inserted into frame 5 instead of frame 6. Suppose we now need to insert a timestamp of 70. At time 70, since $(70 \bmod 2^1) = 0$ but $(70 \bmod 2^2) \neq 0$, we would insert 70 into *frame_number* 1. This would knock out the oldest snapshot of 58, given the slot capacity of 3. From the table, we see that the closer a timestamp is to the current time, the denser are the snapshots stored. ■

In the logarithmic and progressive logarithmic models discussed above, we have assumed that the base is 2. Similar rules can be applied to any base α, where α is an integer and $\alpha > 1$. All three tilted time frame models provide a natural way for incremental insertion of data and for gradually fading out older values.

The tilted time frame models shown are sufficient for typical time-related queries, and at the same time, ensure that the total amount of data to retain in memory and/or to be computed is small.

[2] We align the time axis with the natural calendar time. Thus, for each granularity level of the tilted time frame, there might be a partial interval, which is less than a full unit at that level.

Depending on the given application, we can provide different fading factors in the titled time frames, such as by placing more weight on the more recent time frames. We can also have flexible alternative ways to design the tilted time frames. For example, suppose that we are interested in comparing the stock average from each day of the current week with the corresponding averages from the same weekdays last week, last month, or last year. In this case, we can single out Monday to Friday instead of compressing them into the whole week as one unit.

Critical Layers

Even with the *tilted time frame* model, it can still be too costly to dynamically compute and store a materialized cube. Such a cube may have quite a few dimensions, each containing multiple levels with many distinct values. Because stream data analysis has only limited memory space but requires fast response time, we need additional strategies that work in conjunction with the tilted time frame model. One approach is to compute and store only some mission-critical cuboids of the full data cube.

In many applications, it is beneficial to dynamically and incrementally compute and store two critical cuboids (or **layers**), which are determined based on their conceptual and computational importance in stream data analysis. The first layer, called the **minimal interest layer**, is the minimally interesting layer that an analyst would like to study. It is necessary to have such a layer because it is often neither cost effective nor interesting in practice to examine the minute details of stream data. The second layer, called the **observation layer**, is the layer at which an analyst (or an automated system) would like to continuously study the data. This can involve making decisions regarding the signaling of exceptions, or drilling down along certain paths to lower layers to find cells indicating data exceptions.

Example 8.3 **Critical layers for a power supply stream data cube.** Let's refer back to Example 8.1 regarding the multidimensional analysis of stream data for a power supply station. Dimensions at the lowest level of granularity (i.e., the raw data layer) included *individual_user, street_address*, and *second*. At the minimal interest layer, these three dimensions are *user_group, street_block*, and *minute*, respectively. Those at the observation layer are $*$ (meaning **all** *user*), *city*, and *quarter*, respectively, as shown in Figure 8.2.

Based on this design, we would not need to compute any cuboids that are lower than the minimal interest layer because they would be beyond user interest. Thus, to compute our base cuboid, representing the cells of minimal interest, we need to compute and store the (three-dimensional) aggregate cells for the (*user_group*, *street_block*, *minute*) group-by. This can be done by aggregations on the dimensions *user* and *address* by rolling up from *individual_user* to *user_group* and from *street_address* to *street_block*, respectively, and by rolling up on the *time* dimension from *second* to *minute*.

Similarly, the cuboids at the observation layer should be computed dynamically, taking the tilted time frame model into account as well. This is the layer that an analyst takes as an observation deck, watching the current stream data by examining the slope of changes at this layer to make decisions. This layer can be obtained by rolling up the

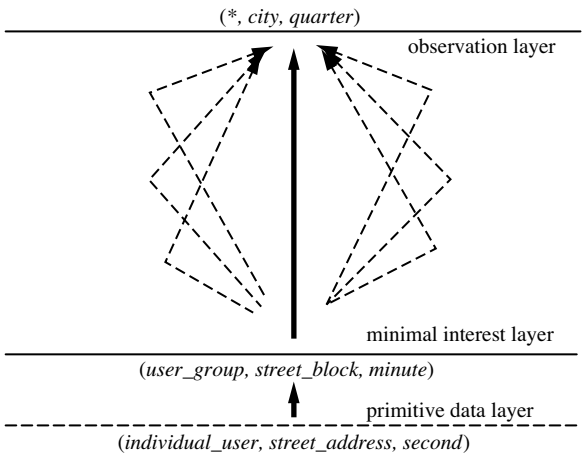


Figure 8.2 Two critical layers in a "power supply station" stream data cube.

cube along the *user* dimension to ∗ (for **all** *user*), along the *address* dimension to *city*, and along the *time* dimension to *quarter*. If something unusual is observed, the analyst can investigate by drilling down to lower levels to find data exceptions. ■

Partial Materialization of a Stream Cube

"*What if a user needs a layer that would be between the two critical layers?*" Materializing a cube at only two critical layers leaves much room for how to compute the cuboids in between. These cuboids can be precomputed fully, partially, or not at all (i.e., leave everything to be computed on the fly). An interesting method is **popular path cubing**, which rolls up the cuboids from the minimal interest layer to the observation layer by following one popular drilling path, materializes only the layers along the path, and leaves other layers to be computed only when needed. This method achieves a reasonable trade-off between space, computation time, and flexibility, and has quick incremental aggregation time, quick drilling time, and small space requirements.

To facilitate efficient computation and storage of the popular path of the stream cube, a compact data structure needs to be introduced so that the space taken in the computation of aggregations is minimized. A hyperlinked tree structure called *H-tree* is revised and adopted here to ensure that a compact structure is maintained in memory for efficient computation of multidimensional and multilevel aggregations.

Each branch of the H-tree is organized in the same order as the specified popular path. The aggregate cells are stored in the nonleaf nodes of the H-tree, forming the computed cuboids along the popular path. Aggregation for each corresponding slot in the tilted time frame is performed from the minimal interest layer all the way up to the observation layer by aggregating along the popular path. The step-by-step aggregation is performed while inserting the new generalized tuples into the corresponding time slots.

The H-tree ordering is based on the popular drilling path given by users or experts. This ordering facilitates the computation and storage of the cuboids along the path. The aggregations along the drilling path from the minimal interest layer to the observation layer are performed during the generalizing of the stream data to the minimal interest layer, which takes only one scan of stream data. Because all the cells to be computed are the cuboids along the popular path, and the cuboids to be computed are the nonleaf nodes associated with the H-tree, both space and computation overheads are minimized.

Although it is impossible to materialize all the cells of a stream cube, the stream data cube so designed facilitates fast on-line drilling along any single dimension or along combinations of a small number of dimensions. The H-tree-based architecture facilitates incremental updating of stream cubes as well.

8.1.3 Frequent-Pattern Mining in Data Streams

As discussed in Chapter 5, frequent-pattern mining finds a set of patterns that occur frequently in a data set, where a pattern can be a set of items (called an *itemset*), a subsequence, or a substructure. A pattern is considered frequent if its count satisfies a *minimum support*. Scalable methods for mining frequent patterns have been extensively studied for static data sets. However, mining such patterns in dynamic data streams poses substantial new challenges. Many existing frequent-pattern mining algorithms require the system to scan the whole data set more than once, but this is unrealistic for infinite data streams. How can we perform incremental updates of frequent itemsets for stream data since an infrequent itemset can become frequent later on, and hence cannot be ignored? Moreover, a frequent itemset can become infrequent as well. The number of infrequent itemsets is exponential and so it is impossible to keep track of all of them.

To overcome this difficulty, there are two possible approaches. One is to keep track of only a predefined, limited set of items and itemsets. This method has very limited usage and expressive power because it requires the system to confine the scope of examination to only the set of predefined itemsets beforehand. The second approach is to derive an *approximate* set of answers. In practice, approximate answers are often sufficient. A number of approximate item or itemset counting algorithms have been developed in recent research. Here we introduce one such algorithm: the Lossy Counting algorithm. It approximates the frequency of items or itemsets within a user-specified error bound, ε. This concept is illustrated as follows.

Example 8.4 **Approximate frequent items.** A router is interested in all items whose frequency is at least 1% (*min_support*) of the entire traffic stream seen so far. It is felt that 1/10 of *min_support* (i.e., $\varepsilon = 0.1\%$) is an acceptable margin of error. This means that all frequent items with a support of at least *min_support* will be output, but that some items with a support of at least (*min_support* $- \varepsilon$) will also be output. ■

Lossy Counting Algorithm

We first introduce the Lossy Counting algorithm for frequent items. This algorithm is fairly simple but quite efficient. We then look at how the method can be extended to find approximate frequent itemsets.

"*How does the Lossy Counting algorithm find frequent items?*" A user first provides two input parameters: (1) the *min_support* threshold, σ, and (2) the error bound mentioned previously, denoted as ε. The incoming stream is conceptually divided into buckets of width $w = \lceil 1/\varepsilon \rceil$. Let N be the current *stream length*, that is, the number of items seen so far. The algorithm uses a frequency-list data structure for all items with frequency greater than 0. For each item, the list maintains f, the approximate frequency count, and Δ, the maximum possible error of f.

The algorithm processes buckets of items as follows. When a new bucket comes in, the items in the bucket are added to the frequency list. If a given item already exists in the list, we simply increase its frequency count, f. Otherwise, we insert it into the list with a frequency count of 1. If the new item is from the bth bucket, we set Δ, the maximum possible error on the frequency count of the item, to be $b-1$. Based on our discussion so far, the item frequency counts hold the actual frequencies rather than approximations. They become approximates, however, because of the next step. Whenever a bucket "boundary" is reached (that is, N has reached a multiple of width w, such as w, $2w$, $3w$, etc.), the frequency list is examined. Let b be the current bucket number. An item entry is deleted if, for that entry, $f + \Delta \leq b$. In this way, the algorithm aims to keep the frequency list small so that it may fit in main memory. The frequency count stored for each item will either be the true frequency of the item or an *underestimate* of it.

"*By how much can a frequency count be underestimated?*" One of the most important factors in approximation algorithms is the approximation ratio (or error bound). Let's look at the case where an item is deleted. This occurs when $f + \Delta \leq b$ for an item, where b is the current bucket number. We know that $b \leq N/w$, that is, $b \leq \varepsilon N$. The actual frequency of an item is at most $f + \Delta$. Thus, the most that an item can be underestimated is εN. If the actual support of this item is σ (this is the minimum support or lower bound for it to be considered frequent), then the actual frequency is σN, and the frequency, f, on the frequency list should be at least $(\sigma N - \varepsilon N)$. Thus, if we output all of the items in the frequency list having an f value of at least $(\sigma N - \varepsilon N)$, then all of the frequent items will be output. In addition, some subfrequent items (with an actual frequency of at least $\sigma N - \varepsilon N$ but less than σN) will be output, too.

The Lossy Counting algorithm has three nice properties: (1) there are no false negatives, that is, there is no true frequent item that is not output; (2) false positives are quite "positive" as well, since the output items will have a frequency of at least $\sigma N - \varepsilon N$; and (3) the frequency of a frequent item can be underestimated by at most εN. For frequent items, this underestimation is only a small fraction of its true frequency, so this approximation is acceptable.

"*How much space is needed to save the frequency list?*" It has been shown that the algorithm takes at most $\frac{1}{\varepsilon} \log(\varepsilon N)$ entries in computation, where N is the stream length so far. If we assume that elements with very low frequency tend to occur

more or less uniformly at random, then it has been shown that Lossy Counting requires no more than $\frac{7}{\varepsilon}$ space. Thus, the space requirement for this algorithm is reasonable.

It is much more difficult to find frequent itemsets than to find frequent items in data streams, because the number of possible itemsets grows exponentially with that of different items. As a consequence, there will be many more frequent itemsets. If we still process the data bucket by bucket, we will probably run out of memory. An alternative is to process as many buckets as we can at a time.

To find frequent itemsets, transactions are still divided by buckets with bucket size, $w = \lceil 1/\varepsilon \rceil$. However, this time, we will read as many buckets as possible into main memory. As before, we maintain a frequency list, although now it pertains to itemsets rather than items. Suppose we can read β buckets into main memory. After that, we update the frequency list by all these buckets as follows. If a given itemset already exists in the frequency list, we update f by counting the occurrences of the itemset among the current batch of β buckets. If the updated entry satisfies $f + \Delta \leq b$, where b is the current bucket number, we delete the entry. If an itemset has frequency $f \geq \beta$ and does not appear in the list, it is inserted as a new entry where Δ is set to $b - \beta$ as the maximum error of f.

In practice, β will be large, such as greater than 30. This approach will save memory because all itemsets with frequency less than β will not be recorded in the frequency list anymore. For smaller values of β (such as 1 for the frequent item version of the algorithm described earlier), more spurious subsets will find their way into the frequency list. This would drastically increase the average size and refresh rate of the frequency list and harm the algorithm's efficiency in both time and space.

In general, Lossy Counting is a simple but effective algorithm for finding frequent items and itemsets approximately. Its limitations lie in three aspects: (1) the space bound is insufficient because the frequency list may grow infinitely as the stream goes on; (2) for frequent itemsets, the algorithm scans each transaction many times and the size of main memory will greatly impact the efficiency of the algorithm; and (3) the output is based on all of the previous data, although users can be more interested in recent data than that in the remote past. A tilted time frame model with different time granularities can be integrated with Lossy Counting in order to emphasize the recency of the data.

8.1.4 Classification of Dynamic Data Streams

In Chapter 6, we studied several methods for the classification of static data. Classification is a two-step process consisting of *learning*, or *model construction* (where a model is constructed based on class-labeled tuples from a training data set), and *classification*, or *model usage* (where the model is used to predict the class labels of tuples from new data sets). The latter lends itself to stream data, as new examples are immediately classified by the model as they arrive.

"*So, does this mean we can apply traditional classification methods to stream data as well?*" In a traditional setting, the training data reside in a relatively static database. Many

classification methods will scan the training data multiple times. Therefore, the first step of model construction is typically performed off-line as a batch process. With data streams, however, there is typically no off-line phase. The data flow in so quickly that storage and multiple scans are infeasible.

To further illustrate how traditional classification methods are inappropriate for stream data, consider the practice of constructing decision trees as models. Most decision tree algorithms tend to follow the same basic top-down, recursive strategy, yet differ in the statistical measure used to choose an optimal splitting attribute. To review, a decision tree consists of internal (nonleaf) nodes, branches, and leaf nodes. An attribute selection measure is used to select the *splitting attribute* for the current node. This is taken to be the attribute that best discriminates the training tuples according to class. In general, branches are grown for each possible value of the splitting attribute, the training tuples are partitioned accordingly, and the process is recursively repeated at each branch. However, in the stream environment, it is neither possible to collect the complete set of data nor realistic to rescan the data, thus such a method has to be re-examined.

Another distinguishing characteristic of data streams is that they are time-varying, as opposed to traditional database systems, where only the current state is stored. This change in the nature of the data takes the form of changes in the target classification model over time and is referred to as **concept drift**. Concept drift is an important consideration when dealing with stream data.

Several methods have been proposed for the classification of stream data. We introduce four of them in this subsection. The first three, namely the *Hoeffding tree algorithm*, *Very Fast Decision Tree (VFDT)*, and *Concept-adapting Very Fast Decision Tree (CVFDT)*, extend traditional decision tree induction. The fourth uses a *classifier ensemble* approach, in which multiple classifiers are considered using a voting method.

Hoeffding Tree Algorithm

The **Hoeffding tree algorithm** is a decision tree learning method for stream data classification. It was initially used to track Web clickstreams and construct models to predict which Web hosts and Web sites a user is likely to access. It typically runs in sublinear time and produces a nearly identical decision tree to that of traditional batch learners. It uses *Hoeffding trees*, which exploit the idea that a small sample can often be enough to choose an optimal splitting attribute. This idea is supported mathematically by the *Hoeffding bound* (or *additive Chernoff bound*). Suppose we make N independent observations of a random variable r with range R, where r is an attribute selection measure. (For a probability, R is one, and for an information gain, it is $\log c$, where c is the number of classes.) In the case of Hoeffding trees, r is information gain. If we compute the mean, $\bar{r}$, of this sample, the **Hoeffding bound** states that the true mean of r is at least $\bar{r} - \varepsilon$, with probability $1 - \delta$, where δ is user-specified and

$$\varepsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2N}}. \tag{8.4}$$

The Hoeffding tree algorithm uses the Hoeffding bound to determine, with high probability, the smallest number, N, of examples needed at a node when selecting a splitting attribute. This attribute would be the same as that chosen using infinite examples! We'll see how this is done shortly. The Hoeffding bound is independent of the probability distribution, unlike most other bound equations. This is desirable, as it may be impossible to know the probability distribution of the information gain, or whichever attribute selection measure is used.

"*How does the Hoeffding tree algorithm use the Hoeffding bound?*" The algorithm takes as input a sequence of training examples, S, described by attributes A, and the accuracy parameter, δ. In addition, the evaluation function $G(A_i)$ is supplied, which could be information gain, gain ratio, Gini index, or some other attribute selection measure. At each node in the decision tree, we need to maximize $G(A_i)$ for one of the remaining attributes, A_i. Our goal is to find the smallest number of tuples, N, for which the Hoeffding bound is satisfied. For a given node, let A_a be the attribute that achieves the highest G, and A_b be the attribute that achieves the second highest G. If $G(A_a) - G(A_b) > \varepsilon$, where ε is calculated from Equation (8.4), we can confidently say that this difference is larger than zero. We select A_a as the best splitting attribute with confidence $1 - \delta$.

The only statistics that must be maintained in the Hoeffding tree algorithm are the counts n_{ijk} for the value v_j of attribute A_i with class label y_k. Therefore, if d is the number of attributes, v is the maximum number of values for any attribute, c is the number of classes, and l is the maximum depth (or number of levels) of the tree, then the total memory required is $O(ldvc)$. This memory requirement is very modest compared to other decision tree algorithms, which usually store the entire training set in memory.

"*How does the Hoeffding tree compare with trees produced by traditional decision trees algorithms that run in batch mode?*" The Hoeffding tree becomes asymptotically close to that produced by the batch learner. Specifically, the expected disagreement between the Hoeffding tree and a decision tree with infinite examples is at most δ/p, where p is the leaf probability, or the probability that an example will fall into a leaf. If the two best splitting attributes differ by 10% (i.e., $\varepsilon/R = 0.10$), then by Equation (8.4), it would take 380 examples to ensure a desired accuracy of 90% (i.e., $\delta = 0.1$). For $\delta = 0.0001$, it would take only 725 examples, demonstrating an exponential improvement in δ with only a linear increase in the number of examples. For this latter case, if $p = 1\%$, the expected disagreement between the trees would be at most $\delta/p = 0.01\%$, with only 725 examples per node.

In addition to high accuracy with a small sample, Hoeffding trees have other attractive properties for dealing with stream data. First, multiple scans of the same data are never performed. This is important because data streams often become too large to store. Furthermore, the algorithm is incremental, which can be seen in Figure 8.3 (adapted from [GGR02]). The figure demonstrates how new examples are integrated into the tree as they stream in. This property contrasts with batch learners, which wait until the data are accumulated before constructing the model. Another advantage of incrementally building the tree is that we can use it to classify data even while it is being built. The tree will continue to grow and become more accurate as more training data stream in.

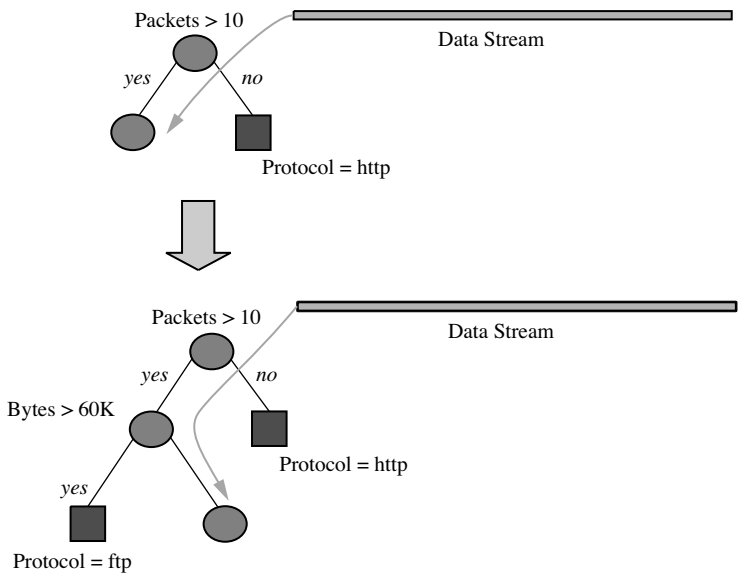


Figure 8.3 The nodes of the Hoeffding tree are created incrementally as more samples stream in.

There are, however, weaknesses to the Hoeffding tree algorithm. For example, the algorithm spends a great deal of time with attributes that have nearly identical splitting quality. In addition, the memory utilization can be further optimized. Finally, the algorithm cannot handle concept drift, because once a node is created, it can never change.

Very Fast Decision Tree (VFDT) and Concept-adapting Very Fast Decision Tree (CVFDT)

The **VFDT** (**Very Fast Decision Tree**) **algorithm** makes several modifications to the Hoeffding tree algorithm to improve both speed and memory utilization. The modifications include breaking near-ties during attribute selection more aggressively, computing the G function after a number of training examples, deactivating the least promising leaves whenever memory is running low, dropping poor splitting attributes, and improving the initialization method. VFDT works well on stream data and also compares extremely well to traditional classifiers in both speed and accuracy. However, it still cannot handle concept drift in data streams.

"*What can we do to manage concept drift?*" Basically, we need a way to identify in a timely manner those elements of the stream that are no longer consistent with the current concepts. A common approach is to use a *sliding window*. The intuition behind it is to incorporate new examples yet eliminate the effects of old ones. We

can repeatedly apply a traditional classifier to the examples in the sliding window. As new examples arrive, they are inserted into the beginning of the window; a corresponding number of examples is removed from the end of the window, and the classifier is reapplied. This technique, however, is sensitive to the window size, w. If w is too large, the model will not accurately represent the concept drift. On the other hand, if w is too small, then there will not be enough examples to construct an accurate model. Moreover, it will become very expensive to continually construct a new classifier model.

To adapt to concept-drifting data streams, the VFDT algorithm was further developed into the **Concept-adapting Very Fast Decision Tree algorithm** (**CVFDT**). CVFDT also uses a sliding window approach; however, it does not construct a new model from scratch each time. Rather, it updates statistics at the nodes by incrementing the counts associated with new examples and decrementing the counts associated with old ones. Therefore, if there is a concept drift, some nodes may no longer pass the Hoeffding bound. When this happens, an alternate subtree will be grown, with the new best splitting attribute at the root. As new examples stream in, the alternate subtree will continue to develop, without yet being used for classification. Once the alternate subtree becomes more accurate than the existing one, the old subtree is replaced.

Empirical studies show that CVFDT achieves better accuracy than VFDT with time-changing data streams. In addition, the size of the tree in CVFDT is much smaller than that in VFDT, because the latter accumulates many outdated examples.

A Classifier Ensemble Approach to Stream Data Classification

Let's look at another approach to classifying concept drifting data streams, where we instead use a *classifier ensemble*. The idea is to train an ensemble or group of classifiers (using, say, C4.5, or naïve Bayes) from sequential chunks of the data stream. That is, whenever a new chunk arrives, we build a new classifier from it. The individual classifiers are weighted based on their expected classification accuracy in a time-changing environment. Only the top-k classifiers are kept. The decisions are then based on the weighted votes of the classifiers.

"*Why is this approach useful?*" There are several reasons for involving more than one classifier. Decision trees are not necessarily the most natural method for handling concept drift. Specifically, if an attribute near the root of the tree in CVFDT no longer passes the Hoeffding bound, a large portion of the tree must be regrown. Many other classifiers, such as naïve Bayes, are not subject to this weakness. In addition, naïve Bayesian classifiers also supply relative probabilities along with the class labels, which expresses the confidence of a decision. Furthermore, CVFDT's automatic elimination of old examples may not be prudent. Rather than keeping only the most up-to-date examples, the ensemble approach discards the least accurate classifiers. Experimentation shows that the ensemble approach achieves greater accuracy than any one of the single classifiers.

8.1.5 Clustering Evolving Data Streams

Imagine a huge amount of dynamic stream data. Many applications require the automated clustering of such data into groups based on their similarities. Examples include applications for network intrusion detection, analyzing Web clickstreams, and stock market analysis. Although there are many powerful methods for clustering static data sets (Chapter 7), clustering data streams places additional constraints on such algorithms. As we have seen, the data stream model of computation requires algorithms to make a single pass over the data, with bounded memory and limited processing time, whereas the stream may be highly dynamic and evolving over time.

For effective clustering of stream data, several new methodologies have been developed, as follows:

- **Compute and store summaries of past data:** Due to limited memory space and fast response requirements, compute summaries of the previously seen data, store the relevant results, and use such summaries to compute important statistics when required.
- **Apply a divide-and-conquer strategy:** Divide data streams into chunks based on order of arrival, compute summaries for these chunks, and then merge the summaries. In this way, larger models can be built out of smaller building blocks.
- **Incremental clustering of incoming data streams:** Because stream data enter the system continuously and incrementally, the clusters derived must be incrementally refined.
- **Perform microclustering as well as macroclustering analysis:** Stream clusters can be computed in two steps: (1) compute and store summaries at the *microcluster* level, where microclusters are formed by applying a hierarchical bottom-up clustering algorithm (Section 7.5.1), and (2) compute *macroclusters* (such as by using another clustering algorithm to group the microclusters) at the user-specified level. This two-step computation effectively compresses the data and often results in a smaller margin of error.
- **Explore multiple time granularity for the analysis of cluster evolution:** Because the more recent data often play a different role from that of the remote (i.e., older) data in stream data analysis, use a tilted time frame model to store snapshots of summarized data at different points in time.
- **Divide stream clustering into on-line and off-line processes:** While data are streaming in, basic summaries of data snapshots should be computed, stored, and incrementally updated. Therefore, an on-line process is needed to maintain such dynamically changing clusters. Meanwhile, a user may pose queries to ask about past, current, or evolving clusters. Such analysis can be performed off-line or as a process independent of on-line cluster maintenance.

Several algorithms have been developed for clustering data streams. Two of them, namely, STREAM and CluStream, are introduced here.

STREAM: A k-Medians-based Stream Clustering Algorithm

Given a data stream model of points, $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$, with timestamps, $T_1, \ldots, T_N$, the objective of stream clustering is to maintain a consistently good clustering of the sequence seen so far using a small amount of memory and time.

STREAM is a single-pass, constant factor approximation algorithm that was developed for the *k-medians problem*. The k-medians problem is to cluster N data points into k clusters or groups such that the sum squared error (SSQ) between the points and the cluster center to which they are assigned is minimized. The idea is to assign similar points to the same cluster, where these points are dissimilar from points in other clusters.

Recall that in the stream data model, data points can only be seen once, and memory and time are limited. To achieve high-quality clustering, the STREAM algorithm processes data streams in buckets (or batches) of m points, with each bucket fitting in main memory. For each bucket, b_i, STREAM clusters the bucket's points into k clusters. It then summarizes the bucket information by retaining only the information regarding the k centers, with each cluster center being weighted by the number of points assigned to its cluster. STREAM then discards the points, retaining only the center information. Once enough centers have been collected, the weighted centers are again clustered to produce another set of $O(k)$ cluster centers. This is repeated so that at every level, at most m points are retained. This approach results in a one-pass, $O(kN)$-time, $O(N^{\varepsilon})$-space (for some constant $\varepsilon < 1$), constant-factor approximation algorithm for data stream k-medians.

STREAM derives quality k-medians clusters with limited space and time. However, it considers neither the evolution of the data nor time granularity. The clustering can become dominated by the older, outdated data of the stream. In real life, the nature of the clusters may vary with both the moment at which they are computed, as well as the time horizon over which they are measured. For example, a user may wish to examine clusters occurring last week, last month, or last year. These may be considerably different. Therefore, a data stream clustering algorithm should also provide the flexibility to compute clusters over user-defined time periods in an interactive manner. The following algorithm, CluStream, addresses these concerns.

CluStream: Clustering Evolving Data Streams

CluStream is an algorithm for the clustering of evolving data streams based on user-specified, on-line clustering queries. It divides the clustering process into on-line and off-line components. The on-line component computes and stores summary statistics about the data stream using *microclusters*, and performs incremental on-line computation and maintenance of the microclusters. The off-line component does macroclustering and answers various user questions using the stored summary statistics, which are based on the tilted time frame model.

To cluster evolving data streams based on both historical and current stream data information, the tilted time frame model (such as a progressive logarithmic model) is adopted, which stores the snapshots of a set of microclusters at different levels of

granularity depending on recency. The intuition here is that more information will be needed for more recent events as opposed to older events. The stored information can be used for processing history-related, user-specific clustering queries.

A **microcluster** in CluStream is represented as a *clustering feature*. CluStream extends the concept of the clustering feature developed in BIRCH (see Section 7.5.2) to include the temporal domain. As a *temporal extension of the clustering feature*, a microcluster for a set of d-dimensional points, $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$, with timestamps, $T_1, \ldots, T_n$, is defined as the $(2d+3)$ tuple $(\boldsymbol{CF}2^x, \boldsymbol{CF}1^x, CF2^t, CF1^t, n)$, wherein $\boldsymbol{CF}2^x$ and $\boldsymbol{CF}1^x$ are d-dimensional vectors while $CF2^t$, $CF1^t$, and n are scalars. $\boldsymbol{CF}2^x$ maintains the sum of the squares of the data values per dimension, that is, $\sum_{i=1}^{n} \boldsymbol{X}_i^2$. Similarly, for each dimension, the sum of the data values is maintained in $\boldsymbol{CF}1^x$. From a statistical point of view, $\boldsymbol{CF}2^x$ and $\boldsymbol{CF}1^x$ represent the second- and first-order moments (Section 8.1.1) of the data, respectively. The sum of squares of the timestamps is maintained in $CF2^t$. The sum of the timestamps is maintained in $CF1^t$. Finally, the number of data points in the microcluster is maintained in n.

Clustering features have additive and subtractive properties that make them very useful for data stream cluster analysis. For example, two microclusters can be merged by adding their respective clustering features. Furthermore, a large number of microclusters can be maintained without using a great deal of memory. Snapshots of these microclusters are stored away at key points in time based on the tilted time frame.

The on-line microcluster processing is divided into two phases: (1) statistical data collection and (2) updating of microclusters. In the first phase, a total of q microclusters, $M_1, \ldots, M_q$, are maintained, where q is usually significantly larger than the number of natural clusters and is determined by the amount of available memory. In the second phase, microclusters are updated. Each new data point is added to either an existing cluster or a new one. To decide whether a new cluster is required, a maximum boundary for each cluster is defined. If the new data point falls within the boundary, it is added to the cluster; otherwise, it is the first data point in a new cluster. When a data point is added to an existing cluster, it is "absorbed" because of the additive property of the microclusters. When a data point is added to a new cluster, the least recently used existing cluster has to be removed or two existing clusters have to be merged, depending on certain criteria, in order to create memory space for the new cluster.

The off-line component can perform user-directed macroclustering or cluster evolution analysis. Macroclustering allows a user to explore the stream clusters over different time horizons. **A time horizon**, h, is a history of length h of the stream. Given a user-specified time horizon, h, and the number of desired macroclusters, k, macroclustering finds k high-level clusters over h. This is done as follows: First, the snapshot at time $t_c - h$ is subtracted from the snapshot at the current time, t_c. Clusters older than the beginning of the horizon are not included. The microclusters in the horizon are considered as weighted pseudo-points and are reclustered in order to determine higher-level clusters. Notice that the clustering process is similar to the method used in STREAM but requires only two snapshots (the beginning and end of the horizon) and is more flexible over a range of user queries.

"*What if a user wants to see how clusters have changed over, say, the last quarter or the last year?*" Cluster evolution analysis looks at how clusters change over time. Given a

user-specified time horizon, h, and two clock times, t_1 and t_2 (where $t_1 < t_2$), **cluster evolution analysis** examines the evolving nature of the data arriving between $(t_2 - h, t_2)$ and that arriving between $(t_1 - h, t_1)$. This involves answering questions like whether new clusters in the data at time t_1 were not present at time t_2, or whether some of the original clusters were lost. This also involves analyzing whether some of the original clusters at time t_1 shifted in position and nature. With the available microcluster information, this can be done by computing the net snapshots of the microclusters, $N(t_1, h)$ and $N(t_2, h)$, and then computing the snapshot changes over time. Such evolution analysis of the data over time can be used for network intrusion detection to identify new types of attacks within the network.

CluStream was shown to derive high-quality clusters, especially when the changes are dramatic. Moreover, it offers rich functionality to the user because it registers the essential historical information with respect to cluster evolution. The tilted time frame along with the microclustering structure allow for better accuracy and efficiency on real data. Finally, it maintains scalability in terms of stream size, dimensionality, and the number of clusters.

In general, stream data mining is still a fast-evolving research field. With the massive amount of data streams populating many applications, it is expected that many new stream data mining methods will be developed, especially for data streams containing additional semantic information, such as time-series streams, spatiotemporal data streams, and video and audio data streams.

8.2 Mining Time-Series Data

"*What is a time-series database?*" A **time-series database** consists of sequences of values or events obtained over repeated measurements of time. The values are typically measured at equal time intervals (e.g., hourly, daily, weekly). Time-series databases are popular in many applications, such as stock market analysis, economic and sales forecasting, budgetary analysis, utility studies, inventory studies, yield projections, workload projections, process and quality control, observation of natural phenomena (such as atmosphere, temperature, wind, earthquake), scientific and engineering experiments, and medical treatments. A time-series database is also a sequence database. However, a **sequence database** is any database that consists of sequences of ordered events, with or without concrete notions of time. For example, Web page traversal sequences and customer shopping transaction sequences are sequence data, but they may not be time-series data. The mining of sequence data is discussed in Section 8.3.

With the growing deployment of a large number of sensors, telemetry devices, and other on-line data collection tools, the amount of time-series data is increasing rapidly, often in the order of gigabytes per day (such as in stock trading) or even per minute (such as from NASA space programs). How can we find correlation relationships within time-series data? How can we analyze such huge numbers of time series to find similar or regular patterns, trends, bursts (such as sudden sharp changes), and outliers, with

fast or even on-line real-time response? This has become an increasingly important and challenging problem. In this section, we examine several aspects of mining time-series databases, with a focus on trend analysis and similarity search.

8.2.1 Trend Analysis

A time series involving a variable Y, representing, say, the daily closing price of a share in a stock market, can be viewed as a function of time t, that is, $Y = F(t)$. Such a function can be illustrated as a time-series graph, as shown in Figure 8.4, which describes a point moving with the passage of time.

"*How can we study time-series data?*" In general, there are two goals in time-series analysis: (1) *modeling time series* (i.e., to gain insight into the mechanisms or underlying forces that generate the time series), and (2) *forecasting time series* (i.e., to predict the future values of the time-series variables).

Trend analysis consists of the following four major **components** or **movements** for characterizing time-series data:

- **Trend or long-term movements**: These indicate the general direction in which a time-series graph is moving over a long interval of time. This movement is displayed by a **trend curve**, or a **trend line**. For example, the trend curve of Figure 8.4 is indicated by a dashed curve. Typical methods for determining a trend curve or trend line include the *weighted moving average method* and the *least squares method*, discussed later.
- **Cyclic movements or cyclic variations**: These refer to the *cycles*, that is, the long-term oscillations about a trend line or curve, which may or may not be periodic. That is, the cycles need not necessarily follow exactly similar patterns after equal intervals of time.

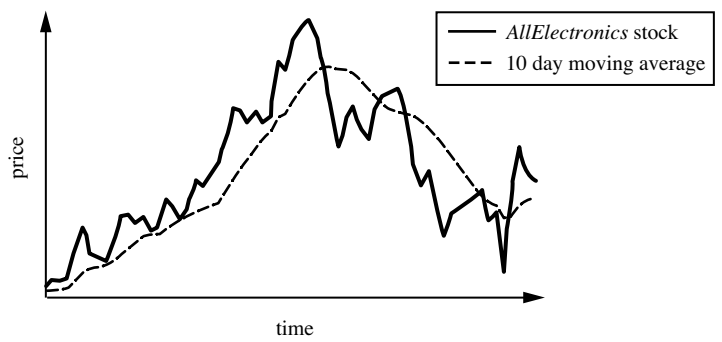


Figure 8.4 Time-series data of the stock price of *AllElectronics* over time. The *trend* is shown with a dashed curve, calculated by a moving average.

- **Seasonal movements or seasonal variations**: These are systematic or calendar related. Examples include events that recur annually, such as the sudden increase in sales of chocolates and flowers before Valentine's Day or of department store items before Christmas. The observed increase in water consumption in summer due to warm weather is another example. In these examples, seasonal movements are the identical or nearly identical patterns that a time series appears to follow during corresponding months of successive years.
- **Irregular or random movements**: These characterize the sporadic motion of time series due to random or chance events, such as labor disputes, floods, or announced personnel changes within companies.

Note that *regression analysis* has been a popular tool for modeling time series, finding trends and outliers in such data sets. Regression is a fundamental topic in statistics and is described in many textbooks. Thus, we will not spend much time on this theme.[3] However, pure regression analysis cannot capture all of the four movements described above that occur in real-world applications. Hence, our discussion of trend analysis and modeling time series focuses on the above movements.

The trend, cyclic, seasonal, and irregular movements are represented by the variables T, C, S, I, respectively. Time-series modeling is also referred to as the **decomposition** of a time series into these four basic movements. The time-series variable Y can be modeled as either the product of the four variables (i.e., $Y = T \times C \times S \times I$) or their sum. This choice is typically empirical.

"*Given a sequence of values for* Y $(i.e., y_1, y_2, y_3, \ldots)$ *for analysis, how can we adjust the data for seasonal fluctuations?*" This can be performed by estimating and then removing from the time series the influences of the data that are systematic or calendar related. In many business transactions, for example, there are expected regular seasonal fluctuations, such as higher sales volumes during the Christmas season. Such fluctuations can conceal both the true underlying movement of the series as well as certain nonseasonal characteristics that may be of interest. Therefore, it is important to identify such seasonal variations and "deseasonalize" the data. For this purpose, the concept of **seasonal index** is introduced, as a set of numbers showing the relative values of a variable during the months of a year. For example, if the sales during October, November, and December are 80%, 120%, and 140% of the average monthly sales for the whole year, respectively, then 80, 120, and 140 are the **seasonal index numbers** for the year. If the original monthly data are divided by the corresponding seasonal index numbers, the resulting data are said to be **deseasonalized**, or *adjusted for seasonal variations*. Such data still include trend, cyclic, and irregular movements.

To detect seasonal patterns, we can also look for correlations between each ith element of the series and $(i-k)$th element (where k is referred to as the **lag**) using **autocorrelation analysis**. For example, we can measure the correlation in sales for every twelfth month,

[3] A simple introduction to regression is included in Chapter 6: Classification and Prediction.

where here, $k = 12$. The correlation coefficient given in Chapter 2 (Equation (2.8)) can be used. Let $\langle y_1, y_2, \ldots, y_N \rangle$ be the time series. To apply Equation (2.8), the two attributes in the equation respectively refer to the two random variables representing the time series viewed with lag k. These times series are $\langle y_1, y_2, \ldots, y_{N-k} \rangle$ and $\langle y_{k+1}, y_{k+2}, \ldots, y_N \rangle$. A zero value indicates that there is no correlation relationship. A positive value indicates a positive correlation, that is, both variables increase together. A negative value indicates a negative correlation, that is, one variable increases as the other decreases. The higher the positive (or negative) value is, the greater is the positive (or negative) correlation relationship.

"*How can we determine the trend of the data?*" A common method for determining trend is to calculate a **moving average of order** n as the following sequence of arithmetic means:

$$\frac{y_1 + y_2 + \cdots + y_n}{n}, \frac{y_2 + y_3 + \cdots + y_{n+1}}{n}, \frac{y_3 + y_4 + \cdots + y_{n+2}}{n}, \cdots \tag{8.5}$$

A moving average tends to reduce the amount of variation present in the data set. Thus the process of replacing the time series by its moving average eliminates unwanted fluctuations and is therefore also referred to as the **smoothing of time series**. If weighted arithmetic means are used in Sequence (8.5), the resulting sequence is called a **weighted moving average of order** n.

Example 8.5 **Moving averages.** Given a sequence of nine values, we can compute its moving average of order 3, and its weighted moving average of order 3 using the weights $(1, 4, 1)$. This information can be displayed in tabular form, where each value in the moving average is the mean of the three values immediately above it, and each value in the weighted moving average is the weighted average of the three values immediately above it.

Original data:	3	7	2	0	4	5	9	7	2
Moving average of order 3:		4	3	2	3	6	7	6	
Weighted $(1, 4, 1)$ moving average of order 3:		5.5	2.5	1	3.5	5.5	8	6.5	

Using the first equation in Sequence 8.5, we calculate the first moving average as $\frac{3+7+2}{3} = 4$. The first weighted average value is calculated as $\frac{1\times3 + 4\times7 + 1\times2}{1+4+1} = 5.5$. The weighted average typically assigns greater weights to the central elements in order to offset the smoothing effect. ■

A moving average loses the data at the beginning and end of a series; may sometimes generate cycles or other movements that are not present in the original data; and may be strongly affected by the presence of extreme values. Notice that the influence of extreme values can be reduced by employing a weighted moving average with appropriate weights as shown in Example 8.5. An appropriate moving average can help smooth out irregular variations in the data. In general, small deviations tend to occur with large frequency, whereas large deviations tend to occur with small frequency, following a normal distribution.

"*Are there other ways to estimate the trend?*" Yes, one such method is the **freehand method**, where an approximate curve or line is drawn to fit a set of data based on the

user's own judgment. This method is costly and barely reliable for any large-scale data mining. An alternative is the **least squares method**,[4] where we consider the best-fitting curve C as the *least squares curve*, that is, the curve having the minimum of $\sum_{i=1}^{n} d_i^2$, where the *deviation* or *error*, d_i, is the difference between the value y_i of a point (x_i, y_i) and the corresponding value as determined from the curve C.

The data can then be adjusted for trend by dividing the data by their corresponding trend values. As mentioned earlier, an appropriate moving average will smooth out the irregular variations. This leaves us with only cyclic variations for further analysis. If periodicity or approximate periodicity of cycles occurs, **cyclic indexes** can be constructed in a manner similar to that for seasonal indexes.

In practice, it is useful to first graph the time series and qualitatively estimate the presence of long-term trends, seasonal variations, and cyclic variations. This may help in selecting a suitable method for analysis and in comprehending its results.

Time-series forecasting finds a mathematical formula that will approximately generate the historical patterns in a time series. It is used to make long-term or short-term predictions of future values. There are several models for forecasting: **ARIMA** (**Auto-Regressive Integrated Moving Average**), also known as the Box-Jenkins methodology (after its creators), is a popular example. It is powerful yet rather complex to use. The quality of the results obtained may depend on the user's level of experience. Interested readers may consult the bibliographic notes for references to the technique.

8.2.2 Similarity Search in Time-Series Analysis

"*What is a similarity search?*" Unlike normal database queries, which find data that match the given query *exactly*, a **similarity search** finds data sequences that *differ only slightly* from the given query sequence. Given a set of time-series sequences, S, there are two types of similarity searches: *subsequence matching* and *whole sequence matching*. **Subsequence matching** finds the sequences in S that contain subsequences that are similar to a given query sequence x, while **whole sequence matching** finds a set of sequences in S that are similar to each other (as a whole). Subsequence matching is a more frequently encountered problem in applications. Similarity search in time-series analysis is useful for financial market analysis (e.g., stock data analysis), medical diagnosis (e.g., cardiogram analysis), and in scientific or engineering databases (e.g., power consumption analysis).

Data Reduction and Transformation Techniques

Due to the tremendous size and high-dimensionality of time-series data, *data reduction* often serves as the first step in time-series analysis. Data reduction leads to not only much smaller storage space but also much faster processing. As discussed in Chapter 2, major

[4] The least squares method was introduced in Section 6.11.1 under the topic of linear regression.

strategies for data reduction include *attribute subset selection* (which removes irrelevant or redundant attributes or dimensions), *dimensionality reduction* (which typically employs signal processing techniques to obtain a reduced version of the original data), and *numerosity reduction* (where data are replaced or estimated by alternative, smaller representations, such as histograms, clustering, and sampling). Because time series can be viewed as data of very high dimensionality where each point of time can be viewed as a dimension, dimensionality reduction is our major concern here. For example, to compute correlations between two time-series curves, the reduction of the time series from length (i.e., dimension) n to k may lead to a reduction from $O(n)$ to $O(k)$ in computational complexity. If $k \ll n$, the complexity of the computation will be greatly reduced.

Several dimensionality reduction techniques can be used in time-series analysis. Examples include (1) the *discrete Fourier transform* (*DFT*) as the classical data reduction technique, (2) more recently developed *discrete wavelet transforms (DWT)*, (3) Singular Value Decomposition (SVD) based on Principle Components Analysis (PCA),[5] and (4) random projection-based sketch techniques (as discussed in Section 8.1.1), which can also give a good-quality synopsis of data. Because we have touched on these topics earlier in this book, and because a thorough explanation is beyond our scope, we will not go into great detail here. The first three techniques listed are signal processing techniques. A given time series can be considered as a finite sequence of real values (or *coefficients*), recorded over time in some *object space*. The data or *signal* is transformed (using a specific transformation function) into a signal in a *transformed space*. A small subset of the "strongest" transformed coefficients are saved as *features*. These features form a *feature space*, which is simply a projection of the transformed space. This representation is sparse so that operations that can take advantage of data sparsity are computationally very fast if performed in feature space. The features can be transformed back into object space, resulting in a compressed approximation of the original data.

Many techniques for signal analysis require the data to be in the *frequency domain*. Therefore, **distance-preserving orthonormal transformations** are often used to transform the data from the time domain to the frequency domain. Usually, a data-independent transformation is applied, where the transformation matrix is determined a priori, independent of the input data. Because the distance between two signals in the time domain is the same as their Euclidean distance in the frequency domain, the DFT does a good job of preserving essentials in the first few coefficients. By keeping only the first few (i.e., "strongest") coefficients of the DFT, we can compute the lower bounds of the actual distance.

Indexing Methods for Similarity Search

"Once the data are transformed by, say, a DFT, how can we provide support for efficient search in time-series data?" For efficient accessing, a *multidimensional index* can be

[5]The Discrete Fourier transform, wavelet transforms, and principal components analysis are briefly introduced in Section 2.5.3.

constructed using the first few Fourier coefficients. When a similarity query is submitted to the system, the index can be used to retrieve the sequences that are at most a certain small distance away from the query sequence. Postprocessing is then performed by computing the actual distance between sequences in the time domain and discarding any false matches.

For subsequence matching, each sequence can be broken down into a set of "pieces" of *windows* with length w. In one approach, the features of the subsequence inside each window are then extracted. Each sequence is mapped to a "trail" in the feature space. The trail of each sequence is divided into "subtrails," each represented by a minimum bounding rectangle. A multipiece assembly algorithm can then be used to search for longer sequence matches.

Various kinds of indexing methods have been explored to speed up the similarity search. For example, *R-trees* and *R***-trees* have been used to store the minimal bounding rectangles mentioned above. In addition, the ε-*kdB tree* has been developed for faster spatial similarity joins on high-dimensional points, and *suffix trees* have also been explored. References are given in the bibliographic notes.

Similarity Search Methods

The above trail-based approach to similarity search was pioneering, yet has a number of limitations. In particular, it uses the Euclidean distance as a similarity measure, which is sensitive to outliers. Furthermore, what if there are differences in the baseline and scale of the two time series being compared? What if there are gaps? Here, we discuss an approach that addresses these issues.

For similarity analysis of time-series data, Euclidean distance is typically used as a similarity measure. Here, the smaller the distance between two sets of time-series data, the more similar are the two series. However, we cannot directly apply the Euclidean distance. Instead, we need to consider differences in the *baseline* and *scale* (or amplitude) of our two series. For example, one stock's value may have a baseline of around \$20 and fluctuate with a relatively large amplitude (such as between \$15 and \$25), while another could have a baseline of around \$100 and fluctuate with a relatively small amplitude (such as between \$90 and \$110). The distance from one baseline to another is referred to as the **offset**.

A straightforward approach to solving the baseline and scale problem is to apply a **normalization transformation**. For example, a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ can be replaced by a normalized sequence $X' = \langle x'_1, x'_2, \ldots, x'_n \rangle$, using the following formula,

$$x'_i = \frac{x_i - \mu}{\sigma} \tag{8.6}$$

where μ is the mean value of the sequence X and σ is the standard deviation of X. We can transform other sequences using the same formula, and then compare them for similarity.

Most real-world applications do not require the matching subsequences to be perfectly aligned along the time axis. In other words, we should allow for pairs of

subsequences to match if they are of the *same shape*, but differ due to the presence of **gaps** within a sequence (where one of the series may be missing some of the values that exist in the other) or differences in offsets or amplitudes. This is particularly useful in many similar sequence analyses, such as stock market analysis and cardiogram analysis.

"*How can subsequence matching be performed to allow for such differences?*" Users or experts can specify parameters such as a sliding window size, the width of an envelope for similarity, the maximum gap, a matching fraction, and so on. Figure 8.5 illustrates the process involved, starting with two sequences in their original form. First, gaps are removed. The resulting sequences are normalized with respect to offset translation (where one time series is adjusted to align with the other by shifting the baseline or phase) and amplitude scaling. For this normalization, techniques such

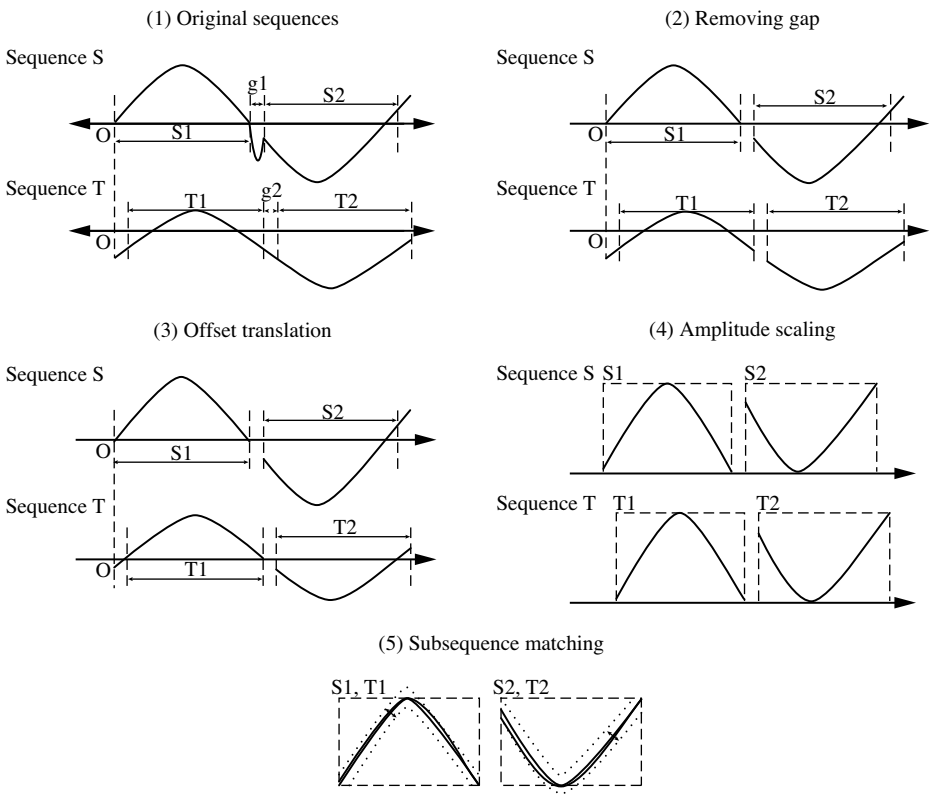


Figure 8.5 Subsequence matching in time-series data: The original sequences are of the same shape, yet adjustments need to be made to deal with differences in gaps, offsets, and amplitudes. These adjustments allow subsequences to be matched within an envelope of width ε. Based on [ALSS95].

as those described in Section 2.4.2 may be used. Two subsequences are considered *similar* and can be matched if one lies within an envelope of ε width around the other (where ε is a small number, specified by a user or expert), ignoring outliers. Two sequences are *similar* if they have enough nonoverlapping time-ordered pairs of similar subsequences.

Based on the above, a similarity search that handles gaps and differences in offsets and amplitudes can be performed by the following steps:

1. **Atomic matching**: Normalize the data. Find all pairs of gap-free windows of a small length that are similar.
2. **Window stitching**: Stitch similar windows to form pairs of large similar subsequences, allowing gaps between atomic matches.
3. **Subsequence ordering**: Linearly order the subsequence matches to determine whether enough similar pieces exist.

With such processing, sequences of similar shape but with gaps or differences in offsets or amplitudes can be found to match each other or to match query templates.

Query Languages for Time Sequences

"*How can I specify the similarity search to be performed?*" We need to design and develop powerful query languages to facilitate the specification of similarity searches in time sequences. A **time-sequence query language** should be able to specify not only simple similarity queries like "*Find all of the sequences similar to a given subsequence Q*," but also sophisticated queries like "*Find all of the sequences that are similar to some sequence in class C_1, but not similar to any sequence in class C_2*." Moreover, it should be able to support various kinds of queries, such as range queries and nearest-neighbor queries.

An interesting kind of time-sequence query language is a **shape definition language**. It allows users to define and query the *overall shape* of time sequences using human-readable series of sequence transitions or macros, while ignoring the specific details.

Example 8.6 **Using a shape definition language.** The pattern *up, Up, UP* can be used to describe increasing degrees of rising slopes. A macro, such as *spike*, can denote a sequence like (*SteepUps*, *flat*, *SteepDowns*), where *SteepUps* is defined as ({*Up*, *UP*}, {*Up*, *UP*}, {*Up*, *UP*}), which means that one *SteepUps* consists of three steep up-slopes, each corresponding to either *Up* or *UP*. *SteepDowns* is similarly defined. ■

Such a shape definition language increases the users' flexibility at specifying queries of desired shapes for sequence similarity search.

8.3 Mining Sequence Patterns in Transactional Databases

A **sequence database** consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. There are many applications involving sequence data. Typical examples include customer shopping sequences, Web clickstreams, biological sequences, sequences of events in science and engineering, and in natural and social developments. In this section, we study *sequential pattern mining* in transactional databases. In particular, we start with the basic concepts of sequential pattern mining in Section 8.3.1. Section 8.3.2 presents several scalable methods for such mining. Constraint-based sequential pattern mining is described in Section 8.3.3. Periodicity analysis for sequence data is discussed in Section 8.3.4. Specific methods for mining sequence patterns in biological data are addressed in Section 8.4.

8.3.1 Sequential Pattern Mining: Concepts and Primitives

"*What is sequential pattern mining?*" **Sequential pattern mining** is the mining of frequently occurring ordered events or subsequences as patterns. An example of a sequential pattern is "*Customers who buy a Canon digital camera are likely to buy an HP color printer within a month.*" For retail data, sequential patterns are useful for shelf placement and promotions. This industry, as well as telecommunications and other businesses, may also use sequential patterns for targeted marketing, customer retention, and many other tasks. Other areas in which sequential patterns can be applied include Web access pattern analysis, weather prediction, production processes, and network intrusion detection. Notice that most studies of sequential pattern mining concentrate on *categorical* (or *symbolic*) *patterns*, whereas numerical curve analysis usually belongs to the scope of trend analysis and forecasting in statistical time-series analysis, as discussed in Section 8.2.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in 1995 [AS95] based on their study of customer purchase sequences, as follows: "*Given a set of sequences, where each sequence consists of a list of events (or elements) and each event consists of a set of items, and given a user-specified minimum support threshold of min_sup, sequential pattern mining finds all* ***frequent*** *subsequences, that is, the subsequences whose occurrence frequency in the set of sequences is no less than min_sup.*"

Let's establish some vocabulary for our discussion of sequential pattern mining. Let $I = \{I_1, I_2, \ldots, I_p\}$ be the set of all *items*. An **itemset** is a nonempty set of items. A **sequence** is an ordered list of **events**. A sequence s is denoted $\langle e_1 e_2 e_3 \cdots e_l \rangle$, where event e_1 occurs before e_2, which occurs before e_3, and so on. Event e_j is also called an **element** of s. In the case of customer purchase data, an event refers to a shopping trip in which a customer bought items at a certain store. The event is thus an itemset, that is, an unordered list of items that the customer purchased during the trip. The itemset (or event) is denoted $(x_1 x_2 \cdots x_q)$, where x_k is an item. For brevity, the brackets are omitted if an element has only one item, that is, element (x) is written as x. Suppose that a customer made several shopping trips to the store. These ordered events form a sequence for the customer. That is, the customer first bought the items in s_1, then later bought

the items in s_2, and so on. An item can occur at most once in an event of a sequence, but can occur multiple times in different events of a sequence. The number of instances of items in a sequence is called the **length** of the sequence. A sequence with length l is called an l-**sequence**. A sequence $\alpha = \langle a_1 a_2 \cdots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \cdots b_m \rangle$, and β is a **supersequence** of α, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_n \subseteq b_{j_n}$. For example, if $\alpha = \langle (ab), d \rangle$ and $\beta = \langle (abc), (de) \rangle$, where a, b, c, d, and e are items, then α is a subsequence of β and β is a supersequence of α.

A **sequence database**, S, is a set of tuples, $\langle SID, s \rangle$, where SID is a *sequence_ID* and s is a sequence. For our example, S contains sequences for all customers of the store. A tuple $\langle SID, s \rangle$ is said to **contain** a sequence α, if α is a subsequence of s. The **support** of a sequence α in a sequence database S is the number of tuples in the database containing α, that is, $support_S(\alpha) = |\{\langle SID, s \rangle | (\langle SID, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\}|$. It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer *min_sup* as the **minimum support threshold**, a sequence α is **frequent** in sequence database S if $support_S(\alpha) \geq min\_sup$. That is, for sequence α to be frequent, it must occur at least *min_sup* times in S. *A frequent sequence is called a* **sequential pattern**. A sequential pattern with length l is called an l-**pattern**. The following example illustrates these concepts.

Example 8.7 **Sequential patterns.** Consider the sequence database, S, given in Table 8.1, which will be used in examples throughout this section. Let $min\_sup = 2$. The set of *items* in the database is $\{a, b, c, d, e, f, g\}$. The database contains four sequences.

Let's look at *sequence* 1, which is $\langle a(abc)(ac)d(cf) \rangle$. It has five *events*, namely (a), (abc), (ac), (d), and (cf), which occur in the order listed. Items a and c each appear more than once in different events of the sequence. There are nine instances of items in sequence 1; therefore, it has a *length* of nine and is called a 9-*sequence*. Item a occurs three times in sequence 1 and so contributes three to the length of the sequence. However, the entire sequence contributes only one to the *support* of $\langle a \rangle$. Sequence $\langle a(bc)df \rangle$ is a *subsequence* of sequence 1 since the events of the former are each subsets of events in sequence 1, and the order of events is preserved. Consider subsequence $s = \langle (ab)c \rangle$. Looking at the sequence database, S, we see that sequences 1 and 3 are the only ones that *contain* the subsequence s. The support of s is thus 2, which satisfies minimum support.

Table 8.1 A sequence database

Sequence_ID	*Sequence*
1	$\langle a(abc)(ac)d(cf) \rangle$
2	$\langle (ad)c(bc)(ae) \rangle$
3	$\langle (ef)(ab)(df)cb \rangle$
4	$\langle eg(af)cbc \rangle$

Therefore, s is frequent, and so we call it a *sequential pattern*. It is a 3-*pattern* since it is a sequential pattern of length three. ■

This model of sequential pattern mining is an abstraction of customer-shopping sequence analysis. Scalable methods for sequential pattern mining on such data are described in Section 8.3.2, which follows. Many other sequential pattern mining applications may not be covered by this model. For example, when analyzing Web clickstream sequences, gaps between clicks become important if one wants to predict what the next click might be. In DNA sequence analysis, *approximate* patterns become useful since DNA sequences may contain (symbol) insertions, deletions, and mutations. Such diverse requirements can be viewed as *constraint relaxation* or *enforcement*. In Section 8.3.3, we discuss how to extend the basic sequential mining model to *constrained* sequential pattern mining in order to handle these cases.

8.3.2 Scalable Methods for Mining Sequential Patterns

Sequential pattern mining is computationally challenging because such mining may generate and/or test a combinatorially explosive number of intermediate subsequences.

"*How can we develop efficient and scalable methods for sequential pattern mining?*" Recent developments have made progress in two directions: (1) efficient methods for mining the *full set* of sequential patterns, and (2) efficient methods for mining only the *set of closed* sequential patterns, where a sequential pattern s is **closed** if there exists no sequential pattern s' where s' is a proper supersequence of s, and s' has the same (frequency) support as s.[6] Because all of the subsequences of a frequent sequence are also frequent, mining the set of closed sequential patterns may avoid the generation of unnecessary subsequences and thus lead to more compact results as well as more efficient methods than mining the full set. We will first examine methods for mining the full set and then study how they can be extended for mining the closed set. In addition, we discuss modifications for mining multilevel, multidimensional sequential patterns (i.e., with multiple levels of granularity).

The major approaches for mining the full set of sequential patterns are similar to those introduced for frequent itemset mining in Chapter 5. Here, we discuss three such approaches for sequential pattern mining, represented by the algorithms GSP, SPADE, and PrefixSpan, respectively. GSP adopts a *candidate generate-and-test* approach using *horizonal data format* (where the data are represented as $\langle sequence\_ID : sequence\_of\_itemsets \rangle$, as usual, where each itemset is an event). SPADE adopts a candidate generate-and-test approach using *vertical data format* (where the data are represented as $\langle itemset : (sequence\_ID, event\_ID) \rangle$). The vertical data format can be obtained by transforming from a horizontally formatted sequence database in just one scan. PrefixSpan is a *pattern growth* method, which does not require candidate generation.

[6] Closed frequent itemsets were introduced in Chapter 5. Here, the definition is applied to sequential patterns.

All three approaches either directly or indirectly explore the **Apriori property**, stated as follows: *every nonempty subsequence of a sequential pattern is a sequential pattern*. (Recall that for a pattern to be called sequential, it must be frequent. That is, it must satisfy minimum support.) The Apriori property is antimonotonic (or downward-closed) in that, if a sequence cannot pass a test (e.g., regarding minimum support), all of its supersequences will also fail the test. Use of this property to prune the search space can help make the discovery of sequential patterns more efficient.

GSP: A Sequential Pattern Mining Algorithm Based on Candidate Generate-and-Test

GSP (Generalized Sequential Patterns) is a sequential pattern mining method that was developed by Srikant and Agrawal in 1996. It is an extension of their seminal algorithm for frequent itemset mining, known as Apriori (Section 5.2). GSP uses the downward-closure property of sequential patterns and adopts a multiple-pass, candidate generate-and-test approach. The algorithm is outlined as follows. In the first scan of the database, it finds all of the frequent items, that is, those with minimum support. Each such item yields a 1-event frequent sequence consisting of that item. Each subsequent pass starts with a *seed set* of sequential patterns—the set of sequential patterns found in the previous pass. This seed set is used to generate new potentially frequent patterns, called *candidate sequences*. Each candidate sequence contains one more item than the seed sequential pattern from which it was generated (where each event in the pattern may contain one or multiple items). Recall that the number of instances of items in a sequence is the *length* of the sequence. So, all of the candidate sequences in a given pass will have the same length. We refer to a sequence with length k as a k-sequence. Let C_k denote the set of candidate k-sequences. A pass over the database finds the support for each candidate k-sequence. The candidates in C_k with at least *min_sup* form L_k, the set of all *frequent* k-sequences. This set then becomes the seed set for the next pass, $k+1$. The algorithm terminates when no new sequential pattern is found in a pass, or no candidate sequence can be generated.

The method is illustrated in the following example.

Example 8.8 **GSP: Candidate generate-and-test (using horizontal data format).** Suppose we are given the same sequence database, S, of Table 8.1 from Example 8.7, with *min_sup* = 2. Note that the data are represented in horizontal data format. In the first scan ($k = 1$), GSP collects the support for each item. The set of candidate 1-sequences is thus (shown here in the form of "*sequence*:*support*"): $\langle a \rangle : 4$, $\langle b \rangle : 4$, $\langle c \rangle : 3$, $\langle d \rangle : 3$, $\langle e \rangle : 3$, $\langle f \rangle : 3$, $\langle g \rangle : 1$.

The sequence $\langle g \rangle$ has a support of only 1 and is the only sequence that does not satisfy minimum support. By filtering it out, we obtain the first seed set, $L_1 = \{\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle e \rangle, \langle f \rangle\}$. Each member in the set represents a 1-event sequential pattern. Each subsequent pass starts with the seed set found in the previous pass and uses it to generate new candidate sequences, which are potentially frequent.

Using L_1 as the seed set, this set of six length-1 sequential patterns generates a set of $6 \times 6 + \frac{6 \times 5}{2} = 51$ candidate sequences of length 2, $C_2 = \{\langle aa \rangle, \langle ab \rangle, \ldots, \langle af \rangle, \langle ba \rangle, \langle bb \rangle, \ldots, \langle ff \rangle, \langle (ab) \rangle, \langle (ac) \rangle, \ldots, \langle (ef) \rangle\}$.

In general, the set of candidates is generated by a self-join of the sequential patterns found in the previous pass (see Section 5.2.1 for details). GSP applies the Apriori property to prune the set of candidates as follows. In the k-th pass, a sequence is a candidate only if each of its length-$(k-1)$ subsequences is a sequential pattern found at the $(k-1)$-th pass. A new scan of the database collects the support for each candidate sequence and finds a new set of sequential patterns, L_k. This set becomes the seed for the next pass. The algorithm terminates when no sequential pattern is found in a pass or when no candidate sequence is generated. Clearly, the number of scans is at least the maximum length of sequential patterns. GSP needs one more scan if the sequential patterns obtained in the last scan still generate new candidates (none of which are found to be frequent).

Although GSP benefits from the Apriori pruning, it still generates a large number of candidates. In this example, six length-1 sequential patterns generate 51 length-2 candidates; 22 length-2 sequential patterns generate 64 length-3 candidates; and so on. Some candidates generated by GSP may not appear in the database at all. In this example, 13 out of 64 length-3 candidates do not appear in the database, resulting in wasted time. ■

The example shows that although an Apriori-like sequential pattern mining method, such as GSP, reduces search space, it typically needs to scan the database multiple times. It will likely generate a huge set of candidate sequences, especially when mining long sequences. There is a need for more efficient mining methods.

SPADE: An Apriori-Based Vertical Data Format Sequential Pattern Mining Algorithm

The Apriori-like sequential pattern mining approach (based on candidate generate-and-test) can also be explored by mapping a sequence database into vertical data format. In **vertical data format**, the database becomes a set of tuples of the form $\langle itemset : (sequence\_ID, event\_ID) \rangle$. That is, for a given itemset, we record the sequence identifier and corresponding event identifier for which the itemset occurs. The **event identifier** serves as a timestamp within a sequence. The *event_ID* of the ith itemset (or event) in a sequence is i. Note than an itemset can occur in more than one sequence. The set of (*sequence_ID*, *event_ID*) pairs for a given itemset forms the **ID_list** of the itemset. The mapping from horizontal to vertical format requires one scan of the database. A major advantage of using this format is that we can determine the support of any k-sequence by simply joining the ID_lists of any two of its $(k-1)$-length subsequences. The length of the resulting ID_list (i.e., unique *sequence_ID* values) is equal to the support of the k-sequence, which tells us whether the sequence is frequent.

SPADE (Sequential PAttern Discovery using Equivalent classes) is an Apriori-based sequential pattern mining algorithm that uses vertical data format. As with GSP, SPADE requires one scan to find the frequent 1-sequences. To find candidate 2-sequences, we join all pairs of single items if they are frequent (therein, it applies the Apriori

property), if they share the same sequence identifier, and if their event identifiers follow a sequential ordering. That is, the first item in the pair must occur as an event before the second item, where both occur in the same sequence. Similarly, we can grow the length of itemsets from length 2 to length 3, and so on. The procedure stops when no frequent sequences can be found or no such sequences can be formed by such joins. The following example helps illustrate the process.

Example 8.9 **SPADE: Candidate generate-and-test using vertical data format.** Let $min\_sup = 2$. Our running example sequence database, S, of Table 8.1 is in horizonal data format. SPADE first scans S and transforms it into vertical format, as shown in Figure 8.6(a). Each itemset (or event) is associated with its ID_list, which is the set of *SID* (*sequence_ID*) and *EID* (*event_ID*) pairs that contain the itemset. The ID_list for individual items, a, b, and so on, is shown in Figure 8.6(b). For example, the ID_list for item b consists of the following (*SID*, *EID*) pairs: $\{(1, 2), (2, 3), (3, 2), (3, 5), (4, 5)\}$, where the entry $(1, 2)$ means that b occurs in sequence 1, event 2, and so on. Items a and b are frequent. They can be joined to form the length-2 sequence, $\langle a, b\rangle$. We find the support of this sequence as follows. We join the ID_lists of a and b by joining on the same *sequence_ID* wherever, according to the *event_ID*s, a occurs before b. That is, the join must preserve the temporal order of the events involved. The result of such a join for a and b is shown in the ID_list for ab of Figure 8.6(c). For example, the ID_list for 2-sequence ab is a set of triples, (*SID*, *EID*(a), *EID*(b)), namely $\{(1, 1, 2), (2, 1, 3), (3, 2, 5), (4, 3, 5)\}$. The entry $(2, 1, 3)$, for example, shows that both a and b occur in sequence 2, and that a (event 1 of the sequence) occurs before b (event 3), as required. Furthermore, the frequent 2-sequences can be joined (while considering the Apriori pruning heuristic that the (k-1)-subsequences of a candidate k-sequence must be frequent) to form 3-sequences, as in Figure 8.6(d), and so on. The process terminates when no frequent sequences can be found or no candidate sequences can be formed. Additional details of the method can be found in Zaki [Zak01]. ■

The use of vertical data format, with the creation of ID_lists, reduces scans of the sequence database. The ID_lists carry the information necessary to find the support of candidates. As the length of a frequent sequence increases, the size of its ID_list decreases, resulting in very fast joins. However, the basic search methodology of SPADE and GSP is breadth-first search (e.g., exploring 1-sequences, then 2-sequences, and so on) and Apriori pruning. Despite the pruning, both algorithms have to generate large sets of candidates in breadth-first manner in order to grow longer sequences. Thus, most of the difficulties suffered in the GSP algorithm recur in SPADE as well.

PrefixSpan: Prefix-Projected Sequential Pattern Growth

Pattern growth is a method of frequent-pattern mining that does not require candidate generation. The technique originated in the FP-growth algorithm for transaction databases, presented in Section 5.2.4. The general idea of this approach is as follows: it finds the frequent single items, then compresses this information into a *frequent-pattern*

SID	*EID*	*itemset*
1	1	a
1	2	abc
1	3	ac
1	4	d
1	5	cf
2	1	ad
2	2	c
2	3	bc
2	4	ae
3	1	ef
3	2	ab
3	3	df
3	4	c
3	5	b
4	1	e
4	2	g
4	3	af
4	4	c
4	5	b
4	6	c

(a) vertical format database

a		b		···
SID	*EID*	*SID*	*EID*	···
1	1	1	2	
1	2	2	3	
1	3	3	2	
2	1	3	5	
2	4	4	5	
3	2			
4	3			

(b) ID_lists for some 1-sequences

ab			ba			···
SID	*EID*(*a*)	*EID*(*b*)	*SID*	*EID*(*b*)	*EID*(*a*)	···
1	1	2	1	2	3	
2	1	3	2	3	4	
3	2	5				
4	3	5				

(c) ID_lists for some 2-sequences

aba			···
SID	*EID*(*a*)	*EID*(*b*)	*EID*(*a*) ···
1	1	2	3
2	1	3	4

(d) ID_lists for some 3-sequences

Figure 8.6 The SPADE mining process: (a) vertical format database; (b) to (d) show fragments of the ID_lists for 1-sequences, 2-sequences, and 3-sequences, respectively.

tree, or *FP-tree*. The FP-tree is used to generate a set of projected databases, each associated with one frequent item. Each of these databases is mined separately. The algorithm builds prefix patterns, which it concatenates with suffix patterns to find frequent patterns, avoiding candidate generation. Here, we look at **PrefixSpan**, which extends the pattern-growth approach to instead mine sequential patterns.

Suppose that all the items within an event are listed alphabetically. For example, instead of listing the items in an event as, say, (bac), we list them as (abc) without loss of generality. Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n \rangle$ (where each e_i corresponds to a frequent event in a sequence database, S), a sequence $\beta = \langle e'_1 e'_2 \cdots e'_m \rangle$ $(m \leq n)$ is called a **prefix** of α if and only if (1) $e'_i = e_i$ for $(i \leq m-1)$; (2) $e'_m \subseteq e_m$; and (3) all the frequent items in $(e_m - e'_m)$ are alphabetically after those in e'_m. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n \rangle$ is called

the **suffix** of α with respect to prefix β, denoted as $\gamma = \alpha/\beta$, where $e''_m = (e_m - e'_m)$.[7] We also denote $\alpha = \beta \cdot \gamma$. Note if β is not a subsequence of α, the suffix of α with respect to β is empty.

We illustrate these concepts with the following example.

Example 8.10 **Prefix and suffix.** Let sequence $s = \langle a(abc)(ac)d(cf)\rangle$, which corresponds to sequence 1 of our running example sequence database. $\langle a\rangle$, $\langle aa\rangle$, $\langle a(ab)\rangle$, and $\langle a(abc)\rangle$ are four prefixes of s. $\langle (abc)(ac)d(cf)\rangle$ is the suffix of s with respect to the prefix $\langle a\rangle$; $\langle (\_bc)(ac)d(cf)\rangle$ is its suffix with respect to the prefix $\langle aa\rangle$; and $\langle (\_c)(ac)d(cf)\rangle$ is its suffix with respect to the prefix $\langle a(ab)\rangle$. ■

Based on the concepts of prefix and suffix, the problem of mining sequential patterns can be decomposed into a set of subproblems as shown:

1. Let $\{\langle x_1\rangle, \langle x_2\rangle, \ldots, \langle x_n\rangle\}$ be the complete set of length-1 sequential patterns in a sequence database, S. The complete set of sequential patterns in S can be partitioned into n disjoint subsets. The i^{th} subset $(1 \leq i \leq n)$ is the set of sequential patterns with prefix $\langle x_i\rangle$.
2. Let α be a length-l sequential pattern and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all length-$(l+1)$ sequential patterns with prefix α. The complete set of sequential patterns with prefix α, except for α itself, can be partitioned into m disjoint subsets. The j^{th} subset $(1 \leq j \leq m)$ is the set of sequential patterns prefixed with β_j.

Based on this observation, the problem can be partitioned recursively. That is, each subset of sequential patterns can be further partitioned when necessary. This forms a *divide-and-conquer* framework. To mine the subsets of sequential patterns, we construct corresponding *projected databases* and mine each one recursively.

Let's use our running example to examine how to use the prefix-based projection approach for mining sequential patterns.

Example 8.11 **PrefixSpan: A pattern-growth approach.** Using the same sequence database, S, of Table 8.1 with $min\_sup = 2$, sequential patterns in S can be mined by a prefix-projection method in the following steps.

1. *Find length-1 sequential patterns.* Scan S once to find all of the frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are $\langle a\rangle : 4$, $\langle b\rangle : 4$, $\langle c\rangle : 4$, $\langle d\rangle : 3$, $\langle e\rangle : 3$, and $\langle f\rangle : 3$, where the notation "$\langle pattern\rangle : count$" represents the pattern and its associated support count.

[7] If e''_m is not empty, the suffix is also denoted as $\langle (\_ \text{ items in } e''_m)e_{m+1} \cdots e_n\rangle$.

Table 8.2 Projected databases and sequential patterns

prefix	*projected database*	*sequential patterns*
$\langle a \rangle$	$\langle (abc)(ac)d(cf) \rangle$, $\langle (\_d)c(bc)(ae) \rangle$, $\langle (\_b)(df)eb \rangle$, $\langle (\_f)cbc \rangle$	$\langle a \rangle$, $\langle aa \rangle$, $\langle ab \rangle$, $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, $\langle abc \rangle$, $\langle (ab) \rangle$, $\langle (ab)c \rangle$, $\langle (ab)d \rangle$, $\langle (ab)f \rangle$, $\langle (ab)dc \rangle$, $\langle ac \rangle$, $\langle aca \rangle$, $\langle acb \rangle$, $\langle acc \rangle$, $\langle ad \rangle$, $\langle adc \rangle$, $\langle af \rangle$
$\langle b \rangle$	$\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)(ae) \rangle$, $\langle (df)cb \rangle$, $\langle c \rangle$	$\langle b \rangle$, $\langle ba \rangle$, $\langle bc \rangle$, $\langle (bc) \rangle$, $\langle (bc)a \rangle$, $\langle bd \rangle$, $\langle bdc \rangle$, $\langle bf \rangle$
$\langle c \rangle$	$\langle (ac)d(cf) \rangle$, $\langle (bc)(ae) \rangle$, $\langle b \rangle$, $\langle bc \rangle$	$\langle c \rangle$, $\langle ca \rangle$, $\langle cb \rangle$, $\langle cc \rangle$
$\langle d \rangle$	$\langle (cf) \rangle$, $\langle c(bc)(ae) \rangle$, $\langle (\_f)cb \rangle$	$\langle d \rangle$, $\langle db \rangle$, $\langle dc \rangle$, $\langle dcb \rangle$
$\langle e \rangle$	$\langle (\_f)(ab)(df)cb \rangle$, $\langle (af)cbc \rangle$	$\langle e \rangle$, $\langle ea \rangle$, $\langle eab \rangle$, $\langle eac \rangle$, $\langle eacb \rangle$, $\langle eb \rangle$, $\langle ebc \rangle$, $\langle ec \rangle$, $\langle ecb \rangle$, $\langle ef \rangle$, $\langle efb \rangle$, $\langle efc \rangle$, $\langle efcb \rangle$.
$\langle f \rangle$	$\langle (ab)(df)cb \rangle$, $\langle cbc \rangle$	$\langle f \rangle$, $\langle fb \rangle$, $\langle fbc \rangle$, $\langle fc \rangle$, $\langle fcb \rangle$

2. *Partition the search space*. The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones with prefix $\langle a \rangle$, (2) the ones with prefix $\langle b \rangle$, ..., and (6) the ones with prefix $\langle f \rangle$.

3. *Find subsets of sequential patterns*. The subsets of sequential patterns mentioned in step 2 can be mined by constructing corresponding *projected databases* and mining each recursively. The projected databases, as well as the sequential patterns found in them, are listed in Table 8.2, while the mining process is explained as follows:

   (a) *Find sequential patterns with prefix* $\langle a \rangle$. Only the sequences containing $\langle a \rangle$ should be collected. Moreover, in a sequence containing $\langle a \rangle$, only the subsequence prefixed with the first occurrence of $\langle a \rangle$ should be considered. For example, in sequence $\langle (ef)(ab)(df)cb \rangle$, only the subsequence $\langle (\_b)(df)cb \rangle$ should be considered for mining sequential patterns prefixed with $\langle a \rangle$. Notice that $(\_b)$ means that the last event in the prefix, which is a, together with b, form one event.

   The sequences in S containing $\langle a \rangle$ are projected with respect to $\langle a \rangle$ to form the $\langle a \rangle$*-projected database*, which consists of four suffix sequences: $\langle (abc)(ac)d(cf) \rangle$, $\langle (\_d)c(bc)(ae) \rangle$, $\langle (\_b)(df)cb \rangle$, and $\langle (\_f)cbc \rangle$.

   By scanning the $\langle a \rangle$-projected database once, its locally frequent items are identified as $a : 2$, $b : 4$, $\_b : 2$, $c : 4$, $d : 2$, and $f : 2$. Thus all the length-2 sequential patterns prefixed with $\langle a \rangle$ are found, and they are: $\langle aa \rangle : 2$, $\langle ab \rangle : 4$, $\langle (ab) \rangle : 2$, $\langle ac \rangle : 4$, $\langle ad \rangle : 2$, and $\langle af \rangle : 2$.

Recursively, all sequential patterns with prefix $\langle a \rangle$ can be partitioned into six subsets: (1) those prefixed with $\langle aa \rangle$, (2) those with $\langle ab \rangle, \ldots,$ and finally, (6) those with $\langle af \rangle$. These subsets can be mined by constructing respective projected databases and mining each recursively as follows:

i. The $\langle aa \rangle$-projected database consists of two nonempty (suffix) subsequences prefixed with $\langle aa \rangle$: $\{\langle (\_bc)(ac)d(cf) \rangle, \{\langle (\_e) \rangle\}$. Because there is no hope of generating any frequent subsequence from this projected database, the processing of the $\langle aa \rangle$-projected database terminates.

ii. The $\langle ab \rangle$-projected database consists of three suffix sequences: $\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)a \rangle$, and $\langle c \rangle$. Recursively mining the $\langle ab \rangle$-projected database returns four sequential patterns: $\langle (\_c) \rangle$, $\langle (\_c)a \rangle$, $\langle a \rangle$, and $\langle c \rangle$ (i.e., $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, and $\langle abc \rangle$.) They form the complete set of sequential patterns prefixed with $\langle ab \rangle$.

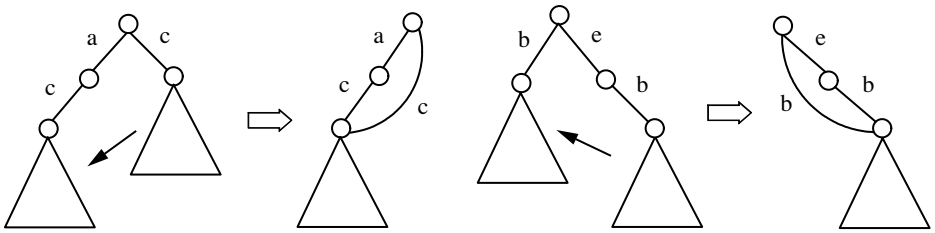
iii. The $\langle (ab) \rangle$-projected database contains only two sequences: $\langle (\_c)(ac)\, d(cf) \rangle$ and $\langle (df)cb \rangle$, which leads to the finding of the following sequential patterns prefixed with $\langle (ab) \rangle$: $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle dc \rangle$.

iv. The $\langle ac \rangle$-, $\langle ad \rangle$-, and $\langle af \rangle$- projected databases can be constructed and recursively mined in a similar manner. The sequential patterns found are shown in Table 8.2.

(b) *Find sequential patterns with prefix* $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, $\langle e \rangle$, *and* $\langle f \rangle$, *respectively*. This can be done by constructing the $\langle b \rangle$-, $\langle c \rangle$-, $\langle d \rangle$-, $\langle e \rangle$-, and $\langle f \rangle$-projected databases and mining them respectively. The projected databases as well as the sequential patterns found are also shown in Table 8.2.

4. *The set of sequential patterns is the collection of patterns found in the above recursive mining process.* ■

The method described above generates no candidate sequences in the mining process. However, it may generate many projected databases, one for each frequent prefix-subsequence. Forming a large number of projected databases recursively may become the major cost of the method, if such databases have to be generated physically. An important optimization technique is **pseudo-projection**, which registers the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence instead of performing physical projection. That is, a physical projection of a sequence is replaced by registering a sequence identifier and the projected position index point. Pseudo-projection reduces the cost of projection substantially when such projection can be done in main memory. However, it may not be efficient if the pseudo-projection is used for disk-based accessing because random access of disk space is costly. The suggested approach is that if the original sequence database or the projected databases are too big to fit in memory, the physical projection should be applied; however, the execution should be swapped to pseudo-projection once the projected databases can fit in memory. This methodology is adopted in the PrefixSpan implementation.



(a) backward subpattern (b) backward superpattern

Figure 8.7 A backward subpattern and a backward superpattern.

A performance comparison of GSP, SPADE, and PrefixSpan shows that PrefixSpan has the best overall performance. SPADE, although weaker than PrefixSpan in most cases, outperforms GSP. Generating huge candidate sets may consume a tremendous amount of memory, thereby causing candidate generate-and-test algorithms to become very slow. The comparison also found that when there is a large number of frequent subsequences, all three algorithms run slowly. This problem can be partially solved by closed sequential pattern mining.

Mining Closed Sequential Patterns

Because mining the complete set of frequent subsequences can generate a huge number of sequential patterns, an interesting alternative is to mine frequent *closed subsequences* only, that is, those containing no supersequence with the same support. Mining closed sequential patterns can produce a significantly less number of sequences than the full set of sequential patterns. Note that the full set of frequent subsequences, together with their supports, can easily be derived from the closed subsequences. Thus, closed subsequences have the same expressive power as the corresponding full set of subsequences. Because of their compactness, they may also be quicker to find.

CloSpan is an efficient closed sequential pattern mining method. The method is based on a property of sequence databases, called **equivalence of projected databases**, stated as follows: *Two projected sequence databases,* $S|_\alpha = S|_\beta$,[8] $\alpha \sqsubseteq \beta$ *(i.e.,* α *is a subsequence of* β*), are equivalent if and only if the total number of items in* $S|_\alpha$ *is equal to the total number of items in* $S|_\beta$.

Based on this property, CloSpan can prune the nonclosed sequences from further consideration during the mining process. That is, whenever we find two prefix-based projected databases that are exactly the same, we can stop growing one of them. This can be used to prune *backward subpatterns* and *backward superpatterns* as indicated in Figure 8.7.

[8] In $S|_\alpha$, a sequence database S is projected with respect to sequence (e.g., prefix) α. The notation $S|_\beta$ can be similarly defined.

After such pruning and mining, a postprocessing step is still required in order to delete nonclosed sequential patterns that may exist in the derived set. A later algorithm called BIDE (which performs a bidirectional search) can further optimize this process to avoid such additional checking.

Empirical results show that CloSpan often derives a much smaller set of sequential patterns in a shorter time than PrefixSpan, which mines the complete set of sequential patterns.

Mining Multidimensional, Multilevel Sequential Patterns

Sequence identifiers (representing individual customers, for example) and sequence items (such as products bought) are often associated with additional pieces of information. Sequential pattern mining should take advantage of such additional information to discover interesting patterns in multidimensional, multilevel information space. Take customer shopping transactions, for instance. In a sequence database for such data, the additional information associated with sequence IDs could include customer age, address, group, and profession. Information associated with items could include item category, brand, model type, model number, place manufactured, and manufacture date. Mining *multidimensional, multilevel* sequential patterns is the discovery of interesting patterns in such a broad dimensional space, at different levels of detail.

Example 8.12 **Multidimensional, multilevel sequential patterns.** The discovery that "*Retired customers who purchase a digital camera are likely to purchase a color printer within a month*" and that "*Young adults who purchase a laptop are likely to buy a flash drive within two weeks*" are examples of multidimensional, multilevel sequential patterns. By grouping customers into "*retired customers*" and "*young adults*" according to the values in the age dimension, and by generalizing items to, say, "*digital camera*" rather than a specific model, the patterns mined here are associated with additional dimensions and are at a higher level of granularity. ■

"*Can a typical sequential pattern algorithm such as PrefixSpan be extended to efficiently mine multidimensional, multilevel sequential patterns?*" One suggested modification is to associate the multidimensional, multilevel information with the *sequence_ID* and *item_ID*, respectively, which the mining method can take into consideration when finding frequent subsequences. For example, (*Chicago*, *middle_aged*, *business*) can be associated with *sequence_ID_1002* (for a given customer), whereas (*Digital_camera*, *Canon*, *Supershot*, *SD*400, *Japan*, 2005) can be associated with *item_ID_543005* in the sequence. A sequential pattern mining algorithm will use such information in the mining process to find sequential patterns associated with multidimensional, multilevel information.

8.3.3 Constraint-Based Mining of Sequential Patterns

As shown in our study of frequent-pattern mining in Chapter 5, mining that is performed without user- or expert-specified constraints may generate numerous patterns that are

of no interest. Such unfocused mining can reduce both the efficiency and usability of frequent-pattern mining. Thus, we promote **constraint-based mining**, which incorporates user-specified constraints to reduce the search space and derive only patterns that are of interest to the user.

Constraints can be expressed in many forms. They may specify desired relationships between attributes, attribute values, or aggregates within the resulting patterns mined. Regular expressions can also be used as constraints in the form of "pattern templates," which specify the desired form of the patterns to be mined. The general concepts introduced for constraint-based frequent pattern mining in Section 5.5.1 apply to constraint-based sequential pattern mining as well. The key idea to note is that these kinds of constraints can be used *during* the mining process to confine the search space, thereby improving (1) the efficiency of the mining and (2) the interestingness of the resulting patterns found. This idea is also referred to as *"pushing the constraints deep into the mining process."*

We now examine some typical examples of constraints for sequential pattern mining. First, constraints can be related to the **duration**, T, of a sequence. The duration may be the maximal or minimal length of the sequence in the database, or a user-specified duration related to time, such as the year 2005. Sequential pattern mining can then be confined to the data within the specified duration, T.

Constraints relating to the maximal or minimal length (duration) can be treated as *antimonotonic* or *monotonic* constraints, respectively. For example, the constraint $T \leq 10$ is **antimonotonic** since, if a sequence does not satisfy this constraint, then neither will any of its supersequences (which are, obviously, longer). The constraint $T > 10$ is **monotonic**. This means that if a sequence satisfies the constraint, then all of its supersequences will also satisfy the constraint. We have already seen several examples in this chapter of how antimonotonic constraints (such as those involving minimum support) can be pushed deep into the mining process to prune the search space. Monotonic constraints can be used in a way similar to its frequent-pattern counterpart as well.

Constraints related to a specific duration, such as a particular year, are considered *succinct* constraints. A constraint is **succinct** if we can enumerate all and only those sequences that are guaranteed to satisfy the constraint, even before support counting begins. Suppose, here, $T = 2005$. By selecting the data for which $year = 2005$, we can enumerate all of the sequences *guaranteed to satisfy* the constraint before mining begins. In other words, we don't need to generate and test. Thus, such constraints contribute toward efficiency in that they avoid the substantial overhead of the generate-and-test paradigm.

Durations may also be defined as being related to sets of partitioned sequences, such as every year, or every month after stock dips, or every two weeks before and after an earthquake. In such cases, *periodic patterns* (Section 8.3.4) can be discovered.

Second, the constraint may be related to an **event folding window**, w. A set of events occurring within a specified period can be viewed as occurring together. If w is set to be as long as the duration, T, it finds time-insensitive frequent patterns—these are essentially frequent patterns, such as "*In 1999, customers who bought a PC bought a digital camera as well*" (i.e., without bothering about which items were bought first). If w is set to 0

(i.e., no event sequence folding), sequential patterns are found where each event occurs at a distinct time instant, such as "*A customer who bought a PC and then a digital camera is likely to buy an SD memory chip in a month*." If w is set to be something in between (e.g., for transactions occurring within the same month or within a sliding window of 24 hours), then these transactions are considered as occurring within the same period, and such sequences are "folded" in the analysis.

Third, a desired (time) **gap** between events in the discovered patterns may be specified as a constraint. Possible cases are: (1) $gap = 0$ (no gap is allowed), which is to find strictly consecutive sequential patterns like $a_{i-1}a_ia_{i+1}$. For example, if the event folding window is set to a week, this will find frequent patterns occurring in consecutive weeks; (2) $min\_gap \leq gap \leq max\_gap$, which is to find patterns that are separated by at least min_gap but at most max_gap, such as "*If a person rents movie A, it is likely she will rent movie B within 30 days*" implies $gap \leq 30$ (days); and (3) $gap = c \neq 0$, which is to find patterns with an exact gap, c. It is straightforward to push gap constraints into the sequential pattern mining process. With minor modifications to the mining process, it can handle constraints with approximate gaps as well.

Finally, a user can specify constraints on the kinds of sequential patterns by providing "pattern templates" in the form of *serial episodes* and *parallel episodes* using *regular expressions*. A **serial episode** is a set of events that occurs in a total order, whereas a **parallel episode** is a set of events whose occurrence ordering is trivial. Consider the following example.

Example 8.13 **Specifying serial episodes and parallel episodes with regular expressions.** Let the notation (E, t) represent *event type* E at *time* t. Consider the data $(A, 1)$, $(C, 2)$, and $(B, 5)$ with an event folding window width of $w = 2$, where the serial episode $A \rightarrow B$ and the parallel episode A & C both occur in the data. The user can specify constraints in the form of a regular expression, such as $(A|B)C * (D|E)$, which indicates that the user would like to find patterns where event A and B first occur (but they are parallel in that their relative ordering is unimportant), followed by one or a set of events C, followed by the events D and E (where D can occur either before or after E). Other events can occur in between those specified in the regular expression. ■

A regular expression constraint may be neither antimonotonic nor monotonic. In such cases, we cannot use it to prune the search space in the same ways as described above. However, by modifying the PrefixSpan-based pattern-growth approach, such constraints can be handled elegantly. Let's examine one such example.

Example 8.14 **Constraint-based sequential pattern mining with a regular expression constraint.** Suppose that our task is to mine sequential patterns, again using the sequence database, S, of Table 8.1. This time, however, we are particularly interested in patterns that match the regular expression constraint, $C = \langle a \star \{bb|(bc)d|dd\}\rangle$, with minimum support.

Such a regular expression constraint is neither antimonotonic, nor monotonic, nor succinct. Therefore, it cannot be pushed deep into the mining process. Nonetheless, this constraint can easily be integrated with the pattern-growth mining process as follows.

First, only the $\langle a \rangle$-projected database, $S|_{\langle a \rangle}$, needs to be mined, since the regular expression constraint C starts with a. Retain only the sequences in $S|_{\langle a \rangle}$ that contain items within the set $\{b, c, d\}$. Second, the remaining mining can proceed from the suffix. This is essentially the *SuffixSpan* algorithm, which is symmetric to PrefixSpan in that it grows suffixes from the end of the sequence forward. The growth should match the suffix as the constraint, $\langle \{bb|(bc)d|dd\} \rangle$. For the projected databases that match these suffixes, we can grow sequential patterns either in prefix- or suffix-expansion manner to find all of the remaining sequential patterns. ■

Thus, we have seen several ways in which constraints can be used to improve the efficiency and usability of sequential pattern mining.

8.3.4 Periodicity Analysis for Time-Related Sequence Data

"*What is periodicity analysis?*" **Periodicity analysis** is the mining of periodic patterns, that is, the search for recurring patterns in time-related sequence data. Periodicity analysis can be applied to many important areas. For example, seasons, tides, planet trajectories, daily power consumptions, daily traffic patterns, and weekly TV programs all present certain periodic patterns. Periodicity analysis is often performed over time-series data, which consists of sequences of values or events typically measured at equal time intervals (e.g., hourly, daily, weekly). It can also be applied to other time-related sequence data where the value or event may occur at a nonequal time interval or at any time (e.g., on-line transactions). Moreover, the items to be analyzed can be numerical data, such as daily temperature or power consumption fluctuations, or categorical data (events), such as purchasing a product or watching a game.

The problem of mining periodic patterns can be viewed from different perspectives. Based on the coverage of the pattern, we can categorize periodic patterns into *full* versus *partial* periodic patterns:

- A **full periodic pattern** is a pattern where every point in time contributes (precisely or approximately) to the cyclic behavior of a time-related sequence. For example, all of the days in the year *approximately* contribute to the season cycle of the year.
- A **partial periodic pattern** specifies the periodic behavior of a time-related sequence at some but not all of the points in time. For example, Sandy reads the *New York Times* from 7:00 to 7:30 every weekday morning, but her activities at other times do not have much regularity. Partial periodicity is a looser form of periodicity than full periodicity and occurs more commonly in the real world.

Based on the precision of the periodicity, a pattern can be either *synchronous* or *asynchronous*, where the former requires that an event occur at a relatively fixed offset in each "stable" period, such as 3 p.m. every day, whereas the latter allows that the event fluctuates in a somewhat loosely defined period. A pattern can also be either *precise* or *approximate*, depending on the data value or the offset within a period. For example, if

Sandy reads the newspaper at 7:00 on some days, but at 7:10 or 7:15 on others, this is an approximate periodic pattern.

Techniques for full periodicity analysis for numerical values have been studied in signal analysis and statistics. Methods like FFT (Fast Fourier Transformation) are commonly used to transform data from the time domain to the frequency domain in order to facilitate such analysis.

Mining partial, categorical, and asynchronous periodic patterns poses more challenging problems in regards to the development of efficient data mining solutions. This is because most statistical methods or those relying on time-to-frequency domain transformations are either inapplicable or expensive at handling such problems.

Take mining partial periodicity as an example. Because partial periodicity mixes periodic events and nonperiodic events together in the same period, a time-to-frequency transformation method, such as FFT, becomes ineffective because it treats the time series as an inseparable flow of values. Certain periodicity detection methods can uncover some partial periodic patterns, but only if the period, length, and timing of the segment (subsequence of interest) in the partial patterns have certain behaviors and are explicitly specified. For the newspaper reading example, we need to explicitly specify details such as "Find the regular activities of Sandy during the half-hour after 7:00 for a period of 24 hours." A naïve adaptation of such methods to the partial periodic pattern mining problem would be prohibitively expensive, requiring their application to a huge number of possible combinations of the three parameters of period, length, and timing.

Most of the studies on mining partial periodic patterns apply the Apriori property heuristic and adopt some variations of Apriori-like mining methods. Constraints can also be pushed deep into the mining process. Studies have also been performed on the efficient mining of partially periodic event patterns or asynchronous periodic patterns with unknown or with approximate periods.

Mining partial periodicity may lead to the discovery of **cyclic or periodic association rules,** which are rules that associate a set of events that occur periodically. An example of a periodic association rule is "*Based on day-to-day transactions, if afternoon tea is well received between 3:00 to 5:00 p.m., dinner will sell well between 7:00 to 9:00 p.m. on weekends.*"

Due to the diversity of applications of time-related sequence data, further development of efficient algorithms for mining various kinds of periodic patterns in sequence databases is desired.

8.4 Mining Sequence Patterns in Biological Data

Bioinformatics is a promising young field that applies computer technology in molecular biology and develops algorithms and methods to manage and analyze biological data. Because DNA and protein sequences are essential biological data and exist in huge volumes as well, it is important to develop effective methods to compare and align biological sequences and discover biosequence patterns.

Before we get into further details, let's look at the type of data being analyzed. DNA and proteins sequences are long linear chains of chemical components. In the case of DNA, these components or "building blocks" are four **nucleotides** (also called *bases*), namely adenine (A), cytosine (C), guanine (G), and thymine (T). In the case of proteins, the components are 20 **amino acids**, denoted by 20 different letters of the alphabet. A gene is a sequence of typically hundreds of individual nucleotides arranged in a particular order. A **genome** is the complete set of genes of an organism. When proteins are needed, the corresponding genes are transcribed into RNA. RNA is a chain of nucleotides. DNA directs the synthesis of a variety of RNA molecules, each with a unique role in cellular function.

"*Why is it useful to compare and align biosequences?*" The alignment is based on the fact that all living organisms are related by evolution. This implies that the nucleotide (DNA, RNA) and proteins sequences of the species that are closer to each other in evolution should exhibit more similarities. An **alignment** is the process of lining up sequences to achieve a maximal level of identity, which also expresses the degree of similarity between sequences. Two sequences are **homologous** if they share a common ancestor. The degree of similarity obtained by sequence alignment can be useful in determining the possibility of homology between two sequences. Such an alignment also helps determine the relative positions of multiple species in an evolution tree, which is called a **phylogenetic tree**.

In Section 8.4.1, we first study methods for *pairwise alignment* (i.e., the alignment of two biological sequences). This is followed by methods for *multiple sequence alignment*. Section 8.4.2 introduces the popularly used Hidden Markov Model (HMM) for biological sequence analysis.

8.4.1 Alignment of Biological Sequences

The problem of alignment of biological sequences can be described as follows: *Given two or more input biological sequences, identify similar sequences with long conserved subsequences*. If the number of sequences to be aligned is exactly two, it is called **pairwise sequence alignment**; otherwise, it is **multiple sequence alignment**. The sequences to be compared and aligned can be either nucleotides (DNA/RNA) or amino acids (proteins). For nucleotides, two symbols align if they are identical. However, for amino acids, two symbols align if they are identical, or if one can be derived from the other by substitutions that are likely to occur in nature. There are two kinds of alignments: *local alignments* versus *global alignments*. The former means that only portions of the sequences are aligned, whereas the latter requires alignment over the entire length of the sequences.

For either nucleotides or amino acids, insertions, deletions, and substitutions occur in nature with different probabilities. **Substitution matrices** are used to represent the probabilities of substitutions of nucleotides or amino acids and probabilities of insertions and deletions. Usually, we use the gap character, "−", to indicate positions where it is preferable not to align two symbols. To evaluate the quality of alignments, a *scoring* mechanism is typically defined, which usually counts identical or similar symbols as positive scores and gaps as negative ones. The algebraic sum of the scores is taken as the alignment measure. The goal of alignment is to achieve the maximal score among all the

possible alignments. However, it is very expensive (more exactly, an NP-hard problem) to find optimal alignment. Therefore, various heuristic methods have been developed to find suboptimal alignments.

Pairwise Alignment

Example 8.15 **Pairwise alignment.** Suppose we have two amino acid sequences as follows, and the substitution matrix of amino acids for pairwise alignment is shown in Table 8.3.

Suppose the penalty for initiating a gap (called the *gap penalty*) is -8 and that for extending a gap (i.e., *gap extension penalty*) is also -8. We can then compare two potential sequence alignment candidates, as shown in Figure 8.8 (a) and (b) by calculating their total alignment scores.

The total score of the alignment for Figure 8.8(a) is $(-2)+(-8)+(5)+(-8)+(-8)+(15)+(-8)+(10)+(6)+(-8)+(6)=0$, whereas that for Figure 8.8(b) is

Table 8.3 The substitution matrix of amino acids.

	A	*E*	*G*	*H*	*W*
A	5	−1	0	−2	−3
E	−1	6	−3	0	−3
H	−2	0	−2	10	−3
P	−1	−1	−2	−2	−4
W	−3	−3	−3	−3	15

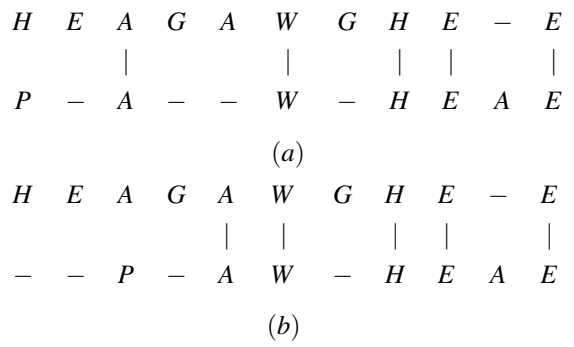


Figure 8.8 Scoring two potential pairwise alignments, (a) and (b), of amino acids.

$(-8)+(-8)+(-1)+(-8)+(5)+(15)+(-8)+(10)+(6)+(-8)+(6)=1$. Thus the alignment of Figure 8.8(b) is slightly better than that in Figure 8.8(a). ■

Biologists have developed 20×20 triangular matrices that provide the weights for comparing identical and different amino acids as well as the penalties that should be attributed to gaps. Two frequently used matrices are PAM (Percent Accepted Mutation) and BLOSUM (BlOcks SUbstitution Matrix). These substitution matrices represent the weights obtained by comparing the amino acid substitutions that have occurred through evolution.

For global pairwise sequence alignment, two influential algorithms have been proposed: the *Needleman-Wunsch Algorithm* and the *Smith-Waterman Algorithm*. The former uses weights for the outmost edges that encourage the best overall global alignment, whereas the latter favors the contiguity of segments being aligned. Both build up "optimal" alignment from "optimal" alignments of subsequences. Both use the methodology of dynamic programming. Since these algorithms use recursion to fill in an intermediate results table, it takes $O(mn)$ space and $O(n^2)$ time to execute them. Such computational complexity could be feasible for moderate-sized sequences but is not feasible for aligning large sequences, especially for entire genomes, where a *genome* is the complete set of genes of an organism. Another approach called *dot matrix plot* uses Boolean matrices to represent possible alignments that can be detected visually. The method is simple and facilitates easy visual inspection. However, it still takes $O(n^2)$ in time and space to construct and inspect such matrices.

To reduce the computational complexity, heuristic alignment algorithms have been proposed. Heuristic algorithms speed up the alignment process at the price of possibly missing the best scoring alignment. There are two influential heuristic alignment programs: (1) BLAST (Basic Local Alignment Search Tool), and (2) FASTA (Fast Alignment Tool). Both find high-scoring local alignments between a query sequence and a target database. Their basic idea is to first locate high-scoring short stretches and then extend them to achieve suboptimal alignments. Because the BLAST algorithm has been very popular in biology and bioinformatics research, we examine it in greater detail here.

The BLAST Local Alignment Algorithm

The **BLAST** algorithm was first developed by Altschul, Gish, Miller, et al. around 1990 at the National Center for Biotechnology Information (NCBI). The software, its tutorials, and a wealth of other information can be accessed at *www.ncbi.nlm.nih.gov/BLAST/*. BLAST finds regions of local similarity between biosequences. The program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches. BLAST can be used to infer functional and evolutionary relationships between sequences as well as to help identify members of gene families.

The NCBI website contains many common BLAST databases. According to their content, they are grouped into nucleotide and protein databases. NCBI also provides specialized BLAST databases such as the vector screening database, a variety of genome databases for different organisms, and trace databases.

BLAST applies a heuristic method to find the highest local alignments between a query sequence and a database. BLAST improves the overall speed of search by breaking the sequences to be compared into sequences of fragments (referred to as **words**) and initially seeking matches between these words. In BLAST, the words are considered as k-tuples. For DNA nucleotides, a word typically consists of 11 bases (nucleotides), whereas for proteins, a word typically consists of 3 amino acids. BLAST first creates a hash table of neighborhood (i.e., closely matching) words, while the threshold for "closeness" is set based on statistics. It starts from exact matches to neighborhood words. Because good alignments should contain many close matches, we can use statistics to determine which matches are significant. By hashing, we can find matches in $O(n)$ (linear) time. By extending matches in both directions, the method finds high-quality alignments consisting of many high-scoring and maximum segment pairs.

There are many versions and extensions of the BLAST algorithms. For example, MEGABLAST, Discontiguous MEGABLAST, and BLASTN all can be used to identify a nucleotide sequence. MEGABLAST is specifically designed to efficiently find long alignments between very similar sequences, and thus is the best tool to use to find the identical match to a query sequence. Discontiguous MEGABLAST is better at finding nucleotide sequences that are similar, but not identical (i.e., gapped alignments), to a nucleotide query. One of the important parameters governing the sensitivity of BLAST searches is the length of the initial words, or *word size*. The word size is adjustable in BLASTN and can be reduced from the default value to a minimum of 7 to increase search sensitivity. Thus BLASTN is better than MEGABLAST at finding alignments to related nucleotide sequences from other organisms. For protein searches, BLASTP, PSI-BLAST, and PHI-BLAST are popular. Standard protein-protein BLAST (BLASTP) is used for both identifying a query amino acid sequence and for finding similar sequences in protein databases. Position-Specific Iterated (PSI)-BLAST is designed for more sensitive protein-protein similarity searches. It is useful for finding very distantly related proteins. Pattern-Hit Initiated (PHI)-BLAST can do a restricted protein pattern search. It is designed to search for proteins that contain a pattern specified by the user and are similar to the query sequence in the vicinity of the pattern. This dual requirement is intended to reduce the number of database hits that contain the pattern, but are likely to have no true homology to the query.

Multiple Sequence Alignment Methods

Multiple sequence alignment is usually performed on a set of sequences of amino acids that are believed to have similar structures. The goal is to find common patterns that are conserved among all the sequences being considered.

The alignment of multiple sequences has many applications. First, such an alignment may assist in the identification of highly conserved residues (amino acids), which are likely to be essential sites for structure and function. This will guide or help pairwise alignment as well. Second, it will help build gene or protein families using conserved regions, forming a basis for phylogenetic analysis (i.e., the inference of evolutionary relationships between genes). Third, conserved regions can be used to develop primers for amplifying DNA sequences and probes for DNA microarray analysis.

From the computational point of view, it is more challenging to align multiple sequences than to perform pairwise alignment of two sequences. This is because multisequence alignment can be considered as a multidimensional alignment problem, and there are many more possibilities for approximate alignments of subsequences in multiple dimensions.

There are two major approaches for approximate multiple sequence alignment. The first method reduces a multiple alignment to a series of pairwise alignments and then combines the result. The popular **Feng-Doolittle alignment** method belongs to this approach. Feng-Doolittle alignment first computes all of the possible pairwise alignments by dynamic programming and converts or normalizes alignment scores to distances. It then constructs a "guide tree" by clustering and performs progressive alignment based on the guide tree in a bottom-up manner. Following this approach, a multiple alignment tool, Clustal W, and its variants have been developed as software packages for multiple sequence alignments. The software handles a variety of input/output formats and provides displays for visual inspection.

The second multiple sequence alignment method uses hidden Markov models (HMMs). Due to the extensive use and popularity of hidden Markov models, we devote an entire section to this approach. It is introduced in Section 8.4.2, which follows.

From the above discussion, we can see that several interesting methods have been developed for multiple sequence alignment. Due to its computational complexity, the development of effective and scalable methods for multiple sequence alignment remains an active research topic in biological data mining.

8.4.2 Hidden Markov Model for Biological Sequence Analysis

Given a biological sequence, such as a DNA sequence or an amino acid (protein), biologists would like to analyze what that sequence represents. For example, is a given DNA sequence a gene or not? Or, to which family of proteins does a particular amino acid sequence belong? In general, given sequences of symbols from some alphabet, we would like to represent the structure or statistical regularities of classes of sequences. In this section, we discuss *Markov chains* and *hidden Markov models*—probabilistic models that are well suited for this type of task. Other areas of research, such as speech and pattern recognition, are faced with similar sequence analysis tasks.

To illustrate our discussion of Markov chains and hidden Markov models, we use a classic problem in biological sequence analysis—that of finding *CpG islands* in a DNA sequence. Here, the alphabet consists of four **nucleotides**, namely, A (adenine), C (cytosine), G (guanine), and T (thymine). **CpG** denotes a pair (or subsequence) of nucleotides, where G appears immediately after C along a DNA strand. The C in a CpG pair is often modified by a process known as *methylation* (where the C is replaced by methyl-C, which tends to mutate to T). As a result, CpG pairs occur infrequently in the human genome. However, methylation is often suppressed around *promotors* or "start" regions of many genes. These areas contain a relatively high concentration of CpG pairs, collectively referred to along a chromosome as **CpG islands**, which typically vary in length from a few hundred to a few thousand nucleotides long. CpG islands are very useful in genome mapping projects.

Two important questions that biologists have when studying DNA sequences are (1) given a short sequence, is it from a CpG island or not? and (2) given a long sequence, can we find all of the CpG islands within it? We start our exploration of these questions by introducing Markov chains.

Markov Chain

A **Markov chain** is a model that generates sequences in which the probability of a symbol depends only on the previous symbol. Figure 8.9 is an example Markov chain model. A Markov chain model is defined by (a) a set of *states*, Q, which emit symbols and (b) a set of *transitions* between states. States are represented by circles and transitions are represented by arrows. Each transition has an associated **transition probability**, a_{ij}, which represents the conditional probability of going to state j in the next step, given that the current state is i. The sum of all transition probabilities from a given state must equal 1, that is, $\sum_{j \in Q} a_{ij} = 1$ for all $j \in Q$. If an arc is not shown, it is assumed to have a 0 probability. The transition probabilities can also be written as a *transition matrix*, $A = \{a_{ij}\}$.

Example 8.16 **Markov chain.** The Markov chain in Figure 8.9 is a probabilistic model for CpG islands. The states are A, C, G, and T. For readability, only some of the transition probabilities are shown. For example, the transition probability from state G to state T is 0.14, that is, $P(x_i = \mathrm{T} | x_{i-1} = \mathrm{G}) = 0.14$. Here, the emitted symbols are understood. For example, the symbol C is emitted when transitioning from state C. In speech recognition, the symbols emitted could represent spoken words or phrases. ■

Given some sequence x of length L, how probable is x given the model? If x is a DNA sequence, we could use our Markov chain model to determine how probable it is that x is from a CpG island. To do so, we look at the probability of x as a *path*, $x_1 x_2 \ldots x_L$, in the chain. This is the probability of starting in the first state, x_1, and making successive transitions to x_2, x_3, and so on, to x_L. In a Markov chain model, the probability of x_L

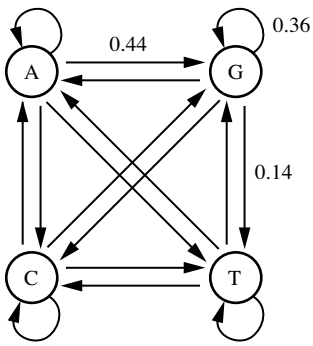


Figure 8.9 A Markov chain model.

depends on the value of only the *previous one state*, x_{L-1}, not on the entire previous sequence.[9] This characteristic is known as the **Markov property**, which can be written as

$$\begin{aligned} P(x) &= P(x_L|x_{L-1})P(x_{L-1}|x_{L-2})\cdots P(x_2|x_1)P(x_1) \\ &= P(x_1)\prod_{i=2}^{L} P(x_i|x_{i-1}). \end{aligned} \tag{8.7}$$

That is, the Markov chain can only "remember" the previous one state of its history. Beyond that, it is "memoryless."

In Equation (8.7), we need to specify $P(x_1)$, the probability of the starting state. For simplicity, we would like to model this as a transition too. This can be done by adding a *begin* state, denoted 0, so that the starting state becomes $x_0 = 0$. Similarly, we can add an *end* state, also denoted as 0. Note that $P(x_i|x_{i-1})$ is the transition probability, $a_{x_{i-1}x_i}$. Therefore, Equation (8.7) can be rewritten as

$$P(x) = \prod_{i=1}^{L} a_{x_{i-1}x_i}, \tag{8.8}$$

which computes the probability that sequence x belongs to the given Markov chain model, that is, $P(x|model)$. Note that the begin and end states are silent in that they do not emit symbols in the path through the chain.

We can use the Markov chain model for classification. Suppose that we want to distinguish CpG islands from other "non-CpG" sequence regions. Given training sequences from CpG islands (labeled "+") and from non-CpG islands (labeled "−"), we can construct two Markov chain models—the first, denoted "+", to represent CpG islands, and the second, denoted "−", to represent non-CpG islands. Given a sequence, x, we use the respective models to compute $P(x|+)$, the probability that x is from a CpG island, and $P(x|-)$, the probability that it is from a non-CpG island. The *log-odds ratio* can then be used to classify x based on these two probabilities.

"But first, how can we estimate the transition probabilities for each model?" Before we can compute the probability of x being from either of the two models, we need to estimate the transition probabilities for the models. Given the CpG (+) training sequences, we can estimate the transition probabilities for the CpG island model as

$$a_{ij}^{+} = \frac{c_{ij}^{+}}{\sum_k c_{ik}^{+}}, \tag{8.9}$$

where c_{ij}^{+} is the number of times that nucleotide j follows nucleotide i in the given sequences labeled "+". For the non-CpG model, we use the non-CpG island sequences (labeled "−") in a similar way to estimate a_{ij}^{-}.

[9] This is known as a **first-order Markov chain model**, since x_L depends only on the previous state, x_{L-1}. In general, for the *k*-**th-order Markov chain model**, the probability of x_L depends on the values of only the *previous k* states.

To determine whether x is from a CpG island or not, we compare the models using the **logs-odds ratio**, defined as

$$log\frac{P(x|+)}{P(x|-)} = \sum_{i=1}^{L} log\frac{a^{+}_{x_{i-1}x_i}}{a^{-}_{x_{i-1}x_i}}. \tag{8.10}$$

If this ratio is greater than 0, then we say that x is from a CpG island.

Example 8.17 **Classification using a Markov chain.** Our model for CpG islands and our model for non-CpG islands both have the same structure, as shown in our example Markov chain of Figure 8.9. Let CpG^+ be the transition matrix for the CpG island model. Similarly, CpG^- is the transition matrix for the non-CpG island model. These are (adapted from Durbin, Eddy, Krogh, and Mitchison [DEKM98]):

$$CpG^+ = \begin{bmatrix} & A & C & G & T \\ A & 0.20 & 0.26 & 0.44 & 0.10 \\ C & 0.16 & 0.36 & 0.28 & 0.20 \\ G & 0.15 & 0.35 & 0.36 & 0.14 \\ T & 0.09 & 0.37 & 0.36 & 0.18 \end{bmatrix} \tag{8.11}$$

$$CpG^- = \begin{bmatrix} & A & C & G & T \\ A & 0.27 & 0.19 & 0.31 & 0.23 \\ C & 0.33 & 0.31 & 0.08 & 0.28 \\ G & 0.26 & 0.24 & 0.31 & 0.19 \\ T & 0.19 & 0.25 & 0.28 & 0.28 \end{bmatrix} \tag{8.12}$$

Notice that the transition probability $a^+_{CG} = 0.28$ is higher than $a^-_{CG} = 0.08$. Suppose we are given the sequence x = CGCG. The log-odds ratio of x is

$$log\frac{0.28}{0.08} + log\frac{0.35}{0.24} + log\frac{0.28}{0.08} = 1.25 > 0.$$

Thus, we say that x is from a CpG island. ■

In summary, we can use a Markov chain model to determine if a DNA sequence, x, is from a CpG island. This was the first of our two important questions mentioned at the beginning of this section. To answer the second question, that of finding all of the CpG islands in a given sequence, we move on to hidden Markov models.

Hidden Markov Model

Given a long DNA sequence, how can we find all CpG islands within it? We could try the Markov chain method above, using a sliding window. For each window, we could

compute the log-odds ratio. CpG islands within intersecting windows could be merged to determine CpG islands within the long sequence. This approach has some difficulties: It is not clear what window size to use, and CpG islands tend to vary in length.

What if, instead, we merge the two Markov chains from above (for CpG islands and non-CpG islands, respectively) and add transition probabilities between the two chains? The result is a *hidden Markov model*, as shown in Figure 8.10. The states are renamed by adding "+" and "−" labels to distinguish them. For readability, only the transitions between "+" and "−" states are shown, in addition to those for the begin and end states. Let $\pi = \pi_1 \pi_2 \ldots \pi_L$ be a path of states that generates a sequence of symbols, $x = x_1 x_2 \ldots x_L$. In a Markov chain, the path through the chain for x is unique. With a hidden Markov model, however, different paths can generate the same sequence. For example, the states C^+ and C^- both emit the symbol C. Therefore, we say the model is "hidden" in that we do not know for sure which states were visited in generating the sequence. The transition probabilities between the original two models can be determined using training sequences containing transitions between CpG islands and non-CpG islands.

A Hidden Markov Model (HMM) is defined by

- a set of states, Q
- a set of transitions, where transition probability $a_{kl} = P(\pi_i = l | \pi_{i-1} = k)$ is the probability of transitioning from state k to state l for $k, l \in Q$
- an **emission probability**, $e_k(b) = P(x_i = b | \pi_i = k)$, for each state, k, and each symbol, b, where $e_k(b)$ is the probability of seeing symbol b in state k. The sum of all emission probabilities at a given state must equal 1, that is, $\sum_b e_k = 1$ for each state, k.

Example 8.18 **A hidden Markov model.** The transition matrix for the hidden Markov model of Figure 8.10 is larger than that of Example 8.16 for our earlier Markov chain example.

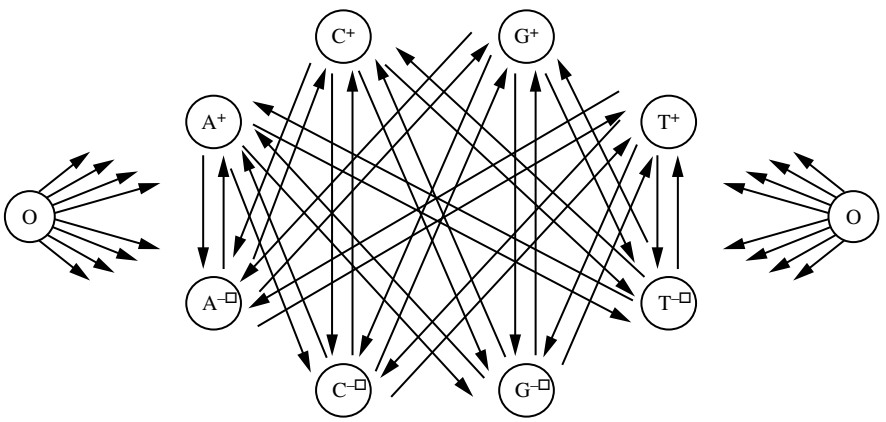


Figure 8.10 A hidden Markov model.

It contains the states A^+, C^+, G^+, T^+, A^-, C^-, G^-, T^- (not shown). The transition probabilities between the "+" states are as before. Similarly, the transition probabilities between the "−" states are as before. The transition probabilities between "+" and "−" states can be determined as mentioned above, using training sequences containing known transitions from CpG islands to non-CpG islands, and vice versa. The emission probabilities are $e_{A^+}(A) = 1$, $e_{A^+}(C) = 0$, $e_{A^+}(G) = 0$, $e_{A^+}(T) = 0$, $e_{A^-}(A) = 1$, $e_{A^-}(C) = 0$, $e_{A^-}(G) = 0$, $e_{A^-}(T) = 0$, and so on. Although here the probability of emitting a symbol at a state is either 0 or 1, in general, emission probabilities need not be zero-one. ■

Tasks using hidden Markov models include:

- *Evaluation*: Given a sequence, x, determine the probability, $P(x)$, of obtaining x in the model.
- *Decoding:* Given a sequence, determine the most probable path through the model that produced the sequence.
- *Learning*: Given a model and a set of training sequences, find the model parameters (i.e., the transition and emission probabilities) that explain the training sequences with relatively high probability. The goal is to find a model that generalizes well to sequences we have not seen before.

Evaluation, decoding, and learning can be handled using the forward algorithm, Viterbi algorithm, and Baum-Welch algorithm, respectively. These algorithms are discussed in the following sections.

Forward Algorithm

What is the probability, $P(x)$, that sequence x was generated by a given hidden Markov model (where, say, the model represents a sequence class)? This problem can be solved using the **forward algorithm**.

Let $x = x_1 x_2 \ldots x_L$ be our sequence of symbols. A path is a sequence of states. Many paths can generate x. Consider one such path, which we denote $\pi = \pi_1 \pi_2 \ldots \pi_L$. If we incorporate the begin and end states, denoted as 0, we can write π as $\pi_0 = 0, \pi_1 \pi_2 \ldots \pi_L$, $\pi_{L+1} = 0$. The probability that the model generated sequence x using path π is

$$\begin{aligned} P(x, \pi) &= a_{0\pi_1} e_{\pi_1}(x_1) \cdot a_{\pi_1\pi_2} e_{\pi_2}(x_2) \cdot \cdots \cdot a_{\pi_{L-1}\pi_L} e_{\pi_L}(x_L) \cdot a_{\pi_L 0} \\ &= a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) a_{\pi_i \pi_{i+1}} \end{aligned} \tag{8.13}$$

where $\pi_{L+1} = 0$. We must, however, consider all of the paths that can generate x. Therefore, the probability of x given the model is

$$P(x) = \sum_{\pi} P(x, \pi). \tag{8.14}$$

That is, we add the probabilities of all possible paths for x.

Algorithm: Forward algorithm. Find the probability, $P(x)$, that sequence x was generated by the given hidden Markov model.

Input:

- A hidden Markov model, defined by a set of states, Q, that emit symbols, and by transition and emission probabilities;
- x, a sequence of symbols.

Output: Probability, $P(x)$.

Method:

(1) Initialization ($i = 0$): $f_0(0) = 1$, $f_k(0) = 0$ for $k > 0$
(2) Recursion ($i = 1 \ldots L$): $f_l(i) = e_l(x_i) \sum_k f_k(i-1) a_{kl}$
(3) Termination: $P(x) = \sum_k f_k(L) a_{k0}$

Figure 8.11 Forward algorithm.

Unfortunately, the number of paths can be exponential with respect to the length, L, of x, so brute force evaluation by enumerating all paths is impractical. The forward algorithm exploits a dynamic programming technique to solve this problem. It defines **forward variables**, $f_k(i)$, to be the probability of being in state k having observed the first i symbols of sequence x. We want to compute $f_{\pi_{L+1=0}}(L)$, the probability of being in the end state having observed all of sequence x.

The forward algorithm is shown in Figure 8.11. It consists of three steps. In step 1, the forward variables are initialized for all states. Because we have not viewed any part of the sequence at this point, the probability of being in the start state is 1 (i.e., $f_0(0) = 1$), and the probability of being in any other state is 0. In step 2, the algorithm sums over all the probabilities of all the paths leading from one state emission to another. It does this recursively for each move from state to state. Step 3 gives the termination condition. The whole sequence (of length L) has been viewed, and we enter the end state, 0. We end up with the summed-over probability of generating the required sequence of symbols.

Viterbi Algorithm

Given a sequence, x, what is the most probable path in the model that generates x? This problem of decoding can be solved using the **Viterbi algorithm**.

Many paths can generate x. We want to find the most probable one, $\pi^\star$, that is, the path that maximizes the probability of having generated x. This is $\pi^\star = argmax_\pi P(\pi|x)$.[10] It so happens that this is equal to $argmax_\pi P(x, \pi)$. (The proof is left as an exercise for the reader.) We saw how to compute $P(x, \pi)$ in Equation (8.13). For a sequence of length L, there are $|Q|^L$ possible paths, where $|Q|$ is the number of states in the model. It is

[10] In mathematics, *argmax* stands for the argument of the maximum. Here, this means that we want the path, π, for which $P(\pi|x)$ attains its maximum value.

infeasible to enumerate all of these possible paths! Once again, we resort to a dynamic programming technique to solve the problem.

At each step along the way, the Viterbi algorithm tries to find the most probable path leading from one symbol of the sequence to the next. We define $v_l(i)$ to be the probability of the most probable path accounting for the first i symbols of x and ending in state l. To find $\pi^\star$, we need to compute $max_k v_k(L)$, the probability of the most probable path accounting for all of the sequence and ending in the end state. The probability, $v_l(i)$, is

$$v_l(i) = e_l(x_i) \cdot max_k(v_l(k)a_{kl}), \tag{8.15}$$

which states that the most probable path that generates $x_1 \ldots x_i$ and ends in state l has to emit x_i in state x_l (hence, the emission probability, $e_l(x_i)$) and has to contain the most probable path that generates $x_1 \ldots x_{i-1}$ and ends in state k, followed by a transition from state k to state l (hence, the transition probability, a_{kl}). Thus, we can compute $v_k(L)$ for any state, k, recursively to obtain the probability of the most probable path.

The Viterbi algorithm is shown in Figure 8.12. Step 1 performs initialization. Every path starts at the begin state (0) with probability 1. Thus, for $i = 0$, we have $v_0(0) = 1$, and the probability of starting at any other state is 0. Step 2 applies the *recurrence formula* for $i = 1$ to L. At each iteration, we assume that we know the most likely path for $x_1 \ldots x_{i-1}$ that ends in state k, for all $k \in Q$. To find the most likely path to the i-th state from there, we maximize $v_k(i-1)a_{kl}$ over all predecessors $k \in Q$ of l. To obtain $v_l(i)$, we multiply by $e_l(x_i)$ since we have to emit x_i from l. This gives us the first formula in step 2. The values $v_k(i)$ are stored in a $Q \times L$ dynamic programming matrix. We keep pointers (*ptr*) in this matrix so that we can obtain the path itself. The algorithm terminates in step 3, where finally, we have $max_k v_k(L)$. We enter the end state of 0 (hence, the transition probability, a_{k0}) but do not emit a symbol. The Viterbi algorithm runs in $O(|Q|^2|L|)$ time. It is more efficient than the forward algorithm because it investigates only the most probable path and avoids summing over all possible paths.

Algorithm: Viterbi algorithm. Find the most probable path that emits the sequence of symbols, x.

Input:

- A hidden Markov model, defined by a set of states, Q, that emit symbols, and by transition and emission probabilities;
- x, a sequence of symbols.

Output: The most probable path, π^*.

Method:

(1)	Initialization ($i = 0$):	$v_0(0) = 1$, $v_k(0) = 0$ for $k > 0$
(2)	Recursion ($i = 1 \ldots L$):	$v_l(i) = e_l(x_i) max_k(v_k(i-1)a_{kl})$
		$ptr_i(l) = argmax_k(v_k(i-1)a_{kl})$
(3)	Termination:	$P(x, \pi^*) = max_k(v_k(L)a_{k0})$
		$\pi_L^* = argmax_k(v_k(L)a_{k0})$

Figure 8.12 Viterbi (decoding) algorithm.

Baum-Welch Algorithm

Given a training set of sequences, how can we determine the parameters of a hidden Markov model that will best explain the sequences? In other words, we want to learn or adjust the transition and emission probabilities of the model so that it can predict the path of future sequences of symbols. If we know the state path for each training sequence, learning the model parameters is simple. We can compute the percentage of times each particular transition or emission is used in the set of training sequences to determine a_{kl}, the transition probabilities, and $e_k(b)$, the emission probabilities.

When the paths for the training sequences are unknown, there is no longer a direct closed-form equation for the estimated parameter values. An iterative procedure must be used, like the **Baum-Welch algorithm**. The Baum-Welch algorithm is a special case of the EM algorithm (Section 7.8.1), which is a family of algorithms for learning probabilistic models in problems that involve hidden states.

The Baum-Welch algorithm is shown in Figure 8.13. The problem of finding the optimal transition and emission probabilities is intractable. Instead, the Baum-Welch algorithm finds a locally optimal solution. In step 1, it initializes the probabilities to an arbitrary estimate. It then continuously re-estimates the probabilities (step 2) until convergence (i.e., when there is very little change in the probability values between iterations). The re-estimation first calculates the expected transmission and emission probabilities. The transition and emission probabilities are then updated to maximize the likelihood of the expected values.

In summary, Markov chains and hidden Markov models are probabilistic models in which the probability of a state depends only on that of the previous state. They are particularly useful for the analysis of biological sequence data, whose tasks include evaluation, decoding, and learning. We have studied the forward, Viterbi, and Baum-Welch algorithms. The algorithms require multiplying many probabilities, resulting in very

Algorithm: Baum-Welch algorithm. Find the model parameters (transition and emission probabilities) that best explain the training set of sequences.

Input:

- A training set of sequences.

Output:

- Transition probabilities, a_{kl};
- Emission probabilities, $e_k(b)$;

Method:

(1) initialize the transmission and emission probabilities;
(2) iterate until convergence
  (2.1) calculate the expected number of times each transition or emission is used
  (2.2) adjust the parameters to maximize the likelihood of these expected values

Figure 8.13 Baum-Welch (learning) algorithm.

small numbers that can cause underflow arithmetic errors. A way around this is to use the logarithms of the probabilities.

8.5 Summary

- **Stream data** flow in and out of a computer system *continuously* and with varying update rates. They are *temporally ordered, fast changing, massive* (e.g., gigabytes to terabytes in volume), and *potentially infinite*. Applications involving stream data include telecommunications, financial markets, and satellite data processing.
- **Synopses** provide *summaries* of stream data, which typically can be used to return *approximate* answers to queries. Random sampling, sliding windows, histograms, multiresolution methods (e.g., for data reduction), sketches (which operate in a single pass), and randomized algorithms are all forms of synopses.
- The **tilted time frame** model allows data to be stored at multiple granularities of time. The most recent time is registered at the finest granularity. The most distant time is at the coarsest granularity.
- A **stream data cube** can store compressed data by (1) using the tilted time frame model on the time dimension, (2) storing data at only some **critical layers**, which reflect the levels of data that are of most interest to the analyst, and (3) performing *partial materialization* based on "popular paths" through the critical layers.
- Traditional methods of **frequent itemset mining, classification**, and **clustering** tend to scan the data multiple times, making them infeasible for stream data. Stream-based versions of such mining instead try to find approximate answers within a user-specified error bound. Examples include the Lossy Counting algorithm for frequent itemset stream mining; the Hoeffding tree, VFDT, and CVFDT algorithms for stream data classification; and the STREAM and CluStream algorithms for stream data clustering.
- A **time-series database** consists of sequences of values or events changing with time, typically measured at equal time intervals. Applications include stock market analysis, economic and sales forecasting, cardiogram analysis, and the observation of weather phenomena.
- **Trend analysis** decomposes time-series data into the following: *trend* (long-term) *movements*, *cyclic movements, seasonal movements* (which are systematic or calendar related), and *irregular movements* (due to random or chance events).
- **Subsequence matching** is a form of *similarity search* that finds subsequences that are similar to a given query sequence. Such methods match subsequences that have the same shape, while accounting for gaps (missing values) and differences in baseline/offset and scale.
- A **sequence database** consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. Examples of sequence data include customer shopping sequences, Web clickstreams, and biological sequences.

- **Sequential pattern mining** is the mining of frequently occurring ordered events or subsequences as patterns. Given a sequence database, any sequence that satisfies minimum support is **frequent** and is called a **sequential pattern**. An example of a sequential pattern is "*Customers who buy a Canon digital camera are likely to buy an HP color printer within a month.*" Algorithms for sequential pattern mining include GSP, SPADE, and PrefixSpan, as well as CloSpan (which mines closed sequential patterns).
- **Constraint-based mining** of sequential patterns incorporates user-specified constraints to reduce the search space and derive only patterns that are of interest to the user. Constraints may relate to the *duration* of a sequence, to an *event folding window* (where events occurring within such a window of time can be viewed as occurring together), and to *gaps* between events. *Pattern templates* may also be specified as a form of constraint using regular expressions.
- **Periodicity analysis** is the mining of periodic patterns, that is, the search for recurring patterns in time-related sequence databases. *Full periodic* and *partial periodic* patterns can be mined, as well as *periodic association rules*.
- **Biological sequence analysis** compares, aligns, indexes, and analyzes biological sequences, which can be either sequences of nucleotides or of amino acids. Biosequence analysis plays a crucial role in bioinformatics and modern biology. Such analysis can be partitioned into two essential tasks: **pairwise sequence alignment** and **multiple sequence alignment**. The dynamic programming approach is commonly used for sequence alignments. Among many available analysis packages, BLAST (Basic Local Alignment Search Tool) is one of the most popular tools in biosequence analysis.
- **Markov chains** and **hidden Markov models** are probabilistic models in which the probability of a state depends only on that of the previous state. They are particularly useful for the analysis of biological sequence data. Given a sequence of symbols, x, the forward algorithm finds the probability of obtaining x in the model, whereas the Viterbi algorithm finds the most probable path (corresponding to x) through the model. The Baum-Welch algorithm learns or adjusts the model parameters (*transition* and *emission* probabilities) so as to best explain a set of training sequences.

Exercises

8.1 A *stream data cube* should be relatively stable in size with respect to infinite data streams. Moreover, it should be incrementally updateable with respect to infinite data streams. Show that the stream cube proposed in Section 8.1.2 satisfies these two requirements.

8.2 In stream data analysis, we are often interested in only the nontrivial or exceptionally large cube cells. These can be formulated as *iceberg conditions*. Thus, it may seem that the iceberg cube [BR99] is a likely model for stream cube architecture. Unfortunately, this is not the case because iceberg cubes cannot accommodate the incremental updates required due to the constant arrival of new data. Explain why.

8.3 An important task in stream data analysis is to *detect outliers* in a multidimensional environment. An example is the detection of unusual power surges, where the dimensions include *time* (i.e., comparing with the normal duration), *region* (i.e., comparing with surrounding regions), *sector* (i.e., university, residence, government), and so on. Outline an efficient stream OLAP method that can detect outliers in data streams. Provide reasons as to why your design can ensure such quality.

8.4 *Frequent itemset mining in data streams* is a challenging task. It is too costly to keep the frequency count for every itemset. However, because a currently infrequent itemset may become frequent, and a currently frequent one may become infrequent in the future, it is important to keep as much frequency count information as possible. Given a fixed amount of memory, can you work out a good mechanism that may maintain high-quality approximation of itemset counting?

8.5 For the above approximate frequent itemset counting problem, it is interesting to incorporate the notion of *tilted time frame*. That is, we can put less weight on more remote itemsets when counting frequent itemsets. Design an efficient method that may obtain high-quality approximation of itemset frequency in data streams in this case.

8.6 A classification model may change dynamically along with the changes of training data streams. This is known as *concept drift*. Explain why decision tree induction may not be a suitable method for such dynamically changing data sets. Is naïve Bayesian a better method on such data sets? Comparing with the naïve Bayesian approach, is lazy evaluation (such as the k-nearest-neighbor approach) even better? Explain your reasoning.

8.7 The concept of microclustering has been popular for on-line maintenance of clustering information for data streams. By exploring the power of microclustering, design an effective *density-based* clustering method for clustering evolving data streams.

8.8 Suppose that a power station stores data regarding power consumption levels by time and by region, in addition to power usage information per customer in each region. Discuss how to solve the following problems in such a *time-series database*:

(a) Find similar power consumption curve fragments for a given region on Fridays.

(b) Every time a power consumption curve rises sharply, what may happen within the next 20 minutes?

(c) How can we find the most influential features that distinguish a stable power consumption region from an unstable one?

8.9 Regression is commonly used in trend analysis for *time-series data sets*. An item in a time-series database is usually associated with properties in multidimensional space. For example, an electric power consumer may be associated with consumer location, category, and time of usage (weekdays vs. weekends). In such a multidimensional space, it is often necessary to perform *regression analysis in an OLAP manner* (i.e., drilling and rolling along any dimension combinations that a user desires). Design an efficient mechanism so that regression analysis can be performed efficiently in multidimensional space.

8.10 Suppose that a restaurant chain would like to mine customers' consumption behavior relating to major sport events, such as "*Every time there is a major sport event on TV, the sales of Kentucky Fried Chicken will go up 20% one hour before the match.*"

(a) For this problem, there are multiple sequences (each corresponding to one restaurant in the chain). However, each sequence is long and contains multiple occurrences of a (sequential) pattern. Thus this problem is different from the setting of sequential pattern mining problem discussed in this chapter. Analyze what are the differences in the two problem definitions and how such differences may influence the development of mining algorithms.

(b) Develop a method for finding such patterns efficiently.

8.11 (**Implementation project**) The sequential pattern mining algorithm introduced by Srikant and Agrawal [SA96] finds sequential patterns among a set of sequences. Although there have been interesting follow-up studies, such as the development of the algorithms SPADE (Zaki [Zak01]), PrefixSpan (Pei, Han, Mortazavi-Asl, et al. [PHMA+01]), and CloSpan (Yan, Han, and Afshar [YHA03]), the basic definition of "sequential pattern" has not changed. However, suppose we would like to find frequently occurring subsequences (i.e., *sequential patterns*) *within one given sequence*, where, say, gaps are not allowed. (That is, we do not consider AG to be a subsequence of the sequence ATG.) For example, the string ATGCTCGAGCT contains a substring GCT with a support of 2. Derive an efficient algorithm that finds the complete set of subsequences satisfying a minimum support threshold. Explain how your algorithm works using a small example, and show some performance results for your implementation.

8.12 Suppose frequent subsequences have been mined from a sequence database, with a given (relative) minimum support, *min_sup*. The database can be updated in two cases: (i) adding new sequences (e.g., new customers buying items), and (ii) appending new subsequences to some existing sequences (e.g., existing customers buying new items). For *each case*, work out an efficient *incremental mining* method that derives the complete subsequences satisfying *min_sup*, without mining the whole sequence database from scratch.

8.13 Closed sequential patterns can be viewed as a lossless compression of a large set of sequential patterns. However, the set of closed sequential patterns may still be too large for effective analysis. There should be some mechanism for *lossy compression* that may further reduce the set of sequential patterns derived from a sequence database.

(a) Provide a good definition of lossy compression of sequential patterns, and reason why such a definition may lead to effective compression with minimal information loss (i.e., high compression quality).

(b) Develop an efficient method for such pattern compression.

(c) Develop an efficient method that mines such compressed patterns directly from a sequence database.

8.14 As discussed in Section 8.3.4, mining partial periodic patterns will require a user to specify the length of the period. This may burden the user and reduces the effectiveness of

mining. Propose a method that will *automatically mine the minimal period of a pattern* without requiring a predefined period. Moreover, extend the method to find *approximate periodicity* where the period will not need to be precise (i.e., it can fluctuate within a specified small range).

8.15 There are several major differences between *biological sequential patterns* and *transactional sequential patterns*. First, in transactional sequential patterns, the gaps between two events are usually nonessential. For example, the pattern "*purchasing a digital camera two months after purchasing a PC*" does not imply that the two purchases are consecutive. However, for biological sequences, gaps play an important role in patterns. Second, patterns in a transactional sequence are usually precise. However, a biological pattern can be quite imprecise, allowing insertions, deletions, and mutations. Discuss how the mining methodologies in these two domains are influenced by such differences.

8.16 BLAST is a typical heuristic alignment method for *pairwise sequence alignment*. It first locates high-scoring short stretches and then extends them to achieve suboptimal alignments. When the sequences to be aligned are really long, BLAST may run quite slowly. Propose and discuss some enhancements to improve the scalability of such a method.

8.17 The Viterbi algorithm uses the equality, $argmax_{\pi} P(\pi|x) = argmax_{\pi} P(x, \pi)$, in its search for the most probable path, π^*, through a *hidden Markov model* for a given sequence of symbols, x. Prove the equality.

8.18 (**Implementation project**) A *dishonest casino* uses a fair die most of the time. However, it switches to a loaded die with a probability of 0.05, and switches back to the fair die with a probability 0.10. The fair die has a probability of $\frac{1}{6}$ of rolling any number. The loaded die has $P(1) = P(2) = P(3) = P(4) = P(5) = 0.10$ and $P(6) = 0.50$.

(a) Draw a hidden Markov model for the dishonest casino problem using two states, Fair (F) and Loaded (L). Show all transition and emission probabilities.

(b) Suppose you pick up a die at random and roll a 6. What is the probability that the die is loaded, that is, find $P(6|D_L)$? What is the probability that it is fair, that is, find $P(6|D_F)$? What is the probability of rolling a 6 from the die you picked up? If you roll a sequence of 666, what is the probability that the die is loaded?

(c) Write a program that, given a sequence of rolls (e.g., $x = 5114362366\ldots$), predicts when the fair die was used and when the loaded die was used. (Hint: This is similar to detecting CpG islands and non-CPG islands in a given long sequence.) Use the Viterbi algorithm to get the most probable path through the model. Describe your implementation in report form, showing your code and some examples.

Bibliographic Notes

Stream data mining research has been active in recent years. Popular surveys on stream data systems and stream data processing include Babu and Widom [BW01], Babcock, Babu, Datar, et al. [BBD+02], Muthukrishnan [Mut03], and the tutorial by Garofalakis, Gehrke, and Rastogi [GGR02].

There have been extensive studies on stream data management and the processing of continuous queries in stream data. For a description of synopsis data structures for stream data, see Gibbons and Matias [GM98]. Vitter introduced the notion of reservoir sampling as a way to select an unbiased random sample of n elements without replacement from a larger ordered set of size N, where N is unknown [Vit85]. Stream query or aggregate processing methods have been proposed by Chandrasekaran and Franklin [CF02], Gehrke, Korn, and Srivastava [GKS01], Dobra, Garofalakis, Gehrke, and Rastogi [DGGR02], and Madden, Shah, Hellerstein, and Raman [MSHR02]. A one-pass summary method for processing approximate aggregate queries using wavelets was proposed by Gilbert, Kotidis, Muthukrishnan, and Strauss [GKMS01]. Statstream, a statistical method for the monitoring of thousands of data streams in real time, was developed by Zhu and Shasha [ZS02, SZ04].

There are also many stream data projects. Examples include Aurora by Zdonik, Cetintemel, Cherniack, et al. [ZCC+02], which is targeted toward stream monitoring applications; STREAM, developed at Stanford University by Babcock, Babu, Datar, et al., aims at developing a general-purpose Data Stream Management System (DSMS) [BBD+02]; and an early system called Tapestry by Terry, Goldberg, Nichols, and Oki [TGNO92], which used continuous queries for content-based filtering over an append-only database of email and bulletin board messages. A restricted subset of SQL was used as the query language in order to provide guarantees about efficient evaluation and append-only query results.

A multidimensional stream cube model was proposed by Chen, Dong, Han, et al. [CDH+02] in their study of multidimensional regression analysis of time-series data streams. MAIDS (Mining Alarming Incidents from Data Streams), a stream data mining system built on top of such a stream data cube, was developed by Cai, Clutter, Pape, et al. [CCP+04].

For mining frequent items and itemsets on stream data, Manku and Motwani proposed sticky sampling and lossy counting algorithms for approximate frequency counts over data streams [MM02]. Karp, Papadimitriou, and Shenker proposed a counting algorithm for finding frequent elements in data streams [KPS03]. Giannella, Han, Pei, et al. proposed a method for mining frequent patterns in data streams at multiple time granularities [GHP+04]. Metwally, Agrawal, and El Abbadi proposed a memory-efficient method for computing frequent and top-k elements in data streams [MAA05].

For stream data classification, Domingos and Hulten proposed the VFDT algorithm, based on their Hoeffding tree algorithm [DH00]. CVFDT, a later version of VFDT, was developed by Hulten, Spencer, and Domingos [HSD01] to handle concept drift in time-changing data streams. Wang, Fan, Yu, and Han proposed an ensemble classifier to mine concept-drifting data streams [WFYH03]. Aggarwal, Han, Wang, and Yu developed a k-nearest-neighbor-based method for classify evolving data streams [AHWY04b].

Several methods have been proposed for clustering data streams. The k-median-based STREAM algorithm was proposed by Guha, Mishra, Motwani, and O'Callaghan [GMMO00] and by O'Callaghan, Mishra, Meyerson, et al. [OMM+02]. Aggarwal, Han, Wang, and Yu proposed CluStream, a framework for clustering evolving data streams

[AHWY03], and HPStream, a framework for projected clustering of high-dimensional data streams [AHWY04a].

Statistical methods for time-series analysis have been proposed and studied extensively in statistics, such as in Chatfield [Cha03], Brockwell and Davis [BD02], and Shumway and Stoffer [SS05]. StatSoft's Electronic Textbook (*www.statsoft.com/textbook/stathome.html*) is a useful online resource that includes a discussion on time-series data analysis. The ARIMA forecasting method is described in Box, Jenkins, and Reinsel [BJR94]. Efficient similarity search in sequence databases was studied by Agrawal, Faloutsos, and Swami [AFS93]. A fast subsequence matching method in time-series databases was presented by Faloutsos, Ranganathan, and Manolopoulos [FRM94]. Agrawal, Lin, Sawhney, and Shim [ALSS95] developed a method for fast similarity search in the presence of noise, scaling, and translation in time-series databases. Language primitives for querying shapes of histories were proposed by Agrawal, Psaila, Wimmers, and Zait [APWZ95]. Other work on similarity-based search of time-series data includes Rafiei and Mendelzon [RM97], and Yi, Jagadish, and Faloutsos [YJF98]. Yi, Sidiropoulos, Johnson, Jagadish, et al. [YSJ+00] introduced a method for on-line mining for co-evolving time sequences. Chen, Dong, Han, et al. [CDH+02] proposed a multidimensional regression method for analysis of multidimensional time-series data. Shasha and Zhu present a state-of-the-art overview of the methods for high-performance discovery in time series [SZ04].

The problem of mining sequential patterns was first proposed by Agrawal and Srikant [AS95]. In the Apriori-based GSP algorithm, Srikant and Agrawal [SA96] generalized their earlier notion to include time constraints, a sliding time window, and user-defined taxonomies. Zaki [Zak01] developed a vertical-format-based sequential pattern mining method called SPADE, which is an extension of vertical-format-based frequent itemset mining methods, like Eclat and Charm [Zak98, ZH02]. PrefixSpan, a pattern growth approach to sequential pattern mining, and its predecessor, FreeSpan, were developed by Pei, Han, Mortazavi-Asl, et al. [HPMA+00, PHMA+01, PHMA+04]. The CloSpan algorithm for mining closed sequential patterns was proposed by Yan, Han, and Afshar [YHA03]. BIDE, a bidirectional search for mining frequent closed sequences, was developed by Wang and Han [WH04].

The studies of sequential pattern mining have been extended in several different ways. Mannila, Toivonen, and Verkamo [MTV97] consider frequent episodes in sequences, where episodes are essentially acyclic graphs of events whose edges specify the temporal before-and-after relationship but without timing-interval restrictions. Sequence pattern mining for plan failures was proposed in Zaki, Lesh, and Ogihara [ZLO98]. Garofalakis, Rastogi, and Shim [GRS99a] proposed the use of regular expressions as a flexible constraint specification tool that enables user-controlled focus to be incorporated into the sequential pattern mining process. The embedding of multidimensional, multilevel information into a transformed sequence database for sequential pattern mining was proposed by Pinto, Han, Pei, et al. [PHP+01]. Pei, Han, and Wang studied issues regarding constraint-based sequential pattern mining [PHW02]. CLUSEQ is a sequence clustering algorithm, developed by Yang and Wang [YW03]. An incremental sequential pattern mining algorithm, IncSpan, was proposed by Cheng, Yan, and Han [CYH04]. SeqIndex, efficient sequence indexing by frequent and

discriminative analysis of sequential patterns, was studied by Cheng, Yan, and Han [CYH05]. A method for parallel mining of closed sequential patterns was proposed by Cong, Han, and Padua [CHP05].

Data mining for periodicity analysis has been an interesting theme in data mining. Özden, Ramaswamy, and Silberschatz [ORS98] studied methods for mining periodic or cyclic association rules. Lu, Han, and Feng [LHF98] proposed intertransaction association rules, which are implication rules whose two sides are totally ordered episodes with timing-interval restrictions (on the events in the episodes and on the two sides). Bettini, Wang, and Jajodia [BWJ98] consider a generalization of intertransaction association rules. The notion of mining partial periodicity was first proposed by Han, Dong, and Yin, together with a max-subpattern hit set method [HDY99]. Ma and Hellerstein [MH01a] proposed a method for mining partially periodic event patterns with unknown periods. Yang, Wang, and Yu studied mining asynchronous periodic patterns in time-series data [YWY03].

Methods for the analysis of biological sequences have been introduced in many textbooks, such as Waterman [Wat95], Setubal and Meidanis [SM97], Durbin, Eddy, Krogh, and Mitchison [DEKM98], Baldi and Brunak [BB01], Krane and Raymer [KR03], Jones and Pevzner [JP04], and Baxevanis and Ouellette [BO04]. BLAST was developed by Altschul, Gish, Miller, et al. [AGM+90]. Information about BLAST can be found at the NCBI Web site *www.ncbi.nlm.nih.gov/BLAST/*. For a systematic introduction of the BLAST algorithms and usages, see the book "BLAST" by Korf, Yandell, and Bedell [KYB03].

For an introduction to Markov chains and hidden Markov models from a biological sequence perspective, see Durbin, Eddy, Krogh, and Mitchison [DEKM98] and Jones and Pevzner [JP04]. A general introduction can be found in Rabiner [Rab89]. Eddy and Krogh have each respectively headed the development of software packages for hidden Markov models for protein sequence analysis, namely HMMER (pronounced "hammer," available at *http://hmmer.wustl.edu/*) and SAM (*www.cse.ucsc.edu/research/ compbio/sam.html*).