

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 $I = \{I_1, I_2, \dots, I_p\}$ 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 $\langle e_1 e_2 e_3 \dots e_l \rangle$, 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 $(x_1 x_2 \dots x_q)$, 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 $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \dots, a_n \subseteq b_{j_n}$. 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, $support_S(\alpha) = |\{ \langle SID, s \rangle \mid (\langle SID, s \rangle \in S) \wedge (\alpha \sqsubseteq s) \}|$. 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 $\langle a(bc)d(f) \rangle$ 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 *horizontal data format* (where the data are represented as $\langle \text{sequence_ID} : \text{sequence_of_itemsets} \rangle$, 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 $\langle \text{itemset} : (\text{sequence_ID}, \text{event_ID}) \rangle$). 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.

⁶Closed 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 (Generalize 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 “[old:item][new: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, $C_2 = \{\langle aa \rangle, \langle ab \rangle, \dots, \langle af \rangle, \langle ba \rangle, \langle bb \rangle, \dots, \langle ff \rangle, \langle (ab) \rangle, \langle (ac) \rangle, \dots, \langle (ef) \rangle\}$.

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 $\langle \text{itemset} : (\text{sequence_ID}, \text{event_ID}) \rangle$. 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 i th itemset (or event) in a sequence is i . Note that an itemset can occur in more than one sequence. The set of $(\text{sequence_ID}, \text{event_ID})$ 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), share the same sequence identifier, and 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 horizontal 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 IDs*, 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*

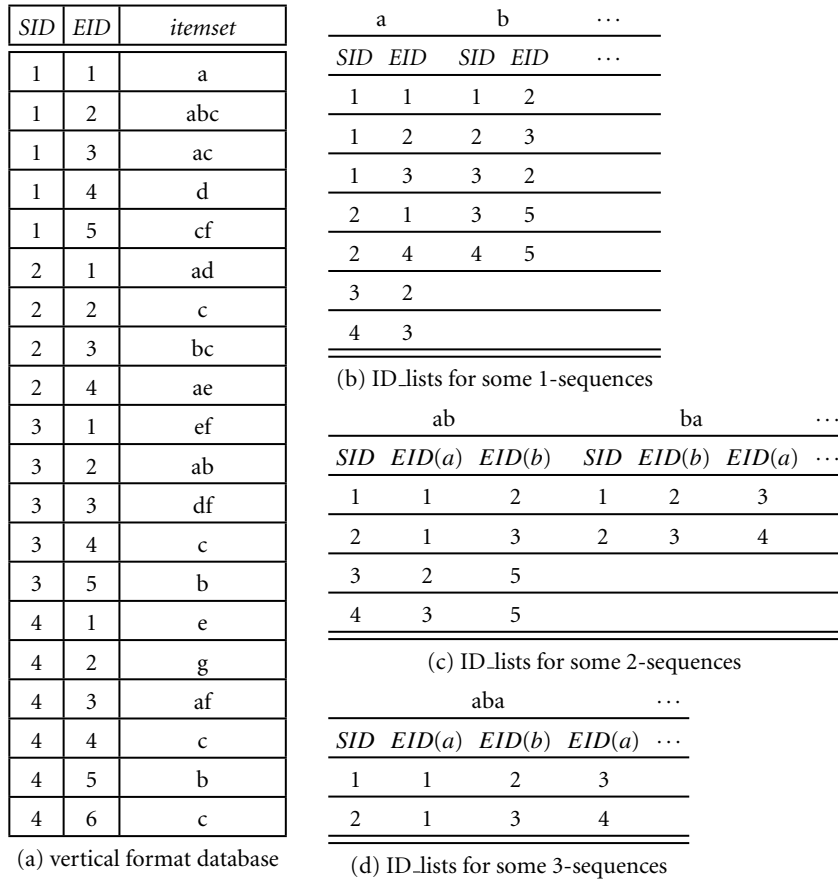


Figure 8.6 The SPADE mining process: (a) vertical format database; (b) to (d) show fragments of the ID_lists for 1-sequences, 2-sequences, and 3-sequences, respectively.

tree, or FP-tree. The FP-tree is used to generation 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 $\alpha = \langle e_1 e_2 \dots e_n \rangle$ (where each e_i corresponds to a frequent event in a sequence database, S), a sequence $\beta = \langle e'_1 e'_2 \dots e'_m \rangle$ ($m \leq n$) 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 $\gamma = \langle e''_m e_{m+1} \dots e_n \rangle$ 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 $\{\langle x_1 \rangle, \langle x_2 \rangle, \dots, \langle x_n \rangle\}$ 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 $\{\beta_1, \beta_2, \dots, \beta_m\}$ 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 $\text{min_sup} = 2$, 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 " $\langle \text{pattern} \rangle : \text{count}$ " represents the pattern and its associated support count.

⁷If e_m'' is not empty, the suffix is also denoted as $\langle (.)\text{items in } e_m'' e_{m+1} \dots e_n \rangle$.

Table 8.2 Projected databases and sequential patterns

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

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 $a : 2$, $b : 4$, ${}_b : 2$, $c : 4$, $d : 2$, 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 $\langle ab \rangle$, ..., 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.

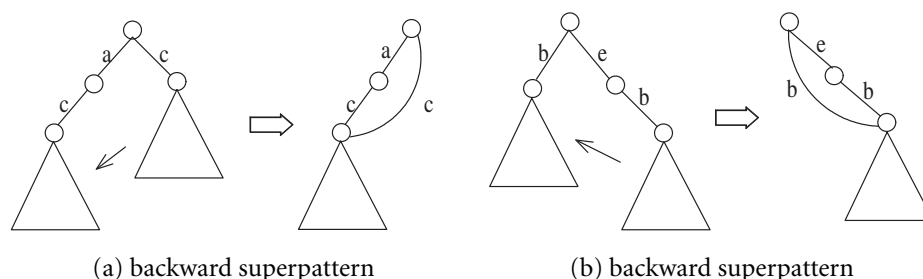


Figure 8.7 A backward subpattern and a backward superpattern.

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.

⁸In $S|_{\alpha}$, 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, SD400, 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 $a_{i-1}a_i a_{i+1}$. 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 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 *Suffix-Span* 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 in 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

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*:
- Find similar power consumption curve fragments for a given region on Fridays.
 - Every time a power consumption curve rises sharply, what may happen within the next 20 minutes?
 - 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, a 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* 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, $\operatorname{argmax}_{\pi} P(\pi|x) = \operatorname{argmax}_{\pi} P(x, \pi)$, 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 $P(6) = 0.50$.
- Draw a hidden Markov model for the dishonest casino problem using two states, Fair (F) and Loaded (L). Show all transition and emission probabilities.
 - 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?
 - Write a program that, given a sequence of rolls (e.g., $x = 5114362366\dots$), 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].

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, was 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 Baxeavanis and Ouellette [BO04]. 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).