# Chapter 12
# Process Mining in the Large

Process mining provides the technology to leverage the ever-increasing amounts of event data in modern organizations and societies. Despite the growing capabilities of modern computing infrastructures, event logs may be too large or too complex to be handled using conventional approaches. This chapter focuses on handling "Big Event Data" and relates process mining to Big Data technologies. Moreover, it is shown that process mining problems can be decomposed in two ways, *case-based decomposition* and *activity-based decomposition*. Many of the analysis techniques described can be made scalable using such decompositions. Also other performance-related topics such as streaming process mining and process cubes are discussed. The chapter shows that the lion's share of process mining techniques can be "applied in the large" by using the right infrastructure and approach.

## 12.1 Big Event Data

Some of the process mining tools described in Chap. 11 can discover process models for logs with billions of events. However, performance highly depends on the characteristics of the event log (e.g., number of distinct activities and redundancy) and the questions asked (e.g., conformance checking is often more time consuming than discovery). Moreover, for some applications event logs may be even larger or results need to be provided instantly.

In Chap. 1, we listed the "four V's of Big Data": Volume, Velocity, Variety, and Veracity (Fig. 1.4). The term "Internet of Events", introduced in Sect. 1.1, refers to the growing availability of event data. These data are omnipresent and an enabler for process mining. This chapter will focus on event data and the first two V's. The first 'V' (Volume) refers to the size of some data sets, in our case event logs. We will discuss various *decomposition* and *distribution* strategies to turn large process mining problems into multiple smaller ones. The second 'V' (Velocity) refers to the speed of the incoming events that need to be processed. It may be impossible or undesirable to store all data. Therefore, we will also introduce the topic of *streaming* process mining.



W. van der Aalst, *Process Mining*, DOI 10.1007/978-3-662-49851-4_12

Big Data is not limited to process-related data. However, Big Data infrastructures enable us to collect, store, and process huge event logs (see Sect. 2.5.9). Process mining tools can exploit such infrastructures. Therefore, we describe current trends in hardware and software (Sect. 12.1.2), before describing the characteristic features of event logs (Sect. 12.1.3). However, first we briefly discuss the possibilities and risks when going from sampled "small data" to "all data".

### *12.1.1 N = All*

In the past, conclusions were often based on human judgment or analysis of sample data. Either data were not available, unreliable, or it was impossible to process all data. In many businesses, we now witness a change from collecting *some data* to collecting *all data* [100]: "$N = \text{All}$" where $N$ refers to the sample size. As described in Sect. 1.1, the digital universe and physical universe are becoming more aligned. Money has become a predominantly digital entity. Queries on the availability of products are answered based on data in some database rather than a visit to the warehouse. The direct coupling between data and reality combined with our improved abilities to store and process data forms the playground of data science as described in Chap. 1.

Sampling was needed in the "analog era" characterized by information *scarcity*. Due to sampling error and sampling bias, it may be risky to extrapolate conclusions from sample data. Moreover, the granularity of analysis using sampled data is often too coarse making it impossible to draw conclusions for smaller subcategories and submarkets. Hence, the concept of sampling makes no sense if *all* data are available and we have the computing power to analyze all events. Consider, for example, conformance checking. Why just check the conformance of a few cases if we can check all cases and detect all deviations? Clearly, "$N = \text{All}$" requires a new way of thinking. For example, auditors and accountants may be afraid of uncovering all deviations. Also the work of marketers and social scientists is changing: large-scale data analysis is replacing sampling and questionnaires.

Having all data ($N = \text{All}$) may also create problems:

- Hardware, software and analysis techniques need to be able to cope with the associated volumes.
- Overfitting the data may lead to "bogus conclusions" (cf. Bonferroni's principle).

This chapter will focus on the first problem. However, to illustrate the second problem we consider the following example inspired by a similar example in [114].

Suppose some Dutch government agency is searching for terrorists by examining hotel visits of all of its 18 million citizens ($18 \times 10^6$). The hypothesis is that terrorists meet multiple times at some hotel to plan an attack. Hence, the agency looks for suspicious "events" $\{p_1, p_2\} \dagger \{d_1, d_2\}$ where persons $p_1$ and $p_2$ meet on days $d_1$ and $d_2$ in some hotel. How many of such suspicious events will the agency find if the behavior of people is completely random? To estimate this number, we make some

additional assumptions. On average, Dutch people go to a hotel every 100 days and a hotel can accommodate 100 people at the same time. We further assume that there are $\frac{18\times10^6}{100\times100} = 1800$ Dutch hotels where potential terrorists can meet.

The probability that two persons ($p_1$ and $p_2$) visit a hotel on a given day $d$ is $\frac{1}{100} \times \frac{1}{100} = 10^{-4}$. The probability that $p_1$ and $p_2$ visit the *same* hotel on day $d$ is $10^{-4} \times \frac{1}{1800} = 5.55 \times 10^{-8}$. The probability that $p_1$ and $p_2$ visit the same hotel on two different days $d_1$ and $d_2$ is $(5.55 \times 10^{-8})^2 = 3.086 \times 10^{-15}$. Note that different hotels may be used on both days. Hence, the probability of suspicious event $\{p_1, p_2\} \dagger \{d_1, d_2\}$ is $3.086 \times 10^{-15}$.

How many candidate events are there? Assume an observation period of 1000 days. Hence, there are $\binom{1000}{2} = \frac{1000\times(1000-1)}{2} = 499,500$ combinations of days $d_1$ and $d_2$. Note that the order of days does not matter, but the days need to be different. There are $\binom{18\times10^6}{2} = \frac{18\times10^6\times(18\times10^6-1)}{2} = 1.62 \times 10^{14}$ combinations of persons $p_1$ and $p_2$. Again the ordering of $p_1$ and $p_2$ does not matter, but $p_1 \neq p_2$. Hence, there are $499,500 \times 1.62 \times 10^{14} = 8.09 \times 10^{19}$ candidate events $\{p_1, p_2\} \dagger \{d_1, d_2\}$.

The expected number of suspicious events is equal to the product of the number of candidate events $\{p_1, p_2\} \dagger \{d_1, d_2\}$ and the probability of such events (assuming independence), $8.09 \times 10^{19} \times 3.086 \times 10^{-15} = 249,749$. Hence, there will be around a quarter million observed suspicious events $\{p_1, p_2\} \dagger \{d_1, d_2\}$ in a 1000 day period!

Suppose that there are only a handful of terrorists and related meetings in hotels. The Dutch government agency will need to investigate around a quarter million suspicious events involving hundreds of thousands innocent citizens. This is an illustration of Bonferroni's principle.

**Bonferroni's principle**

In statistics, the *Bonferroni's correction* is a method (named after the Italian mathematician Carlo Emilio Bonferroni) to compensate for the problem of multiple comparisons. Normally, one rejects the null hypothesis if the likelihood of the observed data under the null hypothesis is low. If we test many hypotheses, we also increase the likelihood of a rare event. Hence, the likelihood of incorrectly rejecting a null hypothesis increases. If the desired significance level for the whole collection of null hypotheses is $\alpha$, then the Bonferroni correction suggests that one should test each individual hypothesis at a significance level of $\frac{\alpha}{k}$ where $k$ is the number of null hypotheses. For example, if $\alpha = 0.05$ and $k = 20$, then $\frac{\alpha}{k} = 0.0025$ is the required significance level for testing the individual hypotheses.

*Bonferroni's principle* aims to avoid treating random observations as if they are real and significant [114]. To apply the principle, compute the number of observations of some phenomena one is interested in under the assumption that things occur at random. If this number is significantly larger than the real number of instances one expects, then most of the findings will be false positives.

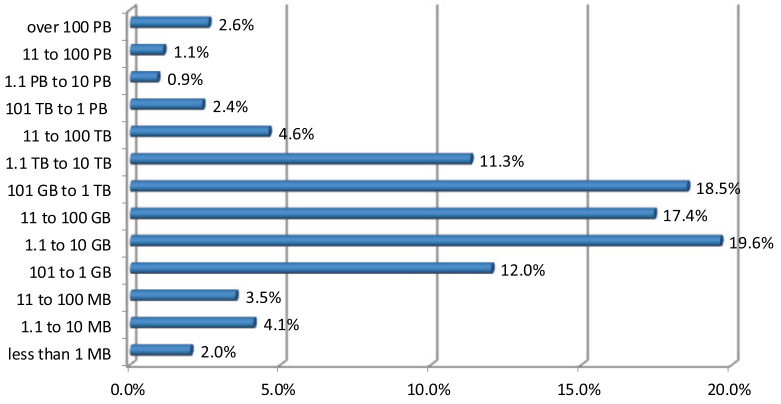



**Fig. 12.1** Results of the KDnuggets poll (August 2015): "What was the largest data set you analyzed/data mined?" (1 Gigabyte (GB) equals 1000 MB, 1 Terabyte (TB) equals 1000 GB, 1 Petabyte (PB) equals 1000 TB)

Bonferroni's principle is highly relevant for large data sets with many instances. The number of rare events of a certain type will increase as the volume of data grows even if there is no pattern and behavior is completely random.

If we are looking for terrorists and expect only a few terrorists to be active, then any hypothesis that points to hundreds of thousands of citizens behaving randomly is pointless. Bonferroni's principle states that one can only find terrorists by looking for events that are so rare that they are unlikely to occur in random data.

Figure 12.1 is another illustration of the challenges posed by today's data sets. In a recent KDnuggets poll, over 78% of respondents reported to have analyzed data sets of more than 1 Gigabyte and over 22% of respondents reported to have analyzed data sets of more than 1 Terabyte. Before discussing process-mining specific ways of dealing with large event logs, we first discuss some general technological developments relevant for mining massive data sets.

### *12.1.2 Hardware and Software Developments*

In 1965, Gordon Moore predicted that the number of transistors would double every year until 1975 [104]. In 1975, Moore revised his prediction to a doubling of components every two years. This prediction turned out to be remarkably accurate, as shown in Fig. 12.2. The diagonal line shows *Moore's law* predicting that the number of transistors on a chip is $1000 \times 2^{(y-1970)/2}$. Here, 1970 is used as basis.

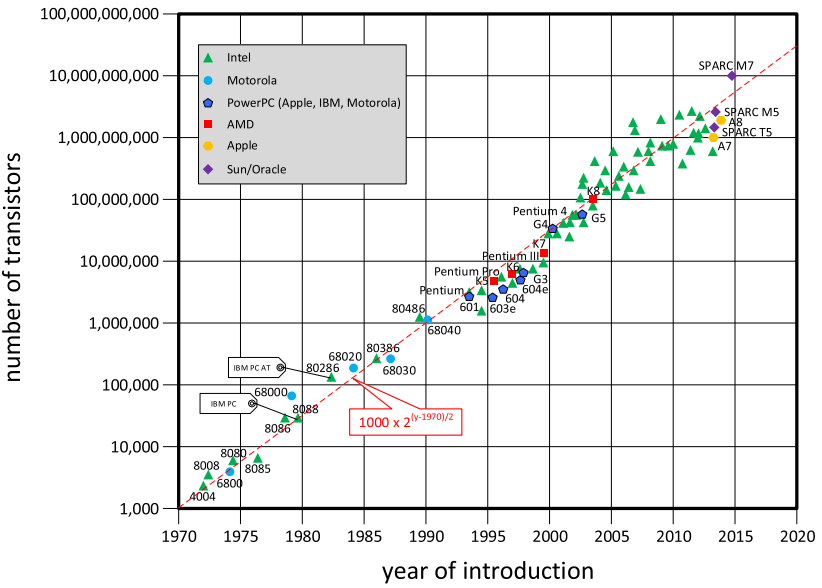



**Fig. 12.2** Moore's law : The number of transistors on a chip has been growing exponentially since the early 1970s

For 2016 this formula predicts $1000 \times 2^{(2016-1970)/2} = 8.4 \times 10^9$ transistors which seems consistent with reality, e.g., the Intel Xeon E7-8890 v3 processor released in May 2015 has a total of 5.6 billion transistors.

The exponential growth is not limited to the number of transistors per chip. Performance of CPUs has been growing at a similar pace. Although clock speeds leveled off around 2004, multicore architectures continued boosting performance. Similar developments can be seen in memory and storage (e.g., size of hard disks and flash drives), graphics (e.g., pixels on a screen), and networking (e.g., the capacity of wireless networks). Also costs have been decreasing exponentially, e.g., the price of storing one Gigabyte of data or making a particular computation.

It is difficult to grasp the incredible developments in IT as reflected by Moore's law. In 1970, it took 1.5 hours to travel by train from Eindhoven to Amsterdam. If transportation would have followed the same developments, then the train trip would now take only $(1.5 \times 60 \times 60)/2^{(2016-1970)/2} = 0.00064$ seconds (i.e., less than a millisecond). Flying from Amsterdam to New York would only take 3.4 milliseconds ($(8 \times 60 \times 60)/2^{(2016-1970)/2} = 0.0034$ seconds). In 1970, a car would have consumed approximately 5000 liter of fuel to drive around the world (40,075 km). If fuel efficiency would have followed similar developments, less than 1 milliliter of fuel would be needed now ($5000/2^{(2016-1970)/2} = 0.00059$ liter). These examples illustrate that our ability to process data has developed at a spectacular pace. Although this development has been ongoing since 1970, data analysis has now reached a "tipping point" thus explaining the current "Big Data" hype.

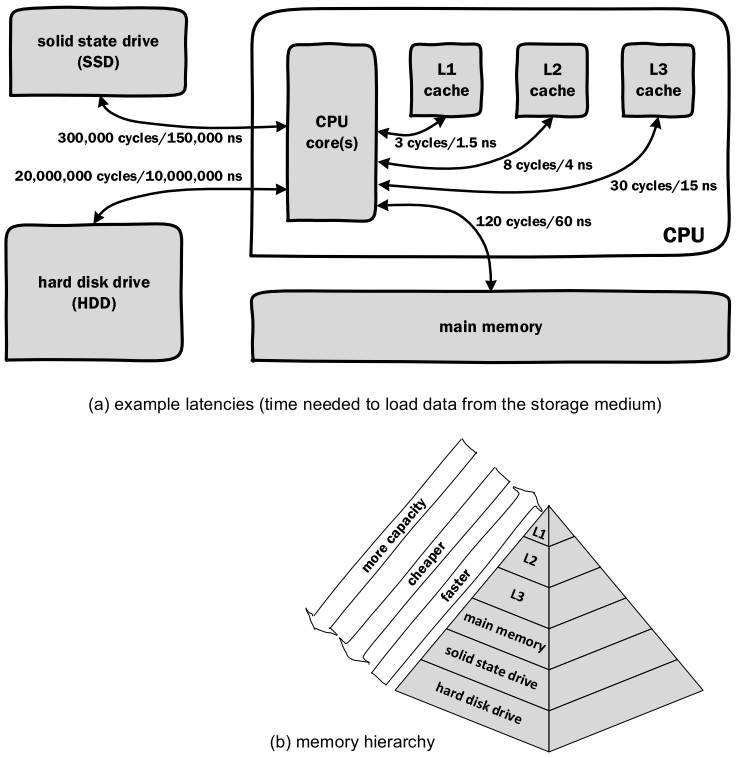



**Fig. 12.3** Conceptual view of a typical computer's memory hierarchy going from slow, spacious and cheap (HDD) to fast, small and expensive (L1 cache)

Let us not take a look inside a computer (see Fig. 12.3) and focus on the different types of storage and their latencies. The hard disk drive (HDD) is the cheapest, but also slowest form of storage. HDDs offer large amounts of storage. The solid state drive (SSD) is more expensive and has less capacity, but is considerably faster. Main memory is even more expensive, but orders of magnitude faster than drives. Fastest and most expensive are the different caches (L1, L2, and L3).

Figure 12.3 shows some typical latencies which come into play when the CPU needs to access data. Latency is the time delay experienced by the computer to load data from the storage medium until it is available in the CPU register. The L1 cache latency in Fig. 12.3 is 1.5 nanoseconds corresponding to 3 cycles of the processor (1 nanosecond is $10^{-9}$ second). Loading data from the L2 and L3 cache takes a bit more time, but is still faster than loading data from main memory. The main memory latency in Fig. 12.3 is 60 nanoseconds (120 cycles). The latencies for the different types of drives are much larger: 150,000 nanoseconds for SSD and 10,000,000 nanoseconds for HDD. Note that the numbers in Fig. 12.3 are just examples. They are used to exemplify the spectacular differences between the different types of storage media. The prices per byte of storage are inverse proportional to the latencies.

To illustrate the differences in Fig. 12.3, we assume that loading data corresponds to fetching a cup of coffee. Getting a data element from the L1 cache corresponds to getting a Nespresso coffee from the kitchen. For the analogy, let us assume that 1.5 nanoseconds corresponds to 7.5 meters (distance from desk to kitchen). Getting a data element from main memory then corresponds to getting a coffee from the Starbucks around the corner (300 meters equals 60 ns). Getting a data element from SSD then corresponds to flying from Eindhoven to Aosta (Italy) to get a really good cappuccino (750 kilometers equals 150,000 ns). Getting a data element from hard disk then corresponds to flying to Colombia, Ethiopia, and Kenya to hand-pick the best beans, process them in Amsterdam, take the beans to Rome and ask a barista to make a ristretto from the ground coffee beans (50,000 kilometers equals 10 milliseconds).

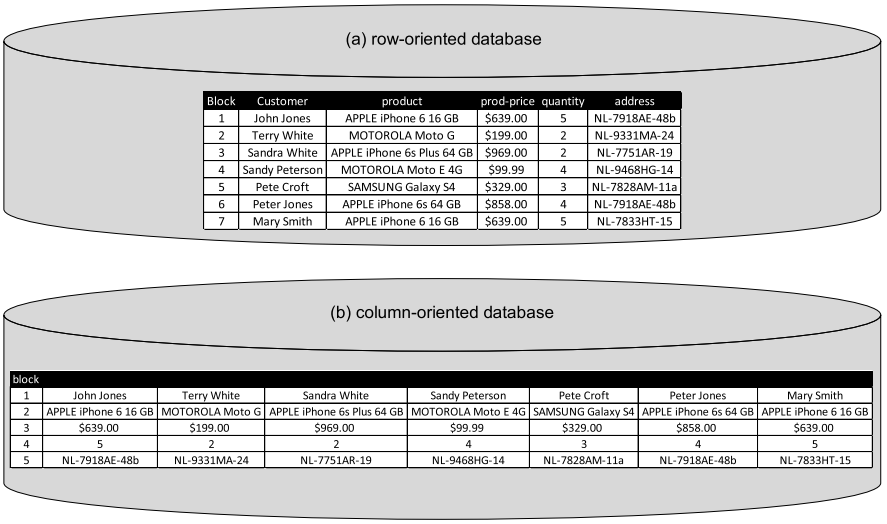
When implementing a process mining technique, it is important to be aware of these differences in latency. Whether an event is on disk or in memory makes a huge difference.

Figures 12.2 and 12.3 provide the context for a discussion on the rapidly changing IT landscape for Big Data analytics. A lot has changed in database technology over the last decade. Edgar Codd defined the relational model in 1970 [32]. For many years Relational Database Management Systems (RDBMS) and the Structured Query language (SQL) were the de facto standard. Database systems like Oracle V2, IBM's DB2, dBase, Sybase, Ingres, Informix, Access, Postgres, and MySQL released in the period 1975–2005 were all relational. However, due to the challenges of extremely large data sets (at the "scale of Google"), the dominance of relational databases and SQL ended. Since 2005 many new non-relational database systems have been released and the emergence of big data technologies like Hadoop caused a revolution in the way IT systems are organized. We sketch some of these developments in the remainder.

#### 12.1.2.1 In-Memory Databases and Analytics

The numbers in Fig. 12.3 show that getting data from a hard disk is like traveling around the world to get a cup of coffee. An *in-memory database management system* primarily relies on main memory for data storage. This requires computer systems with a large main memory (e.g., a terabyte of main memory). Since the size of main memory is still limited compared to disk storage, these in-memory database management systems compress the data using a variety of compression mechanisms. When the limit of available main memory is reached, larger chunks of data are unloaded from main memory based on the characteristics of the application and are reloaded into main memory when they are required again.

Since data is stored in volatile memory, all stored information is lost when the device loses power or is reset. Durability, one of the standard ACID (atomicity, consistency, isolation, durability) properties, does not hold without special provisions. Solutions are snapshot files, checkpoint images, and transaction logging to recover an in-memory database after system failure.




| Block | Customer | product | prod-price | quantity | address |
|---|---|---|---|---|---|
| 1 | John Jones | APPLE iPhone 6 16 GB | $639.00 | 5 | NL-7918AE-48b |
| 2 | Terry White | MOTOROLA Moto G | $199.00 | 2 | NL-9331MA-24 |
| 3 | Sandra White | APPLE iPhone 6s Plus 64 GB | $969.00 | 2 | NL-7751AR-19 |
| 4 | Sandy Peterson | MOTOROLA Moto E 4G | $99.99 | 4 | NL-9468HG-14 |
| 5 | Pete Croft | SAMSUNG Galaxy S4 | $329.00 | 3 | NL-7828AM-11a |
| 6 | Peter Jones | APPLE iPhone 6s 64 GB | $858.00 | 4 | NL-7918AE-48b |
| 7 | Mary Smith | APPLE iPhone 6 16 GB | $639.00 | 5 | NL-7833HT-15 |

| block | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | John Jones | Terry White | Sandra White | Sandy Peterson | Pete Croft | Peter Jones | Mary Smith |
| 2 | APPLE iPhone 6 16 GB | MOTOROLA Moto G | APPLE iPhone 6s Plus 64 GB | MOTOROLA Moto E 4G | SAMSUNG Galaxy S4 | APPLE iPhone 6s 64 GB | APPLE iPhone 6 16 GB |
| 3 | $639.00 | $199.00 | $969.00 | $99.99 | $329.00 | $858.00 | $639.00 |
| 4 | 5 | 2 | 2 | 4 | 3 | 4 | 5 |
| 5 | NL-7918AE-48b | NL-9331MA-24 | NL-7751AR-19 | NL-9468HG-14 | NL-7828AM-11a | NL-7918AE-48b | NL-7833HT-15 |

**Fig. 12.4** Row-oriented versus column-oriented databases

Next to using an in-memory platform or database management system, the analytics tool itself can also manage data directly. The challenge is to keep the "hot data" in main memory and the "cold data" on disk.

OLAP applications can benefit from in-memory analytics. Users want to "slice and dice" data to look at the data from different angles (see Sect. 12.4). The effect of an OLAP operation needs to be instantaneous to fully support the analyst.

Although the costs of in-memory technology are improving, it is still quite expensive and hence only economic in situations where responsiveness is imperative.

#### 12.1.2.2 Columnar Databases

In a traditional relational database, data is organized in horizontal rows and vertical columns. The rows correspond to instances (e.g., sales transactions) and the columns refer to attributes of these instances. This is very natural and corresponds to the way we plot data in spreadsheets or on paper. However, when analyzing data it is rare to analyze all the columns in a single row. During analysis we tend to do operations on all the rows in a single column, e.g., taking the sum or computing the average. In a *column store*, columns tend to be stored together rather than rows.

Figure 12.4 sketches the idea of organizing data by columns (b) rather than rows (a). *Columnar databases* use the row orientation and store all elements related to a particular attribute together in one block, e.g., all quantities, prices, product names, etc. are stored together. Aggregate operations such as sum and average can be done faster, since all the data needed is grouped together. Also compression can be improved, since there tends to be more repetition in the column-oriented blocks. For example, customer and product names are shared by multiple sales transactions whereas within a sales transaction the attributes will be different.

SAP HANA is a well-known example of an application server that includes an in-memory, column-oriented database management system. HANA employs a mix of columnar and row-based storage. MonetDB is a column-oriented database management system released under an open-source license. Other column-oriented systems include Apache Cassandra, HBase, Accumulo, Druid, and Vertica.

Column-oriented systems are part of a larger class of *NoSQL database management systems*. A NoSQL database provides a mechanism for the storage and retrieval of data that is no longer based on the traditional tabular relations from relational databases. Next to column-oriented systems, there are other classes of NoSQL systems such as document-oriented databases, graph databases, object databases, and key–value databases. Most of these systems were developed to cope with specific scalability challenges. For the ranking of web pages, we may need to perform iterated matrix–vector multiplications with billions of rows and columns. For searches in social networks, we need to analyze graphs with billions of nodes and edges. Traditional database systems cannot handle such problems. In fact, the NoSQL systems illustrate the end of the "one size fits all" approach promoted by relational databases.

#### 12.1.2.3 Large-Scale Distributed File Systems

When datasets are small, analysis can be done using a single computer having its own memory, disk and CPU. However, if datasets get larger, parallel processing is needed using a network of parallel computers. In the past, large scientific computations were done using special-purpose parallel computers using specialized hardware. However, the need for cheap large-scale data processing triggered a trend towards the use of thousands of compute nodes operating more or less independently and using commodity hardware.

Scalability at reasonable costs is key for achieving a competitive advantage. For example, Google was forced to create its own hardware and software stack to realize a scalable commercial solution. In this context, Google developed a distributed file system, *Google File System* (GFS) [58]. GFS supports massive numbers of commodity servers with directly attached storage to be exposed as a single logical file system.

In such a distributed file system, files can be enormous (e.g., terabytes), but are rarely updated. Typically data is appended rather than changed. The files are split into chunks that are replicated at two or more compute nodes.

Compute nodes are typically stored in racks. The nodes are connected by fast networks for inter-rack and intra-rack communication. The replicated chunks of data are stored in different racks to be able to handle rack failure.

Replication is essential when the number of compute nodes increases. Recall that commodity hardware is used for compute nodes to lower costs. Assume that the Mean Time Between Failures (MTBF) for a given compute node is three years. This implies that the MTBF for a distributed system composed of 1000 nodes is approximately one day. Therefore, fault-tolerance needs to be built into the system: the replicated chunks of data are used to automatically recover from hardware failures.

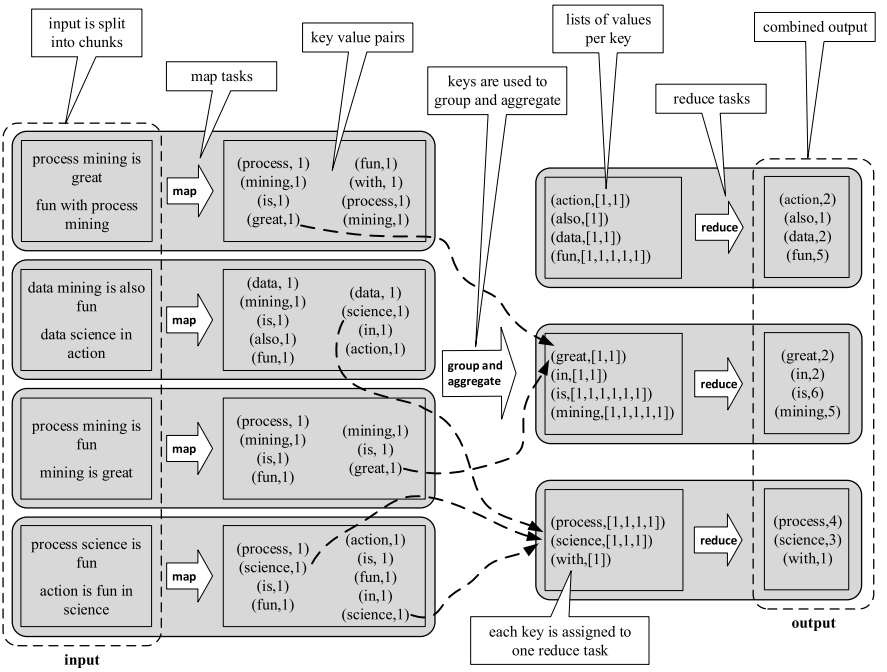



**Fig. 12.5** Counting words using MapReduce

The *Hadoop Distributed File System* (HDFS) is an open-source distributed file system inspired by GFS. HDFS is the core of *Apache Hadoop*, an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. Hadoop is based on the assumption that hardware failures are common and should be automatically handled by the framework.

Apache Hadoop evolved into an extensive ecosystem with many components that go far beyond this book. For example, *Hadoop YARN* is a resource-management platform responsible for managing and scheduling computing resources. *Hadoop MapReduce* is also part of the base framework and provides an implementation of the *MapReduce* programming model for large-scale data processing. MapReduce was originally developed within Google [44].

MapReduce is a programming model that allows complex problems to be broken up into simple map functions that can be executed concurrently together with reduce functions that combine the outputs from each parallel stream.

**MapReduce**
*MapReduce* is a style of distributed computing where the user only needs to provide two functions, called *Map* and *Reduce*. The system takes care of the

rest: managing the parallel execution, coordinating the concurrent tasks, and handling failures. There are two types of tasks, *Map* tasks and *Reduce* tasks.

Each *Map* task gets one or more chunks of data from the distributed file systems (e.g., GFS or HDFS). The *Map* task produces a sequence of *key–value pairs*, e.g., the output of task $i$ could be of the form $\langle(k_{i,1}, v_{i,1}), (k_{i,2}, v_{i,2}), \ldots, (k_{i,n}, v_{i,n})\rangle$. The way the key value pairs are produced from the input data is specified by the *Map* function.

The key–value pairs produced by all *Map* tasks are redistributed by the system as preparation for the *Reduce* tasks. The keys are divided among the *Reduce* tasks, e.g., using hashing. A particular key $k_j$ is assigned to one of the *Reduce* tasks and all values for this key are put into a list. Key and list of values (i.e., $(k_j, \langle v_{j,1}, v_{j,2}, \ldots, v_{j,m}\rangle)$) are input for the *Reduce* task. Function *Reduce* specifies what the output is based on input $(k_j, \langle v_{j,1}, v_{j,2}, \ldots, v_{j,m}\rangle)$. For example, the sum or average could be computed over the list of values for a particular key.

Figure 12.5 shows an example. The input consists of 8 sentences split into four chunks each containing two sentences. Each *Map* task emits one key–value pair per word in its input chunk. In this simple example, the value is always 1. For example, the key value pair (*process*, 1) refers to the first word, (*mining*, 1) refers to the second word, etc. Most keys appear in multiple *Map* tasks. The emitted key–value pairs are grouped using the keys and aggregated into lists. The infrastructure ensures that this is done efficiently. Each key is assigned to a particular *Reduce* task. In the example, there are three *Reduce* tasks each responsible for a few keys. Note that (*great*, $\langle 1, 1\rangle$) is based on key–value pairs emitted by the first and third *Map* tasks. (*science*, $\langle 1, 1, 1\rangle$) in the last *Reduce* task is based on three key–value pairs: one emitted by the second *Map* task and two emitted by the last *Map* task. The *Reduce* function takes as input pairs consisting of a key and its list of associated values and combines these values in some aggregated result. Here, the sum is taken. For example, (*science*, $\langle 1, 1, 1\rangle$) is reduced to (*science*, 3) indicating that the word "science" appears three times in the whole text.

Often the *Reduce* function is commutative and associative. In this case, part of the aggregation can be moved to the *Map* function. For example, the last *Map* task in Fig. 12.5 could have emitted a single key–value pair (*science*, 2) rather than two key–value pairs (*science*, 1). Typically, there are more *Map* tasks than *Reduce* tasks to limit communication.

*Map* and *Reduce* tasks can be done concurrently using possibly thousands of compute nodes. MapReduce is particularly useful in situations where even linear or quadratic algorithms are not fast enough. If an event log is so large that just scanning the log takes too long, the problem needs to be distributed to solve smaller problems in parallel. In the ideal scenario MapReduce scales linearly with the number of compute nodes, i.e., if the number of compute nodes is doubled, the problem is solved in half the time.

MapReduce was first implemented within Google [44]. Later it became a core ingredient of the Apache Hadoop ecosystem. See [114] for an illustration of the broad range of problems that can be tackled using MapReduce.

GFS and HDFS are used in conjunction with commodity hardware distributed over hundreds or thousands of nodes. Since the bandwidth for communication is limited, it is desirable to push computation to the data. Moreover, programming distributed systems is notoriously hard. This explains the relevance of the MapReduce programming model. The user only needs to write "map" and "reduce" functions, and the rest is left to the framework that handles work distribution and faults. Many analysis questions can be translated to "map" and "reduce" functions. In [114], several examples are given. Later, in Sect. 12.2, we will demonstrate that also process discovery can benefit from the MapReduce programming model.

Next to Apache Hadoop, a variety of alternative computing frameworks have been proposed, sometimes also installed on top of or alongside Hadoop. An example is Apache Spark which provides multi-stage in-memory primitives and is particularly suited for machine learning algorithms.

### *12.1.3 Characterizing Event Logs*

This book focuses on a particular type of data, *event data*. The complexity and size of event logs are determined by different factors. This section defines the key characteristics of logs.

Consider, for example, event log $L_1 = [\langle a, b, c, f\rangle^2, \langle a, c, b, f\rangle^2, \langle a, d, e, f\rangle^2]$ in Fig. 12.6. This event log is composed of six cases. The average trace length of a case is four. In total there are 24 events in $L_1$. There are three distinct traces, each of which appears twice. There are six distinct activities ($a$–$f$).

Event log $L_2$ is twice the size of $L_1$ in terms of cases and events. However, the number of distinct traces and distinct activities did not change.

Event log $L_3$ is also twice the size of $L_1$ in terms of events. However, the average trace length of a case has doubled. Event logs $L_1$, $L_2$, and $L_3$ have the same number of distinct traces and distinct activities.

The size of event log $L_4$ is the same as the size of $L_1$ (both in terms of cases and events). However, the number of distinct activities has doubled.

The size of event log $L_5$ is also the same as the size of $L_1$ (both in terms of cases and events). However, now all cases are distinct and the traces vary in length.

Figure 12.6 shows that the complexity of an event log does not just depend on the number of events. A perfectly fitting process model learned for $L_4$ will have twice the number of activities compared to the model learned for $L_1$. The variety in $L_5$ seems higher than in $L_1$ and $L_2$ which are of the same size. Therefore, we require multiple *event log metrics* to adequately characterize the input of a process mining task.

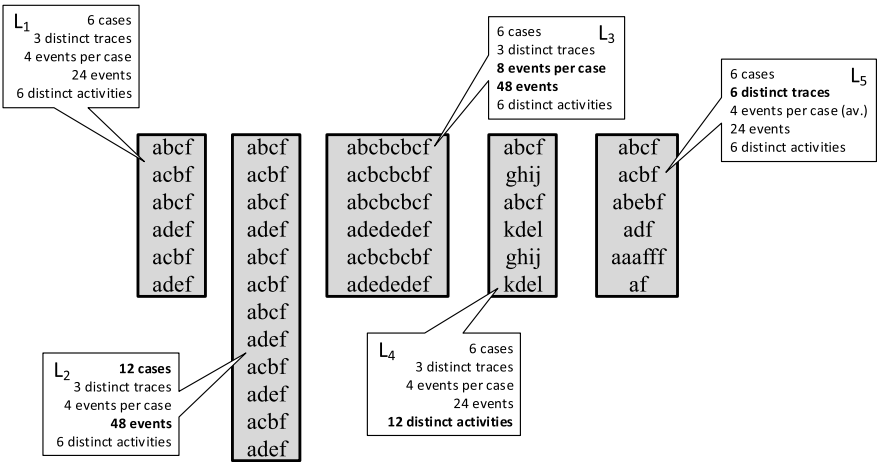



**Fig. 12.6** Illustrating some of the key characteristics of an event log

Let us first focus on *simple event logs* without event attributes other than the activity name, i.e., $L \in \mathbb{B}(\mathscr{A}^*)$ is a multi-set of traces. Let us recall some notations introduced before. These will be used to define several event log metrics. In Definition 7.5, we denoted $G(L) = (A_L, \mapsto_L, A_L^{start}, A_L^{end})$ as the *directly-follows graph* of $L$ with $A_L = \{a \in \sigma \mid \sigma \in L\}$ as the set of observed activities, $\mapsto_L = \{(a, b) \in A \times A \mid a >_L b\}$ as the directly follows relation, $A_L^{start} = \{a \in A \mid \exists_{\sigma \in L}\ a = first(\sigma)\}$ as the set of start activities, and $A_L^{end} = \{a \in A \mid \exists_{\sigma \in L}\ a = last(\sigma)\}$ as the set of end activities. Recall that $\partial_{set}(\sigma) = \{a_1, a_2, \ldots, a_n\}$ for any $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$.

**Definition 12.1** (Event log metrics) Let $L \in \mathbb{B}(\mathscr{A}^*)$ be an event log and $G(L) = (A_L, \mapsto_L, A_L^{start}, A_L^{end})$. We define the following *event log metrics* for $L$:

- Number of cases,

$$\#_{cases}(L) = |L|$$

- Average trace length of cases,

$$av_{tloc}(L) = \frac{\sum_{\sigma \in L}\ L(\sigma) \times |\sigma|}{|L|}$$

  Alternatively one can compute the minimal trace length, the maximal trace length, and the standard deviation of trace lengths.
- Number of distinct activities,

$$\#_{acts}(L) = |A_L|$$

- Average number of distinct activities per case,

$$av_{dapc}(L) = \frac{\sum_{\sigma \in L} L(\sigma) \times |\partial_{set}(\sigma)|}{|L|}$$

  Alternatively one can compute the minimal or maximal number of distinct activities or the standard deviation.
- Average set-based non-overlap of traces,

$$av_{sbnot}(L) = 1 - \frac{\sum_{\sigma_1,\sigma_2 \in L} L(\sigma_1) \times L(\sigma_2) \times \frac{|\partial_{set}(\sigma_1) \cap \partial_{set}(\sigma_2)|}{|\partial_{set}(\sigma_1) \cup \partial_{set}(\sigma_2)|}}{|L|^2}$$

  $av_{sbnot}(L)$ compares pairs of traces in terms of overlap. If all traces refer to the same set of activities, then $av_{sbot}(L) = 0$. If traces tend to refer to disjoint sets of activities, then $av_{sbot}(L)$ will be closer to 1. Other distance measures could be used taking the cardinalities into account (Euclidean or Jaccard distance).
- Number of distinct cases,

$$\#_{dc}(L) = \left|\{\sigma \in L\}\right|$$

- Number of events,

$$\#_{events}(L) = \#_{cases}(L) \times av_{tloc}(L) = \sum_{\sigma \in L} L(\sigma) \times |\sigma|$$

- Number of direct successions,

$$\#_{ds}(L) = |\mapsto_L|$$

  $\#_{ds}(L)$ counts the number of arcs in the directly-follows graph.
- Number of start activities,

$$\#_{sa}(L) = \left|A_L^{start}\right|$$

- Number of end activities,

$$\#_{ea}(L) = \left|A_L^{end}\right|$$

Consider, for example, $L_1 = [\langle a, b, c, f\rangle^2, \langle a, c, b, f\rangle^2, \langle a, d, e, f\rangle^2]$ in Fig. 12.6. Here $\#_{cases}(L_1) = 6$ (number of cases), $av_{tloc}(L_1) = 4$ (average trace length of cases), $\#_{acts}(L_1) = 6$ (number of distinct activities), $av_{dapc}(L_1) = 4$ (average number of distinct activities per case), $av_{sbnot}(L_1) = 0.296$ (average set-based non-overlap of traces), $\#_{dc}(L_1) = 3$ (number of distinct cases), $\#_{events}(L_1) = 24$ (number of events), $\#_{ds}(L_1) = 9$ (number of direct successions), $\#_{sa}(L_1) = 1$ (number of start activities), and $\#_{ea}(L_1) = 1$ (number of end activities). Table 12.1 also shows the event log metrics for the other event logs in Fig. 12.6.

Event logs are considered more challenging if the values for these metrics are higher. $L_2$ is most challenging (of the five toy logs) in terms of the number of cases ($\#_{cases}(L_2) = 12$). $L_3$ is most challenging in terms of the average trace

**Table 12.1** Event log metrics for the five event logs in Fig. 12.6

| Event log metric | | $L_1$ | $L_2$ | $L_3$ | $L_4$ | $L_5$ |
|---|---|---|---|---|---|---|
| Number of cases | $\#_{cases}(L_i)$ | 6 | 12 | 6 | 6 | 6 |
| Average trace length of cases | $av_{tloc}(L_i)$ | 4 | 4 | 8 | 4 | 4 |
| Number of distinct activities | $\#_{acts}(L_i)$ | 6 | 6 | 6 | 12 | 6 |
| Average number of dist. act. per case | $av_{dapc}(L_i)$ | 4 | 4 | 4 | 4 | 3.166 |
| Average set-based non-overlap of traces | $av_{sbnot}(L_i)$ | 0.296 | 0.296 | 0.296 | 0.667 | 0.348 |
| Number of distinct cases | $\#_{dc}(L_i)$ | 3 | 3 | 3 | 3 | 6 |
| Number of events | $\#_{events}(L_i)$ | 24 | 48 | 48 | 24 | 24 |
| Number of direct successions | $\#_{ds}(L_i)$ | 9 | 9 | 10 | 9 | 13 |
| Number of start activities | $\#_{sa}(L_i)$ | 1 | 1 | 1 | 3 | 1 |
| Number of end activities | $\#_{ea}(L_i)$ | 1 | 1 | 1 | 3 | 1 |

length ($av_{tloc}(L_3) = 8$). $L_4$ is most challenging in terms of the number of distinct activities ($\#_{acts}(L_4) = 12$), the least overlap of activities in pairwise comparison of traces ($av_{sbnot}(L_4) = 0.667$), and the number of start and end activities ($\#_{sa}(L_4) = \#_{ea}(L_4) = 3$). $L_5$ is most challenging in terms of the number of distinct cases ($\#_{dc}(L_5) = 6$) and the number of direct successions ($\#_{ds}(L_5) = 13$).

Events may have any number of attributes (see Definition 5.1). Some of the attributes are standard: timestamp, resource, and transaction type. Other attributes may be domain-specific, e.g., blood pressure, sales region, age, voltage, and grade. Attributes may be sparse of not. For example, every event is expected to have a timestamp. However, there may also be attributes like blood pressure that are attached to only a small fraction of all events. Such an attribute is sparse as only few events in the event log have it. Next to the control-flow oriented metrics in Definition 12.1, one can define additional event log metrics such as:

- The number of distinct attributes in an event log $\#_{ndael}(L)$ and
- The average number of attributes per event $av_{nape}(L)$.

The event logs in Fig. 12.6 are, of course, not representative for real-life event logs. In this chapter, we are particularly interested in challenging event logs, e.g., event logs with tens of thousands of cases ($\#_{cases}(L) \gg 10,000$), millions of events ($\#_{events}(L) \gg 1,000,000$), and hundreds of different activities ($\#_{acts}(L) \gg 100$).

For the $\alpha$-algorithm, the heuristic miner, the fuzzy miner, and the inductive miner based on the directly-follows graph (IMD and IMFD algorithms), a single pass through the entire event log suffices. Even if such algorithms are exponential in the number of different activities, the pass through the event log typically remains most time consuming. Note that often the number of distinct activities ($\#_{acts}(L)$) is orders of magnitude smaller than the number of cases ($\#_{cases}(L)$) or the number of events ($\#_{events}(L)$). In later sections, we will show that it is possible to decompose computation based on structures like the directly-follows graph.

Conformance checking based on alignments and discovery based on language-based regions require solving optimization problems (e.g., an ILP problem). These

problems are more challenging and cannot be decomposed as easily. Here the number of distinct activities ($\#_{acts}(L)$) and the average trace length of cases ($av_{tloc}(L)$) are important.

As mentioned in Chap. 11, some of today's process mining tools can handle logs with billions of events and millions of cases. However, this only holds for single-pass algorithms based on counting and does *not* apply to algorithms doing some form of optimization or state exploration (e.g., conformance checking or region-based discovery). Moreover, scalability depends on many factors. An event log with a lot of redundancy ($\#_{cases}(L) \gg \#_{dc}(L)$) is easier to analyze than a log where most cases are unique. Some systems have problems with large logs having many attributes per event (e.g., $av_{nape}(L) \gg 100$) even when these attributes are not used for analysis.

In the remainder, we will focus on process discovery and conformance checking. These are most challenging from a performance point of view and form the backbone for subsequent analyses. Note that after computing alignments, it is easy to extend the model with performance information and other aspects learned from the event log.

## 12.2 Case-Based Decomposition

Process mining is motivated by the availability of event data. However, as event logs become larger (say gigabytes or terabytes), performance becomes a concern. The only way to handle larger applications while ensuring acceptable response times, is to distribute analysis over a network of computers (e.g., multicore systems, grids, and clouds). This ultimately requires splitting the event log: each compute node becomes responsible for a part of the event data. We consider two types of decomposition, *case-based decomposition* and *activity-based decomposition*.

Case-based decomposition (called "vertical partitioning of the event log" in [141]), distributes events based on the case they belong to. Event logs may be composed of millions of cases. These can be distributed over the compute nodes such that each case is assigned to one node. Hence, each compute node works on a subset of the whole log after which the results need to be merged.

Activity-based decomposition (also called "horizontal partitioning of the event log" [141]), distributes events based on the activities they refer to. The number of unique activities is typically much smaller than the number of cases or events. However, many process mining algorithms are non-linear in the number of activities. Creating subproblems working on smaller groups of activities may therefore provide super-linear speedups. Hence, activity-based decomposition can also be used on a single computer solving the subproblems in sequence. For activity-based decomposition cases are split up in smaller traces working on subsets of activities. In principle, each compute node needs to consider all cases. Typically, the activity sets are partly overlapping as will be explained in Sect. 12.3.

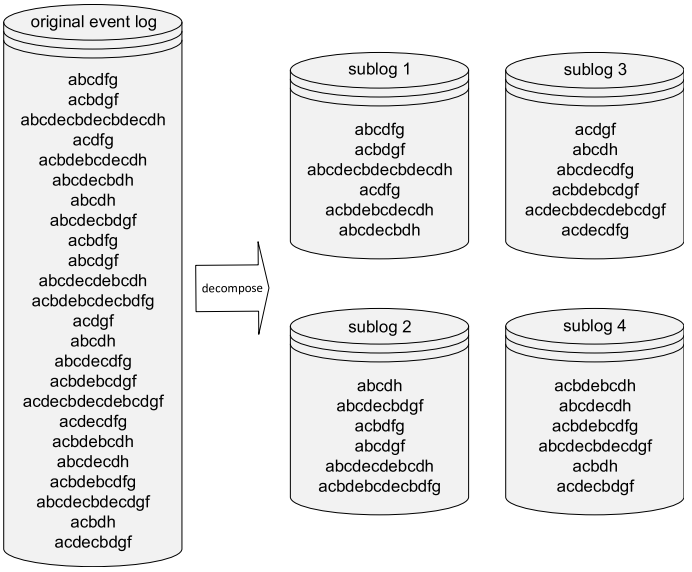



**Fig. 12.7** Case-based decomposition of a small event log: The 24 cases are split into four groups of 6 cases

In this section, we focus on case-based decomposition. As shown in Fig. 12.7, the basic idea is very simple. The cases are simply distributed over the $n$ sublogs. Each of the $n$ sublogs can then be analyzed in parallel.

### *12.2.1 Conformance Checking Using Case-Based Decomposition*

In Chap. 8, we introduced various conformance checking techniques. Token-based replay (Sect. 8.2) and alignment-based conformance checking (Sect. 8.3) are most interesting because they directly relate cases in the event log to the model. Alignment-based conformance checking provides better diagnostics (more detailed, accurate, and easy to understand) than token-based replay, but is more time consuming. To compute optimal alignments, optimization problems need to be solved which can become intractable if model and log are huge.

Case-based decomposition is a straightforward way to achieve a linear speedup for both token-based replay and alignment-based conformance checking. If there are $n$ compute nodes each handling $\frac{1}{n}$ of the cases, then conformance can be checked in $\frac{t_s}{n}$ time where $t_s$ is the time to check conformance using a single node. Such a linear speedup can only be achieved if the overhead is negligible compared to the time to do the actual conformance computations.

Consider the WF-net in Fig. 12.8. Suppose we would like to check conformance of the event log of Fig. 12.7 with respect to this process model. We can sequentially

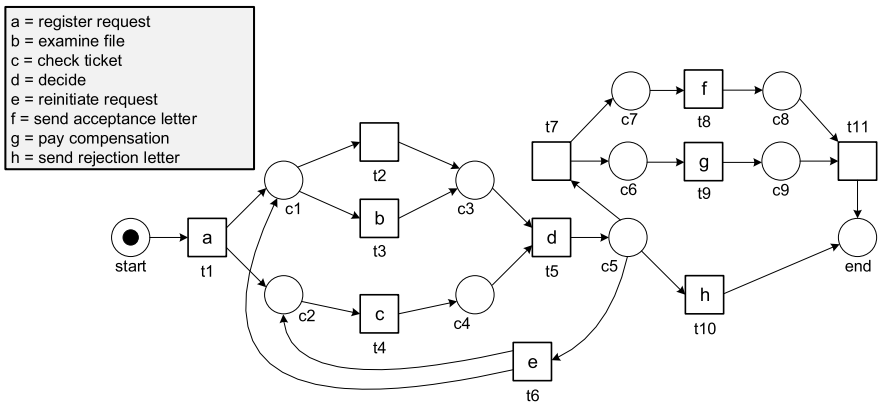



**Fig. 12.8** Process model used to illustrate different decomposition approaches

check all 24 cases on a single compute node or we can split the event log into four parts as shown in Fig. 12.7. This means that $4 \times 6$ cases are checked in parallel. It is easy to combine the diagnostics for the four sublogs. For token-based replay (Sect. 8.2), we can simply *add up the consumed, produced, missing, and remaining tokens* (Sect. 8.2). For alignment-based conformance checking, we can *add up all misalignment costs* (Sect. 8.3). Also metrics such as the percentage of (non-) perfectly fitting cases, moves in model only, and moves in log only can be easily computed based on the intermediate results for the four sublogs.

When splitting the cases, we may exploit redundancy in the event log. By putting similar cases in the same sublog, further speedups are possible. If two cases have the same trace, then conformance only needs to be checked for one of them. Moreover, sophisticated alignment-based conformance checking techniques cache intermediate results and therefore handle similar cases faster. Obviously, there is a trade-off since it may take time to cluster similar cases. Also a simple hashing function can be used to group cases.

### *12.2.2 Process Discovery Using Case-Based Decomposition*

In Chap. 6, we introduced the $\alpha$-algorithm. More advanced process discovery algorithms were introduced in Chap. 7: heuristic mining (Sect. 7.2), genetic process mining (Sect. 7.3), region-based mining (Sect. 7.4), and inductive mining (Sect. 7.5) These algorithms have very different performance characteristics.

*Generic process mining* (Sect. 7.3) can be trivially parallelized in a number of ways. We can decompose the event log as shown in Fig. 12.7. However, we can also replicate the whole event log at each compute node. Per node there are separate generations of individuals (candidate models). Selection (including fitness computations) and reproduction (e.g., crossover and mutation) are done per node. Periodically, individuals are exchanged between compute nodes. By sharing the best

individuals between the parallel nodes, the evolutionary process typically converges faster.

*Region-based mining techniques* (Sect. 7.4) are difficult to parallelize. If region-based techniques are applied to sublogs, there is no easy way to merge the resulting models. For example, a place that can be added according to one sublog may not be feasible according to another sublog.

The *inductive mining techniques* (Sect. 7.5) that actually split the event log (i.e., IM, IMF, and IMC) are also difficult to decompose. The different sublogs can be analyzed separately, however, finding the initial exclusive-choice, sequence, parallel, or redo-loop cut is most time consuming. Only after splitting the event logs based on activities, the work can be distributed. This does not correspond to the notion of case-based decomposition. Learning a model per sublog is possible, but merging these models is problematic. Most likely the models disagree to certain ordering relations. Fortunately, inductive mining based on the directly-follows graph (without log splitting) can be decomposed, as will be explained next.

In the remainder, we focus on the $\alpha$-algorithm, the heuristic miner, the fuzzy miner, and the directly-follows based inductive miners (IMD, IMFD, and IMCD). These have in common that they are based on the *directly-follows graph* or a similar internal aggregate structure. Key for the $\alpha$-algorithm is relation $>_L$ that contains all pairs of activities in a "directly follows" relation, and the sets $T_I$ and $T_O$ (cf. Definition 6.4). The dependency graph used by the heuristic miner is similar to the "directly follows" relation, but incorporates frequencies. Miners such as IMD, IMFD, and IMCD start from a directly-follows graph $G(L) = (A_L, \mapsto_L, A_L^{start}, A_L^{end})$ (see Definition 7.5). Despite subtle differences the basis is the same—*counting local patterns in cases*.

*Any process discovery technique that is based on counting local patterns in individual cases can benefit from case-based decomposition.* Consider Fig. 12.7 and the directly follows relation between $b$ and $c$. In the first sublog, $b$ is four times directly followed by $c$, in the second sublog six times, in the third sublog four times, and in the fourth sublog also four times. Hence, the frequency of the directly follows relation between $b$ and $c$ is $4 + 6 + 4 + 4 = 18$. The same can be done with start and end activities. Frequencies of local patterns can be counted per case and therefore simply added per sublog. These frequencies can be aggregated over all sublogs, making decomposition easy. Note that case-based decomposition does not influence the outcome: The same process model is discovered.

Discovery approaches that count patterns in cases (e.g., the frequency of the directly follows relation) can exploit the *MapReduce* programming model introduced in Sect. 12.1.2. Consider the example in Fig. 12.9. An event log consisting of 16 cases is split into four chunks. The *Map* function emits a key–value pair for every direct succession. To keep track of initial and final activities, dummy starts and ends have been added denoted by $\triangleright$ (start) and $\square$ (end). The first trace $\langle a, b, c, f \rangle$ handled by the first *Map* task results in five key–value pairs: $(\triangleright a, 1)$, $(ab, 1)$, $(bc, 1)$, $(cf, 1)$, and $(f \square, 1)$. The MapReduce framework takes care of the grouping and aggregation of these key–value pairs. The first *Reduce* task is responsible for four keys ($\triangleright a$, $ab$, $ac$, and $ad$), and simply adds up the values in the lists. The output of this

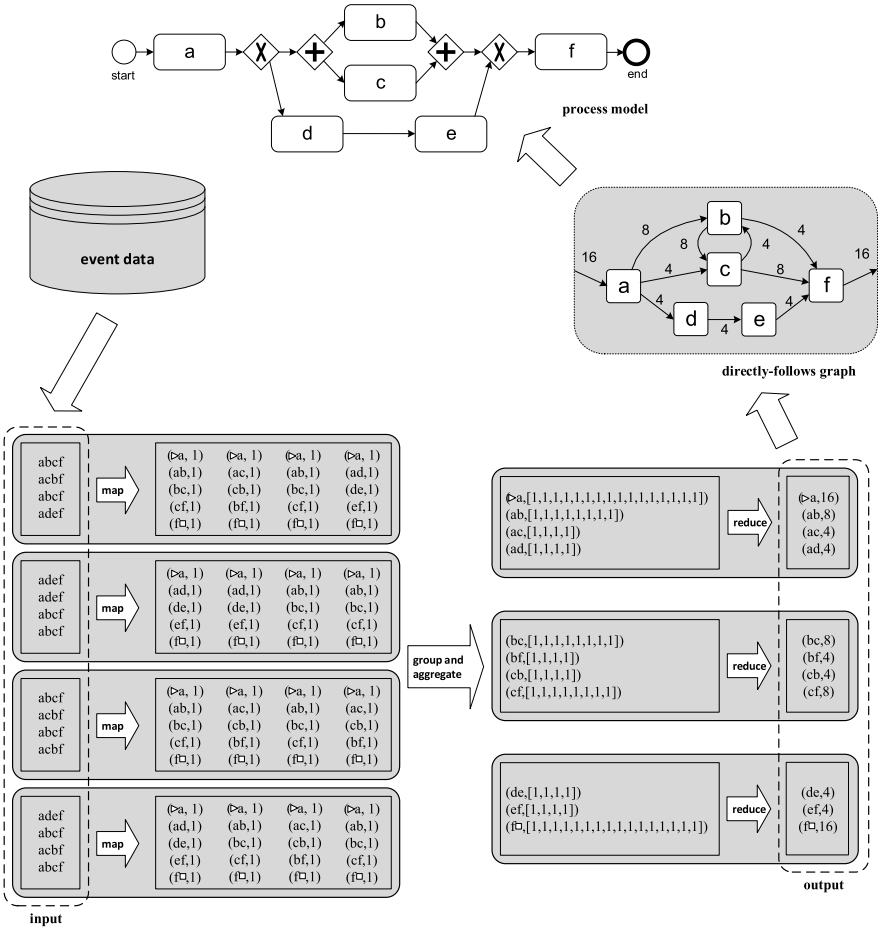



**Fig. 12.9** Computing the directly-follows graph using MapReduce

*Reduce* task is $(\triangleright a, 16)$, $(ab, 8)$, $(ac, 4)$, and $(ad, 4)$. The combined output of all *Reduce* tasks provides all information needed to produce the directly-follows graph shown in Fig. 12.9 (including frequencies). Based on this graph we can apply the $\alpha$-algorithm and the directly-follows based inductive miner (IMD). Other process discovery algorithms like the heuristic miner, the fuzzy miner, and other variants of the $\alpha$-algorithm and inductive miner use similar inputs. Figure 12.9 shows the process model returned by both the $\alpha$-algorithm and the IMD inductive miner (using the directly-follows graph).

The example logs in Figs. 12.7 and 12.9 are too small to really illustrate case-based decomposition. For small examples the overhead caused by decomposition will only slow down analysis. One needs to imagine that event logs contain millions

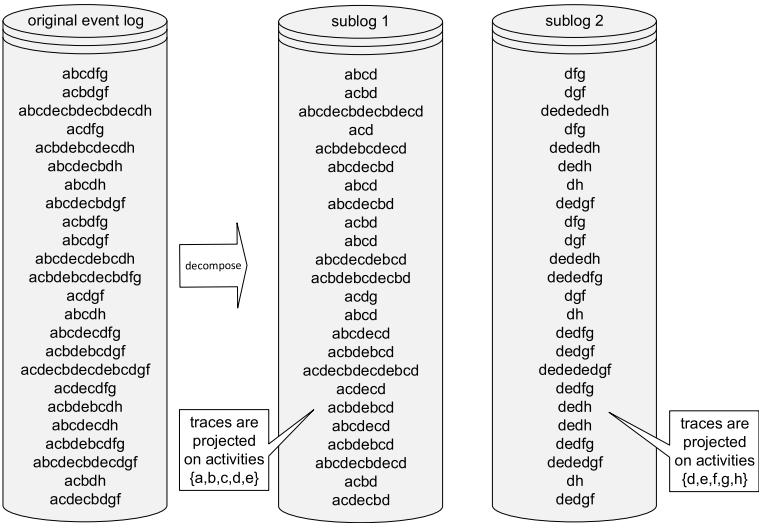



**Fig. 12.10** Activity-based decomposition of a small event log: The two sublogs are projections of the original event log

or even billions of cases referring to hundreds or thousands of different activities. In such cases, it is vital to be able to distribute analysis. Figure 12.9 shows that it is fairly easy to formulate process discovery in terms of the MapReduce programming model and use a Hadoop-like infrastructure.

## 12.3 Activity-Based Decomposition

Case-based decomposition is most suitable when using techniques that ultimately count local patterns, e.g., to construct a directly-follows or dependency graph. However, discovery techniques that do not use such patterns (e.g., region-based discovery or inductive mining based on log-splitting) cannot use case-based decomposition. Conformance techniques can also use case-based decomposition, but the complexity is in the number of activities and the average trace length. Therefore, a linear speedup may not be enough. *If computing an alignment for a single trace takes too long or is even infeasible, case-based decomposition is not a viable solution.*

For situations where case-based decomposition is not good enough, one can consider *activity-based decomposition* as an alternative decomposition approach. Figure 12.10 sketches the idea. Sublogs are created by projecting cases onto subsets of activities. Each sublog is responsible for a particular subset of activities. The first sublog in Fig. 12.10 is responsible for subset $\{a, b, c, d, e\}$ and the second sublog is responsible for subset $\{d, e, f, g, h\}$. Note that the two activity sets are overlapping. Strictly speaking, this is not a requirement. However, later we will see that some overlap is desirable.

We use the projection operator introduced in Chap. 5 to explain activity-based decomposition. $\sigma \uparrow X$ is the projection of $\sigma$ onto some subset $X \subseteq A$, e.g., $\langle a, b, c, a, b, c, d\rangle \uparrow \{a, b\} = \langle a, b, a, b\rangle$. In Fig. 12.10, the set of all activities is $A = \{a, b, c, d, e, f, g, h\}$. The two sublogs are based on subsets $A_1 = \{a, b, c, d, e\}$ and $A_2 = \{d, e, f, g, h\}$. Consider, for example, the first trace in the original log $\sigma = \langle a, b, c, d, f, g\rangle$. This corresponds to trace $\sigma \uparrow A_1 = \langle a, b, c, d\rangle$ in the first sublog and trace $\sigma \uparrow A_2 = \langle d, f, g\rangle$ in the second sublog.

We can also apply the projection operator to event logs. $L = [\langle a, b, c, d, f, g\rangle, \langle a, c, b, d, g, f\rangle, \ldots, \langle a, c, d, e, c, b, d, g, f\rangle]$ is the original event log. $L_1 = L \uparrow A_1 = [\langle a, b, c, d\rangle, \langle a, c, b, d\rangle, \ldots, \langle a, c, d, e, c, b, d\rangle]$ is first sublog, and $L_2 = L \uparrow A_2 = [\langle d, f, g\rangle, \langle d, g, f\rangle, \ldots, \langle d, e, d, g, f\rangle]$ is the second sublog.

In the general setting, we can have any event log $L$ with activities $A$ decomposed in $k$ subsets $A_1, A_2, \ldots, A_k$ such that $A = A_1 \cup A_2 \cup \cdots \cup A_k$. The corresponding $k$ sublogs are $L \uparrow A_1, L \uparrow A_2, \ldots, L \uparrow A_k$. All sublogs have the same number of cases as the original log, but traces are much shorter if the subsets are relatively small. Process mining algorithms can be applied to each of the $k$ sublogs, after which the partial results need to be merged. This is often surprisingly easy, as is shown next.

### 12.3.1 Conformance Checking Using Activity-Based Decomposition

In case of decomposed conformance checking, we have a model $N$ and log $L$ with activities $A$. Although the decomposition approach does not depend on Petri nets (see [142]), let us assume that $N$ is a Petri net with an initial and final marking. We allow for duplicate and silent activities, e.g., transitions with a $\tau$ label or multiple transitions with the same label.

Assume we would like to check the conformance of $L = [\langle a, b, c, d, f, g\rangle, \langle a, c, b, d, g, f\rangle, \ldots, \langle a, c, d, e, c, b, d, g, f\rangle]$ with respect to the Petri net in Fig. 12.8. *How to decompose the event log?* We need to identify subsets of activities and decompose model and log based on this. The basic idea is that we must cut the Petri net in Fig. 12.8 into fragments such that we only cut through transitions that are visible and unique. We cannot cut through places or silent transitions. We can also not cut through transitions having a label appearing at multiple places. In fact, such transitions should also not become separated. The exact requirements are given in [144].

Using the approach just described we can split the Petri net $N$ in two parts as shown in Fig. 12.11. This way we find activity sets $A_1 = \{a, b, c, d, e\}$ and $A_2 = \{d, e, f, g, h\}$. Next we create two sublogs based on these activity sets, $L_1 = L \uparrow A_1$ and $L_2 = L \uparrow A_2$. These are the two sublogs shown in Fig. 12.10. The fragments also form two smaller process models. $N_1$ is the Petri net on the left-hand side of the dotted line (including the transitions labeled $d$ and $e$). $N_2$ is the Petri net on the right-hand side of the dotted line (also including the boundary transitions). $N_1$ has

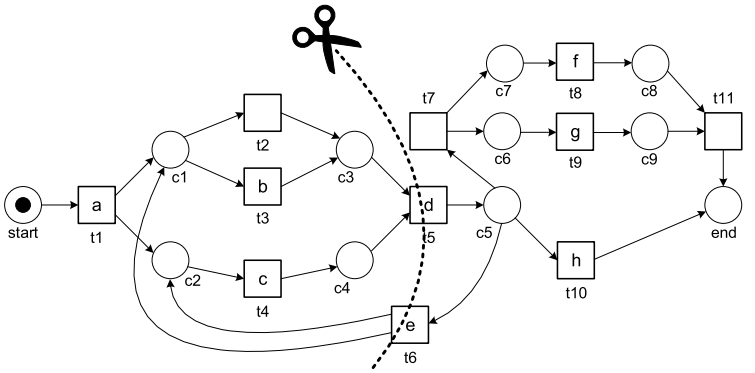



**Fig. 12.11** By cutting through transitions that are visible and unique, we find activity sets $A_1 = \{a, b, c, d, e\}$ and $A_2 = \{d, e, f, g, h\}$

initial marking [*start*] and the empty final marking ([ ]). $N_2$ has initial marking [ ] and final marking [*end*].

Next we check conformance of $L_1 = L \uparrow A_1$ and $L_2 = L \uparrow A_2$ with respect to the two Petri net fragments $N_1$ and $N_2$. In [144], it is shown that $L$ is perfectly fitting $N$ if and only if $L_1$ is perfectly fitting $N_1$ and $L_2$ is perfectly fitting $N_2$. *Hence, we can translate a larger conformance checking problem into two smaller ones without losing accuracy.* Since $L_1$ is perfectly fitting $N_1$ and $L_2$ is perfectly fitting $N_2$, we conclude that $L$ is perfectly fitting $N$.

The Petri net can be decomposed into more parts using the rule explained before: The net can be further decomposed by cutting through transitions that are visible and unique. Figure 12.12 shows that the Petri net can also be split into three fragments. Again we have the same guarantee: the overall log is perfectly fitting if and only if each of the sublogs is perfectly fitting. The partitioning of the net into fragments following the rules described in [144] is called a *valid decomposition*. There is always a unique *maximal* decomposition having fragments which cannot be split further. The maximal decomposition of the Petri net in Fig. 12.8 has six fragments resulting in activity sets: $\{a\}$ (transitions and arcs connected to *start*), $\{a, b, d, e\}$ (transitions and arcs connected to $c1$ and $c3$), $\{a, c, e\}$ (transitions and arcs connected to $c2$), $\{c, d\}$ (transitions and arcs connected to $c4$), $\{d, e, f, g, h\}$ (transitions and arcs connected to $c5$, $c6$ and $c7$), and $\{f, g, h\}$ (transitions and arcs connected to $c8$, $c9$ and *end*). Note that each place and each arc of the original Petri net appears in precisely one of the fragments.

Given a valid decomposition (maximal or not) of an arbitrary Petri net $N$ into fragments $N_1, N_2, \ldots, N_k$ with activity sets $A_1, A_2, \ldots, A_k$: $L$ is perfectly fitting $N$ if and only if for all $i \in \{1, \ldots, k\}$ $L_i$ is perfectly fitting $N_i$. *If a deviation is found in one of the sublogs, then there is a deviation in the overall log. If no deviation is found in any of the sublogs, then there are no deviations in the overall log.*

The approach in [144] is very general and can be applied to other process models [142] and combined with a variety of decomposition strategies [106]. For larger, relatively structured process models, the time for conformance checking may go from

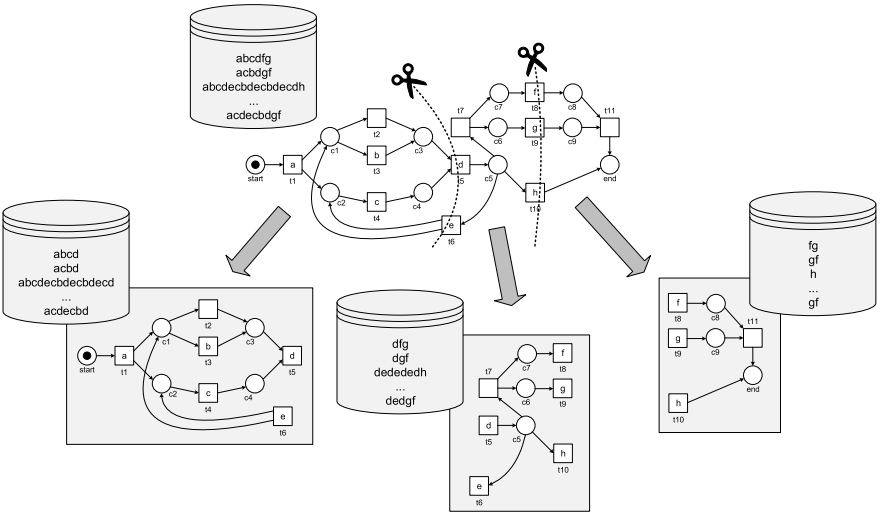



**Fig. 12.12** Another valid decomposition of the process model resulting in three sublogs: $L_1 = L \uparrow \{a, b, c, d, e\}$, $L_2 = L \uparrow \{d, e, f, g, h\}$ and $L_3 = L \uparrow \{f, g, h\}$

hours or days to seconds or minutes (on a single computer). *Even when computation is not distributed over multiple computing nodes, decomposition can help to speed up conformance checking.* This can be explained by the fact that some algorithms are exponential in the number of different activities or the average length of the traces.

### *12.3.2 Process Discovery Using Activity-Based Decomposition*

Assume we have an "Oracle" that, given an event log $L$ over $A$, provides activity sets $A_1, A_2, \ldots, A_k$ with $A = A_1 \cup A_2 \cup \cdots \cup A_k$. If we discover a perfectly fitting model $N_i$ for each of the sublogs $L_i = L \uparrow A_i$, then $L$ is also perfectly fitting the composed model $N = N_1 \oplus N_2 \oplus \cdots \oplus N_k$. The composition synchronizes the different submodels based on shared activity labels. In terms of a Petri net with unique labels, this means that the composition is the net obtained by fusing transitions having the same label (see [144]). This is the reverse of the decompositions in Fig. 12.11 and Fig. 12.12.

We can apply any existing discovery technique to the sublogs $L_i = L \uparrow A_i$. It is even possible to apply different discovery techniques to different sublogs. This can also be done in parallel. Hence, given a suitable Oracle, we get a highly configurable approach to discover process models in a decomposed or distributed manner.

Figure 12.13 sketches the approach using an example. Based on the original log $L = [\langle a, b, c, d, e, f, g, i\rangle, \langle a, c, d, f, h, e, i\rangle, \langle a, c, b, d, g, f, e, i\rangle, \langle a, c, d, h, e, f, i\rangle, \ldots]$, the Oracle suggests: $A_1 = \{a, b, c, d\}$, $A_2 = \{d, g, h, i\}$, and $A_3 =$

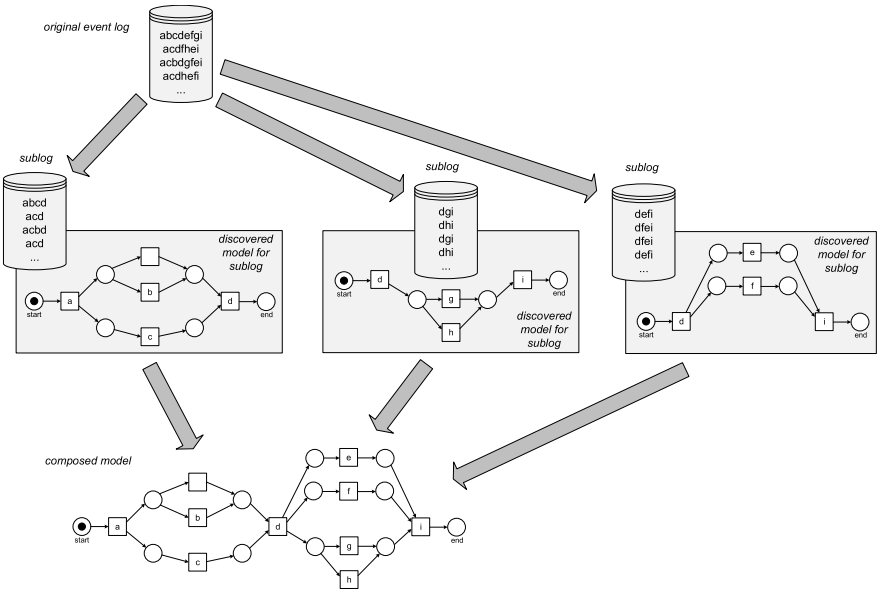



**Fig. 12.13** Decomposed discovery: The event log is projected onto subsets of activities and the resulting sublogs are input for a standard discovery technique. The resulting process models are merged into an overall model by synchronizing overlapping activities

$\{d, e, f, i\}$. The three sublogs $L_1 = L \uparrow A_1$, $L_2 = L \uparrow A_2$, and $L_3 = L \uparrow A_3$ serve as input for conventional discovery approaches. This step can be distributed. The resulting three models are shown in Fig. 12.13 and can be merged into an overall model. Superfluous start/end places are removed while composing the models. Since each of the intermediate models is perfectly fitting, the overall model is also perfectly fitting [144].

All of this is done under the assumption of some Oracle providing $A_1, A_2, \ldots, A_k$. If these sets are poorly chosen (e.g., no overlap between activity sets), then the resulting model may be severely underfitting. The Oracle may be based on *domain knowledge*, e.g., exploiting the natural hierarchy of a system or organization. Note that a software architecture provides information on which components can interact. Such information can be exploited to select partly overlapping activity sets.

We can also use sampling to quickly build a directly-follows graph (or use the MapReduce approach described before) and then use heuristics to decompose the graph [74]. There are various other approaches to quickly decide on activity sets without trying to discover an overall process model first. After projecting the event log on sublogs, more time-consuming approaches can be used.

A well-chosen collection $A_1, A_2, \ldots, A_k$ may also help in the balance between overfitting and underfitting and thus lead to better results in less time. For activities in different subsets we do not need to observe all interleavings to derive concurrency: We only need to see to see the "local" interleavings.

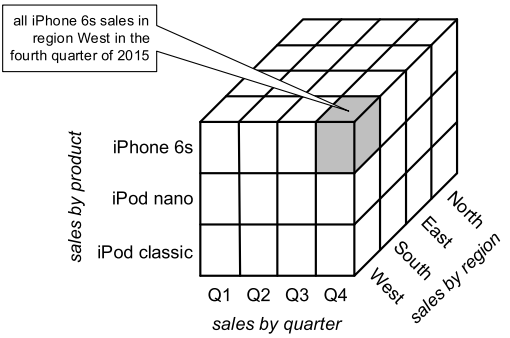



**Fig. 12.14** Three dimensional OLAP cube containing sales data. Each cell refers to all sales of a particular product in a particular region and in a particular period. For each cell we can compute metrics such as the number of items sold or the total value

The goal of this section was to illustrate the different ways in which logs can be decomposed and used for both conformance checking and discovery. The goal was not to describe a concrete algorithm. Therefore, we skipped many of the details. However, the examples show that there are many ways to decompose process mining problems when event logs get extremely large.

## 12.4 Process Cubes

Case-based and activity-based decomposition distribute events over sublogs. For case-based decomposition, each event was assigned to a particular sublog based on the case it belongs to. In case of activity-based decomposition an event may be assigned to multiple sublogs (activity sets may overlap). Decomposition was done for performance reasons. However, there may be more reasons for grouping events. These are discussed in this section using the notion of a *process cube*.

In a process cube, events are organized using different *dimensions* (e.g., case types, regions, subprocesses, departments, and time windows). The cells in such a process cube can be analyzed using process mining techniques by creating a sublog per cell. The results of different cells can be compared. Note that a process cube does not need to be limited to a single process: All events recorded in an organization can be organized in a single cube.

Process cubes are closely related to the cubes used in OLAP (Online Analytical Processing). Figure 12.14 shows an example of an OLAP cube. The example cube has three dimensions and the elements in each cell refer to sales transactions. Assume we are interested in the number of items sold. In this case, the OLAP tool can be used to show the number of products sold for each cell in the OLAP cube. Suppose that in the fourth quarter (Q4) very few iPhones were sold in region West. Then one can drill down into this cell. For instance, one can look at individual sales, view the sales per month (refinement of Q4 into October, November, and December), or view the sales per shop (refinement of the region dimension). When drilling down the information is refined. Pivoting the data, often referred to as "slicing and dicing", helps to see particular patterns. By "slicing" the OLAP cube, the analyst can zoom into a selected slice of the overall data, e.g., only looking at sales of the

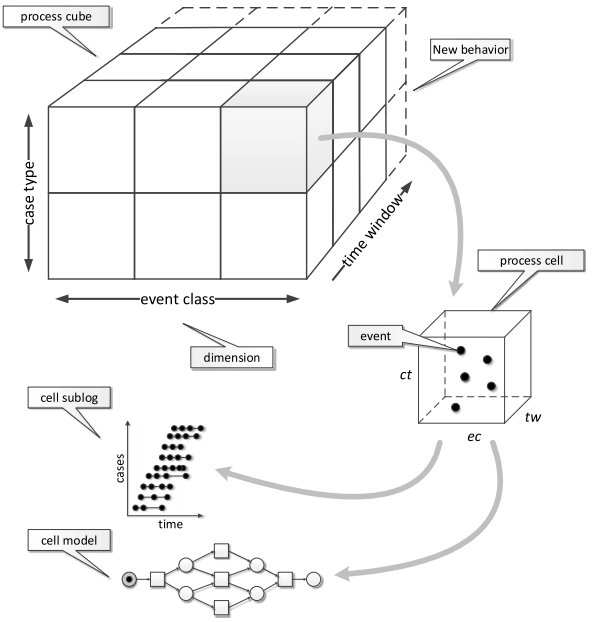



**Fig. 12.15** Each cell in a process cube contains a collection of events that can be converted into an event log. Any process mining technique can be applied to the corresponding event log and subsequently results (e.g., a process model and social network) are associated to the cell. OLAP operations such as slice, dice, roll-up, and drill-down facilitate exploration and comparison of behavior

iPod nano. Slicing effectively removes a dimension from the cube. The term "dicing" refers to applying filters on (possibly) multiple dimensions of the cube. Dicing corresponds to selecting a subcube rather than removing a dimension by selecting a value for it. Views can also be changed by rotating the cube, e.g., swapping the rows and columns. The results can be viewed in tabular form or visualized using various charts. Many BI tools support the OLAP functionality described. These tools support a broad range of charts, e.g., pie charts, bar charts, radar plots, scatter plots, speedometers, Pareto charts, box plots, and scorecards. These can be used to view the data from different angles.

OLAP cubes and notions such as slicing and dicing are *not* process centric. BI products can analyze an OLAP cube with sales data as shown in Fig. 12.14, but do this without considering the underlying process. The sales events are immediately aggregated without trying to distill the underlying process.

*Process cubes* [22, 145] add process specific elements to OLAP cubes and can be viewed as a crossover between process mining and OLAP. Event attributes like activity, timestamp, resource, transaction type, etc. are handled in a specific manner and cells can be converted to logs as shown in Fig. 12.15. Note that results in OLAP cubes are numbers, e.g., the number of transactions in a shop or the average value of sales in October. Results associated to cells in a process cube may include a variety of models (e.g., process models, social networks) and are not limited to numbers (e.g., a sum or average). Earlier we compared process mining with spreadsheets (Sect. 1.3) and noted that spreadsheets are dealing with numbers rather than events and dynamic behavior. A similar distinction can be made between OLAP cubes and process cubes.

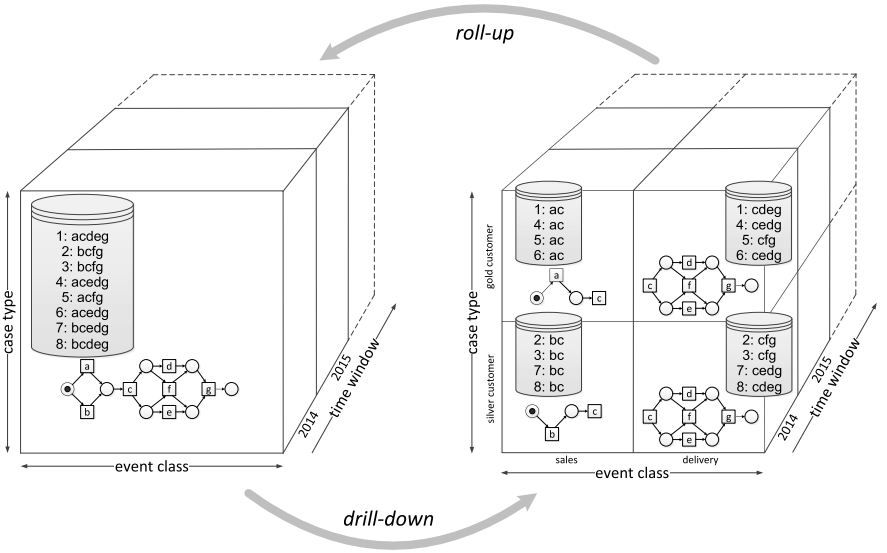



**Fig. 12.16** Illustration of roll-up and drill-down. The drill-down operation uses case-based decomposition for the case type dimension and activity-based decomposition for the event class dimension [145]

Figure 12.16 (taken from [145]) is used to illustrate roll-up and drill-down operations. The cube has three dimensions: *case type*, *event class* and *time window*. In the left cube, there is only one case type and only one event class. The cube covers multiple time windows, but only one is shown (all cases completed in 2014). In this toy example, there are only eight cases (i.e., process instances) and seven distinct activities. The process may be split by identifying multiple case types and/or multiple event classes. The cube shown on the right-hand side of Fig. 12.16 has two case types (gold customer and silver customer) and two event classes (sales and delivery).

As shown in Fig. 12.16, cases 1, 4, 5, and 6 refer to gold customers. Hence, the cells in the "gold customer" row in Fig. 12.16(right) include events related to these four cases. The event class dimension is based on the event's activity. The event class "sales" includes activities $a$, $b$, and $c$. The event class "delivery" refers to activities $c$, $d$, $e$, $f$, and $g$. The time window dimension uses the timestamps of events. A time window may refer to a particular day, week, month, or any other period.

Each dimension in a process cube may have an associated hierarchy, e.g., years composed of months and months composed of days. Using roll-up and drill-down operations, the granularity can be changed as shown in Fig. 12.16.

As mentioned, each cell in a process cube refers to a collection of events and possibly also process mining results (e.g., a discovered process model). Events may be included in *multiple* cells, e.g., sales and delivery cells share $c$ events in Fig. 12.16. There are many situations where cells may be overlapping and share events. A person may be part of multiple departments or have multiple roles. The events gener-

ated by this person are therefore associated to the cells of the different departments and roles. Activities may also be part of multiple processes.

Figure 12.16 assumes a fixed case notion. In process cubes, we normally use a more relaxed case notion. This way we can look at the data from different angles as discussed in Sect. 5.5. When creating event logs from cells, we need to "flatten the event data" to have a clear process instance notion. When considering events related to customer orders, we may view the same event data from the viewpoint of orders, order lines, and deliveries (see Sect. 5.5). Therefore, a process cube needs to support multiple case notions. The same event may be part of multiple cells and multiple cases.

Given a process cube with suitably chosen dimensions, we can compare process mining results generated for an array of cells. We refer to this as *comparative process mining*. The goal is to highlight differences between cells. Next to cross-checking conformance (the log for cell $i$ is replayed on the model for cell $j$), we can compare process models visually or overlay the models as is done in tools like myInvenio (see Fig. 11.15). In the context of ProM several implementations of the process cube concept exist [22, 145]. These have in common that arbitrary plug-ins can be applied to an array cells after which the results can be compared.

OLAP technology and the notion of process cubes can help to deal with large heterogeneous event collections. Also note the relation with event log decomposition (Sects. 12.2 and 12.3). The drill-down operation in Fig. 12.16 uses *case-based* decomposition for the case type dimension and *activity-based* decomposition for the event class dimension.

## 12.5 Streaming Process Mining

The process mining algorithms discussed before assume that all events are stored in a file or database. The algorithms can access all event data at any time. In some scenarios, such assumptions are unrealistic: Events arrive in *streams* and, if not processed immediately, the corresponding information is lost. Events may arrive so rapidly that it is not feasible to store them all in active storage where they can be processed. At best we may be able to archive the events, but there is no time to process these archived data. Moreover, at any point in time we may need to answer questions related to these streams of events (including the recent events). Hence, there is no time to access archived data. Questions may refer to recent data and need to be answered immediately. Handling streams of events can be viewed as "drinking from a firehose": Trying to store and process all event data using conventional process mining algorithms is impossible. Therefore, dedicated algorithms are needed to handle streams of events.

Figure 12.17 shows a stream of events. Suppose that we would like to know the process at any point in time. The process may be changing over time. This phenomenon corresponds to the notion of *concept drift* [81] discussed in Sect. 10.6.3. This triggers the question: *Should the discovered model at time t describe the process over the last day, week, month or year?* New activities may appear and other

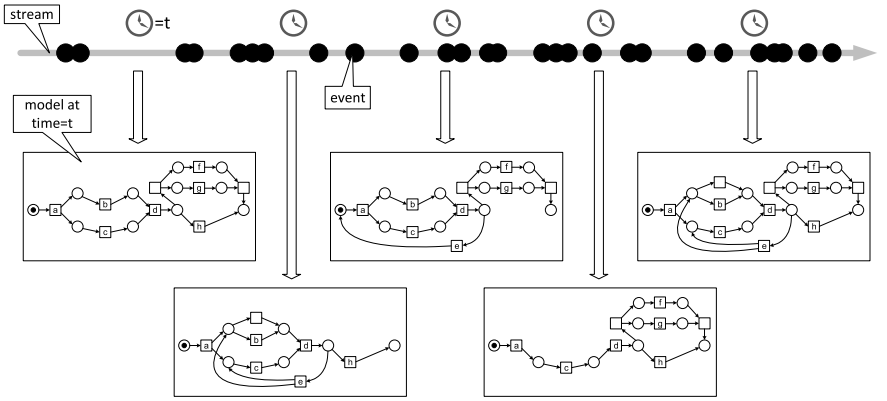



**Fig. 12.17** Given a stream of events we would like to provide an up-to-date process model at any point in time

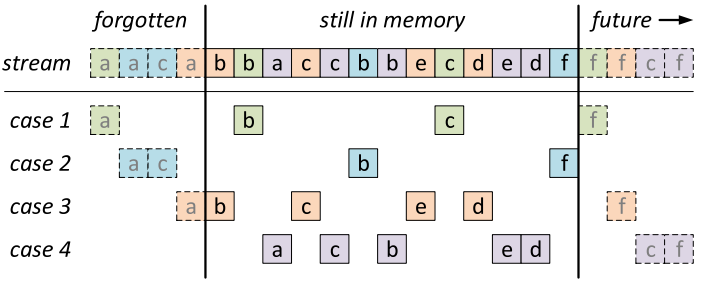



**Fig. 12.18** Using a fixed window of events may lead to incorrect conclusions

activities may fade out. Also the ordering of activities may change. Concept drift refers to all perspectives, e.g., bottlenecks may emerge without changing the ordering of activities and the distribution of work over people may shift. However, for simplicity we focus on control-flow in this section.

In a streaming context, we cannot pass over the data multiple times. Processing time is limited, and questions need to be answered in (near) real-time. *Memory is limited and cannot be increased over time or when the arrival rate of events increases.* Given these constraints, we can often only aim for approximate results. There is a trade-off between handling larger volumes and ensuring accuracy. Often only summaries or samples can be stored. Yet, some streaming algorithms produce results of similar quality as traditional algorithms at a fraction of the computational cost. Hence, they can also be applied to large collections of non-streaming event data.

To understand the challenges, let us consider the example stream in Fig. 12.18. Suppose we are able to store a limited number of events and simply forget the oldest events when new events arrive. Many approaches for streaming data use a fixed-length "window" consisting of the last $n$ elements for some (typically large) $n$. Ap-

plying this in the process mining context may lead to misleading results. As shown in Fig. 12.18, we may loose the prefixes of traces. The initial activities of a case may be forgotten, e.g., the prefix $\langle a, c \rangle$ is missing for case 2. This could lead to the incorrect assumption that cases can start with activity $b$. There is also the problem that we do not know whether cases are finished. The final $f$ activity for case 1 may still occur in the future (or not).

Suppose we need to create a process model for the cases handled during the last month, but can only store 1 million events. Assume this corresponds to roughly 5% of events occurring per month. Hence, for every 20 events that happen on average only one can be stored.

- If we use the fixed-length window approach shown in Fig. 12.18, we have two problems. First of all, the traces may be incomplete (missing prefixes). We are only storing the last 1.5 day (5% of a month). Hence, this is a significant problem if cases are running for days rather than minutes. Second, the process model will be biased towards the most recent period and may not be representative for the whole month.
- We can also randomly sample events. Every event has a 5% probability of being stored in the fixed-length window. Each time a new event is stored, the oldest event is removed. This approach will also provide very misleading results. Suppose a case has 20 events. The probability that the whole case will be stored for a selected event is less than $0.05^{20-1} = 1.9 \times 10^{-25} \approx 0$. Many cases will be reduced to a few random events. Hence, the discovered process model on such data will not make any sense.

To address the problem, we can store *summarized data* or use *smarter forms of sampling*.

Instead of randomly sampling events, we can also sample cases, i.e., use a window where we only keep events of a selected 5% of all cases. We cannot *randomly sample cases* because we would need to keep a list of selected cases and a list of non-selected cases. We do not know the set of cases beforehand and, even if we would, we could not store this information. However, we can use hashing as a kind of controlled randomization. By using a suitable hash function, we can select cases without actually storing them [114]. This will allow us to retain all events of a selected subset of cases and thus create a representative process model.

Next to careful sampling, we can also *store summarized data rather than individual events* (see Fig. 12.19). This is based on the observation that many algorithms basically count frequencies of activities and local patterns like the directly-follows relation. The $\alpha$-miner, the heuristic miner, and the directly-follows based inductive miners (i.e., IMD, IMFD, and IMCD) basically use the following sources of information: the frequencies of observed activities and the frequencies of the elements in the directly-follows relation. The number of unique activities is typically limited compared to the number of cases and events. Hence, these frequencies can be stored compactly. The memory that can be used for this has a fixed upper bound as shown in Fig. 12.19. In [27], a particular approach based on this idea is presented. The approach uses three queues $Q = (Q_{act}, Q_{df}, Q_{last})$.

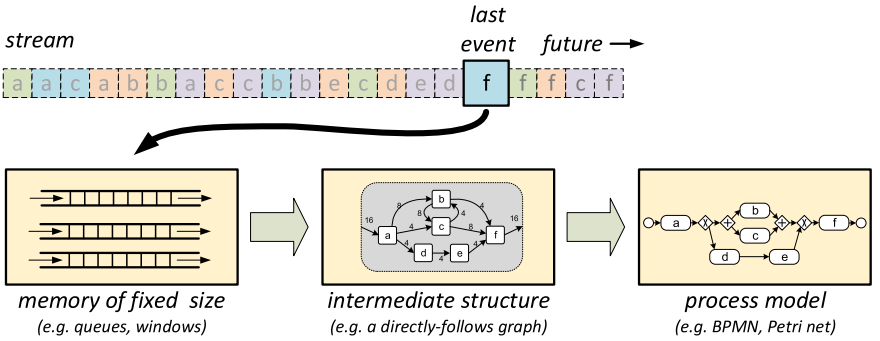



**Fig. 12.19** To deal with streams, we can only use a predefined amount of memory. Instead of storing all events, we need to store summarized or sampled data. Often this limited memory is organized is such a way that some intermediate structure like a weighted directly-follows graph can be constructed from it (e.g., using queues $Q = (Q_{act}, Q_{df}, Q_{last})$). This intermediate structure can be used to create a process model at any point in time

- Queue $Q_{act}$ is used to count the number of times an activity has occurred. It has elements of the form $(a, n)$ where $a$ is the activity and $n$ the frequency.
- Queue $Q_{df}$ is used to count the number of direct successions. It has elements of the form $((a, b), k)$ where $k$ is the number of times that $a$ was followed by $b$ for the same case.
- Queue $Q_{last}$ is used to keep track of the last activity executed for a particular case. It has elements of the form $(c, a)$ where $a$ is the last activity observed for $c$.

Queue $Q_{last}$ is needed to construct $Q_{df}$. Each queue has maximum size and is updated each time a new event arrives. If a queue is full, then the addition of a new element implies the deletion of the least significant element. Queues can be sorted and operated using different heuristics. It is also possible to add *aging* to deal with concept drift. In this case the updates of the queues take into account an aging factor that gives more weight to recent events. Based on $Q = (Q_{act}, Q_{df}, Q_{last})$, we can apply existing process discovery techniques using information similar to the directly-follows graph (e.g., the $\alpha$-miner, the heuristic miner, and the directly-follows based inductive miners). A detailed discussion of such approaches is beyond the scope of this book (see [27]).

Summarizing the above: We can adapt process mining to streams of event data by carefully sampling cases and by keeping only summarized data. We can use existing techniques for this [6, 114].

## 12.6 Beyond the Hype

The techniques described in this chapter enable the application of process mining when event logs are huge. As demonstrated, process mining techniques can exploit the MapReduce programming model, modern database technology (e.g., in-memory

databases and columnar databases), and large-scale distributed file systems (e.g., Hadoop). In this chapter, we focused on techniques to decompose event logs, but also discussed related notions such as process cubes and streaming process mining.

These topics fit perfectly with the current attention for "Big Data" in industry and society. Numerous books appeared in recent years. These discuss "Big Data" from different angles: distributed algorithms [114], analytics [17], societal impact [99, 100], and management [54]. However, the real challenges are often related to data acquisition, data preparation, and the interpretation of results. The majority of real-life applications of process mining would benefit more from a "data science mindset" rather than new Hadoop-like infrastructures.

Moreover, data sets that are "Big" today may be "small" tomorrow. The book "Concise Survey of Computer Methods" [107] by Peter Naur (1928–2016), published in 1974, used already the term "data science" and contains several chapters describing techniques for processing and managing "large datasets". These were written at a time where hard disks had a capacity of just a few megabytes. Although the dimensions have changed dramatically, many of the core principles remained invariant. *To change data into real value, one needs to ask the right questions, use the right analysis techniques, and be able to interpret the results.* It does not suffice to just have a "Big Data" infrastructure.