Graph-Based Process Mining

Amin Jalali(✉)

Department of Computer and Systems Sciences, Stockholm University, Stockholm, Sweden
aj@dsv.su.se

Abstract. Process mining is an area of research that supports discovering information about business processes from their execution event logs. One of the challenges in process mining is to deal with the increasing amount of event logs and the interconnected nature of events in organizations. This issue limits the organizations to apply process mining on a large scale. Therefore, this paper introduces and formalizes a new approach to store and retrieve event logs into/from graph databases. It defines an algorithm to compute Directly Follows Graph (DFG) inside the graph database, which shifts the heavy computation parts of process mining into the graph database. Calculating DFG in graph databases enables leveraging the graph databases' horizontal and vertical scaling capabilities to apply process mining on a large scale. We implemented this approach in Neo4j and evaluated its performance compared with some current techniques using a real log file. The result shows the possibility of using a graph database for doing process mining in organizations, and it shows the pros and cons of using this approach in practice.

Keywords: Process mining · Graph database · Big data · Neo4j

1 Introduction

Business Process Management (BPM) is a research area that aims to enable organizations to narrow the gap between business goals and information technology support [21]. Business process evaluation is a key support in narrowing down this gap. There are two evaluation techniques to analyze business processes, a.k.a., model-based analysis, and data-based analysis [17]. While model-based analysis deals with analyzing business process models, the data-based analysis mostly focuses on analyzing business processes based on their execution event logs.

Process Mining is a discipline in the BPM area that enables data-based analysis for business processes in organizations [18]. It allows analysts not only to evaluate the business processes but also to perform process discovery, compliance checking, and process enhancement based on the execution result, a.k.a., event logs. As the volume of logs increases, new opportunities and challenges also appear. The large volume of logs enables the discovery of more information about business processes; while also raises some challenges, such as feasibility, performance, and data management.

S. Leemans and H. Leopold (Eds.): ICPM 2020 Workshops, LNBIP 406, pp. 273–285, 2021.
https://doi.org/10.1007/978-3-030-72693-5_21

The large volume of data is a challenge to perform process mining in organizations. There are different approaches to deal with this problem. This paper proposes and formalizes a new approach to store and retrieve event logs in graph databases to do process mining on a large volume of data. It also defines an algorithm to compute Directly Follows Graph (DFG) inside the graph database. As a result, it enables i) removing the requirement to move data into analysts' computer, and ii) scaling the DFG computation vertically and horizontally.

The approach is implemented in Neo4j, and its performance is evaluated in comparison with some current techniques based on a real log file. The result shows the feasibility of this approach in discovering process models when the data is much bigger than the computational memory. It also shows better performance when dicing data into small chunks.

The remainder of this paper is organized as follows. Section 2 gives a short background on process mining and graph database. Section 3 introduces the graph-based process mining approach, and Sect. 4 elaborates on the implementation of the approach in Neo4j. Section 5 reports the evaluation results. Section 6 discuss alternative approaches and related works, and finally, Sect. 7 concludes the paper and introduces future research.

2 Background

2.1 Process Mining

Process Mining is a research area that supports business process data-based analysis [18]. Process discovery is a sort of process mining technique that enables identifying process models from event logs automatically. There are different sorts of perspectives that can be discovered from event logs. Control-flow, which describes the flow of activities that happened in a business process, is one of the most important ones. Directly-Follows Graphs (DFGs) is a simple notation widely used and considered a de-facto standard for commercial process mining tools [19].

Figure 1 shows an overview of how a process model can be discovered from event logs using DFG graphs. The process discovery starts by loading a log file that stores business process execution results, a.k.a., log files. Each log contains

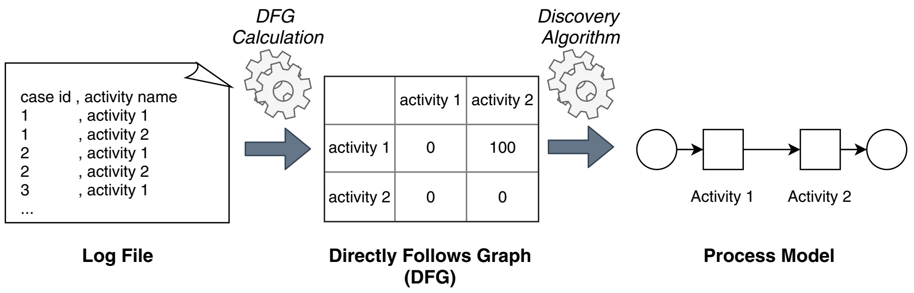

Fig. 1. Steps in a process discovery algorithm

a set of traces representing different cases that are performed in the business process. Each trace contains a set of events representing the execution result of activities in the business process. Thus, a log file shall contain information about traces and events at a minimum. Note that the events should be stored according to the execution order with this basic setup, unless we have information about execution time. It is usual to have more information like the execution time and the resource who has done the activity in the log file.

The next step is calculating the Directly Follows Graph (DFG). This graph shows the frequency of direct relations between activities that are captured in the log file. The result can be considered as a square matrix with the activity names as the index for rows and columns. Let's consider the cell with the index of *activity 1* for the row and *activity 2* for the column (see Fig. 1). The value of the cell shows the number of times that the *activity 2* happened after *activity 1*. Although the calculation of DFG comes back to alpha miner, which was introduced around 20 years ago, it is still the backbone for many process mining algorithms and tools [20]. There are different variations of DFG that store more information, but the basic idea is the same.

The last step is to infer the process model from DFG matrix based on rules that are specified by a process discovery algorithm. This step usually does not take much time since the computation is performed on top of DFG.

2.2 Graph Database

Graph databases are Database Management Systems (DBMS) that support creating, storing, retrieving, and managing graph database models. Graph database models are defined as the data structure where schema and instances are modeled as graphs, and the operation on graphs are graph-oriented [2]. The idea is not new, and it comes back to the late eighties when the object-oriented models were also introduced [2]. However, it recently got much attention in both research and industry due to its ability to handle the huge amount of data and networks. It enables leveraging parallel computing capabilities to analyze massive graphs. As a result, a new discipline is emerged in research, called Parallel Graph Analytics [15].

There are different sorts of graph databases with different features. For example, Neo4j is a graph DBMS that supports both vertical and horizontal scaling, meaning that not only the hardware of the system that runs the DBMS can be scaled out, but the number of physical nodes that run the DBMS as a network can be increased. These features enable having a considerable performance at runtime.

3 Approach

This paper proposes a new approach to store event logs and retrieve a DFG using a graph database. In this way, the scalability capabilities in graph databases can be used in favor of applying process mining. The aim is to introduce an alternative approach to enable discovering process models from large event logs.

Thus, the formal definitions of event repository in graph form are introduced. Then, the soundness property of such a repository log is defined. Finally, an algorithm to discover DFG is introduced.

Note that the formal definition is simplified by limiting the set of attributes to hold information about activities. In practice, the definition of attributes can be extended to store all information about the data perspective.

3.1 Definitions

Definition 1 (Event Repository). *An event repository is a tuple $G = (N = L \cup T \cup E \cup A, R)$, where:*

- *N is the superset of L, T, E, and A subsets which are pairwise disjoint, where:*
  - *L represents the set of logs,*
  - *T represents the set of traces,*
  - *E represents the set of events,*
  - *A represents the set of attributes, representing activities, where:*
  - *$L \cap T \cap E \cap A = \emptyset$.*
- *$R = L \times T \cup T \times E \cup E \times E \cup E \times A$ is the set of relations connecting:*
  - *logs to traces, i.e., $L \times T$*
  - *traces to events, i.e., $T \times E$,*
  - *events to events, i.e., $E \times E$,*
  - *events to attributes, i.e., $E \times A$, where:*
  - *$N \cap R = \emptyset$*

Let's also define two operators on the graph's nodes as:

- *$\bullet n$ represents the operator that retrieves the set of nodes from which there are relations to node n, i.e., $\bullet n = \{\forall e \in N | (e, n) \in R\}$.*
  - *This operator enables retrieving incoming nodes for a given node, e.g., retrieving the set of events that occurred for an activity.*
- *$n\bullet$ represents the operator that retrieves the set of nodes to which there are relations from node n, i.e., $n\bullet = \{\forall e \in N | (n, e) \in R\}$.*
  - *This operator enables retrieving outcoming coming nodes for a given node, e.g., retrieving the set of events that occurred for a trace.*

Note that the relations among logs, traces, events, and attributes are adopted from the eXtensible Event Stream (XES) standard [1]. The information is stored in attributes like XES standard which states: "Information on any component (log, trace, or event) is stored in attribute components" [1]. This is the reason why the activities are represented as attributes in this work. Note that we limit attributes to represent activities only in this work for making formalization simple for the sake of presentation. In practice, the attributes can have types to represent different properties. For example, they can be used to store different data properties of an event, e.g., who has performed it, what data it generates, etc. The usage of attributes in practice can also be extended to hold case id properties for traces and metadata information for the log node. Despite it is good to have the case id as an attribute, we kept the formalization simple by ignoring that as traces represent cases in this structure. Note that you need to know the case id to create such a structure, which is needed in the ETL process.

Definition 2 (Soundness). *An event repository $G = (N = L \cup T \cup E \cup A, R)$, where N, L, T, E, A, R, represent the set of Nodes, Logs, Traces, Events, Attributes, Relations respectively, is sound iff:*

- $\forall t \in T, |\bullet t| = 1$, *meaning that a trace must belong to 1 and only 1 log.*
- $\forall e \in E, |\bullet e \cap T| = 1$, *meaning that an event must belong to 1 and only 1 trace.*
- $\forall e \in E, |\bullet e \cap E| <= 1$, *meaning that an event can only have at most 1 input flow from another event.*
- $\forall e \in E, |e \bullet \cap E| <= 1$, *meaning that an event can only have at most 1 output flow to another event.*
- $\forall e \in E, |e \bullet \cap A| = 1$, *meaning that an event must be related to 1 and only 1 attribute.*

Note that the soundness is a property of event repository and shall not be mistaken by the soundness property of a modeling notation like Petri nets. It is worth mentioning that this formalization can be extended to enable several types of sequences among event logs. To calculate DFG, we need to count the number of direct relations among events for each activity pairs. Algorithm 1 defines how the DFG for a given sound event repository can be calculated.

Algorithm 1: Algorithm for calculating dfg

```
1 Algorithm dfgcalculator($G = (N = L \cup T \cup E \cup A, R)$)
2     $\Psi \leftarrow \emptyset$;
3     foreach two attributes $a, b \in A$ do
4         $c \leftarrow 0$;
5         foreach $e \in \bullet a, e' \in \bullet b$ do
6             if $(e, e') \in R$ then
7                 $c \leftarrow c + 1$;
8         $\Psi \leftarrow \Psi \cup \{(a, b, c)\}$;
9 return $\Psi$;
```

3.2 Example

This section elaborates on the definitions through an example.

Figure 2 shows an example of a sound event repository graph. The set of nodes for Log, Trace, Event, and Attribute are colored as green, red, white, and

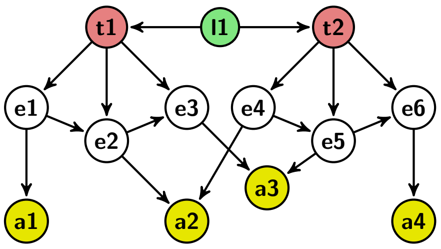

Fig. 2. An example of a sound event repository graph

yellow, respectively. This repository includes one log file, called *l1*, which has two traces, i.e., *t1* and *t2*. *t1* has three events that occurred in this order *e1* → *e2* → *e3*. *t2* also has three events that occurred in this order *e4* → *e5* → *e6*.

As it can be seen, each event is related to one activity, e.g., *e1* is the execution of activity *a1*. To get the list of events that happened for an activity *a1*, we can use $\bullet a1$ operator, which returns $\{e1\}$. For some activities, there might be more than one event, e.g., $\bullet a2$ returns $\{e2, e4\}$. Applying Algorithm 1 on this event repository will return the DFG. The DFG calculation is described as below:

– for each pair of activities, the algorithm will calculate the frequency. We show the calculation for one pair example, i.e., $a2, a3$:
  – $\bullet a2$ retreives $\{e2, e4\}$
  – $\bullet a3$ retreives $\{e3, e5\}$
  – $c = \sum_{\forall e \in \bullet a2, e' \in \bullet a3} |(e, e') \in R| = \sum_{\forall e \in \{e2,e4\}, e' \in \{e3,e5\}} |(e, e') \in R|$
    $= |\{(e2, e3), (e4, e5)\}| = 2$

If we calculate the frequencies for all pairs of activities, the result will be like Table 1.

Table 1. DFG calculation for the sample event repository graph

	a1	a2	a3	a4
a1	0	1	0	0
a2	0	0	2	0
a3	0	0	0	1
a4	0	0	0	0

4 Implementation

The approach presented in this paper is implemented using the Neo4j, which was chosen because it supports i) storing graphs and doing graph operations,

ii) both vertical and horizontal scaling, iii) querying the graph using Cypher, iv) containerizing the database, which allows controlling the computational CPU and memory.

We implemented a data-aware version of the approach. The main differences with the formalization are:

– Attributes store activity names and other attributes that might be associated with an event like resource id, case id, etc. To comply with PM4Py, we stored the log, case, and activity name by 'log_concept_name', 'case_concept_name', and 'concept_name' respectively.
– events have timestamps to enable dicing information based on time. Note that the timestamp cannot be defined as an attribute with its own key since we will end up with many extra nodes due to many timestamps that exist for each event. Thus, they are kept as an attribute of Event class, following the same practice to deal with times in data warehousing [14].

The calculation of DFG is implemented using a Cypher query as below:

```
match
(a1:Attribute {key:'concept_name'})<--(:Event)-[n]->(:Event)
    -->(a2:Attribute {key:'concept_name'})
return
a1.val as dfg_from, a2.val as dfg_to, count(n) as dfg_freq
```

The match clause in the query identifies all patterns in sub-graphs that match the expression. This expression selects two attributes *a1* and *a2* with the type of *concept_name*, which indicates that they are activities' names. Then, it selects all incoming events to those attributes where there is a direct relationship between those two events. The return clause retrieves all combinations of attributes in addition to the number of total direct relations between their events, which is the calculation that we formalized in Algorithm 1.

To limit the number of events based on their timestamp, we can easily add a where clause to the cypher query to limit the timestamp. For other attributes, the associated attribute node can be filtered.

5 Evaluation

This section reports the evaluation result of the approach, which is presented in this paper[1]. To evaluate the approach, we calculated DFG for a real public log file [6] using Process Mining for Python (PM4Py) library [3]. This dataset [6] is selected because it is published openly, which makes the experiment repeatable. It is also the biggest log file that we could find in the BPI challenges, which can help us to evaluate the performance.

[1] The data, code and instructions can be found at https://github.com/neo4pm/supporting_materials/tree/master/papers/Graph-based%20process%20mining.

To evaluate the performance, we need to control the resources that are available for performing process mining. Thus, we decided to containerize the experiments and run them with Docker. Docker is a Platform as a Service (PaaS) product that enables creating, running, and managing containers. It also enables the control of the resources that are available for each container, such as RAM and CPU.

Among different process mining tools, we chose PM4Py [3], because i) it is open-source; ii) the DFG calculation step and discovery step can be separated easily, and iii) it can easily be encapsulated in a container. The separation of DFG calculation and discovery step in this library also enables reusing all discovery algorithms along using our approach, which makes our approach very reusable.

We designed two experiments to evaluate our approach. In *Experiment 1*, we loaded the whole log file into both containers running neo4j and PM4Py, so we kept the number of event logs constant. We calculated DFG several times by changing the RAM and CPU, so we defined the computational resources as a variable. In *Experiment 2*, we kept RAM and CPU constant for both containers, and we calculated DFG by dicing the data. The dicing is done based on a time constraint, and we added more days in an accumulative way to increase the number of events. We ran the experiments for each container separately to make sure that the assigned resources are free and available (Table 2).

Table 2. Evaluation setting

	Constant	Variable
Experiment 1	Events in the log (9 million events)	CPU & RAM
Experiment 2	CPU & RAM	Events in the log

5.1 Experiment 1

To simulate the situation where the computational memory is less than the log size, we started by assigning 512 megabytes of ram to each container. We added the same amount of RAM in each experiment round until we reached 4 gigabytes. We also changed the CPU starting from half of a CPU (0.5), by adding the same amount at each round until we reached 4.0.

Figure 3 shows the execution result for both containers, where the x, y and z axes refer to the available memory (RAM) (in megabytes), DFG calculation time (in seconds), and available CPU quotes, respectively. The experiment related to neo4j and PM4Py containers is plotted in red and blue, respectively. As can be seen, PM4Py could not compute DFG when the memory was less than the size of the log, i.e., around 1.5 gigabytes, while neo4j could calculate DFG in that setting. This shows that the graph database can compute DFG when computational memory is less than the log size, which is an enabler when applying process mining on a very large volume of data.

As it can be seen in the figure, the increasing amount of memory reduced the time that neo4j computed the DFG, while it has very little effect on PM4Py. This is no surprise for in-memory calculation since if the log fits the memory, then the performance will not be increased much by adding more memory. It is also visible that assigning more CPU does not affect the performance of either of these approaches.

It should also be mentioned that despite increasing memory can reduce the DFG calculation time for neo4j significantly; it cannot be faster than PM4Py when calculating the DFG on the complete log file. The reason can be that graph databases shall process metadata, which adds more computation than in-memory calculation approaches. Thus, for small log files that can fit the computer's memory, the in-memory approach can be better if the security and access control are not necessary.

5.2 Experiment 2

Event logs usually contain different variations that exist in the enactment of business processes [4]. These variations make process mining challenging because discovering the process based on the whole event log usually produces the so-called spaghetti models, which usually cannot be comprehended by humans, so they have very little value. Thus, analysts need to filter data to produce a meaningful model, which is a common practice in applying process mining [4,11]. Therefore, we designed this experiment to compare our approach and PM4Py when calculating DFG on a filtered subset of data without scaling the infrastructure.

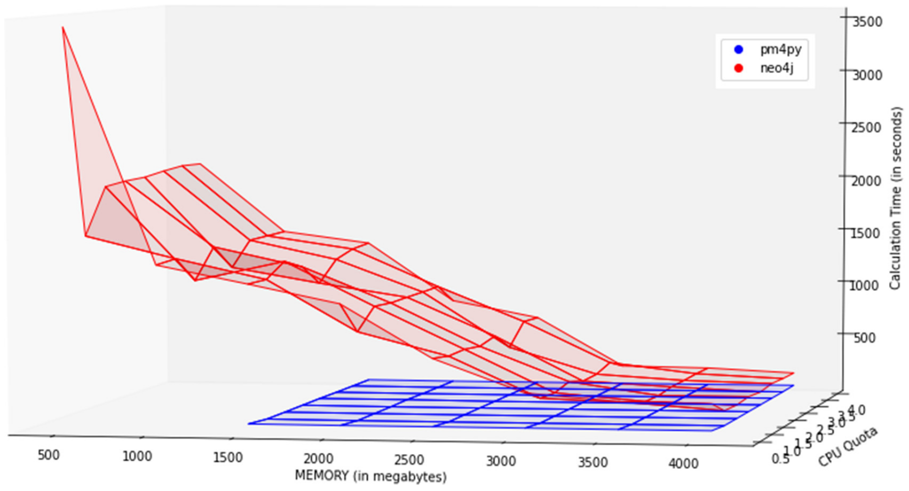

Fig. 3. Evaluating DFG calculation time by scaling resources

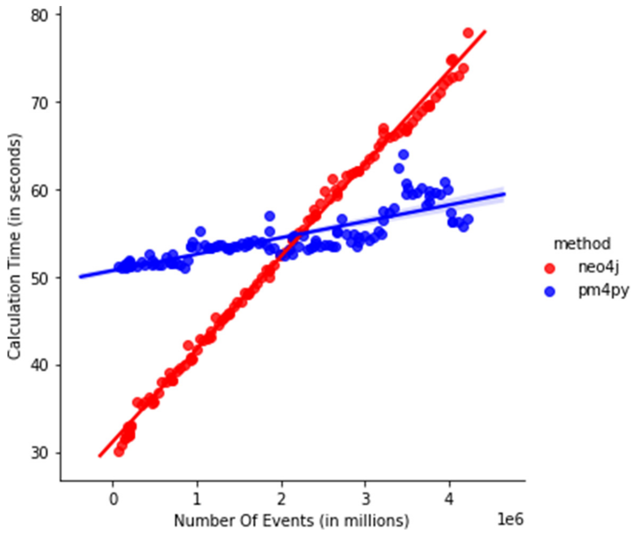

Fig. 4. Evaluating DFG calculation time by dicing the log

To evaluate this scenario, we kept the resources (RAM and CPU) constant for both containers, but we changed the condition for filtering the data. The condition is set based on the dates in which the event occurred. We started by filtering events for a day range, and we calculated DFG for the filtered data. Then, we expanded the filter range by including events that occurred the day after, and we calculated DFG again. We repeated the process for 30 days. As we expanded the filter range by including events that occurred on more days, we increased the number of events. This means that we kept the number of events in the log as a variable. We assigned 14 Gb for RAM and 4 CPU for each container, which was run separately. We diced the data in both settings by filtering events that happened during the first day; then, we added one more day to the filter condition to increase the events in an accumulative way. We repeated this step for almost four months. In this way, we could compare the performance by considering how the size of the filtered events affects the performance of calculating DFG.

Figure 4 shows the evaluation result, where the x and y axes refer to the number of events (in millions) and DFG calculation time (in seconds). As can be seen, our approach performed better when the number of events is less than 2 million. Note that this is still a very big sub-log to analyze for process mining, so this shows that our approach can improve the performance of process mining when dealing with sub-sets of the log. However, PM4Py performed better when the number of events exceeded 2 million. This is no surprise since PM4Py loaded logs into memory first, so increasing the size will have less effect on its performance. Indeed, the difference is only related to filtering the log and retrieving the biggest chunk of data in each iteration.

6 Related Work and Discussion

The related work can be divided into two categories: those related to scalability and those using graph databases.

6.1 Scalability

The scalability issue in process mining is a big concern for applying the techniques on a large volume of data. Thus, different researchers investigated this problem through different techniques.

Hernández, S. et al. computed intermediate DFG and other matrixes through the MapReduce technique over a Hadoop cluster [10]. The evaluation of their approach shows a similar trend for a performance like what we presented in Fig. 4. The performance cannot be compared precisely due to different setup and resources. This is the closest approach to ours.

MapReduce has been used by other researchers for the aim of process mining, e.g., [9,16]. As discussed by [10], MapReduce has been used to support only event correlation discovery in [16], and it is used to discover process models using Alpha Miner and the Flexible Heuristics Miner in [9].

6.2 Graph Database

There are different attempts to use graph databases with process mining.

Esser S. and Fahland D. used the graph database to query multi-dimensional aspects from event logs. This is one important use case that has been introduced by a graph database, i.e., adding more features to the data [7]. They have used Neo4j as the graph database and used Cypher to query the logs. The approach uses a graph database as a log repository to store data without any predefined structure, which is quite different from the topic of this paper. In this regard, the approach is similar to [5], where a relational database is used to store the data. The main difference is that [7] demonstrates that the graph database has more capability to add more features to data, which is a very important topic in any machine learning related approach in general.

Joishi J. and Sureka A. also used a graph database for storing non-structured event logs [12,13]. They also demonstrated that Actor-activity matrix could be calculated using Cypher. However, the approach is context-dependent since the logs are not standardized like our approach. Also, the approach cannot be used with other process discovery algorithms since it does not shift and separate the computation of DFG to a graph database.

Parallel to this work, we realized that Esser S. and Fahland D. [8] extended their approach [7] to discover different perspectives from events which are stored in neo4j. They also introduced an approach to discover DFG from their repository. The approach is similar in creating the event repository, yet this paper also focuses on evaluating the performance and scalability to some extent. This study also confirms the benefits of using a graph database for process mining, which can extend the application of process mining in practice.

7 Conclusion

This paper introduced and formalized a new approach to support process mining using graph databases. The approach defines how log files shall be stored in a graph database, and it also defines how Directly Follows Graphs (DFG) can be calculated in the graph database. The approach is evaluated in comparison with PM4Py by applying it to a real log file. The evaluation result shows that the approach supports mining processes when the event log is bigger than computational memory. It also shows that it is scalable, and the performance is better when dicing the event log in a small chunk.

Graph databases can bring more benefits to process mining than what we have presented in this paper. They are useful to support complex analysis, which requires taking the interconnected nature of data into account. Thus, they can enable more advanced analysis by incorporating data relations while applying different process mining techniques. As future work, we aim to extend the formalization to represent the data-aware event repository. It is also interesting to compare this approach with process discovery approaches that can be implemented in Apache Spark. We also intend to develop a new library to support the use of a graph database for process mining for practitioners and researchers.

References

1. IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams. IEEE Std 1849–2016, pp. 1–50 (2016)
2. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. (CSUR) **40**(1), 1–39 (2008)
3. Berti, A., van Zelst, S., van der Aalst, W.: Process Mining for Python (PM4Py): bridging the gap between process-and data science, pp. 13–16 (2019)
4. Bolt, A., De Leoni, M., van der Aalst, W., Gorissen, P.: Exploiting process cubes, analytic workflows and process mining for business process reporting: a case study in education. In: SIMPDA, pp. 33–47 (2015)
5. De Murillas, E., Reijers, H., van der Aalst, W.: Connecting databases with process mining: a meta model and toolset (2016)
6. Dees, M., van Dongen, B.: BPI challenge 2016: clicks not logged in (2016)
7. Esser, S., Fahland, D.: Storing and querying multi-dimensional process event logs using graph databases. In: Di Francescomarino, C., Dijkman, R., Zdun, U. (eds.) BPM 2019. LNBIP, vol. 362, pp. 632–644. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37453-2_51
8. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. arXiv preprint arXiv:2005.14552 (2020)
9. Evermann, J.: Scalable process discovery using map-reduce. IEEE Trans. Serv. Comput. **9**(3), 469–481 (2014)
10. Hernández, S., Ezpeleta, J., van Zelst, S., van der Aalst, W.: Assessing process discovery scalability in data intensive environments. In: Big Data Computing (BDC), pp. 99–104. IEEE (2015)
11. Jalali, A.: Exploring different aspects of users behaviours in the Dutch autonomous administrative authority through process cubes. Business Process Intelligence (BPI) Challenge (2016)

12. Joishi, J., Sureka, A.: Vishleshan: performance comparison and programming process mining algorithms in graph-oriented and relational database query languages. In: International Database Engineering & Applications Symposium, pp. 192–197 (2015)
13. Joishi, J., Sureka, A.: Graph or relational databases: a speed comparison for process mining algorithm. arXiv preprint arXiv:1701.00072 (2016)
14. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. Wiley, Hoboken (2011)
15. Lenharth, A., Nguyen, D., Pingali, K.: Parallel graph analytics. Commun. ACM **59**(5), 78–87 (2016)
16. Reguieg, H., Toumani, F., Motahari-Nezhad, H.R., Benatallah, B.: Using Mapreduce to scale events correlation discovery for business processes mining. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 279–284. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32885-5_22
17. van der Aalst, W.: Business process management: a comprehensive survey. Int. Sch. Res. Not. **2013**, 37 (2013). ISRN Software Engineering
18. van der Aalst, W.: Process Mining: Data Science in Action. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
19. van der Aalst, W.: A practitioner's guide to process mining: limitations of the directly-follows graph (2019)
20. van der Aalst, W.: Academic view: development of the process mining discipline. In: Process Mining in Action: Principles, Use Cases and Outlook (2020)
21. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-59432-2