Enabling efficient process mining on large data sets: realizing an in-database process mining operator

Remco Dijkman¹ · Juntao Gao² · Alifah Syamsiyah¹ ·
Boudewijn van Dongen¹ · Paul Grefen¹ · Arthur ter Hofstede³

Published online: 9 May 2019
© The Author(s) 2019

Abstract

Process mining can be used to analyze business processes based on logs of their execution. These execution logs are often obtained by querying a database and storing the results in a file. The mining itself is then done on the file, such that the data processing power of the database cannot be used after the log is extracted. Enabling process mining directly on a database therefore provides additional flexibility and efficiency. To help facilitate this, this paper formally defines a database operator that extracts the ‘directly follows’ relation—one of the relations that is at the heart of process mining—from an operational database. It defines the operator using the well-known relational algebra and formally proves equivalence properties of the operator that are useful for query optimization. Subsequently, it presents time-complexity properties of the operator. Finally, it presents an implementation of the operator as part of the H2 DBMS and demonstrates that this implementation extracts the ‘directly follows’ relation from a database with an arbitrary database structure within a fraction of a second; several orders of magnitude faster than is currently possible.

Keywords Process mining · Relational algebra · Formal methods · Database management system

1 Introduction

Process mining can be used to analyze business processes, based on logs of their execution. These execution logs contain the activities that were performed in the business process, the customer cases for which they were performed, the time at which they were performed, and additional information about the resources involved and the properties of the customer case. Many different analysis techniques have been

✉ Remco Dijkman
r.m.dijkman@tue.nl

Extended author information available on the last page of the article

developed (e.g., [1–3]) that can help to analyze the causal and temporal relations between the activities, to analyze whether or not the process has been executed as it is supposed to be executed, and to analyze performance properties of the process.

Table 1 shows a simple event log as it could be stored in a database. In practice, an event log contains thousands of rows (e.g., [4]) and may be distributed over multiple tables (such as e.g., in SAP [5]). Table 1 shows the activities that were performed in an organization, the (customer) case for which these activities were performed and the start and end time of the activities. Given such a table, it is important in process mining to be able to retrieve the ‘directly follows’ relation, because this is the basis for many process mining techniques, including the alpha algorithm [1] (and its variants), the inductive miner [6], the heuristic miner [2], and the fuzzy miner [3]. The directly follows relation retrieves the events that follow each other directly in some instance of process execution (i.e., case). The SQL query that retrieves this relation from relation *R* in Table 1 has two parts: one part that extracts all pairs of activities that follow each other either directly or indirectly and another, nested part that excludes activities that indirectly follow each other. Consequently, it is defined as follows:

```
SELECT DISTINCT a.activity, b.activity
FROM Log a, Log b WHERE a.case = b.case AND
    a.end_time < b.end_time AND
    NOT EXISTS SELECT * FROM Log c WHERE c.case = a.case
    AND
    a.end_time < c.end_time AND c.end_time < b.end_time;
```

The result of applying this query to *Log* in Table 1 is the table that contains the tuples (*A*, *B*), (*B*, *E*), (*A*, *D*), (*D*, *E*), (*A*, *C*), (*C*, *E*).

The query illustrates the challenges that arise when process mining directly on a database. First, it is inconvenient: even a conceptually simple process mining request like ‘retrieve all directly follows relations between events’ is relatively difficult to phrase in SQL. Second, it is inefficient, because it requires a nested query (i.e., the ‘NOT EXISTS’ part) and nested queries are known to cause performance problems [7,8]. This is discussed in detail in Sect. 4.

Table 1 Database table *Log* that contains a log

Case	Activity	Start_time	End_time	Case	Activity	Start_time	End_time
1	A	00:20	00:22	4	A	03:06	03:10
1	B	02:04	02:08	4	B	05:04	05:09
1	E	02:32	02:32	4	E	07:26	07:29
2	A	02:15	02:20	5	A	03:40	03:44
2	D	03:14	03:19	5	B	05:59	06:06
2	E	05:06	05:07	5	E	07:49	07:52
3	A	02:27	02:29	6	A	04:18	04:20
3	D	04:17	04:20	6	C	07:08	07:12
3	E	06:51	06:53	6	E	09:05	09:07

Fig. 1 Strategies for process mining on a database

Consequently, measures must be taken to make process mining feasible on relational databases. Figure 1 shows three possible strategies. Figure 1(i) shows a strategy that is frequently used in practice, in which a user constructs an SQL query to extract a log from the database, writes this log to disk, and then imports it in memory in a process mining tool for further processing. Consequently, the complete log must be read or written three times and in current practice there is manual work involved in getting the log to and from disk. It is easy to imagine a process mining tool that does not need intermediate storage to disk. Such a tool would only need to read the log once and would not require manual intervention to get the log to and from disk. However, it would require that the information is obtained from the database in the form of a log, which limits flexibility and performance in some cases as we will explain further on in the paper. Figure 1(ii) illustrates this strategy.

This paper proposes a third strategy, illustrated in Fig. 1(iii), in which the database management system supports a native ‘directly follows’ operator. This strategy combines the benefits of the first and the second approach, in that it does not require intermediate storage on disk, but does facilitates flexible and efficient querying. To realize this strategy, this paper defines the ‘directly follows’ operator in relational algebraic form. This definition can be used to implement the operator as part of a database management system, including query optimization measures. Finally, the paper presents an implementation of the operator as part of the H2 DBMS [9] as well as the effect that the operator has on the time performance of that system.

Against this background the remainder of this paper is structured as follows. Section 2 explains process mining and relational algebra as background for this paper. Section 3 presents a relational algebraic definition of the ‘directly follows’ operator. Section 4 shows the theoretical computational cost of executing this operator and the potential effects of query optimization with respect to this operator. Section 5 presents the implementation of the operator in the H2 DBMS and the time performance of that operator. Section 6 discusses alternative approaches for extracting the directly

Fig. 2 Example of a business process

follows relation from a database. Finally, Sect. 7 presents related work and Sect. 8 the conclusions.

2 Background

This section presents background information that is necessary to understand the goal of this paper: developing a ‘directly follows’ relation for process mining on a database: process mining, relational algebra, and how relational algebra is used in a database system.

2.1 Process mining

Process mining is the extraction of a process from an event log, where a process defines possible sequences of actions. Process mining is most commonly used in the area of business processes. In this context, a process defines the tasks that must be performed to achieve a certain business goal, such as approving a mortgage or processing a sales order.

Figure 2 shows a simple example of a business process that concerns processing an application. After the process starts, the application must be registered. After that, there is a choice between approving the application, rejecting the application, or postponing a decision. Finally, the result of the decision is filed and the process ends. Each application that is processed has to go through the process and a choice must be made to either approve, reject, or postpone the application. Consequently, identifying the tasks by their letters, each application will either go through the sequence $[A, B, E]$, the sequence $[A, C, E]$, or the sequence $[A, D, E]$. Assuming that the result of each task that is performed is stored in a database along with a timestamp and an identifier for the application on which it was performed, each application will leave a trace in the database, like the traces that are shown in Table 1. The challenge of process mining is to reconstruct the business process when execution traces such as the ones in Table 1 are available, but the process itself is not.

There exist a variety of algorithms for process mining, and in many of them the ‘directly follows’ relation plays an important role [1–3,6]. The ‘directly follows’ relation identifies pairs of events that ever follow each other directly (in time) in a log of execution traces. We will define the operator precisely in Sect. 3, but for

now let $x > y$ represent that event y follows event x directly in some execution trace in the log, such that in Table 1 we have the directly follows relations: $A > B, B > E, A > D, D > E, A > C, C > E$. The directly follows relation is the basis for mining the relations that form the process model. For example, the Alpha algorithm [1] has the rule that, if there exist x, y, z in the log, such that $x > y, x > z$, and neither $y > z$ nor $z > y$, a process model must be constructed in such a way that x occurs first, after which there is a choice between y and z . This is the case in the log in Table 1 for the event A and the events B, C, D , which indeed leads to a task A followed by a choice between B, C, D as in Fig. 2. While the precise manner in which the ‘directly follows’ relation is used to construct a process model from a log differs for different algorithms, it is the basis for constructing the process model in the alpha algorithm [1] (and its variants), the inductive miner [6], the heuristic miner [2], and the fuzzy miner [3]. Consequently, for in-database process mining it is important that the directly follows relation can be extracted efficiently.

2.2 Relational algebra

In this section we briefly define the basic relational algebraic operators. We refer to the many textbooks on databases (e.g., [10]) for more detailed definitions.

Definition 1 (*Attribute, schema, relation*) An attribute is a combination of a name and a domain of possible values. A schema is a set of attributes. Any two elements s_1, s_2 of schema s with $s_1 \neq s_2$ have different names. A relation is a combination of a schema and a set of tuples, where a tuple maps each attribute name from the schema to a value from the domain of that attribute.

For example, let C be the domain of case identifiers, E be the domain of activities, and T be the time domain. The relation of Table 1 has the schema $\{case : C, activity : E, start_time : T, end_time : T\}$ and the set of tuples $\{case \mapsto 1, activity \mapsto A, start_time \mapsto 00:20, end_time \mapsto 00:22\}, \dots\}$

In the remainder of this paper, we will also refer to R as the set of tuples of a relation R . For a relation R with a schema that defines an attribute with name a , we will use e_a to refer to the value of attribute a in element $e \in R$.

Definition 2 (*Relational algebra*) Let R, S be relations with schemas r, s . Furthermore, let a, b be attribute names, and ϕ a condition over attributes that is constructed using the logic operators conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), and the binary conditions ($>, \geq, =, \neq, \leq, <$) over attributes and attribute values. We define the usual relational algebra operators [10,11]:

- *Selection* $\sigma_\phi R = \{e | e \in R, \phi(e)\}$, where $\phi(e)$ is derived from ϕ by replacing each attribute name a by its value in tuple e : e_a . The schema of $\sigma_\phi R$ is r .
- *Projection* $\pi_{a,b,\dots} R = \{\{a \mapsto e_a, b \mapsto e_b, \dots\} | e \in R\}$. The schema of $\pi_{a,b,\dots} R$ is r restricted to the attributes with names $a, b, \dots$.
- *Renaming* $\rho_{a/b} R = R$. The schema of $\rho_{a/b} R$ is derived from r by replacing the name of attribute a by b , while the set of tuples in R does not change. In the remainder of this paper, we will also use $\rho_x R$ to represent prefixing all attribute names of R with x .

Table 2 Example relational algebra expressions

$\sigma_{case=1} Log$				$\pi_{case} Log$		$\rho_{case/id} Log$		
Case	Activity	Start_time	End_time	Case	Id	Activity	Start_time	End_time
1	A	00:20	00:22	1	1	A	00:20	00:22
1	B	02:04	02:08	2	1	B	02:04	02:08
1	E	02:32	02:32	3	1	E	02:32	02:32
				...	...	...	...	...

In addition, we define the usual set theoretic operators $R \cup S$, $R \cap S$, $R - S$. These operators have the usual set-theoretic interpretation, but they require that R and S have the same schema. We define the Cartesian product of R with schema $r = \{r_1 : R_1, r_2 : R_2, \dots, r_n : R_n\}$ and S with schema $s = \{s_1 : S_1, s_2 : S_2, \dots, s_m : S_m\}$ as $R \times S = \{\{r_1 \mapsto e_{r_1}, r_2 \mapsto e_{r_2}, \dots, r_n \mapsto e_{r_n}, s_1 \mapsto u_{s_1}, s_2 \mapsto u_{s_2}, \dots, s_m \mapsto u_{s_m}\} | e \in R, u \in S\}$. The schema of $R \times S$ is $r \cup s$.

Finally, a join operator is usually defined for the commonly used operator of joining tuples from two relations that have some property in common. The join operator is a shorthand for a combination of Cartesian product and selection: $R \bowtie_{\phi} S = \sigma_{\phi} R \times S$. The theta join is a special case of the join operator, in which ϕ has the form $a\theta b$ and θ is a binary condition ($>$, $\geq$, $=$, $\neq$, $\leq$, $<$) on the attributes a and b .

Table 2 shows examples of the selection, projection, and renaming operators, applied to the relation in Table 1.

2.3 Query optimization

Relational algebra is at the core of every relational database system. It is used to define the execution semantics of an SQL query and to define equivalence rules to determine which execution semantics (among a set of equivalent ones) is the most efficient to execute. There is a large number of proven relational algebraic equivalences that can be used to rewrite relational algebraic equations into the equivalent that is most efficient to execute [10, 11]. In the remainder of this paper, we use the following ones. Let R , S be relations, a , b , c be attributes, x , y be attribute values, ϕ , ψ be conditions, and θ be a binary condition ($>$, $\geq$, $=$, $\neq$, $\leq$, $<$). Then:

$$\sigma_{\phi \wedge \psi} R = \sigma_{\phi}(\sigma_{\psi} R) \quad (1)$$

$$\sigma_{\phi}(\sigma_{\psi} R) = \sigma_{\psi}(\sigma_{\phi} R) \quad (2)$$

$$R \bowtie_{\phi} S = S \bowtie_{\phi} R \quad (3)$$

$$(R \bowtie_{\phi} S) \bowtie_{\psi} T = R \bowtie_{\phi} (S \bowtie_{\psi} T) \quad (4)$$

$$\sigma_{\psi}(R \bowtie_{\phi} S) = (\sigma_{\psi} R) \bowtie_{\phi} S, \text{ if } \psi \text{ only has attributes from } R \quad (5)$$

$$\sigma_{\psi}(R - S) = (\sigma_{\psi} R) - (\sigma_{\psi} S) \quad (6)$$

$$\sigma_{a\theta x}(\rho_{b/a} R) = \rho_{b/a}(\sigma_{b\theta x} R) \quad (7)$$

$$\pi_a(\rho_{b/a} R) = \rho_{b/a}(\pi_b R) \quad (8)$$

$$\pi_{a,b,\dots}(\sigma_{\phi} R) = \sigma_{\phi}(\pi_{a,b,\dots} R), \text{ if } \phi \text{ only has attributes from } a, b, \dots \quad (9)$$

$$\pi_{a,b,\dots}(R \bowtie_{\phi} S) = (\pi_{a,\dots} R) \bowtie_{\phi} (\pi_{b,\dots} S), \text{ if } a, b, \dots \text{ can be split over } R, S \quad (10)$$

$$\pi_{a,b}(\pi_{b,c} R) = \pi_b R \quad (11)$$

$$\pi_{a,\dots}(\pi_{b,\dots} R) = \pi_{b,\dots}(\pi_{a,\dots} R) \quad (12)$$

$$\rho_{b/a}(R \bowtie_{b\theta c} S) = (\rho_{b/a} R) \bowtie_{a\theta c} S, \text{ if } a, b \text{ only in } R \quad (13)$$

$$\pi_{Rs} R = R, \text{ if } Rs \text{ contains all attributes from } R \quad (14)$$

$$\pi_{Rs}(R \bowtie_{\phi} S) = R, \text{ if } R \bowtie_{\phi} S \text{ includes each tuple of } R \\ \text{and } Rs \text{ contains all and only attributes from } R \quad (15)$$

$$(R - T) \bowtie_{a\theta b} S = R \bowtie_{a\theta b} S - T \bowtie_{a\theta b} S \quad (16)$$

In practice these equivalences are used to generate alternative formulas that lead to the same result, but represent alternative execution strategies. For example, $\sigma_{\psi}(\sigma_{\phi} R \times \sigma_{\theta} S)$ can be proven to be equivalent to $\sigma_{\psi \wedge \phi \wedge \theta}(R \times S)$. However, $\sigma_{\psi}(\sigma_{\phi} R \times \sigma_{\theta} S)$ represents the execution strategy in which we first execute the selections and then the Cartesian product, while $\sigma_{\psi \wedge \phi \wedge \theta}(R \times S)$ represents the execution strategy where we first execute the Cartesian product and then the selection. The first execution strategy is more efficient than the second, because it only requires the Cartesian product to be computed for a subset of R and S .

3 Relational algebra for process mining

This section defines the ‘directly follows’ relation as a relational algebraic operator. It also presents and proves equivalences for this operator that can be used for query optimization, analogous to the equivalences that are presented in Sect. 2.3.

3.1 Directly follows operator

The directly follows operator retrieves events that directly follow each other in some case. Let Log be a database table that contains log events and that has a column c , which represents the case identifier of an event, a column t , which represents the completion timestamp of an event, and further arbitrary columns $a_1, \dots$, which represent other event attributes. We define the directly follows operator, denoted $>_{c,t} Log$, as follows:

Definition 3 (*Directly follows operator*)

$$>_{c,t} Log = \left\{ \{ \downarrow c \mapsto r_c, \downarrow t \mapsto r_t, \downarrow a_1 \mapsto r_{a_1}, \dots, \uparrow c \mapsto s_c, \uparrow t \mapsto s_t, \uparrow a_1 \mapsto s_{a_1}, \dots \}, \right. \\ \left. \begin{array}{l} r \in Log, s \in Log, r_c = s_c, r_t < s_t, \\ \neg \exists q \in Log : r_c = q_c \wedge r_t < q_t \wedge q_t < s_t \end{array} \right\}$$

Table 3 Result of $>_{case, end_time} Log$

$\downarrow$ Case	$\downarrow$ Activity	$\downarrow$ Start_time	$\downarrow$ End_time	$\uparrow$ Case	$\uparrow$ Activity	$\uparrow$ Start_time	$\uparrow$ End_time
1	A	00:20	00:22	1	B	02:04	02:08
1	B	02:04	02:08	1	E	02:32	02:32
2	A	02:15	02:20	2	D	03:14	03:19
2	D	03:14	03:19	2	E	05:06	05:07
3	A	02:27	02:29	3	D	04:17	04:20
3	D	04:17	04:20	3	E	06:51	06:53
4	A	03:06	03:10	4	B	05:04	05:09
4	B	05:04	05:09	4	E	07:26	07:29
5	A	03:40	03:44	5	B	05:59	06:06
5	B	05:59	06:06	5	E	07:49	07:52
6	A	04:18	04:20	6	C	07:08	07:12
6	C	07:08	07:12	6	E	09:05	09:07

The definition renames existing attributes by prefixing them with $\downarrow$ and $\uparrow$. The resulting attributes refer to the event that precedes ($\downarrow$) the other event and the event that succeeds ($\uparrow$) the other event. Note that the prefixes $\downarrow$ and $\uparrow$ are arbitrary labels, we could also have used the labels ‘preceeding’ and ‘succeeding’, but prefer to use $\downarrow$ and $\uparrow$ for brevity. The directly follows operator takes all events r and s that have the same case identifier and that follow each other in time, and for which there is no event q that has the same case identifier and that happens in between r and s in time. For example, applying the operator to the example log from Table 1 (i.e., $>_{case, end_time} Log$) returns Table 3.

Similar to the way in which the join operator is defined in terms of other relational algebra operators, we can also define the ‘directly follows’ operator in terms of the traditional relational algebra operators.

Proposition 17 (Composite definition of the directly follows operator)

$$\begin{aligned}
 >_{c,t} Log = & \rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log \\
 & - \pi_{As}((\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} Log)
 \end{aligned}$$

where As is the set of attributes that are in $\rho_{\downarrow} Log$ or $\rho_{\uparrow} Log$.

Proof The correctness of this definition follows trivially from the definition of the weakly follows operator (Definition 3) and the definitions of the relational algebra operators (Definition 2). $\square$

The directly follows operator can both be used in an algorithm for process mining that is based on it (or on ‘footprints’ which are derived from it [12]) and for querying for process-mining related information. Some example queries include:

- The activities in which the amount of a loan is changed:

$$\pi_{\uparrow activity} \sigma_{\uparrow amount \neq \downarrow amount} >_{case, end_time} Log$$

This query first extracts the directly follows relation from a log. The resulting table contains a directly follows relation with, among others, columns for the preceeding ($\downarrow activity$) and the succeeding ($\uparrow activity$) activity, as well as the loan amount that is associated with the preceeding ($\downarrow amount$) and the succeeding ($\uparrow amount$) activity. Selecting the rows in which the succeeding amount differs from the preceeding amount identifies the activities that change the amount. Subsequently, we can project the activities for which this is the case.

- The resources that ever changed the amount of a loan:

$\pi_{\uparrow resource \sigma_{\uparrow amount \neq \downarrow amount} >_{case, end_time} Log}$

This is a alternative to the previous query, in which we project the resource that changes the amount rather than the activity in which the amount is changed.

- The two activities that precede a rejection:

$\sigma_{\uparrow activity = reject} (>_{\uparrow case, \uparrow end_time} >_{case, end_time} Log)$

This query first extracts the directly follows relation as usual, it then applies the directly follows relation again to the results, such that we do not only get the activities that succeed ($\uparrow activity$) another activity, but also the activities that succeed ($\uparrow \uparrow activity$) that activity.

3.2 Directly follows query optimization

To facilitate query optimization for the directly follows operator, we must define how it behaves with respect to the other operators and prove that behavior. We present this behavior as propositions along with their proofs. In each of these propositions, we use a, b, c, t as attributes (where—as convention—we use c to denote the case identifier attribute and t to denote the time attribute), θ as a binary operator from the set $\{>, \geq, =, \neq, \leq, <\}$, and x as a value.

The first proposition states that the directly follows operator and the selection operator commute, i.e., you can change the order in which they are executed. It holds for case attributes and event attributes, but not for other types of attributes. We define case attributes as attributes that keep the same value for all events in a case, from the moment that they get a value. An example of a case attribute is an attribute ‘customer name’ that gets a value when ‘register customer’ occurs and keeps that value for the remainder of the case. We define event attributes as attributes that have a value for at most one event in each case. An example of an event attribute is an attribute ‘amount’ that may have a value for activities labeled ‘determine if fine must be paid’. As this proposition only holds for case and event attributes, we can only use it for optimizing queries that involve a selection on a case or event attribute. Selections on attributes that may have different values for different activities with the same case identifier cannot be optimized.

Proposition 18 (directly follows and selection commute) $>_{c,t} \sigma_{a\theta x} Log$
 $= \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x} >_{c,t} Log$, if a is a case or event attribute.

Proof

$>_{c,t} \sigma_{a\theta x} Log$

$$\begin{aligned}
&= (\text{Proposition 17}) \\
&\quad \rho \downarrow (\sigma_{a\theta x} \text{Log}) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow (\sigma_{a\theta x} \text{Log}) \\
&\quad - \pi_{As}(((\rho \downarrow (\sigma_{a\theta x} \text{Log}) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow (\sigma_{a\theta x} \text{Log})) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \sigma_{a\theta x} \text{Log})) \\
&= (\text{Proposition 7}) \\
&\quad \sigma_{\downarrow a\theta x}(\rho \downarrow \text{Log}) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \sigma_{\uparrow a\theta x}(\rho \uparrow \text{Log}) \\
&\quad - \pi_{As}(((\sigma_{\downarrow a\theta x}(\rho \downarrow \text{Log}) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \sigma_{\uparrow a\theta x}(\rho \uparrow \text{Log})) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \\
&\sigma_{a\theta x} \text{Log})) \\
&= (\text{Propositions 1, 3, 5}) \\
&\quad \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x}(\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \\
&\quad - \pi_{As}(\sigma_{\downarrow a\theta x \wedge \uparrow a\theta x \wedge a\theta x}((\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \text{Log})) \\
&= (\text{assume } \downarrow a\theta x \wedge \uparrow a\theta x \Rightarrow a\theta x) \\
&\quad \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x}(\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \\
&\quad - \pi_{As}(\sigma_{\downarrow a\theta x \wedge \uparrow a\theta x}((\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \text{Log})) \\
&= (\text{Proposition 9}) \\
&\quad \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x}(\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \\
&\quad - \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x}(\pi_{As}((\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \text{Log})) \\
&= (\text{Proposition 6}) \\
&\quad \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x}(\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \\
&\quad - \pi_{As}((\rho \downarrow \text{Log} \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho \uparrow \text{Log}) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \text{Log})) \\
&= (\text{Proposition 17}) \\
&\quad \sigma_{\downarrow a\theta x \wedge \uparrow a\theta x} >_{c,t} \text{Log}
\end{aligned}$$

□

Note that Proposition 9 requires that the condition only contains attributes that are also projected (in this case $\downarrow a, \uparrow a$ must be in As). This condition is satisfied as per Proposition 17. Also note that the proof uses an assumption, which states that if any two events in a case have the same value for an attribute, all events for that case that are between these two (in time) must also have that value ($\downarrow a\theta x \wedge \uparrow a\theta x \Rightarrow a\theta x$). This assumption holds for case attributes and for event attributes, which are the scope of this proposition. As a consequence, the proposition can only be used to optimize a query on such attributes, but not on other attributes, such as the resource attribute.

The next proposition is a variant of the previous one, in which there is a selection condition on two attributes instead of an attribute and a value.

Proposition 19 (directly follows and selection commute 2) $>_{c,t} \sigma_{a\theta b} \text{Log} = \sigma_{\downarrow a\theta \downarrow b \wedge \downarrow a\theta \uparrow b \wedge \uparrow a\theta \downarrow b \wedge \uparrow a\theta \uparrow b} >_{c,t} \text{Log}$, if a, b are case or event attributes.

Proof Analogous to the proof of Proposition 18

□

To prove that directly follows and projection commute, we first need to prove that projection and set minus commute, because the set minus operator is an important part

of the definition of the directly follows operator. However, for the general case it is not true that projection and set minus commute. A counter example is easily constructed. Let $R = \{ \{a \mapsto 1, b \mapsto 2\} \}$ and $S = \{ \{a \mapsto 1, b \mapsto 3\} \}$. For these relations it does not hold that $\pi_a(R - S) = (\pi_a R) - (\pi_a S)$. However, we can prove this proposition for the special case that S is a subset of R and a uniquely identifies tuples in R . Since these conditions are satisfied for the directly follows operator, it is sufficient to prove the proposition under these conditions.

Proposition 20 (projection and restricted set minus commute) $\pi_a(R - S) = (\pi_a R) - (\pi_a S)$, if $S \subseteq R$ and a uniquely identifies each tuple in R .

Proof This equivalence is proven by observing that $S \subseteq R$ implies that a non-surjective injective function $f : S \rightarrow R$ exists that matches each tuple s in S to a unique tuple r in R . The fact that a uniquely identifies tuples in R (and also in S , because S is a subset of R) implies that f is completely determined by the values of tuples in a , i.e., the values of attributes other than a have no consequence for f . Therefore, projecting R and S onto a does not change the tuple mapping defined by f .

Now, looking at the left side of Proposition 20, calculating the projection over the difference, means removing the attributes not in a from the selected tuples in R that are not in the range of f . Looking at the right side, calculating the difference over the projections, means removing the attributes not in a from R and S (which does not affect f) and then selecting the tuples in R that are not in the range of f . These two are equivalent. $\square$

Proposition 21 (directly follows and restricted projection commute) $>_{c,t} \pi_{c,t,a} Log = \pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a} >_{c,t} Log$

Proof

$$\begin{aligned}
&>_{c,t} \pi_{c,t,a} Log \\
&= (\text{Proposition 17}) \\
&\quad \rho_{\downarrow}(\pi_{c,t,a} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow}(\pi_{c,t,a} Log) \\
&\quad - \pi_{As}((\rho_{\downarrow}(\pi_{c,t,a} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow}(\pi_{c,t,a} Log)) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \pi_{c,t,a} Log) \\
&= (\text{Proposition 8}) \\
&\quad \pi_{\downarrow c, \downarrow t, \downarrow a}(\rho_{\downarrow} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \pi_{\uparrow c, \uparrow t, \uparrow a}(\rho_{\uparrow} Log) \\
&\quad - \pi_{As}((\pi_{\downarrow c, \downarrow t, \downarrow a}(\rho_{\downarrow} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \pi_{\uparrow c, \uparrow t, \uparrow a}(\rho_{\uparrow} Log)) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \pi_{c,t,a} Log) \\
&= (\text{Proposition 10}) \\
&\quad \pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a}(\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \\
&\quad - \pi_{As}(\pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a, c, t, a}((\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} Log)) \\
&= (\text{Propositions 11, 12}) \\
&\quad \pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a}(\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \\
&\quad - \pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a}(\pi_{As}((\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} Log)) \\
&= (\text{Proposition 20}) \\
&\quad \pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a}((\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \\
&\quad - \pi_{As}((\rho_{\downarrow} Log \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} Log) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} Log))
\end{aligned}$$

= (Proposition 17)

$$\pi_{\downarrow c, \downarrow t, \downarrow a, \uparrow c, \uparrow t, \uparrow a} >_{c,t} Log$$

□

The next proposition states that the directly follows relation and the theta join (see Definition 2) commute. This proposition relies on the fact that the directly follows operator duplicates each attribute a into an attribute $\uparrow a$ and an attribute $\downarrow a$. Table 3 illustrates this. However, if the case, activity and start time attribute uniquely identify an event, then there is no need to duplicate the end time attribute or any other attribute, and the duplicate attributes just take up space. This redundancy can easily be fixed later on with a projection operator and in future work additional efficiency may be achieved by avoiding this redundancy altogether.

Proposition 22 (directly follows and theta join commute) $>_{c,t} (R \bowtie_{a\theta b} S) = (>_{c,t} R) \bowtie_{\downarrow a\theta b} S \bowtie_{\uparrow a\theta b} S$, if each tuple from R is combined with a tuple in S .

Proof

$$\begin{aligned}
 &>_{c,t} (R \bowtie_{a\theta b} S) \\
 &= (\text{Proposition 17}) \\
 &\quad \rho_{\downarrow} (R \bowtie_{a\theta b} S) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} (R \bowtie_{a\theta b} S) \\
 &\quad - \pi_{As} ((\rho_{\downarrow} (R \bowtie_{a\theta b} S) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} (R \bowtie_{a\theta b} S)) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \\
 &\quad (R \bowtie_{a\theta b} S)) \\
 &= (\text{Proposition 13}) \\
 &\quad (\rho_{\downarrow} R) \bowtie_{\downarrow a\theta b} S \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} (\rho_{\uparrow} R) \bowtie_{\uparrow a\theta b} S \\
 &\quad - \pi_{As} (((\rho_{\downarrow} R) \bowtie_{\downarrow a\theta b} S) \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} (\rho_{\uparrow} R) \bowtie_{\uparrow a\theta b} S) \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} \\
 &\quad (R \bowtie_{a\theta b} S)) \\
 &= (\text{Propositions 3, 4, 10, 14, 15}) \\
 &\quad \rho_{\downarrow} R \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} R \bowtie_{\downarrow a\theta b} S \bowtie_{\uparrow a\theta b} S \\
 &\quad - \pi_{As} (\rho_{\downarrow} R \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} R \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} R) \bowtie_{\downarrow a\theta b} S \bowtie_{\uparrow a\theta b} S \\
 &= (\text{Proposition 16}) \\
 &\quad (\rho_{\downarrow} R \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} R \\
 &\quad - \pi_{As} (\rho_{\downarrow} R \bowtie_{\downarrow t < \uparrow t \wedge \downarrow c = \uparrow c} \rho_{\uparrow} R \bowtie_{\downarrow t < t \wedge t < \uparrow t \wedge \downarrow c = c} R)) \bowtie_{\downarrow a\theta b} S \bowtie_{\uparrow a\theta b} S \\
 &= (\text{Proposition 17}) \\
 &\quad (>_{c,t} R) \bowtie_{\downarrow a\theta b} S \bowtie_{\uparrow a\theta b} S
 \end{aligned}$$

□

The properties that are proven in this way help to define different queries with equivalent results, but different execution costs, such that the most efficient query among a set of equivalent queries can be established.

Table 4 Execution costs of process mining on a database

Execution strategy	Order of costs (disk blocks)
Classical process mining	
With intermediate storage	$3 \cdot B$
With database connection	B
With database operator	
Native operator	B
Composite operator	B up to B^3

4 Theoretical execution cost

We determine the computational cost of executing the directly follows operation and compare the costs for the different execution strategies that we identified in the introduction. We also determine the effect of query optimization on the directly follows operator.

4.1 Cost of computing the directly follows relation

The execution cost of a database operation is typically defined in terms of the number of disk blocks read or written, because reading from or writing to disk are the most expensive database operations. In line with the strategies for process mining on a database that are presented in Fig. 1, Table 4 shows four execution strategies with their costs. Note that the ‘with database operator’ strategy from Fig. 1 is split up into two alternatives. The first alternative requires a database management system that contains an implementation of a dedicated directly follows operator. The second alternative can be applied in any database management system and uses a nested query to extract the directly follows relation. The execution cost is presented as an order of magnitude, measured in terms of the number of disk blocks B that must be read or written. The number of disk blocks is linear with the number of events in the log and depends on the number of bytes needed to store an event and the number of bytes per disk block. These measures assume that the complete log fits into memory. If the log does not fit into memory, the measures would worsen. However, the best operators in terms of execution cost (classical process mining with database connection and the native database operator proposed here) would remain the best and the worst operator (the composite database operator) would remain the worst.

Process mining with intermediate storage requires that the log is read and written three times: once to query the database for the log, once to store the log to disk, and once to load the log in the process mining tool. Consequently, the complexity is $3 \cdot B$. Process mining directly on a database requires that the log is read only once. Subsequent processing can be done in memory.

While process mining with a database connection can be fast, in many usage scenarios more flexible querying capabilities are needed, which can benefit from access to all SQL operators. In particular, this is the case if the process mining log must be aggre-

Fig. 3 Execution costs of process mining using the composite operator

gated from multiple database tables, or if—rather than the event log itself—process mining related information must be extracted.

For such usage scenarios, the ‘directly follows’ relation must be extracted directly from the database. Such an operator reads the log from disk once and computes the directly follows relation in memory. Consequently, it has linear cost. For databases that do not have the native ‘directly follows’ operator proposed in this paper, the operator can be emulated using the composite formula from Proposition 17. The drawback of this formula is that it requires that the intermediate results from both sides of the minus operator are stored, after which the minus operator can be applied. While this is no problem as long as the intermediate results fit into memory, the costs become prohibitive once the intermediate results must be written to disk. We will show the precise practical implications of this problem in Sect. 5.

On a theoretical level, the problem is illustrated in Fig. 3. This figure shows that the problem arises when the number of events in the log is high, relative to the number of cases. The mechanism that causes this is easy to derive from Proposition 17, which shows that the intermediate results that must be stored are the pairs of events that directly or indirectly follow each other in some case (the left side and right side of the minus operator). Consequently, if there are many events per case, this number is high (cubic over the number of events per case in the right-hand side of the minus operator).

The precise calculation can be performed as follows. Let V be the number of cases in the log, N be the number of events, F be the block size (i.e., the number of tuples/events that fit in a single disk block), $B_{Log} = \frac{N}{F}$ be the number of disk blocks required to store the log, and M be the total memory size in blocks. Note that the cost of a block nested join (or minus) operator on two relations R and S that take B_R and B_S disk blocks (with $B_R \leq B_S$), is equal to $B_R + B_S$ when one of the two relations

fits in memory, and equal to $B_R + \frac{B_S}{M} \cdot B_R$ otherwise [13]. The cost is split up into five components:

1. The cost of the first join is denoted as B_{join_1} . This equals B_{Log} if the log fits into memory and $B_{Log} + \frac{B_{Log}}{M} \cdot B_{Log}$ otherwise. Note that this join appears twice, but that it only needs to be computed once.
2. The cost of storing the results of the first join to disk is denoted as B_{result_1} . This equals 0 if the result fits in memory. Otherwise, the number of tuples in the result, which we denote as $|t_1|$, equals the number of pairs of events that directly or indirectly follow each other in some case: $V \cdot (\frac{N}{V} \cdot \frac{N}{V} - 1) / 2$ on average. This fits into $\frac{|t_1| \cdot 2}{F}$ disk blocks (times 2 because each tuple in the result is a pair of tuples from the original).
3. The cost of the second join is denoted as B_{join_2} . This equals 0 if the original log fits into memory. Otherwise, the cost equals $B_{Log} + \frac{|t_1| \cdot 2}{F} / M \cdot B_{Log}$.
4. The cost of storing the result of the second join to disk is denoted as B_{join_2} . This equals 0 if the result fits into memory. Otherwise, the number of tuples in the result, which we denote as $|t_2|$, equals the number of pairs of events that indirectly follow each other. This equals the number of pairs of events $|t_1|$ that directly or indirectly follow each other minus the number of pairs of events that directly follow each other: $V \cdot (\frac{N}{V} - 1)$ on average. This fits into $\frac{|t_2| \cdot 2}{F}$ disk blocks (times 2 because each tuple in the result is a triple of tuples from the original and then reduced to a pair by projection).
5. The cost of the minus operator is denoted as B_{minus} . This equals 0 if the result of the second join fits into memory. Otherwise, it equals $B_{result_1} + \frac{B_{result_1}}{M} \cdot B_{result_2}$.

To generate Fig. 3 we used a tuple size of 80 bytes, a 4 GB buffer size, and a block size of 50, such that there is a total memory size of 1 million blocks. The figure shows two ‘thresholds’ in the computational cost. These thresholds are crossed when a particular intermediate result no longer fits into memory.

The order of the cost can be determined more easily. The order of the cost is determined by the cost of the set minus, because this incorporates both intermediate results, which are typically much larger than the original log. Therefore, the order of the cost of computing the intermediate results are $\frac{N^2}{V} / F$. The total order of cost is then obtained by filling these costs out in the right-hand side of the formula for computing the cost of the set minus, which yields: $(\frac{N^2}{V} / F / M) \cdot \frac{N^2}{V} / F$. If we let M be large enough to contain the log itself, but not the intermediate results (i.e., we set $M = \frac{N}{F}$), this can be simplified as: $\frac{N^3}{V^2} / F$.

Summarizing, the execution cost of retrieving a directly follows relation directly from a database can be as low as retrieving it from a process mining tool, if the database supports the native ‘directly follows’ operator proposed in this paper and the process mining tool supports on-database process mining. However, as long as a native ‘directly follows’ operator does not exist, the execution costs increase to third order polynomial cost if the average number of events per case is high (i.e., if intermediate results do not fit into memory anymore).

Table 5 Execution cost orders of different execution sequences

Execution sequence	In memory (blocks)	On disk (blocks)
$\sigma > Log$	B	$B \cdot \left(\frac{N}{V}\right)^2$
$> \sigma Log$	$B \cdot S$	$Q \cdot B \cdot \left(\frac{N}{V}\right)^2$

4.2 The effect of query optimization

An advantage of in-database process mining is that it enables query optimization. Query optimization, using the rewrite rules that are defined in Sect. 3.2 can greatly reduce the cost of executing a query. As an example, we show the cost of executing the query $>_{c,t} \sigma_{a\theta x} Log$ and the equivalent query $\sigma_{\downarrow a\theta x \wedge \uparrow a\theta x} >_{c,t} Log$. These costs decrease at least linearly with the fraction of events that match the selection criteria. Let Q be that fraction. Table 5 shows the different execution situations that can arise. It is possible to either first derive the directly follows relation and then do the selection, or vice versa. It is also possible that the intermediate results fit in memory, or that they must be stored on disk. If the results fit in memory (and the table is indexed with respect to the variable on which the selection is done), then the execution costs are simply the cost of reading the log, or the part that matches the selection criteria, into memory once. If the intermediate results do not fit into memory, the order of the execution cost is $\frac{N^3}{V^2}/F$ as explained in the previous section. Remembering that $B = \frac{N}{F}$ leads to the formulas that are shown in the table.

The most dramatic increase occurs in the situation where, if the selection is done first, the intermediate results fit into memory, while, if the selection is done last, the intermediate results do *not* fit into memory. In practice this is likely to be the case, because the selection can greatly reduce the number of events that are considered. For example, for a log with $N = 10,000$ events over $V = 500$ cases, with a block size of $F = 50$ and a selection fraction $Q = 0.10$, the order of the cost increases from 20 to 8×10^4 according to the formulas from Table 5. The actual computed costs increase (the same order of magnitude) from 21 (plus one, because we need to read one disk block to load the index that is used to optimize the selection) to 9.5×10^4 using the formulas from the previous section.

This shows that the way in which a query that includes the directly follows operator is executed greatly influences the execution cost. Query optimizers, which are parameterized with equivalence relations from Sect. 3.2 and the cost calculation functions from Sect. 4.1, can automatically determine the optimal execution strategy for a particular query.

5 Practical execution time performance

We implemented the native ‘directly follows’ operator into the H2 Database Management System [9]. The implementation is publicly available through GitHub,¹ as

¹ <https://github.com/rmdijkman/h2processmining>.

are the process mining logs that we used to evaluate the implementation and that are referenced in the table. We also created a plugin for the ProM process mining tool [14] that enables process mining directly on an H2 Database. This implementation is publicly available through the ProM packaging system and as open source.²

As process mining logs, we used the logs from the BPI Challenges of the past years, except for the BPI 2016 challenge, which contained click-data rather than process data. For some BPI Challenges, multiple logs were made available, in which case we took the largest log. We implemented the operator as a function on a table, such that for the table R that contains at least case identifiers, activity labels, and completion timestamps in the first three columns, $\text{FOLLOWS}(R)$ returns the directly follows relation of the table.

Subsequently, we evaluated the implementation of the native operator, by storing the logs from practice into an H2 database and extracting the directly follows relation, both using the implementation of the native operator and using the composite operator in the form of the following SQL query (which is the same as the one outlined in Sect. 1).

```
SELECT DISTINCT a.activity, b.activity
FROM Log a, Log b WHERE a.case = b.case AND
      a.end_time < b.end_time AND
      NOT EXISTS SELECT * FROM Log c WHERE c.case = a.case
AND
      a.end_time < c.end_time AND c.end_time < b.end_time;
```

All evaluations were run on a server with two Intel Xeon processors running at 2.4 GHz, 64Gb of memory and a database cache size of 8GB. Clearly, the time performance will differ a lot depending on the hardware that is used. However, we mainly want to use the experiments to illustrate the practical implications of the time complexity that is specified in Table 4 and the time complexity will not differ depending on the hardware that is used.

Table 6 shows the results of the comparison. The table shows some descriptive statistics of the logs and the time performance of the different approaches that are illustrated in Fig. 1, of which the theoretical performance is specified in Table 4. Note that for classical process mining with intermediate storage, the time for storing the log on disk and retrieving it from disk (i.e., the ‘extract’ and ‘load’ steps from Fig. 1) is not included. We left that time out, because it involves manual actions, such as clicking menu buttons and choosing a file name, which will clearly be the dominant factor in the time performance and are irrelevant from the perspective of computational performance. If a fully automated means of extracting the log from the database is required, using classical mining with a database connection or one of the two database operators makes more sense.

As expected based on Table 4, the performance of the first three operators (excluding disk storage for classical process mining) has the same order or magnitude. An analysis on artificial data shows similar results [20]. While these results might suggest that for that reason the choice of technique for extracting the directly follows relation does not

² <https://svn.win.tue.nl/repos/prom/Packages/InDatabasePreprocessing/>.

Table 6 Time performance of deriving a directly follows relation

Log	Cases	Events	Event types	Performance (ms) classical mining intermediate storage ^a	Database connection	Database operator native operator	Composite operator
[4]	1143	150,291	360	64	67	206	919×10^3
[15]	13,087	262,200	36	72	64	95	41×10^3
[16]	42,134	65,533	13	20	23	35	2×10^3
[17]	46,616	466,737	39	113	71	139	29×10^3
[18]	34,015	59,083	254	15	16	30	13×10^3
[19]	31,509	561,671	26	125	171	169	25×10^3

^aThe time for storing on and retrieving from disk is not included

matter, this is not true. It is indeed true that the performance of the three techniques does not differ. However, the three techniques are suitable for very different applications:

- Classical mining with intermediate storage is most suitable for once-off process mining, in which the data that must be extracted from the database has been carefully selected beforehand, because the extraction of the data from the database is labor intensive and often requires administrator privileges.
- Classical mining with a database connection is most suitable for continuous process mining, in which a dedicated database table exists that contains the event log and that is continuously being updated. It cannot be used for ad-hoc process mining from a database, because it only works when a dedicated database table with an event log exists. Examples of such an approach are the DB-XES approach [21] and the approach that is proposed by the professional process mining tool Celonis [22].
- Process mining with a native directly follows operator is most suitable for exploratory process mining, in which no dedicated table exists that contains the event log, or in which data from different tables must be combined before process mining takes place. It allows for fast ad-hoc process mining even when a database does not contain a dedicated table with an event log. An example of such an approach is the Event Cube approach [23].

As also expected based on Table 4, the performance of the composite operator is much worse than that of the other operators. It takes two to three orders of magnitude longer to compute the directly follows relation for the composite operator. In the worst case the computation time goes from milliseconds to over an hour. Moreover, the performance of the composite operator is heavily affected by the number of cases and events in the log, while the performance of the other operators is hardly affected by the number of cases and events. This result is in line with the theoretical complexity of the composite operator that is presented in Sect. 4.1, where we theorized that the order of the execution cost is polynomial over the fraction of events per case.

Note that Table 6 only shows the performance of extracting the directly follows relation from the database. It does not include the time that it takes to mine the process model based on the directly follows relation, because once the directly follows relation is derived, mining a process model from it, is the same for each of the approaches. For completeness, Table 7 shows the time performance of mining the process model from the directly follows relation for the different approaches. The table also presents the complexity of the mined models in terms of their nodes and edges. It shows that the time performance is indeed the same for the different approaches. We attribute the minor differences in time performance to noise due to, for example, background processor activity.

In an experiment, we explored the effect of the theoretical computational complexity on the computation time in more detail. To do so, we took the log of the BPI 2011 challenge [4] and varied the number of events per case to measure the effect on the computation time. We varied the number of events per case by randomly removing event types until a desired number of events per case is reached, and also by artificially adding event types. Event types were artificially added by taking an existing event type and copying its properties in terms of the probability that that event type follows and is

Table 7 Time performance of mining from a directly follows relation

Log	Nodes	Edges	Performance (ms) classical mining intermediate storage	Database connection	Database operator native operator	Composite operator
[4]	648	149×10^3	237	238	280	291
[15]	39	249×10^3	12	12	13	26
[16]	33	58×10^3	12	10	11	1
[17]	45	420×10^3	5	3	1	2
[18]	286	58×10^3	29	31	26	27
[19]	43	530×10^3	4	4	2	1

Fig. 4 Computational time with different number events per case

followed by another event type. Figure 4 shows the result of the experiment. It shows the computational time required to extract the weakly follows relation from an event log, required both by the native and the composite operator for a different number of events per case. The figure also shows trend lines. For the native operator the trend line is linear and for the composite operator the trend line is polynomial. Due to the way in which the event logs were generated, it was hard to generate event logs with between 50 and 100 events per case. Regardless, the experiment clearly shows that the computational time of the composite operator is polynomial over the number of events per case, which is in line with the theoretical computational cost.

Summarizing, the native operator returns query results within a fraction of a second. As expected, it performs several orders of magnitude better than the composite operator. Moreover, unlike the composite operator, the native operator is hardly affected by properties of the log for current industry-strength logs. This makes the native operator uniquely suited for interactive in-database exploration of an event log.

6 Alternatives for querying the directly follows relation

On a high-level, we identified three different strategies to doing process mining on a database, as illustrated in Sect. 1 (Fig. 1).

These strategies can be compared along a number of criteria:

1. Computational performance;

2. Flexibility with which a log can be constructed for mining;
3. Flexibility with which the database can be queried for process mining related information; and
4. Effort needed to adapt a DBMS.

Where, by ‘flexibility’ we mean the ease with which queries can be constructed and adapted, for extracting a process mining log (point 2) or for extracting other information (point 3).

All three proposed strategies behave similarly with respect to the first two criteria: the computational performance is linear over the number of events in the log, and an SQL query can be defined that constructs a log file from (multiple tables) in a database. This SQL query can be adapted at will and running a process mining tool on the resulting log should be easy, especially if the process mining tool has a direct database connection and constructs the log file internally.

Consequently, the difference between the strategies primarily becomes a trade-off between the desire to ask process mining-related queries directly on the database and the associated need to adapt the DBMS. Queries that involve the directly follows operator are difficult to specify and perform badly when using the composite operator, while they are easy to specify and perform well when using the native operator. However, being able to use the native operator requires that such an operator is implemented in the DBMS, which clearly is a time-consuming task, especially with respect to the implementation of the query optimization rules that are discussed in Sect. 3.

To avoid the problems of the composite operator, one might be tempted to define a query without a ‘NOT EXISTS’ part, which exploits a time-order of the events to derive the weakly follows relation. For example, assuming that the database management system can return the row number of each tuple, a relation can be ordered by case identifier and time, and subsequently the following SQL query would return the ‘directly follows’ relation for event logs in which no events occur at the same time.

```
SELECT DISTINCT a.activity, b.activity
FROM Log a, Log b WHERE a.case = b.case AND
      a.ROW_NUMBER = b.ROW_NUMBER - 1;
```

However, such a query fails if two events have the same timestamp. For example, for the log (1, A, 0:01), (1, B, 0:02), (1, C, 0:02), (1, D, 0:03), this query would not work, because the query would return (A, B), (B, C), (C, D), while it should return (A, B), (A, C), (B, D), (C, D). What makes matters worse, is that the result would be non-deterministic, since the result that would be returned, depends on the particular order of events that have the same timestamp, which is not specified. This problem may be more or less common in a log, depending on the time-granularity at which events are logged, which—in practice—is often (too) coarse. For example, it may happen that only the date of the event is logged and not the time. This problem is also known as the ‘timestamp challenge’ [24].

Table 8 shows that this problem clearly exists for the BPI logs. The experiments were run on a computer with an Intel i7 processor at 4.2 GHz with 16 GB of RAM, an SSD drive and a database cache size of 8 GB. The table shows the computational performance of the native operator that is proposed in this paper and the operator that works on the ‘sorted’ log. The performance is comparable for the two operators.

Table 8 Performance of the ‘sorted’ directly follows operator

Log	Cases	Events	Performance native operator (ms)	Performance sorted operator (ms)	#Relations	#Relations sorted operator
[4]	1143	150,291	141	141	16, 546	372
[15]	13,087	262,200	172	187	344	263
[16]	42,134	65,533	16	47	90	90
[17]	46,616	466,737	203	297	890	779
[18]	34,015	59,083	16	47	7765	1042
[19]	31,509	561,671	250	343	449	278

The table also shows the number of directly follows relations that is returned by the operator for the ‘sorted’ log and the number of directly follows relations that exists in the log. This clearly shows that the operator for the ‘sorted’ log misses many directly follows relations that do exist in the log.

7 Related work

By defining an operator for efficiently extracting the ‘directly follows’ relation between events from a database, this paper has its basis in a long history of papers that focus on optimizing database operations. In particular, it is related to papers that focus on optimizing database operations for data mining purposes [25,26], of which SAP HANA [27] is a recent development. The idea of proposing domain-specific database operators has also been applied in other domains, such as spatio-temporal databases [28] and scientific databases [29].

In related work on DB-XES [21], which extends the previous Relational XES [30], we developed a standard data warehouse structure to store process logs for efficient process mining. DB-XES also provides dedicated insert operations that insert pre-processed data in data warehouse tables for each inserted log event. This pre-processed data can be queried efficiently, which provides performance benefits that are similar to the performance benefits of the technique described in this paper [31]. However, the technique described in this paper allows for querying any database structure in a computationally fast manner, while the DB-XES data warehouse structures are fixed. In this respect, the functionality that is described in this paper is complementary to that of DB-XES, and a DB-XES data warehouse can even be filled using techniques described in this paper.

The Celonis process mining tool, which is being used in practice, has a similar approach as DB-XES [22]. It creates a data warehouse in SAP HANA that is being kept up-to-date with the operational database and on which process mining is facilitated. Like DB-XES this approach will facilitate fast process mining on the pre-defined data warehouse structure. Also like DB-XES, to facilitate computationally fast process mining on any database structure, additional facilities, such as the one proposed in

this paper, are needed, and the functionality described in this paper can be used to fill the Celonis data warehouse.

By presenting a ‘directly follows’ operator, the primary goal of this paper is to support computationally efficient process mining on a database. There exist some papers that deal with the computational complexity of the process mining algorithms themselves [32,33]. Also, in a research agenda for process mining the computational complexity and memory usage of process mining algorithms have been identified as important topics [34]. However, this paper focuses on a step that precedes the process mining itself: flexibly querying a database to investigate which information is useful for process mining.

More database-related work from the area of process mining comes from shaping data warehouses specifically for process mining [35,36]. There also exists work that focuses on the extraction of logs from a database [5,37], including conceptual problems that arise when lifting the lower-level data and events from a database to a process level [38,39]. Also, several query languages have been developed specifically for databases that contain business processes and related information [40–42] as well as dedicated querying facilities for other forms of process mining [43–45].

8 Conclusions

This paper presents a first step towards in-database process mining. In particular, it defines a relational algebraic operator to extract the ‘directly follows’ relation from a log that is stored in a relational database, possibly distributed over multiple tables. The paper presents and proves relational algebraic properties of this operator, in particular that the operator commutes with the selection, projection, and theta join. These equivalence relations can be used for query optimization. Also, the paper presents and proves formulas to estimate the computational cost of the operator. These formulas can be used in combination with the equivalence relations to determine the most efficient execution strategy for a query.

The operator is primarily meant to facilitate exploratory process mining in the style of online analytical processing. For these purposes, it has the clear benefit that it facilitates efficient process mining, even on databases in which the ‘event log’ or ‘directly follows’ relation is not stored trivially in a single database table, but must be constructed with a complex SQL query.

The operator was implemented as part of the H2 Database Management System, as a function on a table that can be part of an SQL query. An evaluation with logs from practice shows that the operator returns the ‘directly follows’ relation of a log within a fraction of a second. If the ‘directly follows’ relation is extracted from the log using an SQL query with only standard SQL, the query takes several orders of magnitude longer, from a minute to even over an hour, depending on properties of the log. Consequently, the native implementation of the operator makes a database system uniquely suited for in-database process mining.

At this moment, query optimization measures are not implemented for the operator. This remains for future work. Also, the query optimization equations have some limitations, in particular with respect to the conditions under which they hold. These

limitations restrict the possibilities for query optimization. The extent to which these theoretical limitations impact practical performance of the operator must be investigated in future work and, if possible, mitigated.

The work on in-database process mining can be further extended, by defining and implementing additional operators. Currently, it is hard to work with ‘cases’ in an SQL query, because a log table has a row for each event that happens rather than each case that completes. For that reason, we aim to develop operators that can easily aggregate properties from an event level to a case level. For example, operators for easily extracting variants of execution sequences could be envisioned, as well as operators for extracting properties of resource involvement and time characteristics of cases. In addition, we aim to extend the algebraic optimization rules, in particular because we expect that other process mining operators, such as strictly follows, independence, and exclusion - even though they can be defined in terms of the directly follows operator - can be better optimized if they get their own optimization rules. We also aim to do various case studies in which we explore the process-mining related queries that process analysts tend to define, such that we can develop support for those queries and end up with a feature-complete Process Mining DBMS. In parallel, we are working on the DB-XES approach to in-database process mining. This approach keeps separate (cache) tables for process mining-related queries. When the process mining query is requested, this requires a once-off effort to fill the separate tables. From that moment on, these tables are kept up-to-date with specialized insert statements.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. van der Aalst, W., Weijters, A., Maruster, L.: Workflow mining: discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.* **16**(9), 1128–1142 (2004)
2. Weijters, A., van der Aalst, W., De Medeiros, A.: Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Technical Report WP **166**, 1–34 (2006)
3. Günther, C., van der Aalst, W.: Fuzzy mining: adaptive process simplification based on multi-perspective metrics. In: *Proceedings of BPM*, pp. 328–343 (2007)
4. van Dongen, B.: Real-life event logs—hospital log (2011). <https://doi.org/10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffc54>
5. Ingvaldsen, J.E., Gulla, J.A.: Preprocessing support for large scale process mining of sap transactions. In: *Proceedings of BPM*, pp. 30–41 (2007)
6. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs: a constructive approach. In: Colom, J.M., Desel, J. (eds.) *Proceeding of ATPN*, pp. 311–329. Springer, New York (2013)
7. Kim, W.: On optimizing an SQL-like nested query. *ACM Trans. Database Syst.* **7**(3), 443–469 (1982)
8. Chaudhuri, S.: An overview of query optimization in relational systems. In: *Proceedings of PODS*, pp. 34–43. New York (1998)
9. H2: H2 Database Engine (2017). <http://h2database.com>. Accessed 31 Aug 2017
10. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases: The Logical Level*. Addison-Wesley, Boston (1995)

11. Sagiv, Y., Yannakakis, M.: Equivalence among relational expressions with the union and difference operation. In: Proceedings of VLDB, pp. 535–548 (1978)
12. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, New York (2011)
13. Blasgen, M.W., Eswaran, K.P.: Storage and access in relational data bases. IBM Syst. J. **16**(4), 363–377 (1977)
14. van Dongen, B., de Medeiros, A., Verbeek, E., Weijters, T., van der Aalst, W.: The prom framework: a new era in process mining tool support. In: Proceedings of ATPN, pp. 444–454. Springer, New York (2005)
15. van Dongen, B.: BPI challenge 2012 (2012). <https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f>
16. Steeman, W.: BPI challenge 2013, incidents (2013). <https://doi.org/10.4121/500573e6-acc-4b0c-9576-aa5468b10cee>
17. van Dongen, B.: BPI challenge 2014: activity log for incidents (2014). <https://doi.org/10.4121/uuid:86977bac-f874-49cf-8337-80f26bf5d2ef>
18. van Dongen, B.: BPI challenge 2015 municipality 5 (2015). <https://doi.org/10.4121/uuid:b32c6fe5-f212-4286-9774-58dd53511cf8>
19. van Dongen, B.: BPI challenge 2017—offer log (2017). <https://doi.org/10.4121/uuid:7e326e7e-8b93-4701-8860-71213edf0fbe>
20. Syamsiyah, A., van Dongen, B.F., Dijkman, R.M.: A native operator for process discovery. In: Proceedings of DEXA, pp. 292–300. Springer, New York (2018)
21. Syamsiyah, A., van Dongen, B., van der Aalst, W.: Accelerating process mining using relational databases. In: Proceedings of SIMPDA, pp. 137–141 (2016)
22. Celonis: SAP Process Mining by Celonis. <https://www.sap.com/developer/showcases/process-mining-by-celonis.html> (2019). Accessed 22 Feb 2019
23. Ribeiro, J., Weijters, T.: Event cube: another perspective on business processes. In: Proceedings of OTM, pp. 274–283. Springer, New York (2011)
24. van der Aalst, W., et al.: Process mining manifesto. In: Proceedings of BPM Workshops, pp. 169–194 (2012)
25. Chen, M.S., Han, J., Yu, P.S.: Data mining: an overview from a database perspective. IEEE Trans. Knowl. Data Eng. **8**(6), 866–883 (1996)
26. Agrawal, R., Imielinski, T., Swami, A.: Database mining: a performance perspective. IEEE Trans. Knowl. Data Eng. **5**(6), 914–925 (1993)
27. Färber, F., Cha, S.K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W.: SAP HANA database: data management for modern business applications. SIGMOD Rec. **40**(4), 45–51 (2012)
28. Abraham, T., Roddick, J.F.: Survey of spatio-temporal databases. GeoInformatica **3**(1), 61–99 (1999)
29. Cudre-Mauroux, P., Kimura, H., Lim, K.T., Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Wang, D.L., Balazinska, M., Becla, J., DeWitt, D., Heath, B., Maier, D., Madden, S., Patel, J., Stonebraker, M., Zdonik, S.: A demonstration of SciDB: a science-oriented DBMS. Proc. VLDB Endow. **2**(2), 1534–1537 (2009)
30. van Dongen, B.F., Shabani, S.: Relational XES: data management for process mining. In: Proceedings of the CAiSE Forum, pp. 169–176 (2015)
31. Syamsiyah, A., van Dongen, B.F., van der Aalst, W.M.P.: DB-XES: enabling process discovery in the large. In: Data-Driven Process Discovery and Analysis, pp. 53–77. Springer, New York (2018)
32. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Proceedings of CAiSE, pp. 270–285 (2012)
33. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Proceedings of BPM, pp. 375–383 (2007)
34. van der Aalst, W., Weijters, A.: Process mining: a research agenda. Comput. Ind. **53**(3), 231–244 (2004)
35. Eder, J., Olivotto, G.E., Gruber, W.: A data warehouse for workflow logs. In: Proceedings of EDCIS, pp. 1–15 (2002)
36. zur Mühlen, M.: Process-driven management information systems—combining data warehouses and workflow technology. In: Proceedings of ICECR, pp. 550–566 (2001)
37. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Connecting databases with process mining: a meta model and toolset. In: Proceedings of BPMDS 2016, pp. 231–249 (2016)
38. Chirigati, F., Freire, J.: Towards integrating workflow and database provenance. In: Proceedings of IPAW, pp. 11–23. Springer, New York (2012)

39. van der Aalst, W.: Extracting event data from databases to unleash process mining. In: BPM—Driving Innovation in a Digital World, pp. 105–128 (2015)
40. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Process mining on databases: unearthing historical data from redo logs. In: Proceedings of BPM, pp. 296–309 (2017)
41. Momotko, M., Subieta, K.: Process query language: a way to make workflow processes more flexible. In: Proceedings of ADBIS, pp. 306–321 (2004)
42. Beheshti, S.M.R., Benatallah, B., Motahari-Nezhad, H.R., Sakr, S.: A query language for analyzing business processes execution. In: Proceedings of BPM, pp. 281–297 (2011)
43. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: Proceedings of CAiSE, pp. 290–305. Springer, New York (2016)
44. Schönig, S., Di Ciccio, C., Maggi, F.M., Mendling, J.: Discovery of multi-perspective declarative process models. In: Proceeding of ICSOC, pp. 87–103. Springer, New York (2016)
45. Syamsiyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Discovering social networks instantly: moving process mining computations to the database and data entry time. In: Proceedings of BPMDS, pp. 51–67. Springer, New York (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Remco Dijkman¹ · Juntao Gao² · Alifah Syamsiyah¹ ·
Boudewijn van Dongen¹ · Paul Grefen¹ · Arthur ter Hofstede³

Juntao Gao
gjt@nepu.edu.cn

Alifah Syamsiyah
a.syamsiyah@tue.nl

Boudewijn van Dongen
b.f.v.dongen@tue.nl

Paul Grefen
p.w.p.j.grefen@tue.nl

Arthur ter Hofstede
a.terhofstede@qut.edu.au

¹ Eindhoven University of Technology, Eindhoven, The Netherlands

² Northeast Petroleum University, Daqing, China

³ Queensland University of Technology, Brisbane, Australia