Process Mining on Databases: Unearthing Historical Data from Redo Logs

E. González López de Murillas[1,2], W.M.P van der Aalst[1], and H.A. Reijers[1]

[1] Department of Mathematics and Computer Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[2] Lexmark Enterprise Software, Gooimeer 12, 1411DE Naarden, The Netherlands
{e.gonzalez,w.m.p.v.d.aalst,h.a.reijers}@tue.nl

Abstract. Process Mining techniques rely on the existence of event data. However, in many cases it is far from trivial to obtain such event data. Considerable efforts may need to be spent on making IT systems record historic data at all. But even if such records are available, it may not be possible to derive an event log for the case notion one is interested in, i.e., correlating events to form process instances may be challenging. This paper proposes an approach that exploits a commonly available and versatile source of data, i.e. *database redo logs*. Such logs record the writing operations performed in a general-purpose database for a range of objects, which constitute a collection of *events*. By using the relations between objects as specified in the associated data model, it is possible to turn such events into an event log for a wide range of case types. The resulting logs can be analyzed using existing process mining techniques.

Keywords: *Process Mining*, *Database*, *Redo log*, *historical data*, *trace creation*, *transitive relations*, *data model.*

1 Introduction

Process mining heavily depends on event data. But to get proper data, it is either necessary (i) to build a customized storage facility oneself or (ii) to rely on data that is already stored by existing IT systems. The former approach requires extensive knowledge of the application domain and a potentially hybrid technology landscape to create a facility that records all possible events that are related to a pre-defined notion of a case. This is potentially costly and not very flexible.

The second approach requires the transformation of available data – which is not specifically stored for process mining purposes – into an event log. Approaches exist to accomplish this for the data stored in and generated by SAP systems [6, 13, 14], EDI messages [4], and ERP databases in general [16]. Some efforts for generalization have been made here, as can be seen in [15]: XESame is a tool that allows transforming database records into events, traces and logs, prior definition of the mappings between database elements and log concepts like timestamp, case, activity and resource. Nonetheless, the drawback of these approaches is that they are restricted to the specific IT system or data format that they are developed for.

Artifact-centric approaches are more generic and also fit within the second strategy [3, 5, 8–11]. These techniques provide a way to get insights into the contents of a database, showing the life-cycle of objects without presenting *data convergence* (one event is related to multiple cases) and *data divergence* (several events of the same type are related to a single case) issues, as happens in classical log extraction. However, both issues are not fully solved in the artifact field. In fact, one could argue that the key problems are evaded by restricting the contents of a single artifact, in order to avoid data divergence, and hiding data convergence in the discovered relations between them.

The technique explained in this paper, based on the ideas introduced in [1], also relies on the use of existing data. However, it exploits so-called *redo logs* that several Data Base Management Systems (DBMSs), like Oracle RDBMS and MySQL, maintain for data integrity and recovery reasons. This source of data has the potential to create a full historic view on what has happened during the handling a wide range of data objects. Note that by simply looking at the regular content of a DBMS, one cannot see which events led to its current state. Fortunately, redo logs provide an opportunity to learn about the historical evolution of data on the basis of a generic-purpose data source, exactly tuned to the purpose of the process mining analysis one wishes to perform.

Redo logs already contain a list of events, but the challenge is how to correlate these events to create the traces one is interested in. In this paper, we explain how to create a trace on the basis of a configurable concept of a case (i.e. the process instance), exploiting the relations expressed in the data model of the DBMS in question. The result is a log which represents a specific point of view on the objects in the database, including the stages of their historical evolution and the causal relations between them.

The analysis technique presented in this paper can be used, whenever redo logs are available, as an alternative to building a specific recording facility. The approach is also generic, in the sense that it can be used to extract data from a technology that is used by a wide variety of organizations. Additionally, it is a viable alternative for artifact-centric approaches, since it allows for a much richer behavior discovery due its incorporation of the data model to infer causal relations between events. Finally, the nature of the extracted logs (events with unique IDs and availability of data schemas) opens the door to developing new discovery techniques that could exploit the additional information that databases provide, in order to solve data convergence and divergence issues.

The paper is structured as follows. Section 2 presents a walkthrough of the approach on a simple example to explain the various phases of creating an event log. Section 3 provides the formalization of the important concepts that this work builds upon. Section 4 describes the tool that implements these concepts to generate an event log. Section 5 provides an example case to show how the technique can be flexibly applied to solve a range of business questions. Finally, Section 6 presents the conclusion and future work.

2 Walkthrough

The aim of this work is to analyze database redo logs, which can be seen as a list of modifications performed on the database content, so that we use these to generate event logs. These event logs will be used to perform process mining analyses like process discovery, conformance checking, and process enhancement. To explain the idea of redo log analysis, a step by step walk-through using a simple case is performed in this section.

Let us consider as an example a database that stores information on a portal for selling concert tickets. At this point, we will focus on three tables only: *customer*, *booking*, and *ticket*. These tables contain information about the customers of the portal, the bookings made by these customers, and the tickets being booked by them.

Table 1: Fragment of a redo log: each line corresponds to the occurrence of an event

#	Time + Op + Table	Redo	Undo
1	2014-11-27 15:57:08.0 + INSERT + CUSTOMER	`insert into "SAMPLEDB". "CUSTOMER" ("ID", "NAME", "ADDRESS", "BIRTH_DATE") values (’17299’, ’Name1’, ’Address1’, TO_DATE( ’01-AUG-06’, ’DD-MON-RR’));`	`delete from "SAMPLEDB". "CUSTOMER" where "ID" = ’17299’ and "NAME" = ’Name1’ and "ADDRESS" = ’Address1’ and "BIRTH_DATE" = TO_DATE( ’01-AUG-06’, ’DD-MON-RR’) and ROWID = ’1’;`
2	2014-11-27 16:07:02.0 + UPDATE + CUSTOMER	`update "SAMPLEDB". "CUSTOMER" set "NAME" = ’Name2’ where "NAME" = ’Name1’ and ROWID = ’1’;`	`update "SAMPLEDB". "CUSTOMER" set "NAME" = ’Name1’ where "NAME" = ’Name2’ and ROWID = ’1’;`
3	2014-11-27 16:07:16.0 + INSERT + BOOKING	`insert into "SAMPLEDB". "BOOKING" ("ID", "CUSTOMER_ID") values (’36846’, ’17299’);`	`delete from "SAMPLEDB". "BOOKING" where "ID" = ’36846’ and "CUSTOMER_ID" = ’17299’ and ROWID = ’2’;`
4	2014-11-27 16:07:16.0 + UPDATE+ TICKET	`update "SAMPLEDB". "TICKET" set "BOOKING_ID" = ’36846’ where "BOOKING_ID" IS NULL and ROWID = ’3’;`	`update "SAMPLEDB". "TICKET" set "BOOKING_ID" = NULL where "BOOKING_ID" = ’36846’ and ROWID = ’3’;`
5	2014-11-27 16:07:17.0 + INSERT + BOOKING	`insert into "SAMPLEDB". "BOOKING" ("ID", "CUSTOMER_ID") values (’36876’, ’17299’);`	`delete from "SAMPLEDB". "BOOKING" where "ID" = ’36876’ and "CUSTOMER_ID" = ’17299’ and ROWID = ’4’;`
6	2014-11-27 16:07:17.0 + TICKET + UPDATE	`update "SAMPLEDB". "TICKET" set "ID" = ’36876’ where "BOOKING_ID" IS NULL and ROWID = ’5’;`	`update "SAMPLEDB". "TICKET" set "ID" = NULL where "BOOKING_ID" = ’36876’ and ROWID = ’5’;`

2.1 Event Extraction

An example of a fragment of a redo log is shown in Table 1. This fragment contains six changes made to the records of the three tables. Each of these events indicates (a) the time at which it occurred, (b) the operation performed and on which table this was done, (c) an SQL sentence to redo the change, and (d) another SQL sentence to undo it. We claim that these basic fields provide enough information to reconstruct the state of the database at any intermediate stage. Also, they allow us to perform an in-depth analysis to detect patterns on the behavior of the process or processes that rely on the support by this database.

The first thing we need to do is to transform each of the records in the redo log in Table 1 to an event that we can manipulate. To do so, it is necessary to split the contents of redo and undo sentences into different attributes. Table 2 shows the attributes for each event extracted from the *redo* and *undo* columns in Table 1. The rows with the symbol = for “Value after event” indicate that the value for an “Attribute name” did not change after the event. Also, the values between braces {} in the “Value before event” column were extracted not from the present event but from previous ones. This is, for instance, the case in the second event: It is an update on the name of the customer, the record of which was already inserted in the first event in the table. Finally, the values between parentheses () identify the ones that could not be

Table 2: Fragment of a redo log: each line corresponds to the occurrence of an event

#	Attribute name	Value after event	Value before event	C
1	Customer:id	17299	-	4
	Customer:name	Name1	-	4
	Customer:address	Address1	-	4
	Customer:birth_date	01-AUG-06	-	4
	RowID	=	1	-
2	Customer:id	=	{17299}	1
	Customer:name	Name2	Name1	4
	Customer:address	=	{Address1}	1
	Customer:birth_date	=	{01-AUG-06}	1
	RowID	=	1	-
3	Booking:id	36846	-	4
	Booking:customer_id	17299	-	4
	RowID	=	2	-
4	Ticket:booking_id	36846	NULL	3
	Ticket:id	=	(317132)	1
	Ticket:belongs_to	=	(172935)	1
	Ticket:for_concert	=	(1277)	1
	RowID	=	3	-
5	Booking:id	36876	-	4
	Booking:customer_id	17299	-	4
	RowID	=	4	-
6	Ticket:booking_id	36876	NULL	3
	Ticket:id	=	(317435)	1
	Ticket:belongs_to	=	(173238)	1
	Ticket:for_concert	=	(1277)	1
	RowID	=	5	-

extracted directly from the redo log, but only from the database content itself. This is because those columns were not modified in that event, as is the case in events 4 and 6, where only the field “ticket:booking_id” is updated. Therefore, the other values remain equal and it is not necessary to specify them in the *redo* and *undo* sentences. It is still necessary to identify on which row of the database the change must be applied. To do so, the redo log system provides a *RowID* identifier to find it. In addition to it, an extra column *C* has been added, which encodes a numeric vector for each event representing which columns had its value (1) not modified, (2) changed from a value to NULL, (3) from NULL to a value or (4) inserted/updated.

2.2 Exploiting the Data Model

After extracting the events from the redo log, the next step required is to obtain the data model from the database. This will be a main ingredient used to correlate events. Finding these correlations will tell us which sets of events can be grouped into traces to finally build an event log. Obtaining the data model involves querying the tables, columns, and keys defined in the database schema. Figure 1 shows the extracted data model. For the selected tables *customer*, *booking* and *ticket*, we see that two key relations exist between them: (a) *(booking_customer_fk, customer_pk)* and (b) *(ticket_booking_fk, booking_pk)*. This means that we must use the first pair of keys (a) to correlate *customer* and *booking* events, and the second pair of keys (b) to correlate *booking* and *ticket* events. Both pairs (a) and (b) must be used to correlate the events of the three tables.

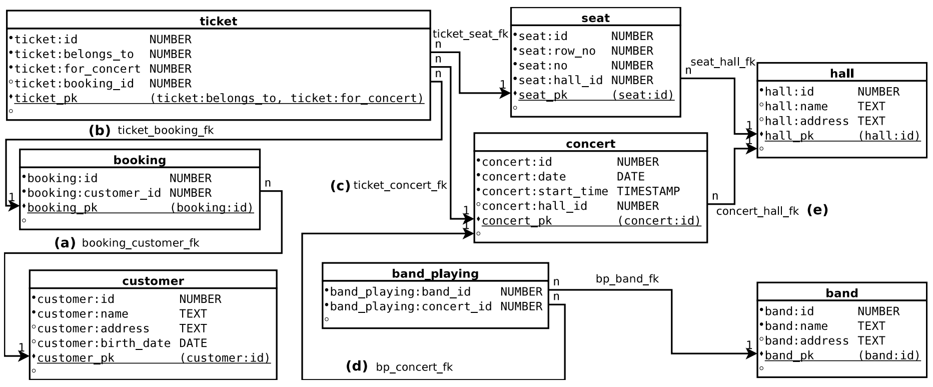


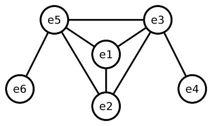
Fig. 1: Database schema for the *Ticket selling* example.

When using pairs of primary and foreign keys, we can consider the attributes referred by them as *equivalent* for our purposes, i.e. relating to the same event. We will do so to actually relate events that belong to different tables, but use different column names (attributes in the events) to store the same values. Therefore, attributes *customer:id* and *booking:customer_id* are considered to be equivalent, and the same can be said of the pair *booking:id* and *ticket:booking_id*. Then, using these equivalences and observing the *value after event* column in Table 2, we see that every event is related to at least one other event by means of some attribute value. That is the case, for instance, for events 1 and 2, given that they share the same value for the attribute *customer:id*. Also, event 3 is related to events 1, 2 and 5, sharing the same value for the attributes *customer:id* and *booking:customer_id*, and to event 4, sharing the same value for the attributes *booking:id* and *ticket:booking_id*. Event 6 is related to event 5 by means of the attributes *booking:id* and *ticket:booking_id* as well. A graph in which events are the vertexes and edges show relations between them would look like the one in Figure 2a. This graph helps to understand the structure of a trace we wish to extract. What needs to be taken care of still is that it contains events of two *different* ticket bookings (events 3-4 and events 5-6), which is behavior that we would like to see separately.

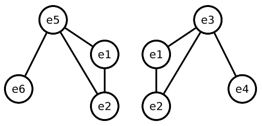
2.3 Process Instance Identification

To decide which events go into which traces, it is necessary to define which view is desired to obtain on a process. In this case, let us assume that it is interesting to see how tickets are booked by customers. Using relations Fig 1.a and Fig 1.b, we can say that individual traces should contain behavior for the same user, booking and ticket: our *case* notion. Applying that notion, we see that events 1 and 2 point to a single customer. Events 3 and 4 point to a single pair of booking and ticket, and still relate to the customer of events 1 and 2. However, events 5 and 6 represent a different pair of booking-ticket but do relate to the customer in events 1 and 2. Therefore, this leads to two separated but not disjoint graphs in which events 1 and 2 are common, as observed in Figure 2b. Each of these graphs represent the structure of a trace for a separate case in our event log.

From the initial change log $CL = \langle e1, e2, e3, e4, e5, e6 \rangle$ we obtain two traces $t1 = (e1, e2, e3, e4)$ and $t2 = (e1, e2, e5, e6)$ following the above reasoning. In this case, applying a discovery algorithm to these two traces will result in a very simple sequential model. In Section 5, we will present samples of questions and answers to understand the process extending the same technique to a more extensive dataset and choosing different views on the data.



(a) Connected graph of related events.



(b) Graphs for the two final traces.

Fig. 2: Traces as graphs of related events (events $e1$ to $e6$ refer to Table 2.

This way we can find patterns and obtain interesting insights regarding the observed behavior. What follows now is the formal basis that precisely captures the ideas discussed so far.

3 Formalizations

The basic idea to use redo logs for the creation of an event log has been introduced in the previous section. This section provides a formal description of the underlying notions. Some of the notation used in this part originate from [1]. First, we need to define the *data model.*

Definition 1 (Data Model) *Assume V to be some universe of values. A data model is a tuple DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) such that*

- *C is a set of class names,*
- *A is a set of attribute names,*
- $classAttr \in C \rightarrow \mathcal{P}(A)$ *is a function mapping each class onto a set of attribute names.* A_c *is a shorthand denoting the set of attributes of class* $c \in C$*, i.e.,* $A_c = classAttr(c)$,
- $val \in A \rightarrow \mathcal{P}(V)$ *is a function mapping each attribute onto a set of values.* $V_a = val(a)$ *is a shorthand denoting the set of possible values of attribute* $a \in A$,
- *PK is a set of primary key names,*
- *FK is a set of foreign key names,*
- *PK and FK are disjoint sets, that is* $PK \cap FK = \emptyset$*. To facilitate further definitions, the shorthand K is introduced, which represents the set of all keys:* $K = PK \cup FK$,
- $keyClass \in K \rightarrow C$ *is a function mapping each key name to a class.* K_c *is a shorthand denoting the set of keys of class* $c \in C$ *such that* $K_c = \{k \in K \mid keyClass(k) = c\}$,
- $keyRel \in FK \rightarrow PK$ *is a function mapping each foreign key onto a primary key,*
- $keyAttr \in K \rightarrow \mathcal{P}(A)$ *is a function mapping each key onto a set of attributes, such that* $\forall k \in K : keyAttr(k) \subseteq A_{keyClass(k)}$,
- *refAttr* $\in FK \times A \not\rightarrow A$ *is a function mapping each pair of a foreign key and an attribute onto an attribute from the corresponding primary key. That is,* $\forall k \in FK : \forall a, a' \in keyAttr(k) : (refAttr(k, a) \in keyAttr(keyRel(k)) \wedge (refAttr(k, a) = refAttr(k, a') \implies a = a')$.

Definition 2 (Notations) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model.*

- $M^{DM} = \{map \in A \not\rightarrow V \mid \forall a \in dom(map) : map(a) \in V_a\}$ *is the set of mappings,*
- $O^{DM} = \{(c, map) \in C \times M^{DM} \mid dom(map) = classAttr(c)\}$ *is the set of all possible objects of DM.*

A data model defines the structure of objects in a database. Such objects can belong to different classes and varied relations can exist between them. A collection of possible objects constitutes an *object model.*

Definition 3 (Object Model) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model. An object model of DM is a set* $OM \subseteq O^{DM}$ *of objects.* $\mathcal{U}^{OM}(DM) = \mathcal{P}(O^{DM})$ *is the set of all object models of CM.*

The objects in an *object model* must have a specific structure according to a certain *data model*. Also, some rules apply to ensure that the *object model* respects the rules stated by the *data model*. This is covered by the notion of a *valid object model*.

Definition 4 (Valid Object Model) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model.* $VOM \subseteq \mathcal{U}^{OM}(DM)$ *is the set of valid object models. We say that* $OM \in VOM$ *if the following requirements hold:*

– $\forall (c, map) \in OM : (\forall k \in K_c \cap FK : (\exists (c', map') \in OM :$
  $keyClass(keyRel(k)) = c' \wedge (\forall a \in keyAttr(k) :$
  $map(a) = map'(refAttr(k, a)))))$*, i.e., referenced objects must exist,*
– $\forall (c, map), (c, map') \in OM : (\forall k \in K_c \cap PK : ((\forall a \in keyAttr(k) :$
  $map(a) = map'(a)) \implies map = map'))$*, i.e., PK and UK values must be unique.*

Different vendors offer *DataBase Management Systems* (DBMSs) like Oracle RDBMS, Microsoft's SQL server, MySQL, etc. All of them allow to store *objects* according to a specific *data model*. The work of this paper focuses on the analysis of redo logs, being conceptually independent of the specific implementation. These redo logs can contain information about changes done either on the data or the structure of the database. We will focus here on the changes *on data*, which include insertions of new objects, updates of objects, and deletions. Each of these changes represent an event, which has a type (Definition 5) and mappings for the value of objects before and after the change (Definition 6). Also, the combination of an event and a specific time stamp represents an *event occurrence* (Definition 7).

Definition 5 (Event Types) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model and VOM the set of valid object models.* $ET = ET_{add} \cup ET_{upd} \cup ET_{del}$ *is the set of event types composed of the following pairwise disjoint sets:*

– $ET_{add} = \{(\oplus, c) \mid c \in C\}$ *are the event types for adding objects,*
– $ET_{upd} = \{(\oslash, c) \mid c \in C\}$ *are the event types for updating objects,*
– $ET_{del} = \{(\ominus, c) \mid c \in C\}$ *are the event types for deleting objects.*

Definition 6 (Events) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, VOM the set of valid object models and* $map_{null} \in \{\emptyset\} \to V$ *a function with the empty set as domain.* $E = E_{add} \cup E_{upd} \cup E_{del}$ *is the set of events composed of the following pairwise disjoint sets:*

– $E_{add} = \{((\oplus, c), map_{old}, map_{new})) \mid (c, map_{new}) \in O^{DM} \wedge map_{old} = map_{null}\}$
– $E_{upd} = \{((\oslash, c), map_{old}, map_{new})) \mid (c, map_{old}) \in O^{DM} \wedge (c, map_{new}) \in O^{DM}\}$
– $E_{del} = \{((\ominus, c), map_{old}, map_{new})) \mid (c, map_{old}) \in O^{DM} \wedge map_{new} = map_{null}\}$

Definition 7 (Event Occurrence, Change Log) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, VOM the set of valid object models and E the set of events. Assume some universe of timestamps TS.* $eo = (e, ts) \in E \times TS$ *is an event occurrence.* $EO(DM, E) = E \times TS$ *is the set of all possible event occurrences. A change log* $CL = \langle eo_1, eo_2, ..., eo_n \rangle$ *is a sequence of event occurrences such that time is non-decreasing, i.e.,* $CL = \langle eo_1, eo_2, ..., eo_n \rangle \in (EO(DM, E))^*$ *and* $ts_i \leq ts_j$ *for any* $eo_i = (e_i, ts_i)$ *and* $eo_j = (e_j, ts_j)$ *with* $1 \leq i < j \leq n$*.*

Definition 8 (Effect of an Event) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, VOM the set of valid object models and E the set of events. For any two object models* $OM_1 \in VOM$ *and* $OM_2 \in VOM$ *and event occurrence* $eo = (((op,c),map_{old},map_{new}),ts) \in EO(DM,E)$*, we denote* $OM_1 \overset{eo}{\rightarrow} OM_2$ *if and only if* $OM_2 = \{(d,map) \in OM_1 \mid map \neq map_{old} \vee op = \oplus\} \cup \{(c,map_{new}) \mid op \neq \ominus\}$*.*

Event e is permissible in object model OM, notation $OM \overset{e}{\rightarrow}$*, if and only if there exists an OM' such that* $OM \overset{e}{\rightarrow} OM'$*. If this is not the case, we denote* $OM \overset{e}{\not\rightarrow}$*, i.e., e is not permissible in OM. If an event is not permissible, it will fail and the object model will remain unchanged. Relation* $\overset{e}{\Rightarrow}$ *denotes the effect of event e. It is the smallest relation such that (a)* $OM \overset{e}{\Rightarrow} OM'$ *if* $OM \overset{e}{\rightarrow} OM'$ *and (b)* $OM \overset{e}{\Rightarrow} OM$ *if* $OM \overset{e}{\not\rightarrow}$*.*

When, in Definition 8, we say that $OM_1 \overset{eo}{\rightarrow} OM_2$, it means that OM_2 must contain (1) all the objects in OM_1 except the one that the event occurrence eo refers to, and (2), the object inserted if eo is an insertion or the modified object if it is an update. If eo is a deletion, the object is not included in OM_2.

Definition 9 (Effect of a Change Log) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, VOM the set of valid object models, E the set of events and* $OM_0 \in VOM$ *the initial valid object model. Let* $CL = \langle e_1, e_2, ..., e_n \rangle \in (EO(DM,E))^*$ *be a change log. There exist object models* $OM_1, OM_2, ..., OM_n \in VOM$ *such that* $OM_0 \overset{e_1}{\Rightarrow} OM_1 \overset{e_2}{\Rightarrow} OM_2 ... \overset{e_n}{\Rightarrow} OM_n$*. Hence, change log CL results in object model* OM_n *when starting in* OM_0*. This is denoted by* $OM_0 \overset{CL}{\Rightarrow} OM_n$*.*

Definitions 1 to 9 establish the basis to understand data models, events and change logs, among other concepts. However, a mechanism to relate events to each other to build traces is still missing. For that purpose, and as one of the main contributions of this paper, the concept of *trace id pattern* is introduced. Then, subsequent definitions will be presented to show the trace building technique.
In a classical approach, the notion of case id is given by an attribute common to all the events in a trace. If traces do not exist yet, they can be created grouping events by the value of the selected attribute. However, in our setting, we have a collection of events of different classes with disjoint attribute sets. This means that it will be impossible to find a single common attribute to be used as case id. A *trace id pattern* substitutes the idea of a case id attribute for a set of attributes and keys. By its use, it becomes possible to find a common set of attributes between events of different classes using foreign-primary key relations. This relations establish the equivalence between pairs of attributes. The example presented in Section 2.2 illustrates this idea using the pair of keys *customer_pk* and *booking_customer_fk* to set the equivalence between the attributes *customer:id* and *booking:customer_id*. Each *trace id pattern* configures a view of a process to focus on, determining also which is the central element of the view, called *root*. This *root* element will determine the start event for each trace and will allow, in further steps, to build traces according to such a view.

Definition 10 (Trace ID Pattern) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model. A Trace ID Pattern on DM is a tuple* $TP_{DM} = (TPA, TPK, ROOT)$ *such that*
– $TPA \subseteq A$ *is a subset of the attributes in the data model,*
– $TPK \subseteq K$ *is a subset of the keys in the data model,*

– *$ROOT \in TPK$ is a key of the data model.*

To find the equivalence between different *attribute names*, we define a *canonical mapping* (Definition 11), i.e., a way to assign a common name to all the attributes linked, directly or transitively, through foreign-primary key relations. To show a simple example, the canonical mapping of the attribute *booking:customer_id* would be *customer:id* since both are linked through the foreign-primary pair of keys *booking_customer_fk* and *customer_pk*.

Definition 11 (Canonical Mapping) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model and TP = (TPA, TPK, ROOT) a trace id pattern on DM. A canonical mapping $canon \in A \to A$ is a function mapping each attribute to its canonical attribute such that:*

$$canon(a) = \begin{cases} a & \text{if } a \notin \bigcup_{k \in FK} keyAttr(k), \\ canon(refAttr(fk,a)) & \text{if } a \in \bigcup_{k \in FK} keyAttr(k). \\ with fk \in \{k \in FK \mid a \in keyAttr(k)\} & \end{cases}$$

The combination of a *trace id pattern* and the *canonical mapping* results in the *canonical pattern attribute set*, i.e., the set of canonical attributes for each of the elements (keys and attributes) configured in the *trace id pattern*. This allows us to obtain a minimum set of attributes to identify traces, avoiding the presence of attributes which, despite being different, map to the same canonical form.

Definition 12 (Canonical Pattern Attribute Set) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model and TP = (TPA, TPK, ROOT) a trace id pattern on DM. The canonical pattern attribute set of TP is a set $CPAS^{TP} = \{canon(a) \mid a \in TPA \cup (\bigcup_{k \in TPK} keyAttr(k))\}$.*

Definition 13 (Notations II) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, TS some universe of timestamps and $eo = (((op,c), map_{old}, map_{new}), ts)$ an event occurrence. We define the following shorthands for event occurrences:*

– *$eventClass(eo) = c$ denotes the class of an event occurrence,*
– *$time(eo) = ts$ denotes the timestamp of an event occurrence,*
– *$mapVal_{eo}$ denotes the right mapping function to obtain the values of the event in an event occurrence such that*

$$mapVal_{eo} = \begin{cases} map_{new} & \text{if } op \in \{\oplus, \oslash\}, \\ map_{old} & \text{if } op = \ominus \end{cases}$$

Each trace we want to build represents a process instance. However, a process instance, which is formed by *event occurrences*, needs to comply with some rules to guarantee that they actually represent a meaningful and valid trace.The first thing we need to accomplish is to build a set of traces that do not contain too much behavior according to the selected view (*trace id pattern*). We will call this the set of *well-formed traces*.

Definition 14 (Traces, Well-Formed Traces) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, $TP_{DM} = (TPA, TPK, ROOT)$ a trace id pattern on DM, CL a change log of event occurrences and $T = \{t \in \mathcal{P}(\{eo_i \mid 1 \leq i \leq n\})\}$ the set of possible traces on that change log.*
$WFT^{CL} \subseteq T$ is the set of well-formed traces such that $WFT^{CL} = \{t \in T \mid (\forall eo_i, eo_j \in t : \forall a_i \in A_{eventClass(eo_i)}, a_j \in A_{eventClass(eo_j)} : (a_i \in dom(mapVal_{eo_i}) \wedge$

$a_j \in dom(mapVal_{eo_j}) \wedge \{canon(a_i), canon(a_j)\} \subseteq CPAS^{TP} \wedge canon(a_i) = canon(a_j))$ $\implies mapVal_{eo_i}(a_i) = mapVal_{eo_j}(a_j))\}$, i.e., the traces that do not contain event occurrences with different values for an attribute of which its canonical form is in the canonical pattern attribute set.

The same way that a *trace id pattern* configures a view of the process, each process instance will be represented by a unique *trace id*. This concept (Definition 15) allows us to distinguish different traces. These traces aggregate events holding relations that can exist even between *events* of different *classes*.

Definition 15 (Trace ID) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, $TP_{DM} = (TPA, TPK, ROOT)$ a trace id pattern on DM and $t \in WFT$ a well-formed trace. $TID_t^{TP} \subseteq CPAS^{TP} \times V$ is a set of pairs attribute-value for a trace t according to a trace id pattern TP such that $TID_t^{TP} = \{(a,v) \in CPAS^{TP} \times V \mid eo \in t \wedge a = canon(b) \wedge b \in dom(mapVal_{eo}) \wedge v = mapVal_{eo}(b)\}$.*

The second goal of the trace building process is to avoid the creation of traces containing events that do not belong to the same instance, according to the selected view (*trace id pattern*). Definition 17 sets such rules. Some of these rules require a way to find connections between events. Such connections or properties are stated in Definition 16 as *trace id properties*.

Definition 16 (Trace ID Properties) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, $TP_{DM} = (TPA, TPK,$ $ROOT)$ a trace id pattern on DM, $\{t, t'\} \subseteq WFT$ are two well-formed traces and $\{TID_t^{TP}, TID_{t'}^{TP}\}$ their corresponding trace ids. We define the following properties*

- *$TID_t^{TP} \sim TID_{t'}^{TP} \iff \exists (a,v) \in CPAS^{TP} \times V : (a,v) \in TID_t^{TP} \wedge (a,v) \in TID_{t'}^{TP}$, i.e., TID_t^{TP} and $TID_{t'}^{TP}$ are related if and only if an attribute exists with the same value in both trace ids,*
- *$TID_t^{TP} \cong TID_{t'}^{TP} \iff \forall ((a,v),(a',v')) \in TID_t^{TP} \times TID_{t'}^{TP} : (a = a' \wedge v = v') \vee (a \neq a')$, i.e., TID_t^{TP} and $TID_{t'}^{TP}$ are compatible if and only if for each common attribute the value is the same in both trace ids,*
- *$TID_t^{TP} \bowtie TID_{t'}^{TP} \iff TID_t^{TP} \sim TID_{t'}^{TP} \wedge TID_t^{TP} \cong TID_{t'}^{TP}$, i.e., TID_t^{TP} and $TID_{t'}^{TP}$ are linkable if and only if they are compatible and related,*
- *$TID_t^{TP} \leq TID_{t'}^{TP} \iff TID_t^{TP} \subseteq TID_{t'}^{TP}$, i.e., TID_t^{TP} is a subtrace of $TID_{t'}^{TP}$ if and only if all the attributes in TID_t^{TP} are contained in $TID_{t'}^{TP}$ with the same value,*

Definition 17 (Valid Traces,Event Logs) *Let DM = (C, A, classAttr, val, PK, FK, keyClass, keyRel, keyAttr, refAttr) be a data model, $TP_{DM} = (TPA, TPK, ROOT)$ a trace id pattern on DM, CL a change log of event occurrences, WFT^{CL} the set of well-formed traces on that change log and $RootCAN \subseteq CPAS^{TP}$ is the set of canonical attributes of root such that $RootCAN = \{b \in CPAS^{TP} \mid \exists a \in keyAttr(ROOT) : canon(a) = b\}$. We define $VT(PT, CL) \subseteq WFT^{CL}$ as the set of valid traces for TP such that $VT(PT, CL) = \{t \in WFT^{CL} \mid \forall eo \in t : ((\forall c \in RootCAN : \exists (c,v) \in TID_{\{eo\}}^{TP}) \wedge (\nexists eo' \in t : time(eo') < time(eo))) \vee (TID_{\{eo\}}^{TP} \bowtie TID_{\{eo' \in t \mid time(eo') < time(eo)\}}^{TP})))\}$.*

Finally, we define an event log L^{PT} as the maximum subset of $VT(PT, CL)$ such that $\forall t, t' \in L^{PT} : (TID_t^{TP} \subseteq TID_{t'}^{TP} \implies t = t')$, i.e., the set of valid traces that does not contain any pair in which one of the traces is a subtrace of the other.

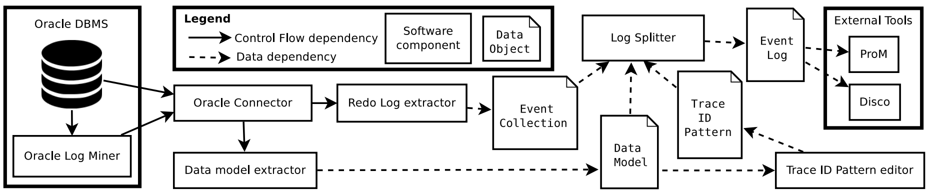


Fig. 3: Architecture of the Redo Log Inspector tool.

In the end, we guarantee that the resulting *event log* contains the minimum set of traces with the maximum behavior (L^{PT}) for a certain view *trace id pattern* TP). The traces in this *event log* start with an event containing values for the configured *root* element of the TP. Also, each of these traces contain events that are directly or transitively related ($\sim$) and compatible ($\cong$).

4 Implementation

The techniques presented in this paper allow for the extraction of events from any type of RDBMS with redo logs. Our implementation, however, is specific for Oracle technology. We can obtain data models directly from an Oracle DBMS, which makes it possible to design the *trace id pattern* (Definition 10) needed to generate an event log in accordance to Definition 17 from an event collection. The Redo Log Inspector [3] tool that we developed to demonstrate the feasibility of our ideas, fully implements the approach described in the previous sections and provides a user interface to control all the aspects of the analysis.

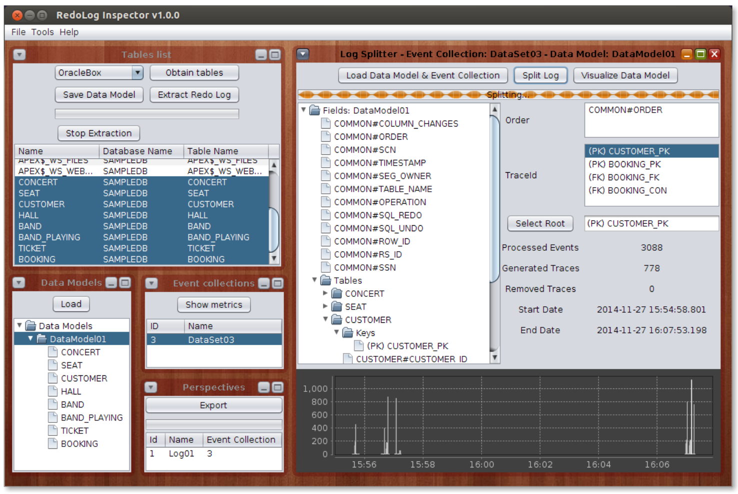
The Redo Log Inspector is composed of different components (see Figure 3). It uses a *Oracle Connector* component to communicate with the Oracle database and with the Oracle Log Miner functions. This component is used by the *Redo Log extractor* to generate an *Event Collection* from the desired tables. The *Data model extractor* also makes use of the *Oracle connector* to automatically obtain a *Data Model*. This last element is used by the *Trace ID Pattern editor* to design the desired *Trace ID Pattern*. Then, the three objects (*Event collection*, *Data model* and *Trace ID Pattern*) are used by the *Log splitter* to compute the traces to form a *Event log*. Finally, the *Event log* can be analyzed with existing process mining tools, such as ProM[4] [2] and Disco[5]. Figure 4 shows a screenshot of the tool while splitting an event collection into traces using a Trace ID Pattern.

5 Demonstration

The database we will use to demonstrate our approach is part of an imaginary portal for selling concert tickets. As stated, the database stores information about customers, bookings, and tickets. In addition, the concert venues are represented in the database, along with the collection of seats they offer. Each concert is also stored in it, with the list of bands performing. Figure 1 shows the data schema of the database. It is composed of eight different tables with several columns each, and a number of relations between them:

– Concerts: date and start time of the concert and the venue in which it will take place.

[3] Redo Log Inspector v1.0: `http://www.win.tue.nl/~egonzale/projects/rlpm/`
[4] ProM: `http://www.promtools.org`
[5] Disco: `http://fluxicon.com/disco/`



Name	Database Name	Table Name
APEX$_WS_FILES	SAMPLEDB	APEX$_WS_FILES
APEX$_WS_WEB...	SAMPLEDB	APEX$_WS_WEB...
CONCERT	SAMPLEDB	CONCERT
SEAT	SAMPLEDB	SEAT
CUSTOMER	SAMPLEDB	CUSTOMER
HALL	SAMPLEDB	HALL
BAND	SAMPLEDB	BAND
BAND_PLAYING	SAMPLEDB	BAND_PLAYING
TICKET	SAMPLEDB	TICKET
BOOKING	SAMPLEDB	BOOKING

- Data Models
  - DataModel01
    - CONCERT
    - SEAT
    - CUSTOMER
    - HALL
    - BAND
    - BAND_PLAYING
    - TICKET
    - BOOKING

ID	Name
3	DataSet03

Id	Name	Event Collection
1	Log01	3

- Fields: DataModel01
  - COMMON#COLUMN_CHANGES
  - COMMON#ORDER
  - COMMON#SCN
  - COMMON#TIMESTAMP
  - COMMON#SEG_OWNER
  - COMMON#TABLE_NAME
  - COMMON#OPERATION
  - COMMON#SQL_REDO
  - COMMON#SQL_UNDO
  - COMMON#ROW_ID
  - COMMON#RS_ID
  - COMMON#SSN
  - Tables
    - CONCERT
    - SEAT
    - CUSTOMER
      - Keys
        - (PK) CUSTOMER_PK
      - CUSTOMER#CUSTOMER_ID

Order: COMMON#ORDER

TraceId:
(PK) CUSTOMER_PK
(PK) BOOKING_PK
(FK) BOOKING_FK
(FK) BOOKING_CON

Select Root: (PK) CUSTOMER_PK

Processed Events	3088
Generated Traces	778
Removed Traces	0
Start Date	2014-11-27 15:54:58.801
End Date	2014-11-27 16:07:53.198

Fig. 4: Screenshot of the tool while splitting an event collection into traces.

– Band: name and address of bands, which could perform in different concerts.
– Band_playing: relates bands to concerts, indicating which ones will perform.
– Hall: details of the venues in which concerts can take place.
– Seat: each of the seats available in a venue.
– Ticket: the product being sold to users. They link a concert to a specific seat in a venue. Also, they refer to a booking if they have been acquired.
– Booking: objects created by customers when buying tickets.
– Customer: address, name and birth date. Each entry represents an user account.

Tables may be linked to other tables by means of foreign keys. For instance, the table *ticket* contains a foreign key named *ticket_concert_fk*: It associates the column *ticket:for_concert* to the primary key of table *concert*, specifically the column *concert:id*. This field relates a *ticket* to a specific *concert* in a $n:1$ relation, which means that a *ticket* must refer to a *concert*, but a *concert* can be related to many *tickets*.

In this database, like in many other settings, only the last state of the process is stored. This means we are able to answer questions of the following types:

1. How many concerts have been organized in the past?
2. Which venue has hosted most events in the last year?
3. What is the average number of tickets bought per customer in a month?

However, we would also like to find answers to other kinds of questions as well. In particular, we wish to pose questions that do not focus so much on the data facts, but on the underlying process that created and modified the data. Some of these questions are:

Q1. Which are the steps followed by a user to book a ticket?
Q2. Do customers book tickets before all the bands were confirmed?
Q3. Do bands ever cancel their performances in concerts?
Q4. Are venues being reserved before or after the bands have confirmed their performance?

It is evident that in order to find answers the inclusion of additional fields in the data schema would have been helpful. That data could be recorded explicitly in the database, adding timestamps to rows in every table and recording historical data of operations. It would be the equivalent of explicitly recording a *log* in the database. However, not every system has been designed to exploit the benefits of data and process mining. In other words, there are situations where we cannot rely on explicitly recorded logs of sorted events.

For instance, the fourth question could be answered querying the database only if the timestamps of execution of every operation are being recorded. However, Figure 1 shows that such timestamps are not present in the data schema of the proposed example. Something similar applies to the third question, which inquires if bands can cancel their performance at concerts. This requires the database to keep record of all the bands that were to perform in concerts and also the ones that canceled, for instance, by means of a status flag. Unfortunately the data schema does not store such information. What happens in case of a cancellation is that the corresponding entry will be removed from the table *band_playing*. This makes it impossible to know afterward which bands were once scheduled to perform but not anymore.

The focus of this section is to use database redo logs to answer the proposed questions using the technique presented in this paper. To do so, a dataset[6] of 8512 events has been generated based on a simulated environment interacting with the Oracle database presented in Figure 1. CPN tools [12] was used to model the creation of concerts and customers, the selling of tickets, and other operations on the elements of the database. The activities of such a process connect through a socket to a Java application managing the communication with an Oracle database. This way the environment of the system is simulated. This last one also generated the set of redo log files used to extract the events in our dataset. In the remainder the four questions are answered step-by-step.

5.1 Which are the steps followed by a user to book a ticket?

In order to answer this first question, we need to obtain the process describing the customer actions from the moment the selling portal is reached until the moment the ticket is sold. To do so, a log that contains traces showing that behavior has to be generated. This question will be answered in the next section in conjunction with the second one for the sake of brevity.

5.2 Could customers book tickets before all the bands were confirmed?

To answer this second question, two parts of the system must be involved: the ticket booking by customers and the concert organizing parts. For the first part, we can assume that the tables *customer*, *booking* and *ticket* must be involved in the process. Using the data schema in Figure 1, we see that tables *customer* and *booking* are linked by means of the pair of primary and foreign keys *customer_pk* and *booking_customer_fk* (Figure 1.a). Also, the tables *booking* and *ticket* are linked by means of the pair of keys *booking_pk* and *ticket_booking_fk* (Figure 1.b). Now, it is necessary to complete it with the concert organizing part. To do so, we have to relate each ticket to the concert it belongs to, and the later one to the bands playing. Observing Figure 1 we see that there is a relation between tables *ticket* and *concert* by means of the pair of keys *ticket_concert_fk* and *concert_pk* (Figure 1.c). Also, tables *concert* and *band_playing* share a relation by means of keys *concert_pk* and *bp_concert_fk* (Figure 1.d). Therefore, we should add these three keys to the Trace ID Pattern $TP = (TPA, TPK, ROOT)$, resulting in the following configuration:

– $TPA = \emptyset$,

[6] `http://www.win.tue.nl/~egonzale/projects/rlpm/datasets/ticket-selling-dataset.zip`

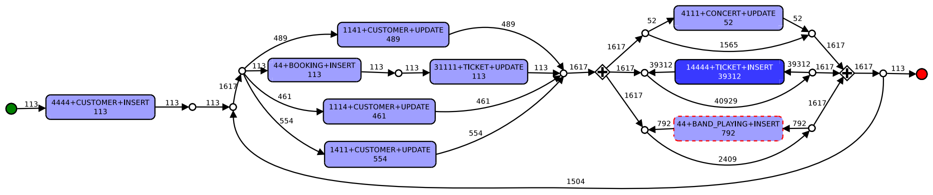


Fig. 5: Model of the ticket purchase and part of the concert organizing process.

- $TPK = \{customer\_pk, booking\_customer\_fk, booking\_pk,$ $ticket\_booking\_fk, concert\_pk, ticket\_concert\_fk, bp\_concert\_fk\}$,
- $ROOT = customer\_pk$.

Given that we want to cover the process from the moment a *customer* enters the system until the ticket is bought, it makes sense to select as root element of our Trace ID Pattern the primary key *customer_pk* in table *customer*. In other words, the customer is the case we want to follow through the process.

After this, the splitting process that follows generates a log with 149 cases. In this case the Inductive Miner [7] is used, and the log is replayed on it. Then, the activity *44+BAND_PLAYING+INSERT* is highlighted, which filters the log to show statistics using only the traces that contain the selected activity. The result is the annotated model in Figure 5. In it we observe that an insertion in the *customer* table can be followed either by modifications on it, or by an insertion in the *booking* table. An update in the *ticket* table can only be preceded by a *booking* creation. This means that, according to the evidence, the process followed by a customer to buy a ticket is as follows: (1) Create an account, which results in the insertion of a record in the table *customer*. (2) Create a booking, inserting a record in the table *booking*. (3) Buy the selected ticket, updating the *booking_id* field in the desired record in the table *ticket*. It can be also observed that modifications on the details of a customer profile can be made at almost any point in time, but not between the insertion of a booking and the update of a ticket. This suggests that both steps are performed automatically and in a strict sequence (Q1). To answer the second question it is interesting to see that insertions in the table *band_playing* can happen at any moment, before or after tickets are booked. This means that new bands are added to the concert not only after a concert is created, but also after a ticket has been booked. This does not require a causal relation in the sense that bands are added because a ticket is booked. However, it shows that both activities can happen in that order, answering the second proposed question (Q2).

5.3 Do bands ever cancel their performance in concerts?

To find out the answer to the third question, we should look at the *band_playing* table and see if any entry has been removed. This would not be possible when just inspecting the current content of the database. Fortunately, thanks to the redo logs, we can reconstruct the life-cycle of concerts. For the sake of brevity, the answer will be provided using the same experiment to answer the fourth question.

5.4 Are venues being reserved before or after the bands have confirmed their performance?

To solve the fourth question, we need to see how halls are being assigned to concerts at the same time that bands are being confirmed to perform on concerts. To do so, we have

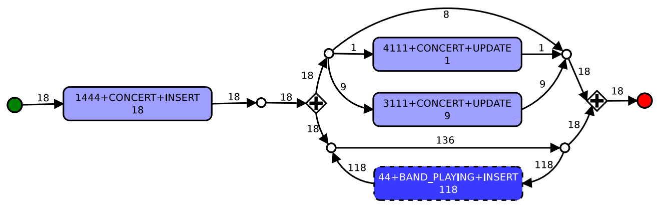


Fig. 6: Model showing the process of organizing a concert.

to focus on tables *concert*, *hall* and *band_playing*. Observing the data schema in Figure 1, we see that tables *concert* and *hall* are linked by means of the pair of primary and foreign keys *concert_hall_fk* and *hall_pk* (Figure 1.e). Also, there is a link between the tables *concert* and *band_playing* by means of the pair of keys *concert_pk* and *bp_concert_fk* (Figure 1.d). Therefore, we will use the four of them in our Trace ID Pattern $TP = (TPA, TPK, ROOT)$:

- $TPA = \emptyset$,
- $TPK = \{hall\_pk, concert\_hall\_fk, concert\_pk, bp\_concert\_fk\}$,
- $ROOT = concert\_pk$.

Knowing that concerts are the main object in this view, *concert_pk* will be selected as the root element. Splitting the dataset using these settings generates a log with 18 traces. Using the Inductive Miner and replaying the log, the annotated model in Figure 6 is obtained. It is evident that no deletions of records on table *band_playing* have been recorded. Therefore, as far as we can tell, none of the bands ever canceled their performance within a concert (Q3). We can also see that *hall* column in concerts can be updated before, after, or at the same time that bands confirm their performance in concerts. Therefore, there are no restrictions on the order of both events (Q4).

6 Conclusion

This work proposes to systematically use database redo logs as a new source of event data. The benefits include the existence of a data model and the historical view we obtain from the database. This represents a considerable innovation compared to the analysis of plain database content. To make sense of the events and obtain logs, the new concepts of *trace id pattern* and *trace id* have been introduced, which enable the discovery of transitive relations between data objects and the causal dependencies of the data modifications. An innovative approach to group the events in traces has been provided as well. Also, the feasibility of the approach has been shown in the form of a prototype. This prototype has been applied on a synthetic dataset to demonstrate its potential usefulness to answer a range of business questions that could not be directly answered by querying the database.

The technique is characterized by some drawbacks. First, the splitting algorithm produces a log where the same event may appear in different traces. This causes the existing process discovery algorithms to generate statistics that must be interpreted from the view we selected on the process. This is due to the fact that they consider events to be unique and only present in a single trace, when, in our case, they can be repeated and be counted more than once. If not interpreted correctly, the numbers could lead to the wrong conclusions. Also, the algorithm produces a number of traces that in some cases exceed the number of original events. These traces need to be analyzed by the discovery algorithms to produce models. This means that, in the end, we are going through the log many times. It would be useful to reduce the analysis to a single pass through the event collection to compute the structures needed by the discovery algorithms, e.g. a *Directly-follows Graph*. The analysis of real-life event logs is an obvious next task. Performing a case study on non-artificial redo logs will, hopefully, support the value of the techniques presented in this paper.

References

1. van der Aalst, W.M.P.: Extracting event data from databases to unleash process mining. In: vom Brocke, J., Schmiedel, T. (eds.) BPM - Driving Innovation in a Digital World, pp. 105–128. Management for Professionals, Springer International Publishing (2015)
2. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Rozinat, A., Verbeek, E., Weijters, T.: Prom: The process mining toolkit. In: Proceedings of the BPM Demonstration Track (BPMDemos 2009), Ulm, Germany, September 8, 2009 (2009)
3. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 32(3), 3–9 (2009)
4. Engel, R., van der Aalst, W.M.P., Zapletal, M., Pichler, C., Werthner, H.: Mining inter-organizational business process models from edi messages: A case study from the automotive sector. In: Advanced Information Systems Engineering. pp. 222–237. Springer (2012)
5. Fahland, D., De Leoni, M., Van Dongen, B.F., van der Aalst, W.M.P.: Behavioral conformance of artifact-centric process models. In: Business Information Systems. pp. 37–49. Springer (2011)
6. Ingvaldsen, J.E., Gulla, J.A.: Preprocessing support for large scale process mining of SAP transactions. In: Business Process Management Workshops. pp. 30–41. Springer (2008)
7. Leemans, S.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs-a constructive approach. In: Application and Theory of Petri Nets and Concurrency, pp. 311–329. Springer (2013)
8. Lu, X.: Artifact-Centric Log Extraction and Process Discovery. Master's thesis, Technische Universiteit Eindhoven, The Netherlands (2013), `http://repository.tue.nl/761324`
9. Mueller-Wickop, N., Schultz, M.: ERP event log preprocessing: Timestamps vs. accounting logic. In: Design Science at the Intersection of Physical and Virtual Design, Lecture Notes in Computer Science, vol. 7939, pp. 105–119. Springer Berlin Heidelberg (2013)
10. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
11. Nooijen, E.H., van Dongen, B.F., Fahland, D.: Automatic discovery of data-centric and artifact-centric processes. In: BPM Workshops. pp. 316–327. Springer (2013)
12. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN tools for editing, simulating, and analysing coloured petri nets. In: Applications and Theory of Petri Nets 2003, pp. 450–462. Springer (2003)
13. Roest, A.: A practitioner's guide for process mining on ERP systems : the case of SAP order to cash. Master's thesis, Technische Universiteit Eindhoven, The Netherlands (2012), `http://repository.tue.nl/748077`
14. Segers, I.: Investigating the Application of Process Mining for Auditing Purposes. Master's thesis, Technische Universiteit Eindhoven, The Netherlands (2007), `http://repository.tue.nl/630348`
15. Verbeek, H., Buijs, J.C., Van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Information Systems Evolution, pp. 60–75. Springer (2011)
16. Yano, K., Nomura, Y., Kanai, T.: A practical approach to automated business process discovery. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International. pp. 53–62 (Sept 2013)