Detecting System-Level Behavior Leading To Dynamic Bottlenecks

Zahra Toosinezhad*, Dirk Fahland*, Özge Köroğlu*, Wil M.P. van der Aalst†*
*Eindhoven University of Technology, Eindhoven, The Netherlands, email: z.toosinezhad@tue.nl, d.fahland@tue.nl
†RWTH Aachen, Department of Computer Science, Aachen, Germany, wvdaalst@pads.rwth-aachen.de

***Abstract*—Dynamic bottlenecks occur when some cases in a particular part of the process are temporarily delayed. In performance-optimized systems such as production systems, warehouse automation systems, and baggage handling systems, such bottlenecks are rare, bounded in time and location, but costly when they occur and propagate through the system. Detecting and understanding the situations that cause such bottlenecks is crucial for mitigating and preventing processing delays. Classical process mining techniques that analyze performance along individual cases cannot detect these phenomena and their causes. We show that undesired system-level behavior can be detected when identifying temporal event patterns across different cases in the same process step. Conceptualizing these patterns as system-level events allows us to correlate them into cascades of system-level behavior using spatio-temporal conditions. We discover classes of frequent patterns in these cascades that describe behaviors that precede bottlenecks. Applied on event data of a major European airport, our approach could fully automatically detect cascades of undesired system-level behavior leading to dynamic bottlenecks. Each detected cascade was verified as a correct causal explanation for a dynamic bottleneck due to the physical system layout and its processing.**

***Index Terms*—Process mining, Performance analysis, Dynamic bottlenecks, Event aggregation, Event correlation**

I. Introduction

A primary objective of applying process mining is detecting and understanding process performance problems to prevent unnecessary delays or *bottlenecks* through process redesign or early detection and mitigation. Bottlenecks are observed when cases or items are processed too slow, leading to an accumulation of unprocessed work. Bottlenecks are caused by the unavailability of a worker or machine or when work volume exceeds work capacity, resulting in growing queues and longer waiting times [1]. The most common approach for identifying bottlenecks from event logs is to calculate for each case or item the time difference between any two subsequent process steps [2]. Aggregating these time differences over all cases reveals which process steps take longer than average, indicating a *"static" bottleneck* observed for the majority of the cases due to structural capacity problems.

A. Dynamic Bottlenecks

Performance-optimized processes in production systems, warehouse automation systems, and Baggage Handling Systems (BHS) already established sufficient processing capacity to avoid static bottlenecks, often through a lean design process [3]. Here, bottlenecks only arise *dynamically* when the process operates outside its standard operating conditions, such as work volume temporarily exceeding expected capacity or processing capacity temporarily falling below intended levels. Fig. 1 illustrates an example from a BHS. We observe a higher load of cases (bags) $4, 5, 6$ between $L1$ and $L2$ that can still be processed normally with additional cases 3 and 7 about to enter. This high load propagates forward via conveyor belts to $L2, L3, L4$. Between $L4$ and $L5$ cases $2, 4, 3, 5$ saturate the physical system capacity. Processing at $L4$ is temporarily slowed down by stopping the conveyor belt from $L3$ to $L4$, and cases $6, 7$ are *blocked* for 2 minutes.

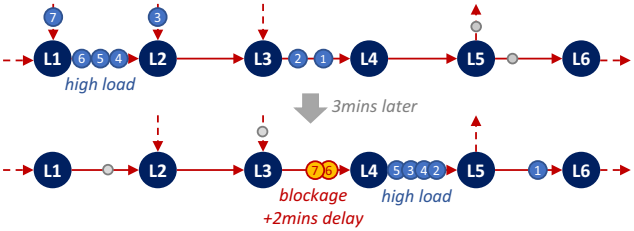

Fig. 1: Formation of a dynamic bottleneck.

Such *dynamic bottlenecks* are rare, bounded in time and location, but costly when they occur. In the BHS example, a delay of 2 minutes could mean bags $6, 7$ might miss a scheduled flight. Bottlenecks may propagate through the system similar to a traffic jam: processing at $L3$ in Fig. 1 may slow down as well, causing further bottlenecks before $L3$. Understanding under which circumstances dynamic bottlenecks emerge and propagate (and how far) would enable process redesign or the development of suitable early-warning prediction models enabling human or automated intervention.

B. Research Problem

In this paper, we consider the *research problem of (1) detecting dynamic bottlenecks and (2) detecting the process behavior which frequently precedes dynamic bottlenecks from basic process event logs*. The process behavior leading to a bottleneck *cannot* be described in terms of a single case or group of cases: $\{4, 5, 6\}$ between $L1$ and $L2$ form the initial high-load situation, but $\{2, 3, 4, 5\}$ between $L4$ and $L5$ form the later high-load situation causing the blocking of $\{6, 7\}$ before $L4$. If the bottleneck propagates further towards $L2$, completely different cases will be affected. As a result *case-oriented process mining techniques are unable to detect dynamic bottlenecks and how they form and propagate*.

C. Approach

We argue that solving the above problems requires studying "system-level" behavior which can only be observed indirectly from case-level events (of different cases) that have a specific characteristic in a specific part of the process, e.g., events related to bags $\{4, 5, 6\}$ between $L1$ and $L2$ forming a "high-load" dynamic. From this angle, the following three sub-problems arise which we propose to solve as follows.

(1) *Automatically detect "outlier" system-level behaviors that are different from normal operations and bounded in location and in time, i.e., the specific process steps and time interval where the process operated outside its regular parameters.* Formally, given an event log L of case-level events E characterized by activity, time, and case identifier, identify all sub-sets $s_1, ..., s_N \subseteq E$ of events of L whose characteristics significantly deviate from other sets of events, e.g., higher load and dynamic bottlenecks. The subsets s_i may overlap if they represent different overlapping characteristics. In this paper, we conceptualize the problem in a general way and propose to aggregate each set s_i into a *system-level event* for further analysis; we provide efficient techniques to detect time-bounded high-load situations and dynamic bottlenecks in performance-optimized systems such as a BHS.

(2) *Automatically identify those system-level events which are meaningfully* correlated *to each other and can be considered as a cascade of causally related system-level events.* This necessitates the inclusion of domain-knowledge. In this paper, we conceptualize the problem in terms of temporal and spatial distance properties between system-level events. We then identify correlated events by *querying* for system-level events with low temporal and spatial distance, i.e., dynamics in adjacent process steps that overlap in time. We show that the correlated, viz. queried, system-level events describe a partially-ordered system-level behavior that we call a *cascade*.

(3) *Automatically identify frequent patterns of correlated system-level events in system-level cascades that result in dynamic bottlenecks.* We show that applying existing sub-graph mining algorithms can efficiently detect such frequent patterns in cascades.

D. Results

We applied the above techniques of system-level event detection, querying, and grouping into system-level cascades, and frequent pattern detection on the complete baggage-level event data of 7 days of a BHS of a major European airport. We identified 1029 instances of dynamic bottlenecks and 123187 instances of other outlier behaviors. We identified four basic types of frequent patterns that precede these bottlenecks; all identified patterns were confirmed as correctly describing cause-effect behavior for forming dynamic bottlenecks.

In the following, we discuss related work in Sect. II. We then introduce an event model that considers events at different levels of abstraction in Sect. III to structure our research problem. We discuss how to detect system-level events in Sect. IV, how to detect cascades of correlated system-level events in Sect. V, and how to find patterns in cascades in Sect. VI. We report on our results on synthetic and real-life event logs in Sect. VII and discuss our findings and future work in Sect. VIII.

II. Related work

The primary approach for *performance outlier analysis in process mining* is to aggregate time differences between any two process steps and identify outlier values [2]. However, these techniques can only identify static bottlenecks affecting all (or most) cases. Event interval analysis [4] allows analyzing durations between any two process steps on a more fine-grained level based on event and case attributes, but cannot capture patterns involving multiple cases. The visual analytics technique of the Performance Spectrum reveals time-bounded performance patterns [5] over multiple process steps. These performance patterns can be understood as emergent system-level behavior formed by a group of cases in a specific time-window, such as batching [6]. We contribute techniques to detect performance patterns of multiple cases (system-level events) with a negative impact on performance, esp. bottlenecks, and how they propagate in time and to other process steps (as system-level dynamic).

Behavioral outlier analysis techniques in process mining identify infrequent event sequences [7] or infrequent event contexts [8] in a case of correlated events. The problem of this paper is to identify frequent patterns over emergent system-level events that are initially uncorrelated.

Techniques for finding sequential patterns in spatio-temporal event data [9] are inapplicable due to the distributed nature of processes. Process event correlation techniques either use case-level attributes to correlate events [10] or a process model describing the expected behavior [11]. This information is not available for emergent system-level events. By abstracting a process as a sequence of queues and visualizing the queue parameters over time, Staged Process Flows [12] allow visually analyzing how performance problems propagate through a process and affect multiple cases. This technique however assumes a sequential process and outlier behavior is not captured as a data feature for further analysis.

Queueing concepts are also used to discover congestion graphs [13] which model workload and resource availability over time in a Markov state for improved remaining time prediction. The congestion graph model is local to one process step and does not consider how performance problems may propagate. Our work aims at detecting, for instance, situations of high workload and resource unavailability and how they propagate to other process steps, thus providing a large description of performance dynamics.

Outside process mining, studies in *distributed system failure analysis* [14], [15] revealed the importance of analyzing correlations of outlier behavior and failures to increase resilience and performance in IT systems [16] and in IoT environments [17]. No existing work studies cascades of correlated failures for frequent patterns. Various works address finding outliers and their causal relations in *traffic data streams*. Outliers are identified through temporal properties in a specific spatial area

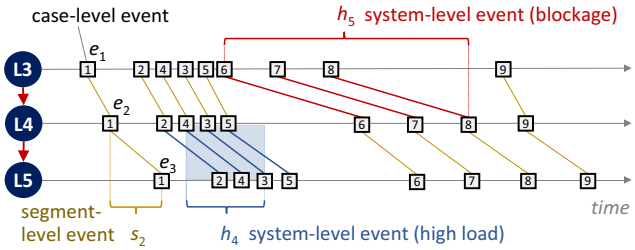

Fig. 2: Case, segment, and system-level events.

[18]–[20]. Outliers of a single type are correlated by building trees based on spatial and temporal properties of outliers [18] or by training a deep neural network which also allows short-term prediction [19]. Spatio-temporal correlations between different outliers can be learned through an attention network [20]. However, these works do not distinguish different types of outlier behavior. These techniques cannot visualize and explain longer cascades of multiple outlier events over time and their frequencies.

III. Event Definitions on Different Levels

To precisely formulate the three research questions of Sect. I, we introduce a conceptual event model having 3 levels. *Case-level* events records updates to a case as usual; *segment-level* events capture behavioral properties of pairs of directly related events; *system-level events* describe sets of consecutive segment-level events with specific properties. Figure 2 illustrates all 3 event notions for the behavior described in Fig. 1.

We assume our input events to be records of actions that happened to a specific case or object at a specific point in time, i.e., the standard event notion in process mining. We call a relation $< \subseteq E \times E$ an *acyclic* order iff its transitive closure $<^*$ is irreflexive, i.e., $<$ has no (self-)cycle.

Definition 1 (Case-level event log)**.** *A* case-level event log $L^c = (E^c, \#^c, <^c)$ *is a set* E^c *of events;* $\#^c$ *defines attributes such as case id* $\#^c_{id}(e)$*, activity* $\#^c_{act}(e)$*, and time* $\#^c_{time}(e)$ *for each* $e \in E^c$*; events are ordered by an acyclic order wrt. case id and time* $<^c \subseteq \{(e_1, e_2) \in E^c \times E^c \mid \#^c_{id}(e_1) = \#^c_{id}(e_2) \wedge \#^c_{time}(e_1) \leq \#^c_{id}(e_2)\}$.

Note that $<^c$ orders only events in the same case and according to time, but $<^c$ can be stricter than a partial order. In the following, we assume that $<^c$ describes "directly causally precedes" which can be derived using binary token flows [21], transitive reduction of a partial order [8], or may simply hold in the data, e.g., a bag moving on a physical conveyor belt. In Fig. 2, each square shows a case-level event, each diagonal line between two case-level events shows $<^c$. Tab. I shows a part of a case-level event log for our running example of Fig. 1 (*not* visualized in Fig. 2); $e_{15} <^c e_{18}$ and $e_{18} <^c e_{22}$ but $e_{15} \not<^c e_{22}$.

The set of activities in L^c is $\Sigma = \{\#^c_{act}(e) \mid e \in E^c\}$. We call a pair $(a, b) \in \Sigma \times \Sigma$ a *segment* of the process describing the passage from activity a to the next activity b [5]. Two

TABLE I: Case-level events of running example

	id	*act*	*time*
...	...	...	...
e_{10}	1	L4	10:21:30
e_{11}	1	L5	10:21:50
e_{12}	2	L4	10:23:10
e_{13}	2	L5	10:23:30
e_{14}	4	L4	10:23:25
e_{15}	5	L3	10:23:40
e_{16}	4	L5	10:23:45
e_{17}	3	L4	10:23:40
e_{18}	5	L4	10:23:55
...	...	...	...

...	...	...	...
e_{19}	3	L5	10:24:00
e_{20}	6	L3	10:23:55
e_{21}	7	L3	10:24:30
e_{22}	5	L5	10:24:05
e_{23}	6	L4	10:26:00
e_{24}	8	L3	10:24:40
e_{25}	7	L4	10:26:15
e_{26}	6	L5	10:26:20
e_{27}	8	L4	10:26:20

case-level events $e_1 <^c e_2$ with $\#^c_{act}(e_1) = a, \#^c_{act}(e_2) = b$ describe that the cases or objects $\#^c_{id}(e_1)$ and $\#^c_{id}(e_2)$ passed through segment (a, b). We capture this behavioral observation as an aggregate event, called a *segment-level event* as shown in Fig. 2. The segment-level events are fully defined by the case-level event log L^c as follows.

Definition 2 (Segment-level log)**.** *Let* $L^c = (E^c, \#^c, <^c)$ *be a case-level event log. A* segment-level log $L^s = (E^s, \#^s, <^s)$ *of* L^c *has events* $E^s = <^c$ *(the causally related events of* L^c*) where each* segment-level *event* $s = (e_1, e_2) \in E^s$ *has case id* $\#^s_{id}(s) = \#^c_{id}(e_1) = \#^c_{id}(e_2)$*, source and target activity* $\#^s_{src}(s) = \#^c_{act}(e_1)$, $\#^s_{tgt}(s) = \#^c_{act}(e_2)$*, start and end time* $\#^s_{start}(s) = \#^c_{time}(e_1)$, $\#^s_{end}(s) = \#^c_{time}(e_2)$*. Events are acyclically ordered wrt. segment and time by* $<^s \subseteq \{(s_1, s_2) \in E^s \times E^s \mid \#^s_{src}(s_1) = \#^s_{src}(s_2) \wedge \#^s_{tgt}(s_1) = \#^s_{tgt}(s_2) \wedge (\#^s_{start}(s_1) < \#^s_{start}(s_2) \vee \#^s_{end}(s_1) < \#^s_{end}(s_2))\}$.

Each diagonal line between two case-level events in Fig. 2 is a segment-level event. Table II shows the segment-level event log for the case-level events in Tab. I; for instance $s_{75} = (e_{15}, e_{18})$ and $s_{77} = (e_{18}, e_{22})$. Each segment-level event defines a spatio-temporal interval from $\#^s_{src}(s)$ to $\#^s_{tgt}(s)$ during time $[\#^s_{start}(s); \#^s_{end}(s)]$.

In contrast to case-level events, $<^s$ orders the segment-level events of cases per segment, e.g., in Tab. II $s_{75} <^s s_{78}$ (for cases 5 and 6 in $(L3, L4)$). Def. 2 requires that the order $<^s$ is consistent with at least start or end time. Though, $<^s$ can be stricter than the temporal order (as in Def. 1). In the following, we assume $s_1 <^s s_2$ iff s_1 *directly starts before* s_2, i.e., their start events are neighbors on the time-axis of their start activity in Fig. 2. We write $E^s(a, b)$ for events $s \in E_s$ with $\#^s_{src}(s) = a$ and $\#^s_{tgt}(s) = b$ in segment (a, b).

Fig. 2 shows how a set of consecutive segment-level events may form a behavior that we can recognize as a pattern at the system level. For instance, the segment-level events of cases $2, 3, 4, 5$ in $(L4, L5)$ have a shorter arrival rate than other segment-level events causing a short interval of *high-load*. The segment-level events of cases $6, 7, 8$ in $(L3, L4)$ have a longer duration causing a *dynamic bottleneck*.

To be able to reason about how the high-load interval relates to the bottleneck, we aggregate a set of consecutive segment-level events into a *system-level event*.

TABLE II: Segment-level event log

	id	*src*	*tgt*	*start*	*end*
s_{72}	1	L4	L5	10:21:30	10:21:50
s_{73}	2	L4	L5	10:23:10	10:23:30
s_{74}	4	L4	L5	10:23:25	10:23:45
s_{75}	5	L3	L4	10:23:40	10:23:55
s_{76}	3	L4	L5	10:23:40	10:24:00
s_{77}	5	L4	L5	10:23:55	10:24:05
s_{78}	6	L3	L4	10:23:55	10:26:00
s_{79}	7	L3	L4	10:24:30	10:26:15
s_{80}	8	L3	L4	10:24:40	10:26:20
s_{81}	6	L4	L5	10:26:00	10:26:20

Definition 3 (System-level event log)**.** *Let L^s be a segment-level event log. A* system-level event log $L^h = (E^h, \#^h, <^h)$ *(over L^s) defines a set of system-level events E^h where for each $h \in E^h$, $\#^h_{type}(h)$ is the event type, $\#^h_{src}(h)$ and $\#^h_{tgt}(h)$ are source and target, $\#^h_{sev}(h) \subseteq E^s(\#^h_{src}(h), \#^h_{tgt}(h))$ are segment-level events from L^s, $\#^h_{start}(h) = \min_{s \in \#^h_{sev}(h)} \#^s_{start}(s)$ is the start time and $\#^h_{end}(h) = \max_{s \in \#^h_{sev}(h)} \#^s_{end}(s)$ is the end time. Event $h \in E^h$ is* closed *iff for any two segment-level events $s_1, s_2 \in \#^h_{sev}(h)$ any event $x \in E^s(\#^h_{src}(h), \#^h_{tgt}(h)), s_1 <^s x <^s s_2$ is also in h, i.e., $x \in \#^h_{sev}(h)$. The order $<^h$ of system-level events is acyclic and must respect time: $<^h \subseteq \{(h_1, h_2) \in E^h \times E^h \mid \#^s_{start}(s_1) < \#^s_{start}(s_2) \vee \#^s_{end}(s_1) < \#^s_{end}(s_2)\}$.*

Figure 2 indicates two closed system-level events h_4 and h_5, i.e., two closed sets of consecutive segment-level events. In a real BHS, the higher load in $(L4, L5)$ may *cause* the conveyor belt $(L3, L4)$ to stop.

In contrast to Def. 2, Def. 3 allows two design decisions. (1) *Which sets of segment-level events shall be aggregated into a system-level event?* (2) *Which system-level events might be causally related and hence should be ordered by $<^h$?* These design decisions correspond to our research questions (1) and (2) which we answer in Sect. IV and V-B, respectively. There we will call sets of system-level events connected by $<^h$ a *cascade* of system-level behavior. We then show in Sect. VI that we can answer research question (3) be searching for frequent patterns in $L^h = (E^h, \#^h, <^h)$ along $<^h$.

IV. Detecting System-level Events

We consider how to detect "outlier" system-level events E^h of interest to build a system-level event log $L^h = (E^h, \#^h, <^h)$ from a normal case-level event log L^c, i.e., research question (1). We already obtained the segment-level event log $L^s = (E^s, \#^s, <^s)$ of L^c as described in Sect. III and shown in Tab. II. We are interested in detecting *closed* system-level events h, i.e., sets of consecutive segment-level events, that together show a "different" behavioral characteristic than other events in the segment, as shown in Fig. 2.

Detecting such differences requires a separate detection method for each type of behavioral characteristic. Thus, solving this problem requires the analyst to define a set of system-level event detection methods $D = \{detect_1, ..., detect_k\}$

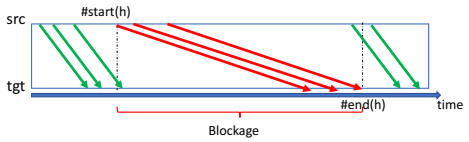

Fig. 3: Segment-level events forming a system-level blockage.

where each detection method $detect_i(E^s, \#^s, <^s) = E^h_i$ returns a set of system-level events of a specific type according to Def. 3. The set of system-level events is then $E^h = \bigcup_{detect_i \in D} detect_i(E^s, \#^s, <^s)$.

In the following, we define two specific system-level event detection methods to detect dynamic bottlenecks (Sect. IV-A) and high-load situations (Sect. IV-B) for the domain of material handling systems (MHS).

A. Detecting Dynamic Bottlenecks as Blockages

A dynamic bottleneck emerges when one or more cases take significantly higher times to flow from activity a to activity b than the baseline duration for segment (a, b). If $\tilde{t}$ is the baseline duration for cases traversing (a, b), then we can observe a *significant delay* as a segment-level event $s \in E^s(a, b)$ where the time-duration $\Delta t(s) = \#^s_{end}(s) - \#^s_{start}(s)$ is significantly higher $\Delta t(s) \gg \tilde{t}$ than the baseline.

In highly optimized systems such as MHS, the median duration in a segment does form the intended baseline, i.e., $\tilde{t}$ is the median of $\Delta t(E^s) = \{\Delta t(s) \mid s \in E^s\}$. Any segment-level event s where $\Delta t(s)$ is an outlier wrt. $\Delta t(E^s)$ in a statistical sense would show a significant delay. We analyzed and compared 4 univariate outlier detection methods and identified the *modified z-score* as the most reliable technique. [22, Ch. 5.2]

The modified z-score relates a concrete deviation $(\Delta t(s) - \tilde{t})$ of a segment-level event s to the median absolute deviation $MAD = \text{median}_{s \in E^s}(|\Delta t(s) - \tilde{t}|)$ of all events:

$$M(s) = (0.6745 \cdot (\Delta t(s) - \tilde{t}))/MAD \quad (1)$$

The factor 0.6745 allows approximating a standard normal distribution. If M_s is higher than a threshold k_{delay}, the $\Delta t(s)$ is an outlier wrt. $\Delta t(E^s)$ and we consider the case $\#^s_{id}(s)$ to suffer a significant delay, i.e., $delay(s) = true$ iff $M_s > k_{delay}$. For the domain of material handling systems, we confirmed normality of the data and identified $k_{delay} = 50$ as reliable together with domain experts. [22, Ch. 5.2]

A *dynamic bottleneck* emerges when several consecutive cases experience a delay. For example, in a MHS it may be that multiple cases are stuck behind each other in the same queue (red cases in Fig. 3) while other cases wait either before the blocked queue or are routed around and only enter the queue after the blockage ends (green cases at the end of Fig. 3). We can formally describe a blockage as a set of consecutive segment-level events s with $delay(s) = true$.

We detect all blockages in a segment (a, b) as follows. Assuming all start time-stamps to be distinct, $E^s(a, b) =$

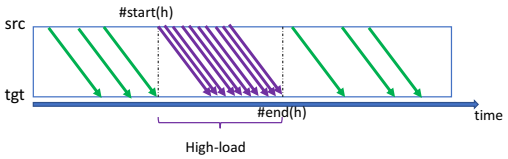

Fig. 4: Segment-level events forming high-load at system-level

$\langle s_1, \ldots, s_n \rangle$ defines a sequence of segment-level events. Iterating over $\langle s_1, \ldots, s_n \rangle$ we find all subsequences $\langle s_v, \ldots, s_w \rangle$ with $delay(s_i) = true, v \leq i \leq w$. Each such subsequence defines a new system-level event h of $\#^h_{type}(h) = blockage$ over $\#^h_{sev}(h) = \{s_v, \ldots, s_w\}$ (see Def. 3).

Suppose that in our running example $(L3, L4)$ has a median duration of $\tilde{t} = 10secs$ and $MAD = 1sec$. Then for $k_{delay} = 50$, segment-level events s_{78}, s_{79}, s_{80} of Tab. II have $delay(s_i) = true$, e.g., $M(s_{78}) = 0.6745 \cdot (125 - 10)/1 = 77.56 > k_{delay}$, resulting in the blockage (BL) system-level event h_5 in Tab. III.

B. Detecting High-load

A process having to handle a significantly higher number of cases in a process step may face performance problems. We can quantify the workload in a segment during a time-interval as the number of segment-level events crossing this time interval. If this workload is significantly above a baseline workload, then this time-interval describes a *high-load* outlier situation as illustrated in Fig. 4.

To detect high-load events in segment (a, b), we bin all segment-level events into bins of length k_{bin}, i.e., $bin_i = \{s \in E^s(a, b) \mid \#^s_{start}(s) \in [i \cdot k_{bin}; (i + 1) \cdot k_{bin})\}$ with $load_i = |bin_i|$. We then use the IQR-method to classify bins as outliers wrt. load. Let $P^{75}_{(a,b)}$ be the 75th percentile of the $load_i, i > 0$. For a bin holds $highload(bin_i) = true$ iff $load_i > P^{75}_{(a,b)}$. As for blockages, we search for subsequences $\langle bin_v, \ldots, bin_w \rangle$ with $highload(bin_i) = true, v \leq i \leq w$. Each such subsequence defines a new system-level event h of $\#^h_{type}(h) = highload$ over $\#^h_{sev}(h) = bin_v \cup \ldots \cup bin_w$ (see Def. 3).

Suppose that in our running example segment $(L4, L5)$ for $k_{bin} = 1min$ has $P^{50} = 2$ and $P^{75} = 3$, i.e., 75% of the bins see at most 3 bags per minute. Then $s_{73}, s_{74}, s_{76}, s_{77}$ of Tab. II fall into a bin with $load = 4$, resulting in the high-load (HL) system-level event h_4 in Tab. III.

V. Detecting System-Level Event Cascades

In Sect. IV, we detected system-level events E^h from a given case-level event log L^c, for example the events in Tab. III. In this section, we detect which system-level events might be causally related, i.e., research question (2). We first discuss how to identify whether two events in E^h are correlated in Sect. V-A. Ordering correlated events by time yields the relation $<^h$ of a system-level event log $L^h = (E^h, \#^h, <^h)$ (see Sect. III). We then observe that a set of events connected by $<^h$ forms a *cascade* of system-level behavior; we formalize this in Sect. V-B.

TABLE III: Detected System-Level Events

	type	*src*	*tgt*	*start*	*end*	*sev*
h_1	HL	L1	L2	10:20	10:23	$s_{65}, \ldots, s_{68}$
h_2	HL	L2	L3	10:21	10:23	$s_{61}, \ldots, s_{64}$
h_3	HL	L3	L4	10:22	10:24	$s_{69}, \ldots, s_{71}$
h_4	HL	L4	L5	10:23	10:24	$s_{73}, s_{74}, s_{76}, s_{77}$
h_5	BL	L3	L4	10:23	10:26	s_{78}, s_{79}, s_{80}
h_6	HL	L5	L6	10:25	10:26	$s_{82}, \ldots, s_{84}$

A. Correlating System-Level Events

In contrast to case-level events, the system-level events E^h lack a unique case identifier that describes which events belong to the same *observable* dynamic. Rather, we infer from emergent system-level event properties whether two events might be connected by a *direct material cause* that explains why one event has an influence on another event. The example in Fig. 2 illustrates this: h_4 and h_5 originate from segment-level events of disjoint sets of cases, yet the high-load h_4 in segment $(L4, L5)$ prevents cases on $(L3, L4)$ from entering $(L4, L5)$ which causes the blockage h_5 in $(L3, L4)$. In the following, we discuss which emergent properties we can use to explain possible causal relations between system-level events.

Each system-level event h in our event model has two emergent properties at the system-level: a *spatial property* (it is located on segment $(\#^h_{src}(h), \#^h_{tgt}(h))$) and a *temporal property* (it happens in the interval $[\#^h_{start}(h); \#^h_{end}(h)]$). There can only be a direct material cause between two system-level events if they are "sufficiently" close to each other in space and time, i.e., events that are too far apart cannot be related. For example, in Fig. 2, h_4 and h_5 are spatially close (share activity $L4$) and temporally close (overlap in time).

We propose to use a spatial distance measure $d_s : E^h \times E^h \to D_s$ and a temporal distance measure $d_t : E^h \times E^h \to D_t$ to characterize the distance between two events by $d_s(h_1, h_2)$ and $d_t(h_1, h_2)$. From domain-knowledge we can derive which spatial and temporal distances together are *Close* $\subseteq D_s \times D_t$. Then two events $h_1, h_2 \in E^h$ are correlated when $(d_s(h_1, h_2), d_t(h_1, h_2)) \in$ *Close*. Note that distance does not have to be continuous.

- Measuring spatial distance: The spatial distance measure d_s could be based on the physical distance of the locations of h_1 and h_2 or whether particular materials or information is being exchanged. For the case of a MHS with physical conveyor belts, we assume that two system-level events are correlated more likely if they happen on neighbouring conveyor belts. We define distance $d_s(h_1, h_2)$ as "proximity" in terms of locations shared between h_1 and h_2, i.e., $d_s(h_1, h_2) = |\{\#^h_{src}(h_1), \#^h_{tgt}(h_1)\} \cap \{\#^h_{src}(h_2), \#^h_{tgt}(h_2)\}|$. We consider h_1 and h_2 spatially close if they are spatially connected, $d_s(h_1, h_2) > 0$.
- Measuring temporal distance: The temporal distance measure d_t could be based on time distance or the amount of overlap between the intervals h_1 and h_2. For the case of a MHS, we consider system-level events as

related if they overlap in time or contain each other in terms of Allen Algebra. Formally, $d_t(h_1, h_2)$ is the Allen relation [23] that holds for $[\#^h_{start}(h_1); \#^h_{end}(h_1)]$ and $[\#^h_{start}(h_2); \#^h_{end}(h_2)]$; specifically $[t_1; t'_1]$ *overlaps* with $[t_2; t'_2]$ iff: $t'_2 - t'_1 > 0$ and $t_2 - t_1 > 0$; $[t_1; t'_1]$ *contains* $[t_2; t'_2]$ iff: $t'_1 - t'_2 > 0$ and $t_2 - t_1 > 0$. We define that h_1 and h_2 are close if $d_t(h_1, h_2) \in \{overlaps, contains\}$. Note that $d_t(h_1, h_2)$ defines that h_1 and h_2 are close only if h_1 starts before h_2, i.e., $d_t(h_1, h_2)$ is directed.

Altogether, two events h_1, h_2 are correlated iff they are spatially and temporally close, i.e., $d_s(h_1, h_2) > 0$ and $d_t(h_1, h_2) \in \{overlaps, contains\}$. In Fig. 2, h_4 and h_5 are correlated.

B. Cascades of Correlated Events

We now want to order the correlated system-level events over time to study system-level behavior. Technically, we will end up defining an acyclic order $<^h$ of a system-level event log (Def. 3). We conceptualize this problem as a directed graph $G = (E^h, F^h)$ where each system-level event is a node $h \in E_h$. There is a directed edge $(h_1, h_2) \in F^h$ between any two events that describes that h_1 and h_2 potentially occur in the same dynamic(directed clique). From this graph, we retain only those edges (h_1, h_2) where h_1 and h_2 are spatially and temporally close as described in Sect. V-A.

In other words, we *query* the graph G for the sub-graph of edges (h_1, h_2) where $(d_s(h_1, h_2), d_t(h_1, h_2)) \in Close$. Each queried edge (h_1, h_2) states that h_1 and h_2 are correlated by overlapping spatially and temporally *and* h_1 starts before h_2. This satisfies our requirement for the acyclic ordering relation $<^h$ of system-level event logs (Def. 3). We thus can define $<^h = \{(h_1, h_2) \in E^h \times E^h \mid (d_s(h_1, h_2), d_t(h_1, h_2)) \in Close\}$. The resulting system-level event log $L^h = (E^h, \#^h, <^h)$ can be understood as directed acyclic graph of correlated events. In general, the system-level event log $L^h = (E^h, \#^h, <^h)$ is not connected when interpreting $<^h$ as directed edges over nodes E^h.

Definition 4 (Cascade)**.** *A* cascade *is a connected component in the system-level event log $L^h = (E^h, \#^h, <^h)$. The events in a cascade are acyclically by local correlations that go forward in time (along the correlation edges $<^h$).*

For instance, the conceptual graph for the system-level events in Tab. III has nodes $h_1, \ldots, h_6$ that are all connected to each other. Querying this graph for correlated events returns the edges (h_1, h_2), (h_2, h_3), (h_3, h_4), (h_4, h_5), (h_4, h_6) resulting in the system-level event log of Fig. 5 for the process in Fig. 1. Together they describe that a high-load h_1 in $(L1, L2)$ propagated forward to h_4 in $(L4, L5)$ from where it further propagated in parallel: backwards as a blockage h_5 in $(L3, L4)$ and forward as high-load h_6 in $(L5, L6)$.

VI. Detecting Frequent Patterns in Cascades

Figure 6 shows some cascades of system-level events detected on real-life data of a BHS. These cascades are complex in structure and not all correlated outlier dynamics may be systematic, that is, occur frequently or even describe cause-effect relations. In the following, we discuss how to identify frequently occurring patterns in the cascades of a system-level event log $L^h = (E^h, \#^h, <^h)$, i.e., research question (3).

Fig. 5: Cascade of the system-level events of Tab. III

Given the graph-based nature of L^h, we propose to describe these patterns as sub-graphs.

Definition 5 (Cascade pattern)**.** *A* cascade pattern *is a connected labeled graph $P = (N^p, \#^p, F^p)$; each node $n \in N^p$ describes a system-level event by type $\#^p_{type}(n)$, and source $\#^p_{src}(n)$ and target $\#^p_{tgt}(n)$ activity; each directed edge (n_1, n_2) describes that n_1 "causes" n_2 in the sense of Sect. III. A cascade pattern P* occurs *in a system-level event log L^h iff P is an isomorphic sub-graph in L^h wrt. $\#^p_{type}(n), \#^p_{src}(n), \#^p_{tgt}(n)$, i.e., ignoring time-intervals of events in L^h.*

Finding all frequent cascade patterns becomes a *frequent sub-graph mining problem*. Frequent sub-graph mining is a data mining task of finding all sub-graphs that appear in at least k_{min} graphs of a graph database. In our situation, L^h is the graph database, each cascade is a graph in L^h, and the frequent sub-graphs are the cascade patterns we are interested in. Various sub-graph mining techniques are available; in our evaluation we use TKG [24] which returns the top-k sub-graphs wrt. their frequency (support) and has low running times also on large problem instances.

VII. Evaluation

We implemented and released the first version of our technique (https://github.com/processmining-in-logistics/cascades/releases/tag/v1.0) and evaluated it on real-life event data of a BHS of a major European airport to answer the following questions. (1) Are there dynamic bottlenecks and do they occur alone or are they correlated? (2) What is the structure and frequency of cascades that precede dynamic bottlenecks? (3) Are there frequent patterns in the cascades that are describing cause-effect relations of a BHS?

(1) In the BHS, each bag is a case and events are recorded when bags pass sensors on conveyors, the sensor location is recorded as activity name. We applied our implementation on an event log which contains 4,220,897 case-level events of 152,518 bags over 7 days. Table IV provides statistics of detected system-level events per day. We distinguish between blockage and high-load events that were isolated (iso) and correlated in a cascade (co): most of the outlier behaviors are correlated. We also report the average delay due to blockages and the maximum number of bags in a blockage (size): the frequency and duration of blockages varies, but overall hundreds of bags are blocked each day, an average 4 minutes of delay may cause bags checked in late to miss their flight.

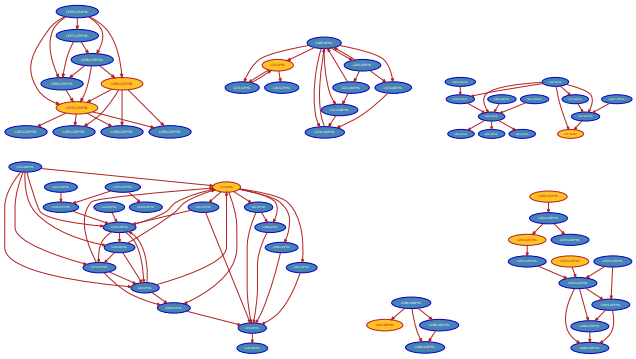

Fig. 6: Detected cascades of system-level events.

TABLE IV: Detected system-level events and cascades

Day	High-load iso	High-load co	Blockage iso	Blockage co	Blockage delay	Blockage size	# Cascades total	# Cascades block.
Mon	1073	15876	23	41	214s	4	586	23
Tue	924	16582	57	115	294s	13	785	40
Wed	920	15719	51	109	243s	9	869	47
Thu	1287	15871	30	71	454s	6	904	38
Fri	1390	18428	55	83	266s	6	1090	50
Sat	1069	16802	38	146	290s	6	1029	55
Sun	1183	16063	59	151	220s	12	991	56

(2) Table IV also reports the number of cascades found in total and cascades with at least one blockage event: most cascades contain no blockage but several cascades contain multiple correlated blockage events. We detect more cascades with and without blockages on Friday to Sunday which is in line with higher flight traffic. Fig. 6 shows a few example cascades: cascades greatly vary in size and complexity.

(3) We applied the frequent subgraph mining algorithm TKG of SMPF [25] on the detected cascades with at least one blockage. We detected 4591 frequent subgraphs; 500 subgraphs contained at least one blockage with a minimum support of 3. We analyzed the structure of the frequent subgraphs and identified the following 4 types, also shown in Fig. 7.

1) Fig. 7a illustrates the first cascade pattern type. Next to each system-level event h_1 we visualize its segment as a node $\#^h_{src}(h_1) : \#^h_{tgt}(h_1)$ connected via a dashed edge. The red edge from one segment node to the next indicates that the target activity of the first is the source of the second activity (they form a sequence). We can see that the causal order $h_1 <^h h_2 <^h h_3$ follows the direction of the segments, i.e., the cascade propagates *forward* in the process.
2) Fig. 7b illustrates the second pattern where $h_1 <^h h_2 <^h h_3$ are ordered in opposite direction of the segments, i.e., the cascade propagates *backwards* in the process.
3) Fig. 7c illustrates the third pattern where the segments are not sequentially ordered but diverge from the same source or converge to the same target, i.e., the cascade propagates *in parallel*.
4) Fig. 7d illustrates the fourth pattern where the cascade propagates *mixed* through the process, i.e., forward, backward, and parallel.

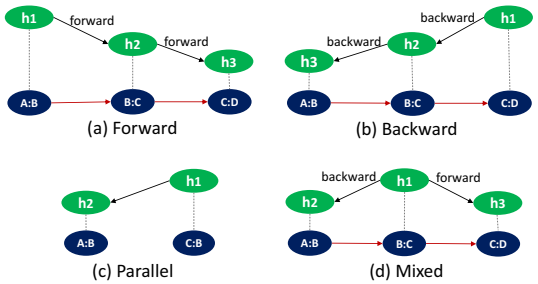

Fig. 7: Frequent types of cascade patterns.

TABLE V: Frequency of detected cascade patterns per type

Patterntype	*Frequency*	*Minsupport*	*Maxsupport*
Forward	50	3	6
Backward	60	3	5
Parallel	68	3	6
Mixed	322	3	4

Table V shows how many patterns of each type were found in the data and their minimum and maximum support. The large number of mixed patterns suggests that dynamic bottlenecks are most often preceded by complex dynamics.

Fig. 8 shows 3 detected *mixed* frequent subgraphs with at least one blockage with the physical layout. Fig 8a shows a simple subgraph detected close to the end of the process. We indicate the average delay between start times of two system-level events on the edge. High-load (h1) on segment $L13$:$L14$ propagates forward (h2) and causes a blockage (h3) after 143 secs at up-stream segment $L16$:$L13$. This pattern is causally explained by the system: $L13$:$L14$ is overloaded causing $L16$:$L13$ to stop forwarding bags. Subgraph 8b was found in baggage screening and is similar to subgraph 8a high-load (h2) on $L3$:$L2$ causes a blockage (h3) at upstream segment $L4$:$L3$. Interestingly (h2) is preceded in this frequent pattern by a parallel high-load (h1). The blockage (h3) is explained causally by (h2) whereas the preceding high-load (h1) is only frequently correlated to (h2) but both causally link to (h4). Fig. 8c illustrates a bigger subgraph detected at the end of the process. Here, high-loads (h1) and (h2) converge via $L6$:$L7$ and $L8$:$L7$ to location $L7$, propagate backwards as high-load (h3) to $L9$:$L8$. Location $L8$ is a split to three different locations $L7$, $L10$, $L11$. The high-load propagates downstream and causes a blockage (h5) on $L8$:$L10$. This complex dynamic occurred repeatedly and the first high-load occurred 194 secs before the blockage.

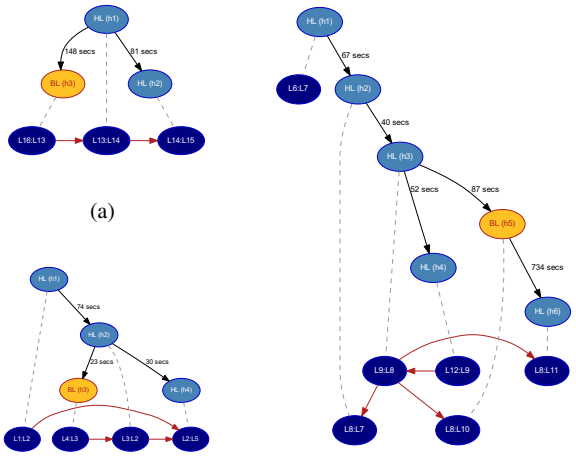

Fig. 8: Examples of detected cascade patterns.

VIII. Conclusion

We presented and validated a new method to detect cascades of system-level behavior which frequently precede dynamic bottlenecks from regular event logs. We could detect dynamic bottlenecks in a highly optimized BHS of a major European airport. The bottlenecks occur in cascades of high-load situations that propagate through the system. We verified that all propagation dynamics are causally explained by the underlying system dynamics.

The current approach has several limitations. The identified patterns occur relatively infrequently in the cascades and show a large variety. All the patterns have low support which requires further investigation. Analysis of longer time scales is required to increase confidence in the findings. Currently, we only consider high-load as an explanation for dynamic bottlenecks. Other causes such as availability of workers and equipment have to be included in future work. We only detect dynamic bottlenecks but would like to predict those bottlenecks, e.g., by splitting patterns in prediction rules with antecedent and consequent. We only demonstrated the applicability of our approach to BHS. The applicability to other systems with strict queuing has to be confirmed while generalization to business processes requires further research.

Acknowledgment

The research leading to these results has received funding from Vanderlande in the project "Process Mining in Logistics". Özge Köroğlu worked at Vanderlande Industries, Veghel, the Netherlands for parts of this study. We thank Marwan Hassani for his valuable input on formulating the research problem in a clear manner.

References

[1] R. W. Hall, "Queueing methods: For services and manufacturing," 1991.

[2] F. Milani and F. M. Maggi, "A comparative evaluation of log-based process performance analysis techniques," in *BIS 2018*, vol. 320 of *LNBIP*, pp. 371–383, Springer, 2018.

[3] T. L. Graafmans, O. Turetken, J. H. Poppelaars, and D. Fahland, "Process mining for six sigma: a guideline and tool support," *BISE*, 2020.

[4] S. Suriadi, C. Ouyang, W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Event interval analysis: Why do processes take time?," *Decis. Support Syst.*, vol. 79, pp. 77–98, 2015.

[5] V. Denisov, D. Fahland, and W. M. P. van der Aalst, "Unbiased, fine-grained description of processes performance from event data," in *BPM 2018*, vol. 11080 of *LNCS*, pp. 139–157, Springer, 2018.

[6] E. L. Klijn and D. Fahland, "Performance mining for batch processing using the performance spectrum," in *BPM 2019 Workshops*, vol. 362 of *LNBIP*, pp. 172–185, Springer, 2019.

[7] F. Folino, G. Greco, A. Guzzo, and L. Pontieri, "Mining usage scenarios in business processes: Outlier-aware discovery and run-time prediction," *Data Knowl. Eng.*, vol. 70, no. 12, pp. 1005–1029, 2011.

[8] X. Lu, D. Fahland, F. J. H. M. van den Biggelaar, and W. M. P. van der Aalst, "Detecting deviating behaviors without models," in *BPM 2015 Workshops*, vol. 256 of *LNBIP*, pp. 126–139, Springer, 2015.

[9] Y. Huang, L. Zhang, and P. Zhang, "A framework for mining sequential patterns from spatio-temporal event data sets," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 4, pp. 433–448, 2008.

[10] H. Reguieg, B. Benatallah, H. R. M. Nezhad, and F. Toumani, "Event correlation analytics: Scaling process mining using mapreduce-aware event correlation discovery techniques," *IEEE Trans. Serv. Comput.*, vol. 8, no. 6, pp. 847–860, 2015.

[11] S. Pourmirza, R. M. Dijkman, and P. Grefen, "Correlation miner: Mining business process models and event correlations without case identifiers," *Int. J. Cooperative Inf. Syst.*, vol. 26, no. 2, pp. 1742002:1–1742002:32, 2017.

[12] H. Nguyen, M. Dumas, A. H. M. ter Hofstede, M. L. Rosa, and F. M. Maggi, "Business process performance mining with staged process flows," in *CAiSE 2016*, vol. 9694 of *LNCS*, pp. 167–185, Springer, 2016.

[13] A. Senderovich, J. C. Beck, A. Gal, and M. Weidlich, "Congestion graphs for automated time predictions," in *EAAI 2019*, pp. 4854–4861, AAAI Press, 2019.

[14] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan, "Subtleties in tolerating correlated failures in wide-area storage systems," in *NSDI 2006*, USENIX, 2006.

[15] M. Sedaghat, E. Wadbro, J. Wilkes, S. de Luna, O. Seleznjev, and E. Elmroth, "Diehard: Reliable scheduling to survive correlated failures in cloud data centers," in *CCGrid 2016*, pp. 52–59, IEEE Computer Society, 2016.

[16] D. Tang and R. K. Iyer, "Analysis and modeling of correlated failures in multicomputer systems," *IEEE Trans. Computers*, vol. 41, no. 5, pp. 567–577, 1992.

[17] A. Perer and F. Wang, "Frequence: interactive mining and visualization of temporal frequent event sequences," in *IUI 2014*, pp. 153–162, ACM, 2014.

[18] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xie, "Discovering spatio-temporal causal interactions in traffic data streams," in *SIGKDD 2011*, pp. 1010–1018, ACM, 2011.

[19] Y. Wu and H. Tan, "Short-term traffic flow forecasting with spatial-temporal correlation in a hybrid deep learning framework," *CoRR*, vol. abs/1612.01022, 2016.

[20] S. Guo, Y. Lin, N. Feng, C. Song, and H. Wan, "Attention based spatial-temporal graph convolutional networks for traffic flow forecasting," in *AAAI 2019*, pp. 922–929, AAAI Press, 2019.

[21] W. M. P. van der Aalst, D. T. G. Unterberg, V. Denisov, and D. Fahland, "Visualizing token flows using interactive performance spectra," in *PETRI NETS 2020*, vol. 12152 of *LNCS*, pp. 369–380, Springer, 2020.

[22] Özge Köroğlu, "Outlier Detection in Event Logs of Material Handling System," Master's thesis, Eindhoven University of Technology, 2019.

[23] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[24] P. Fournier-Viger, C. Cheng, J. C. Lin, U. Yun, and R. U. Kiran, "TKG: efficient mining of top-k frequent subgraphs," in *BDA 2019*, vol. 11932 of *Lecture Notes in Computer Science*, pp. 209–226, Springer, 2019.

[25] P. Fournier-Viger, J. C. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam, "The SPMF open-source data mining library version 2," in *PKDD 2016*, vol. 9853 of *LNCS*, pp. 36–40, Springer, 2016.