Process Discovery from Event Stream Data in the Cloud – A Scalable, Distributed Implementation of the Flexible Heuristics Miner on the Amazon Kinesis Cloud Infrastructure

Joerg Evermann
Memorial University of Newfoundland
St. John's, Canada
jevermann@mun.ca

Jana-Rebecca Rehse, Peter Fettke
Deutsches Forschungszentrum für Künstliche Intelligenz
Saarbrücken, Germany
{Jana-Rebecca.Rehse, Peter.Fettke}@iwi.dfki.de
Universität des Saarlandes
Saarbrücken, Germany

***Abstract*—Cloud computing offers readily available, scalable infrastructure to tackle problems involving high data volume and velocity. Discovering processes from event streams, especially when the business processes execute in a cloud environment, is such a problem. Event stream data is generated rapidly with varying volume and must be processed on-the-fly, making stream processing an important use case for cloud computing. This paper describes a distributed, streaming implementation of the flexible heuristics miner on Amazon Kinesis, a cloud-based event stream infrastructure, showing how mining methods can scale effortlessly to tens of millions of events per minute.**

I. Introduction

Many information systems produce event logs that capture the actions of their users. Examples are page requests of web-servers and business-object method calls in ERP systems. Process discovery deals with the identification of processes from such event logs, for example the process of ordering a product on an e-commerce web-site, or the process of scheduling a manufacturing order in an ERP system [1].

Process discovery in the context of cloud environments is a largely unexplored topic. In this paper we describe process discovery *in* the cloud, not process discovery *of* cloud data. Specifically, we are not concerned with mining process models from data that necessarily originate in cloud-based systems or the cloud infrastructure. Research questions related to cloud issues such as multi-tenancy are discussed in [2]. Instead, we are interested in the use of cloud-based systems to mine process data that may or may not originate in the cloud.

The rapid creation and increased availability of data has been captured in the notion of "big data". The characteristics of "big data", high volume, high velocity and high variability makes business process discovery an important cloud computing use case. The scalability provided by on-demand computing and data management infrastructure allows rapid or real-time process discovery of big data. The large volume of event data, generated at a rapid rate, makes it impractial to store the data for any length of time and requires that the data is processed on-the-fly, using data stream processing methods. A useful abstraction for this are data streams that connect a set of independent data processors that together implement a distributed data analysis algorithm. Cloud infrastructure is ideally suited for dynamically provisioning the required data streams and computing nodes, making business process discovery an important cloud computing use case.

9781-5090-1445-3/16$31.00 © 2016 IEEE (CloudBPM'16)

The Flexible Heuristics Miner (FHM) [3] is a simple yet useful and widely used [4] heuristic for constructing process models from event logs. We implement the FHM algorithm in a distributed way on indepedent processing nodes that ingest streaming event data and are also internally connected by event streams. We use a cloud-based implementation of the event streaming infrastructure, affording us scalability to hundreds of megabytes per minute. The individual processing nodes consist of independent processing threads that do not manage any information proportional in size to the volume of incoming events. Scalability is therefore limited only by processing power and network bandwidth; both limitations are addressed by the distributed implementation and scalability features offered by cloud-based processing environments. In summary, the goals of this research are to demonstrate that

- Process discovery algorithms can be designed or adapted for streaming event data, operating on an event-by-event basis,
- The streaming event processing algorithm can be distributed across multiple processing nodes, in turn connected by event streams,
- The distributed algorithms and the connecting network of event streams can be readily implemented on commercial cloud infrastructure, and
- The implementation of the distributed algorithms scales effortlessly to tens of millions of events per minute.

The remainder of the paper is as follows. Sec. II briefly discusses prior work on stream process dicsovery. Sec. III

introduces the main ideas of the FHM. Next, Sec. IV presents our distributed, scalable implementation of the FHM algorithm. Following this, Sec. V presents our implementation and experimental results. The paper closes with a brief discussion in Sec. VI.

II. Related Work

Process mining of big data has only been a very recent research topic [5] and only a few approaches have been presented, mostly concerning process mining on stationary data. Reguieg et al. [6] apply map-reduce to process discovery with the aim of discovering "event correlations" in systems where events are not explicitly associated with cases. However, the actual process discovery is outside the scope of their map-reduce based approach. In contrast, [7] describe a map-reduce implementation for computing event log abstractions, such as the "follows" relation (Def. 1 in Sec. III), which are used by different process discovery algorithms. Also using map-reduce, [8] present a scalable implementation of the Alpha miner and the FHM. While dealing with "big data", these approaches operate on static data and can therefore not deal well with high data velocity.

Among approaches operating on streaming data, [9] presents a method to discover declarative process models. A declarative process model is one that does not explicitly specify the process control flow, but "describes a set of constraints that must be satisfied throughout the process execution." In contrast, our work is concerned with the mining of explicit process models.

Three streaming variants of the FHM have already been proposed. The streaming version described in [10] uses three event queues to maintain information about the latest observed activities and the resulting latest observed dependency measures. From this information, a process model is mined periodically or when required, using any existing process discovery algorithm. This is in contrast to our approach which continuously updates not only the occurrence counts of the log-based ordering relation instances and the dependency measures (Def. 1, 2 in Sec. III) but the entire process model with the processing of every event.

Expanding on the earlier approach in [10], [11] uses lossy counting to count occurrences of activities and instances of the "follows" relation (Def. 1 in Sec. III) to then periodically mine a process model from this information. Again, in contrast to our work, the algorithm does not continuously update the discovered model. Instead, the idea is that "since the two fundamental measures of Heuristics Miner ... are based on the directly-follows measure ... our idea is to 'replace' the batch version of this frequency measure, with statistics computed over an event stream."[11, p. 2423].

Another streaming implementation of the FHM in [12] stores event counts and information that allows computing the "follows" relation in trace prefix trees. A number of pruning strategies are described and implemented to keep the size of the prefix trees manageable even for streams with many different events and long traces. But, similar to [10], [11], [12] and in contrast to our work, only the basic log relations (Def. 1, 2 in Sec. III) are updated on an event-by-event basis, the actual FHM discovery algorithm (Def. 3 in Sec. III) is performed periodically.

Only indirectly concerned with process discovery are event stream processing applications for process prediction, such as those in [13], [14].

III. The Flexible Heuristics Miner

The Flexible Heuristic Miner (FHM) algorithm [3] is designed to be used with noisy event log data in that it allows the exclusion of rare or unusual event occurrences that should be considered "outliers" for the discovery of the process model. The FHM algorithm defines three log-based ordering relations:

Definition 1. *(Log-based ordering relations for the FHM algorithm) Let T be a set of activities and W be an event log over T. Let $a,\, b \in T$:*

- $a >_w b$ *iff there is a trace* $\sigma = t_1 t_2 t_3 \dots t_{n-1}$ *in* W *such that* $\sigma \in W$ *and* $t_i = a$ *and* $t_{i+1} = b$ *for* $i \in \{1, \dots, n-2\}$
- $a >>_w b$ *iff there is a trace* $\sigma = t_1 t_2 t_3 \dots t_{n-1}$ *in* W *such that* $\sigma \in W$ *and* $t_i = t_{i+2} = a$ *and* $t_{i+1} = b$ *for* $i \in \{1, \dots, n-3\}$
- $a >>>_w b$ *iff there is a trace* $\sigma = t_1 t_2 t_3 \dots t_{n-1}$ *in* W *such that* $\sigma \in W$ *and* $t_i = a$ *and* $t_j = b$ *and* $i < j$ *for* $i, j \in \{1, \dots, n-1\}$

The "follows" relation ($>_w$) represents direct successors of activities in the log. The second relation ($>>_w$) represents loops of length two between activities. The third relation ($>>>_w$) represents the general successor relationship, either direct or indirect, irrespective of the distance of the two activities in the log.

From the occurrence counts of instances of these log-based ordering relations, the FHM algorithm constructs "dependency measures" that express the relative frequency of the occurrence of instances of each log-based ordering relation. These dependency measures indicate the degree of "certainty" that there is a true dependency relation between two events A and B, or that there is truly a loop of length two.

Definition 2. *(Dependency measures for the FHM algorithm) Let T be a set of activities and W be an event log over T. Then:*

$$a \Rightarrow_w b = \left(\frac{|a >_w b| - |b >_w a|}{|a >_w b| + |b >_w a| + 1} \right) \quad \textit{if } (a \neq b)$$

$$a \Rightarrow_w a = \left(\frac{|a >_w a|}{|a >_w a| + 1} \right)$$

$$a \Rightarrow_w^2 b = \left(\frac{|a >>_w b| - |b >>_w a|}{|a >>_w b| + |b >>_w a| + 1} \right)$$

$$a \Rightarrow_w^l b = 2 \left(\frac{|a >>>_w b| - \mathrm{abs}(|a| - |b|)}{|a| + |b| + 1} \right)$$

The FHM algorithm uses these frequency-based dependency measures to define a dependency graph, using the following steps:

Definition 3. *(Dependency graph algorithm for the FHM algorithm) Let T be a set of activities and W be an event log over T. Then:*

$$
\begin{aligned}
&C_1 = (a,a) \in T \times T | a \Rightarrow_w a \geq \sigma_{L1L} \\
&C_2 = \{(a,b) \in T \times T | (a,a) \notin C_1 \wedge (b,b) \notin C_1 \wedge \\
&\quad a \Rightarrow^2_w b \geq \sigma_{L2L}\} \\
&C_{out} = \{(a,b) \in T \times T | b \neq End \wedge a \neq b \wedge \\
&\quad \forall_{y \in T} [a \Rightarrow_w b \geq a \Rightarrow_w y]\} \\
&C_{in} = \{(a,b) \in T \times T | a \neq Start \wedge a \neq b \wedge \\
&\quad \forall_{x \in T} [a \Rightarrow_w b \geq x \Rightarrow_w b]\} \\
&C'_{out} = \{(a,x) \in C_{out} | (a \Rightarrow -wx) < \sigma_a \wedge \\
&\quad \exists_{(b,y) \in C_{out}} [(a,b) \in C_2 \wedge ((b \Rightarrow_w y) - (a \Rightarrow_w x)) > \sigma_r]\} \\
&C_{out} = C_{out} - C'_{out} \\
&C'_{in} = \{(x,a) \in C_{in} | (x \Rightarrow_w x) < \sigma_a \wedge \\
&\quad \exists_{(y,b) \in C_{in}} [(a,b) \in C_2 \wedge ((y \Rightarrow_w b) - (x \Rightarrow_w a)) > \sigma_r]\} \\
&C_{in} = C_{in} - C'_{in} \\
&C''_{out} = \{(a,b) \in T \times T | a \Rightarrow_w b \geq \sigma_a \vee \\
&\quad \exists_{(a,c) \in C_{out}} [((a \Rightarrow_w c) - (a \Rightarrow_w b)) < \sigma_r]\} \\
&C''_{in} = \{(b,a) \in T \times T | (b \Rightarrow a) \geq \sigma_a \vee \\
&\quad \exists_{(b,c) \in C_{in}} [((b \Rightarrow_w c) - (b \Rightarrow wa)) < \sigma_r]\} \\
&DG = C_1 \cup C_2 \cup C_{out} \cup C_{in} \cup C''_{out} \cup C''_{in}
\end{aligned}
$$

As the name implies, the dependency graph indicates only which event types depend on other event types, but does not indicate whether a particular event type is followed by an AND, an XOR, or an OR split (or, conversely, whether a particular event type is preceded by an AND, an XOR, or an OR join). To identify the splits in the process model, each trace is run ”forwards” against the dependency graph. For example, if the dependency graph states that tasks B and C depend on task A (i.e. they can possibly be activated by the occurrence of task A), we wish to find out whether A in some traces activates B and in other traces C, or whether in all traces B and C are activated, or some combination. The first case would represent an XOR split, the second case would represent an AND split, and the third case an OR split. To identify joins in the process model, the same method is used but the traces are run ”backwards” against the dependency graph. The result of this step is an augmenented causal net (CNet) which contains information about the frequencies with which one set of events activates another set of events in the workflow log.

Our implementation of the FHM in a distributed architecture using event streams follows the effects of a single new process event. Specifically, it examines how a each new process event affects the occurrence counts of the log-based ordering relation instances (Def. 1), how changes to these occurrence counts affect dependency relations (Def. 2), how changes in the dependency relations affect the various sets considered in the algorithm in Def. 3, and how changes to the various sets together with each completed trace affect the final augmented causal net.

IV. Distributed Event Stream FHM

Our contribution in this paper is the separation of the FHM algorithm into individual processing stages, suitable for a cloud-based scalable event stream infrastructure, and the distribution of these stages to different computation nodes in the cloud infrastructure.

We use Amazon Kinesis, part of Amazon Web Services (AWS), to provide a scalable stream infrastructure for records of arbitary form. In Kinesis, a stream is a queue for arbitrary records. Each record is associated with a key. A Kinesis stream is logically divided into one or many shards, which are logical divisions of a stream. When writing to a stream, a producer provides a key for each record; records with the same key are written to the same shard. We use the partitioning of records by shards to separate the processing of events using multiple independent processing threads for each stage.

Figure 1 presents an overview of the architecture. The event consumer examines each process trace event and provides information about how the event affects the occurrence counts of instances of the log-based ordering relations (Def. 1) via the relation stream. The relation consumer then updates the dependency relations (Def. 2) and provides updated dependency values to the update consumer via the update stream. The update consumer identifies how the changes to dependency relations affect the dependency graph (Def. 3) and provides information about any graph changes to the graph consumer via the graph update stream. Up to this point, information is logically separated by event pairs, both in the event streams and the processing threads. Only the final graph consumer maintains a global dependency graph, and, using completed traces, computes an updated causal net as final output. The following subsections provide details of the the different processing stages.

A. Event Generators

Event generators produce the raw events to be processed. Events are tuples of the form $(Activity, TimeStamp, CaseId)$ and are written to the event stream using $CaseId$ as key, so that a consumer reading from an event stream shard processes all events for a particular case.

B. Event Consumer

The event consumer ingests events from the event stream. Because cases are independent of each other, processing is performed by independent processing threads, each serving one shard of the stream. Each thread maintains three separate hashmaps for active traces ($a : CaseId \mapsto Trace$), mean interarrival times ($i : CaseId \mapsto Time$), and the times of the last observed event for each active case ($l : CaseId \mapsto Time$). When an event is received for an active case, these maps are updated and the instances of the log-based ordering relations

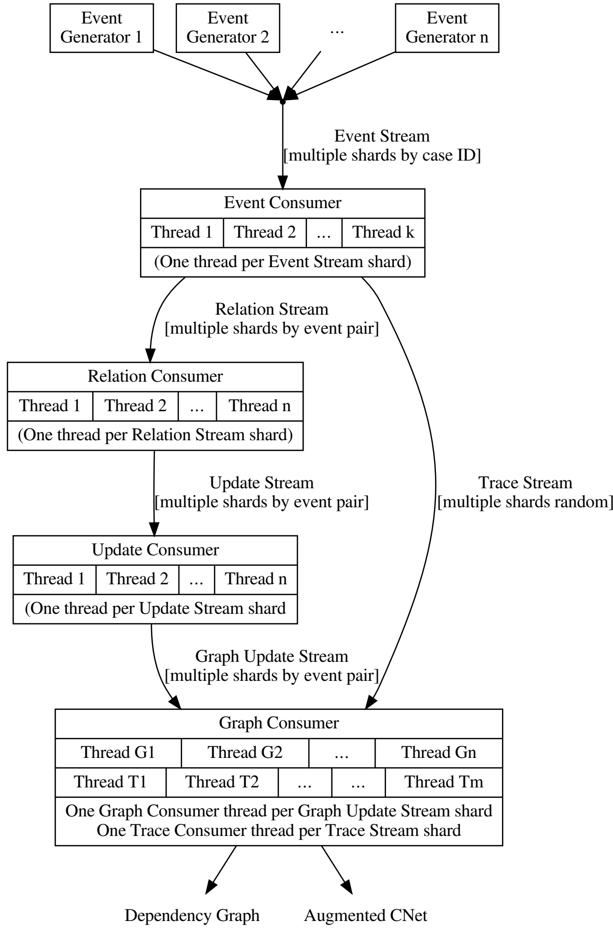

Fig. 1. Architecture of Distributed FHM on Amazon Kinesis Infrastructure

(Def. 1) are computed for this new event with respect to the already processed previous events for the case.

Active cases are periodically retired to conserve memory. In the absence of explicit end-of-trace markers, trace retirement is a serious problem in stream processing as business processes may be long running and events could arrive infrequently or irregularly. Prior work has dealt with this in different ways. The case pruning in [12] is based on a fixed maximum number of active cases. It periodically retires all cases whose time of last observed event is less than the mean of the times of last observed events for all cases. Should a late event arrive for a retired case, it is treated as a new case. The streaming FHM in [10], [11] use fixed-size queues to limit memory consumption and retire the oldest events when the queues are full, and [9] primarily considers explicitly tagged end-of-case events.

In our work, we assume a Poisson distributed event inter-arrival time. A 99.99% confidence interval for the next event arrival time is computed from the mean interarrival time for the case. When the upper bound of this interval is in the past, the case is retired, its trace is written to the trace stream and all information for the case is deleted. The event consumer maintains a hashset of all retired case IDs. When a new event arrives with a case ID that is not in the active cases, it is checked against retired case IDs. If it is not found in that set, a new active case is opened, otherwise the event is discarded as a late event for an already retired case (with probability of 0.01%). The size of the confidence interval represents a trade-off between being able to process late events and conserving thread memory.

Instances of log-based ordering relations for each processed event are collected and emitted into the relation stream as tuples $((Activity1, Activity2), RelationType, Count)$ where $RelationType$ indicates the type of basic relation (Def. 1), and $Count$ indicates how many instances of each activity pair $(Activity1, Activity2)$ are added to this log-based ordering relation based on the currently processed event: Appending an event to an active trace can generate only one instance of $>_w$ and $>>_w$ but can generate multiple instances of $>>>_w$. These records are written to the relation stream using $(Activity1, Activity2)$ as key. Because the subsequent computation of the dependency measures requires information not only about the pair $(Activity1, Activity2)$ but also about its 'inverse' $(Activity2, Activity1)$, these are written with the same key and so to the same shard.

C. Relation Consumer

The relation consumer reads records from the relation stream. Because the dependency measures (Def. 2) for a pair of activities are based only on the occurrence counts of the log-based ordering relation instances for that pair and its inverse pair, processing is performed by independent threads, each serving a shard of the stream. Each thread maintains three hashmaps that map activity pairs to counts of the instances of each relation type for each activity pair ($c : Event \mapsto \mathbb{N}$, $f_{\text{count}} : Event \times Event \mapsto \mathbb{N}$, $l2_{\text{count}} : Event \times Event \mapsto \mathbb{N}$. $ld_{\text{count}} : Event \times Event \mapsto \mathbb{N}$). Additionally, each thread maintains three hashmaps that map activitiy pairs to current values for each dependency measure ($f_{\text{dep}} : Event \times Event \mapsto \mathbb{R}$, $l2_{\text{dep}} : Event \times Event \mapsto \mathbb{R}$. $ld_{\text{dep}} : Event \times Event \mapsto \mathbb{R}$).

Occurence counts and dependency measure values are updated according to Def. 2 when a new record is read from the relation stream. Updated dependency measure values are then emitted to the update stream as tuples of the form $(Activity1, Activity2, RelationType, Value)$ using the activity pair as a key to ensure that tuples for the same activity pair are processed by the same subsequent consumer. To save stream capacity, only values that meet a configurable lower threshold (0.5 by default) are emitted.

D. Update Consumer

The update consumer reads records from the update stream, using independent threads for each shard. Each thread maintains a partial dependency graph, and a list of current values for each dependency measure for each activity pair it processes.

When reading a record containing a new dependency measure value, the processing thread considers each of the sets that form the core of the FHM algorithm in Def. 3 and determines

whether the new dependency measure value yields any changes to the various sets.

The reconceptualization of the core FHM algorithm in terms of set updates (Algorithms 1–3) is a core contribution of this paper. There are two notable considerations: First, the loop-of-length-one and loop-of-length-two dependency measures can only grow, not shrink (cf. Def 2): Updates to these can never cause edge removals from the dependency graph, i.e. there are never any removals from the sets C_1 and C_2 in Def. 3. Second, edges in the dependency graph can result from multiple, different types of dependencies. For example, an edge may be in the sets C_1 and C''_{out}. Hence, edges can only be removed when not supported by *any* dependency. The update consumer thread maintains a hashmap that maps each graph edge (activity pair) to the set of dependency types that support each graph edge. Edges are removed only when the last supporting dependency type is removed.

Changes to the graph are emitted as tuples $(Activity1, Activity2, UpdateOp)$ into the graph update stream where $UpdateOp$ indicates removal or addition of an edge. Records are keyed by activity pair.

The thresholds for the algorithm $(\alpha, \sigma_>, \sigma_{>>}, \sigma_{>>>})$ are increased asymptotically to one because higher thresholds reflect the increasing requirements for practical significance in larger event volumes [3].

```
Data:
  α, σ_>, σ_>>, σ_>>> ← 0.9, initial dependency thresholds
  ρ ← 0.05, initial relative-to-best threshold for dependencies
  d : Event × Event ↦ ℝ, a map of direct dependencies for each
event pair
  l2 : Event × Event ↦ ℝ, a map of loop-two dependencies for
each event pair
  dg : Event × Event ↦ 2^{>_w, >>_w, >>>_w}, a map of edges of the
partial dependency graph to the powerset of dependency relation types
Function UpdateConsumer()
    while true do
        u ← UpdateStream.dequeue()
        switch u.Type do
            case >_w do
                ProcessDirectFollows(u)
            case >>_w do
                o ← l2(u.Event1, u.Event2)
                l2(u.Event1, u.Event2) ← u.Value
                if
                  (u.Value > o)∧(u.Value ≥ σ_>>)∧(o < σ_>>)
                  then
                    addEdge(u.Event1, u.Event2, >>_w)
                    addEdge(u.Event2, uEvent1, >>_w)
            case >>>_w do
                if u.Value > σ_>>> then
                    addEdge(u.Event1, uEvent2, >>>_w)
                if u.Value < σ_>>> then
                    removeEdge(u.Event1, uEvent2, >>>_w)
        end
        updateThresholds()
    end
```

Algorithm 1: Outline of Update Consumer

```
Function ProcessDirectFollows(u)
    o ← d(u.Event1, u.Event2)
    d(u.Event1, u.Event2) ← u.Value
    if u.Event1 = u.Event2 then
        if (o < σ_>) ∧ (u.Value ≥ σ_>) then
            addEdge(u.Event1, u.Event2, >_w)
    else
        cout ← max(d(u.Event1, b)|b ≠ u.Event2)
        c ← {b|d(u.Event1, b) = cout}
        if cout = 0 then
            addEdge(u.Event1, u.Event2, >_w)
        else if (u.Value > cout) then
            removeEdge(u.Event1, c, >_w)
            addEdge(u.Event1, u.Event2, >_w)
        cin ← max(d(a, u.Event2)|a ≠ u.Event1)
        c ← {a|d(a, u.Event2) = cin}
        if cin = 0 then
            addEdge(u.Event1, u.Event2, >_w)
        else if (u.Value > cin) then
            removeEdge(c, u.Event2, >_w)
            addEdge(u.Event1, u.Event2, >_w)
        foreach e ∈ {e|(u.Event1, e) ∈ dg ∧ (e, u.Event1) ∈
          dg ∧ e ≠ u.Event2} do
            f ← max(d(e, b)|b ≠ u.Event1)
            if f − u.Value > ρ then
                removeEdge(u.Event1, u.Event2, >_w)
        end
        foreach e ∈ {e|(e, u.Event2) ∈ dg ∧ (u.Event2, e) ∈
          dg ∧ e ≠ u.Event1} do
            f ← max(d(a, e)|a ≠ u.Event1)
            if f − u.Value > ρ then
                removeEdge(u.Event1, u.Event2, >_w)
        end
        if d > α then
            addEdge(u.Event1, u.Event2, >_w)
        foreach e ∈ {e|(u.Event1, e) ∈ dg ∧ e ≠
          u.Event2 ∧ d(u.Event1, e) − u.Value < ρ} do
            addEdge(u.Event1, u.Event2, >_w)
        end
```

Algorithm 2: Update Consumer (part 2)

```
Function addEdge(event1, event2, dep.type)
    if dg(event1, event2) ≠ ∅ then
        dg(event1, event2) ← dg(event1, event2) ∪ {dep.type}
    else
        dg(event1, event2) ← {dep.type}
        GraphStream.enqeue(event1, event2, ADD)
Function removeEdge(event1, event2, dep.type)
    if dg(event1, event2) ≠ ∅ then
        dg(event1, event2) ← dg(event1, event2) \ {dep.type}
        if dg(event1, event2) = ∅ then
            dom(dg) ← dom(dg) \ (event1, event2)
            GraphStream.enqeue(event1, event2, REM)
```

Algorithm 3: Update Consumer (part 3)

E. Graph Consumer

The graph consumer processes the dependency graph updates from the graph update stream. While multiple threads read graph updates from each shard, the graph consumer process maintains a single, complete dependency graph, synchronized across threads. At the same time, trace consumer threads read the complete retired traces from the trace stream and run each trace forwards and backwards against the current dependency graph, as described in [3], to update a single, complete augmented CNet. Trace consumer threads run traces independently against the dependency graph, but to prevent updates to the dependency graph while a trace is being run, graph consumer threads and trace consumer threads are run alternatingly. The result of this step is a complete dependency

graph and an augmented CNet, written to file output or visualized.

V. Implementation and Experiment

We implemented our method employing the Amazon Web Services (AWS) commercial cloud infrastructure. Source code is available from the first author. AWS Kinesis provides the stream infrastructure, processors are distributed across different AWS EC2 instances, performance data is collected using AWS CloudWatch and visualized in a CloudWatch dashboard (Figs. 2, 3).

Each Kinesis shard supports up to 1000 records per second for writing. Each shard can support up to 5 transactions per second for reading, with up to 10000 records read in each transaction. Shards can be dynamically split to increase capacity. There is no limit to the number of shards per stream. Each shard can be written to by many different client applications but can only be read from by one client application at a time.

For our experiment, we provisioned an event stream and a relation stream with a maximum of 20 shards each, for a total capacity of up to 1,200,000 records per minute for each stream. Because the relation consumer performs significant data reduction, the trace stream, update stream, and graph update stream were provisioned with a maximum of 2 shards each for a capacity of up to 120,000 records per minute for each stream. We further provisioned 5 AWS EC2 instances with 16 compute cores and 32GB memory each (type m4.2xlarge) to run the event generators, the event consumers, the relation consumer, the update consumer, and the graph consumer.

Because ours is a faithful implementation of the FHM algorithm, the model generated from the streaming data is the same as if the streaming data was collected and used as "stationary" data in the original FHM algorithm. Hence, our experimental evaluation does not assess the quality of the mined model, such as replay fitness, precision, or generalization. Instead, we focus on demonstrating the scalability of our approach on a commercial cloud platform.

Each consumer runs one processing thread for each shard of its input stream, as indicated in Fig. 1. AWS Kinesis provides consumers with information about how far the current read transaction is behind the "tip" of the shard, the latest write time. The processing threads adapt their read and write rates to catch up to the tip of the shard while remaining within the AWS Kinesis limits, and throttle their read rates once the tip has been reached to match the stream write rate. For this, threads control the thread sleep time after each read transaction and processing of the read records, and the number of records read per read transaction. Each processing shard can persist its state information (reading position in the shard, retired and active cases for the event consumer; current dependency measure values for the relation consumer; partial dependency graph for the update consumers; augmented CNet for the trace consumer) and be restarted without loss of information.

Using the PLG process log generator [15], we produced an event log stored on AWS S3. To simulate random event arrivals for our experiment, a set of event generator threads inserted events from this log into the event stream with a Poisson-distributed interarrival rate. To change the rate of event production two parameters could be configured: the number of concurrent traces that are simulated, and the mean time between successive events for each simulated trace.

We gradually increased the rate at which events are written to the event stream, from $\approx 2,000,000$ events per minute to $\approx 5,500,000$ events per minute. Fig. 2 shows the AWS CloudWatch dashboard with read and write rates for the event and relation stream over the 3 hour period during which we conducted our experiment. The figure shows that, as the event generators increase the rate at which events arrive, the event consumer matches this rate in reading events from the stream. It also shows that the read and write rates for the relation stream are independent of the rate at which events are processed: The data volume for the relation stream depends on the complexity of each trace (i.e. how many instances of the log-based ordering relations are generated for each event), and the number of separate shards. In our experiment that rate was $\approx 3,500,000$ records per minutes.

Figure 3 shows the AWS CloudWatch dashboard with read and write rates for the trace stream, the update stream, and the graph update stream, over the same 3 hour period. These are graphed in a separate diagram because of their much lower data volumes. The volume for the trace stream mirrors that of the event stream: As more events arrive (either faster, or for more cases), more cases per minute will be retired and their traces emitted into the trace stream, from a low of $\approx 57,000$ traces per minutes, to a high of $\approx 160,000$ traces per minute. On the other hand, the update and graph update stream volumes mirror that of the relation stream, but at significantly lower rates as more and more data reduction is performed. The data rate for the update stream fluctuated around $\approx 110,000$ update records per minute and that of the graph udpate stream around $\approx 87,000$ graph update records per minute.

Both figures show that there is significant variation around the mean data rate for each stream, especially as consumers adjust to variations in the inbound data rate, so that it is important to provision sufficient stream capacity. The CPU load and memory consumption for most processing nodes was negligible even at the highest data volume; only the event consumer experienced a significant CPU load of $\approx 10\%$.

VI. Discussion and Conclusions

This research had four distinct goals (cf. Sec. I). We have demonstrated that a popular and widely used process discovery algorithm can be adapted to process events on an event-by-event basis. For the FHM algorithm, this is primarily the adaptation of the dependency graph construction algorithm in Def. 3 in our algorithms 1–3. We have demonstrated that the event processing algorithm can be distributed. Each of the processors in our algorithm works independently on a separate compute node, connected only by the event stream infrastructure. We have provided an implementation on a commercially available compute cloud, which demonstrates

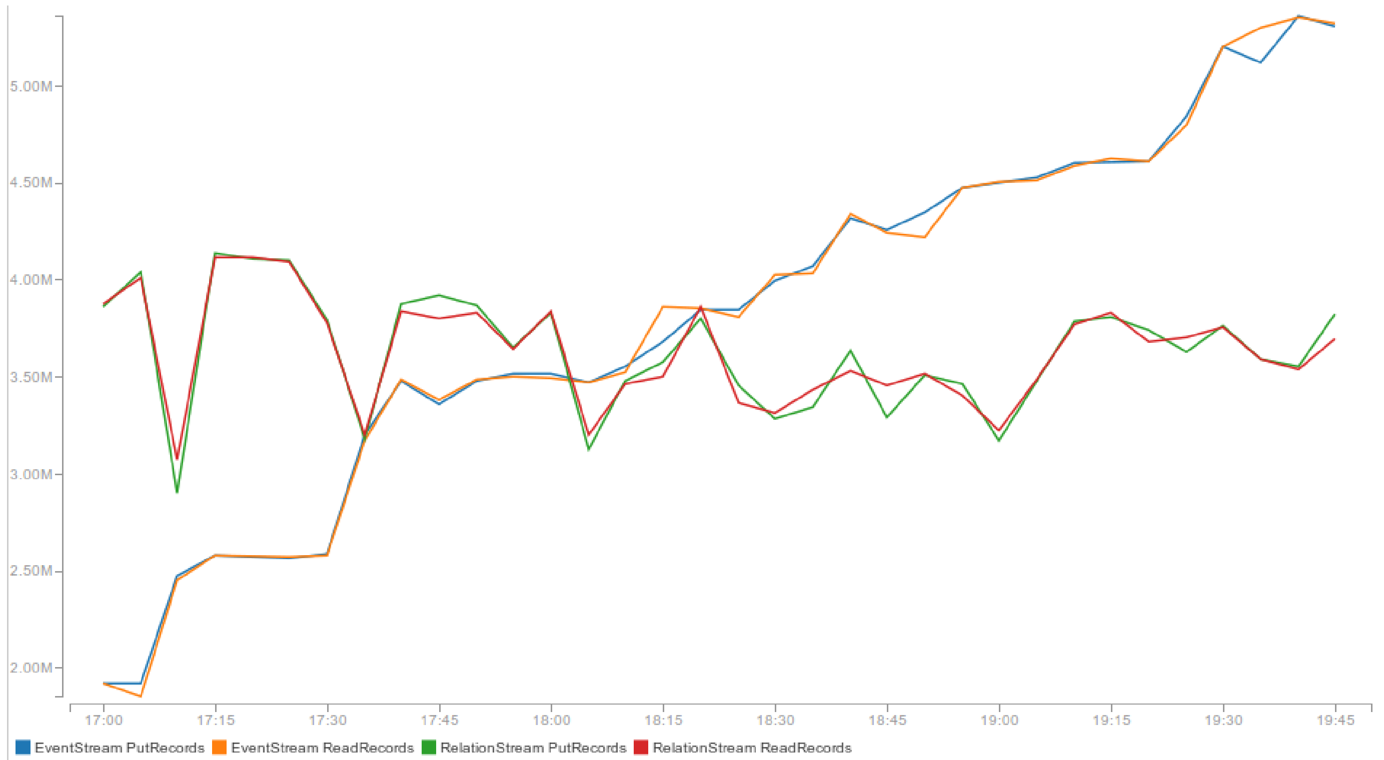

Fig. 2. Records per minute for event and relation streams, 5 minute averages

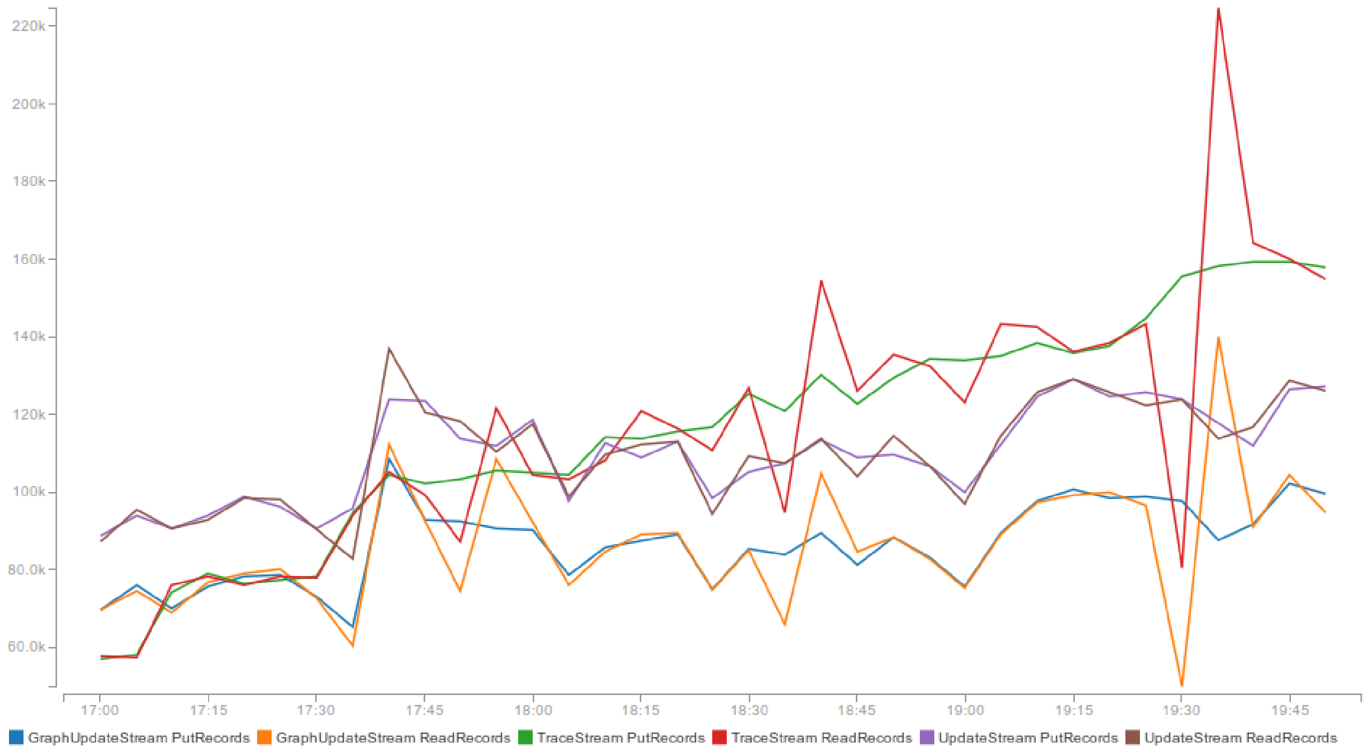

Fig. 3. Records per minute for trace, update and graph update streams, 5 minutes averages

the practical utility of this work. Finally, we have demonstrated the scalability of the solution.

Figures 2 and 3 show that only the event stream requires significant capacity. As the event consumer uses independent threads for each shard, there is in practice no limit to the throughput capacity of our implementation: More shards can be added to the stream as required, and more processing threads can be added, even across multiple AWS EC2 instances. As the description in Section IV indicates, only the event consumer maintains state information that depends on the inbound event volume. However, only the set of case IDs for retired traces grow continuously, whereas the remainder of the state information concerns active cases and is not retained once those cases are retired. With the assumption of a consecutive numbering of case IDs, only the latest retired case ID would need to be stored. The other parts of the algorithm maintain state information that grows with the number of different event types in the trace data, which is significantly smaller, of the order of tens to hundreds.

Our research has some limitations that need to be addressed in further research work. For example, while our distributed implementation on an event stream infrastructure clearly aids the scalability of our approach, the present implementation does not deal with "concept drift" [16], i.e. changes in structure of the process underlying the event stream. Other streaming implementations of the FHM, such as those in [10], [11] are able to identify and react to concept drift by means of "aging" or "decaying" values for log relations and dependency measures. Another area of further work is to empirically explore the relationship between the event stream characteristics (for example, number of event types, complexity of the underlying process) and the relative data volume and therefore the requirements for the various event streams in our architecture.

REFERENCES

[1] W. M. P. van der Aalst, "Process mining: Overview and opportunities," *ACM Trans. Management Inf. Syst.*, vol. 3, no. 2, p. 7, 2012. [Online]. Available: http://doi.acm.org/10.1145/2229156.2229157

[2] ——, "Configurable services in the cloud: Supporting variability while enabling cross-organizational process mining," in *On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Meersman, T. S. Dillon, and P. Herrero, Eds., vol. 6426. Springer, 2010, pp. 8–25. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16934-2_5

[3] A. J. M. M. Weijters and J. T. S. Ribeiro, "Flexible heuristics miner (FHM)," in *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, part of the IEEE Symposium Series on Computational Intelligence 2011, April 11-15, 2011, Paris, France*. IEEE, 2011, pp. 310–317. [Online]. Available: http://dx.doi.org/10.1109/CIDM.2011.5949453

[4] J. Claes and G. Poels, "Process mining and the ProM framework: An exploratory survey," in *Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers*, ser. Lecture Notes in Business Information Processing, M. L. Rosa and P. Soffer, Eds., vol. 132. Springer, 2012, pp. 187–198. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36285-9_19

[5] W. M. P. van der Aalst and E. Damiani, "Processes meet big data: Connecting data science with process science," *IEEE Trans. Services Computing*, vol. 8, no. 6, pp. 810–819, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSC.2015.2493732

[6] 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. Services Computing*, vol. 8, no. 6, pp. 847–860, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSC.2015.2476463

[7] S. Hernández, J. Ezpeleta, S. J. van Zelst, and W. M. P. van der Aalst, "Assessing process discovery scalability in data intensive environments," in *2nd IEEE/ACM International Symposium on Big Data Computing, BDC 2015, Limassol, Cyprus, December 7-10, 2015*, I. Raicu, O. F. Rana, and R. Buyya, Eds. IEEE, 2015, pp. 99–104. [Online]. Available: http://dx.doi.org/10.1109/BDC.2015.31

[8] J. Evermann, "Scalable process discovery using map-reduce," *IEEE Trans. Services Computing*, vol. 9, no. 3, pp. 469–481, 2016. [Online]. Available: http://dx.doi.org/10.1109/TSC.2014.2367525

[9] A. Burattin, M. Cimitile, F. M. Maggi, and A. Sperduti, "Online discovery of declarative process models from event streams," *IEEE Trans. Services Computing*, vol. 8, no. 6, pp. 833–846, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSC.2015.2459703

[10] A. Burattin, A. Sperduti, and W. M. P. van der Aalst, "Heuristics miners for streaming event data," *CoRR*, vol. abs/1212.6383, 2012. [Online]. Available: http://arxiv.org/abs/1212.6383

[11] ——, "Control-flow discovery from event streams," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*. IEEE, 2014, pp. 2420–2427. [Online]. Available: http://dx.doi.org/10.1109/CEC.2014.6900341

[12] M. Hassani, S. Siccha, F. Richter, and T. Seidl, "Efficient process discovery from event streams using sequential pattern mining," in *IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015*. IEEE, 2015, pp. 1366–1373. [Online]. Available: http://dx.doi.org/10.1109/SSCI.2015.195

[13] B. Schwegmann, M. Matzner, and C. Janiesch, "A method and tool for predictive event-driven process analytics," in *11. Internationale Tagung Wirtschaftsinformatik, Leipzig, Germany, February 27 – March 1, 2013*, 2013, p. 46. [Online]. Available: http://aisel.aisnet.org/wi2013/46

[14] ——, "preCEP: Facilitating predictive event-driven process analytics," in *Design Science at the Intersection of Physical and Virtual Design - 8th International Conference, DESRIST 2013, Helsinki, Finland, June 11-12, 2013. Proceedings*, ser. Lecture Notes in Computer Science, J. vom Brocke, R. Hekkala, S. Ram, and M. Rossi, Eds., vol. 7939. Springer, 2013, pp. 448–455. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38827-9_36

[15] A. Burattin and A. Sperduti, "PLG: A framework for the generation of business process models and their execution logs," in *Business Process Management Workshops - BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13-15, 2010, Revised Selected Papers*, ser. Lecture Notes in Business Information Processing, M. zur Muehlen and J. Su, Eds., vol. 66. Springer, 2010, pp. 214–219. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20511-8_20

[16] R. P. J. C. Bose, W. M. P. van der Aalst, I. Zliobaite, and M. Pechenizkiy, "Handling concept drift in process mining," in *Advanced Information Systems Engineering - 23rd International Conference, CAiSE 2011, London, UK, June 20-24, 2011. Proceedings*, ser. Lecture Notes in Computer Science, H. Mouratidis and C. Rolland, Eds., vol. 6741. Springer, 2011, pp. 391–405. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21640-4_30