Process Discovery using Big Data Event Stream Processing - A Scalable, Distributed Implementation of the Flexible Heuristics Miner on the Amazon Kinesis Cloud Infrastructure

Joerg Evermann\textsuperscript{1}, Jana-Rebecca Rehse\textsuperscript{2,3}, and Peter Fettke\textsuperscript{2,3}

\textsuperscript{1} Memorial University of Newfoundland
\textsuperscript{2} Deutsches Forschungszentrum für Künstliche Intelligenz
\textsuperscript{3} Universität des Saarlandes

Abstract. An important characteristic of big data is high velocity. Data is generated rapidly and must be processed quickly, preferably on-the-fly rather than from persistent storage. Event stream processing has become an important tool in managing big data. In this paper, we show how process mining can benefit from stream processing to yield mining methods that scale effortlessly to tens of millions of events per minute. Specifically, we describe a distributed, streaming implementation of the flexible heuristics miner on Amazon Kinesis, a cloud-based event stream infrastructure.

Key words: Process mining, event stream processing, big data, cloud computing, flexible heuristics miner, distributed processing

1 Introduction

Many information systems produce event logs that capture the actions of their users. Examples of event logs are page requests of web-servers and business-object method calls in ERP systems. Process discovery is that area of process mining that deals with the identification of processes from 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].

The increasingly rapid creation and increased availability of such data has been captured in the notion of ”Big Data”. Big data is frequently characterized by a high velocity (volume per time) of data, which makes it impractical to store the data for any length of time and requires that the data be processed on-the-fly. A useful abstraction for this are event streams that connect a set of independent data processors that implement a distributed data analysis algorithm.

Process mining of big data has only been a very recent research topic [2] and only a few approaches have been presented [3, 4, 5, 6, 7]. The Flexible Heuristics Miner (FHM) [9] is a simple yet useful and widely used [8] heuristic for constructing process models from event traces. A streaming version of the FHM has been presented recently [4]. However, this implements the FHM on a
sliding window of batches of event traces. Rather than operating on individual
events as they are generated, it operates on each batch as if it were a complete
log. Thus, it requires significant amount of event storage (the extent of the sliding
window), which limits its scalability. It is also not a distributed implementation,
further limiting its scalability. Finally, events are processed multiple times as
they are within the window during multiple sliding iterations, making it an
inefficient approach.

Our approach addresses these limitations. We build on [7] where it is shown
that the FHM can be separated into discrete processing stages operating in
parallel. We implement the FHM algorithm in a distributed way on independent
processing nodes that ingress event stream 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.

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.

2 Event Stream Processing in the Cloud: The example of
Amazon Kinesis

Amazon Kinesis, part of Amazon Web Services (AWS), provides a scalable event
stream infrastructure for records of arbitrary form. A stream is logically divided
into one or many shards. When writing to a stream, a producer provides a key
for each record; records with the same key are written to the same shard. Each
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. There is no limit to the number of shards per stream.

3 Distributed Event Stream FHM

Building on [7], we separate the FHM algorithm into individual processing stages
and distribute these stages to different computation nodes. The partitioning of
records by shards is used to separate the processing of events using multiple
independent processing threads for each stage. Figure 1 presents an overview of the entire architecture. For space reasons, we cannot describe the original FHM algorithm in this paper, the reader is referred to [9] for details.

3.1 Event Generators

Event generators produce the raw events to be processed. Events are tuples of the form \((Event, TimeStamp, CaseId)\) 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.
3.2 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. Each thread maintains a trace, mean interarrival time, and the time of the last observed event for each active case. When an event is received for an active case, the basic log relations (Def. 3 in [9]) are computed for this new event.

Active cases are periodically retired to conserve memory. Assuming a Poisson distributed interarrival 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. Late events, arriving with less than 0.01% probability, are discarded. The size of the confidence interval is configurable and represents a trade-off between being able to process late events and conserving thread memory.

Basic log relation entries for each processed event are collected and emitted into the relation stream as tuples \( ((\text{Event}_1, \text{Event}_2), \text{RelationType}, \text{Count}) \) where \text{RelationType} indicates the type of basic relation (Def. 3 in [9]), and \text{Count} indicates how many instances of each event pair \( (\text{Event}_1, \text{Event}_2) \) are added to this log relation based on the currently processed event: Appending an event to an active trace generates only one instance of \( >_w \) and \( >>_w \) but can generate multiple instances of \( >>>_w \). These records are written to the relation stream using \( (\text{Event}_1, \text{Event}_2) \) as key. Because the subsequent computation of the dependency measures requires information not only about the pair \( (\text{Event}_1, \text{Event}_2) \) but also about its 'inverse' \( (\text{Event}_2, \text{Event}_1) \), these are written with the same key and so to the same shard.

3.3 Relation Consumer

The relation consumer reads records from the relation stream. Because the dependency measures (Def. 4 in [9]) for a pair of events are based only on the basic log relations for that pair and its inverse pair, processing is performed by independent threads, each serving a shard of the stream. Each thread maintains a count of the instances of each relation type for each event pair, as well as current values for each dependency measure. Counts and dependency measure values are updated when a new record is read from the relation stream. Updated dependency measure values are emitted to the update stream as a tuple \( (\text{Event}_1, \text{Event}_2, \text{Type}, \text{Value}) \) using the event pair as a key to ensure that tuples for the same event 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.

3.4 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
Data:
\( r \), a set of retired cases
\( a : \text{CaseId} \rightarrow \text{Trace} \), traces for active cases
\( i : \text{CaseId} \rightarrow \text{Time} \), mean interarrival times for each case
\( l : \text{CaseId} \rightarrow \text{Time} \), last arrival times for each case

Function \text{EventConsumer}()

\[
\text{while true do}
\]

\[
\text{\hspace{1cm} e \leftarrow \text{EventStream.dequeue}()}
\]

\[
\text{\hspace{1cm} if e.CaseId} \notin r \text{ then}
\]

\[
\text{\hspace{2cm} if e.CaseId} \in \text{dom}(a) \text{ then}
\]

\[
\text{\hspace{3cm} t \leftarrow a(e.\text{CaseId})}
\]

\[
\text{\hspace{3cm} i(e.\text{CaseId})} \leftarrow it(e.\text{CaseId}) \times t \text{.length} + e.\text{Timestamp} - l(e.\text{CaseId})
\]

\[
\text{\hspace{3cm} l(e.\text{CaseId})} \leftarrow e.\text{Timestamp}
\]

\[
\text{\hspace{1cm} else}
\]

\[
\text{\hspace{2cm} t = \text{new Trace}}
\]

\[
\text{\hspace{3cm} a \leftarrow a \cup (e.\text{CaseId}, t)}
\]

\[
\text{\hspace{3cm} i \leftarrow i \cup (e.\text{CaseId}, 0)}
\]

\[
\text{\hspace{3cm} l \leftarrow l \cup (e.\text{CaseId}, e.\text{Timestamp})}
\]

\[
\text{\hspace{1cm} end}
\]

\[
\text{\hspace{1cm} t \leftarrow t \cup (e.\text{Timestamp}, e.\text{Event})}
\]

\[
\text{\hspace{1cm} relations} \leftarrow \text{computeBasicLogRelations(t) } / \text{ Def. 3 in [9]}
\]

\[
\text{\hspace{1cm} RelationStream.enqueue(relations)}
\]

\[
\text{\hspace{1cm} end}
\]

\[
\text{retireTraces()}
\]

\[
\text{end}
\]

Function \text{retireTraces}()

\[
\text{foreach c} \in \text{dom}(a) \text{ do}
\]

\[
\text{\hspace{1cm} if l(c) + qPoisson(i(c), 0.999) < now then}
\]

\[
\text{\hspace{2cm} TraceStream.enqueue(a(c))}
\]

\[
\text{\hspace{1cm} dom(a)} \leftarrow \text{dom(a)} \setminus c; \text{dom(i)} \leftarrow \text{dom(i)} \setminus c; \text{dom(l)} \leftarrow \text{dom(l)} \setminus c
\]

\[
\text{\hspace{1cm} r} \leftarrow r \cup c
\]

\[
\text{end}
\]

\[
\text{end}
\]

Algorithm 1: Outline of the Event Consumer

A list of current values for each dependency measure for each event pair it processes. When reading a record containing a new dependency measure value, the processing thread considers each of the sets that form the FHM algorithm in Def. 6 in [9] and determines whether the new dependency measure value yields any changes to the different sets. The reconceptualization of the core FHM algorithm in terms of updates to the different sets (Algorithms 3–5) is one of the core features of our implementation. There are two notable considerations: First, the loop-of-length-one and loop-of-length-two dependency measures can only grow, not shrink: Updates to these can never cause edge removals from the dependency graph. Second, edges can result from multiple, different types of dependencies. Hence, edges can only be removed when not supported by any dependency. The update consumer thread maintains a set of dependency types that support each graph edge (Alg. 3) and removes edges only when the last supporting dependency type is removed (Alg. 5). Changes to the graph are emitted as tuples \((\text{Event1}, \text{Event2}, \text{UpdateOp})\) into the graph update stream where \text{UpdateOp} indicates removal or addition of an edge. Records are keyed by event 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 [9].
Data:
  \( c : \text{Event} \mapsto \mathbb{N} \), a map of counts \(|\{!\}|\) for each event type \( t \)
  \( f : \text{Event} \times \text{Event} \mapsto \mathbb{N} \), a map of \(|\rightarrow|\) for each event type pair
  \( t_2 : \text{Event} \times \text{Event} \mapsto \mathbb{N} \), a map of \(|\rightarrow|\) for each event type pair
  \( t_2 : \text{Event} \times \text{Event} \mapsto \mathbb{N} \), a map of \(|\rightarrow|\) for each event type pair

Function \( \text{RelationConsumer()} \)

while true do
  \( r \leftarrow \text{RelationStream.dequeue()} \)
  \( ep \leftarrow (r.\text{Event1}, r.\text{Event2}) \)
  \( pe \leftarrow (r.\text{Event2}, r.\text{Event1}) \)
  switch \( r.\text{RelationType} \)
    case \( \rightarrow \) do
      \( f(ep.\text{Event1}, ep.\text{Event2}) \leftarrow f(ep.\text{Event1}, ep.\text{Event2}) + r.\text{Count} \)
      \( d_1 \leftarrow (ep.\text{Event1}) - (ep.\text{Event2}) \)
      \( d_2 \leftarrow (ep.\text{Event2}) - (ep.\text{Event1}) \)
      \( \text{DependencyStream.enqueue}((ep.\text{Event1}, ep.\text{Event2}, \rightarrow, d_1)) \)
      \( \text{DependencyStream.enqueue}((ep.\text{Event2}, ep.\text{Event1}, \rightarrow, d_2)) \)
    case \( \rightarrow \rightarrow \) do
      \( t_2(ep.\text{Event1}, ep.\text{Event2}) \leftarrow t_2(ep.\text{Event1}, ep.\text{Event2}) + r.\text{count} \)
      \( d_1 \leftarrow (ep.\text{Event1}) - (ep.\text{Event2}) \)
      \( d_2 \leftarrow (ep.\text{Event2}) - (ep.\text{Event1}) \)
      \( \text{DependencyStream.enqueue}((ep.\text{Event1}, ep.\text{Event2}, \rightarrow \rightarrow, d_1)) \)
      \( \text{DependencyStream.enqueue}((ep.\text{Event2}, ep.\text{Event1}, \rightarrow \rightarrow, d_2)) \)
    case \( \rightarrow \rightarrow \rightarrow \) do
      \( t_2(ep.\text{Event1}, ep.\text{Event2}) \leftarrow t_2(ep.\text{Event1}, ep.\text{Event2}) + r.\text{count} \)
      \( d_1 \leftarrow 2 \cdot (ep.\text{Event1}) - (ep.\text{Event2}) \)
      \( d_2 \leftarrow 2 \cdot (ep.\text{Event2}) - (ep.\text{Event1}) \)
      \( \text{DependencyStream.enqueue}((ep.\text{Event1}, ep.\text{Event2}, \rightarrow \rightarrow \rightarrow, d_1)) \)
      \( \text{DependencyStream.enqueue}((ep.\text{Event2}, ep.\text{Event1}, \rightarrow \rightarrow \rightarrow, d_2)) \)
    otherwise do
      \( c(ep.\text{Event1}) \leftarrow c(ep.\text{Event1}) + r.\text{count} \)
  end
end

Algorithm 2: Outline of the Relation Consumer

Data:
  \( \sigma_0^\rightarrow, \sigma_0^\rightarrow \rightarrow, \sigma_0^\rightarrow \rightarrow \rightarrow \leftarrow 0.9 \), initial dependency thresholds
  \( \rho \leftarrow 0.05 \), initial relative-to-best threshold for dependencies
  \( d : \text{Event} \times \text{Event} \mapsto \mathbb{R} \), a map of direct dependencies for each event pair
  \( t_2 : \text{Event} \times \text{Event} \mapsto \mathbb{R} \), a map of loop-two dependencies for each event pair
  \( d_2 : \text{Event} \times \text{Event} \mapsto \{ \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \} \), a map of edges of the partial dependency graph to the power set of dependency relation types

Function \( \text{UpdateConsumer()} \)

while true do
  \( u \leftarrow \text{UpdateStream.dequeue()} \)
  switch \( u.\text{Type} \)
    case \( \rightarrow \) do
      \( \text{ProcessDirectFollows}(u) \)
    case \( \rightarrow \rightarrow \) do
      \( o \leftarrow (u.\text{Event1}, u.\text{Event2}) \leftarrow u.\text{Value} \)
      if \((u.\text{Value} > \omega) \land (u.\text{Value} \geq \sigma_0^\rightarrow) \land (\omega < \sigma_0^\rightarrow)\) then
        \( \text{addEdge}(u.\text{Event1}, u.\text{Event2}, \rightarrow \rightarrow) \)
        \( \text{addEdge}(u.\text{Event2}, u.\text{Event1}, \rightarrow \rightarrow) \)
      end
    case \( \rightarrow \rightarrow \rightarrow \) do
      if \( u.\text{Value} \geq \sigma_0^\rightarrow \rightarrow \rightarrow \) then
        \( \text{addEdge}(u.\text{Event1}, u.\text{Event2}, \rightarrow \rightarrow \rightarrow) \)
      end
      if \( u.\text{Value} < \sigma_0^\rightarrow \rightarrow \rightarrow \) then
        \( \text{removeEdge}(u.\text{Event1}, u.\text{Event2}, \rightarrow \rightarrow \rightarrow) \)
      end
    end
end

Algorithm 3: Outline of Update Consumer
3.5 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 maintains a single, complete dependency graph. At the same time, trace consumer threads read the complete and retired traces from the trace stream and run each trace forwards and backwards against the current...
dependency graph, as described in [9], to update a single, complete augmented CNet maintained by the graph consumer. Trace consumer threads can run traces independently against the dependency graph, but to prevent updates to the dependency graph while a trace 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.

\[
\begin{align*}
data: & \quad dg \subseteq Event \times Event, \text{the set of edges of the dependency graph} \\
& \quad CNet_{\text{succ}} : (Event, Event) \rightarrow \mathbb{N}, \text{augmented CNet, successor counts} \\
& \quad CNet_{\text{pred}} : (Event, Event) \rightarrow \mathbb{N}, \text{augmented CNet, predecessor counts} \\
\end{align*}
\]

\begin{algorithm}
\caption{Trace Consumer()}
\begin{algorithmic}
\While{true}
\For{$i \in 1 \ldots \text{length}(t) - 1$}
\State $\text{cand}_{\text{succ}} \leftarrow \emptyset$
\For{$s | (t_i, s) \in dg$}
\If{$(\exists j, t_j \in t_i + 1 \leq j \leq n)$}
\State $s_{\text{ind}} \leftarrow \min(j | t_j \in t \wedge j > i \wedge t_j = s)$
\State $p \leftarrow \{a | (a, s) \in dg\}$
\If{$p \cap t_i+1 \ldots t_{s_{\text{ind}}} \neq \emptyset$}
\State $\text{cand}_{\text{succ}} \leftarrow \text{cand}_{\text{succ}} \cap s$
\EndIf
\EndIf
\EndFor
\State $CNet_{\text{succ}}(t_i, \text{cand}_{\text{succ}}) +$
\EndFor
\For{$i \in 2 \ldots \text{length}(t)$}
\State $\text{cand}_{\text{pred}} \leftarrow \emptyset$
\For{$s | (s, t_i) \in dg$}
\If{$(\exists j, t_j \in t \wedge j \leq i - 1)$}
\State $s_{\text{ind}} \leftarrow \max(j | t_j \in t \wedge j < i \wedge t_j = s)$
\State $p \leftarrow \{a | (s, a) \in dg\}$
\If{$p \cap t_{s_{\text{ind}}} \ldots t_{i-1} \neq \emptyset$}
\State $\text{cand}_{\text{pred}} \leftarrow \text{cand}_{\text{pred}} \cap s$
\EndIf
\EndIf
\EndFor
\State $CNet_{\text{pred}}(t_i, \text{cand}_{\text{pred}}) +$
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

4 Implementation and Experiment

We implemented our method employing the Amazon Web Services (AWS) commercial cloud infrastructure. Source code is available\textsuperscript{1}. 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). For our experiment, we provisioned an event stream and a relation stream with 20 shards each, for a total capacity of 1,200,000 records per minute for each stream. Because the relation consumer performs significant data reduction, the trace stream, update stream, and graph

\textsuperscript{1}http://joerg.evermann.ca/software.html
The update stream were provisioned with only 2 shards each for a capacity of 120,000 records per minute for each stream.

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 (Sec. 2), 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 [10], we produced an event log stored on AWS S3. To simulate random event arrivals, a set of event generator threads inserted events from this log into the event stream with a Poisson-distributed interarrival rate.

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 basic log 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 update 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 $\approx 10\%$.

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

This research had four distinct goals (see Sec. 1). 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 algorithm of Def. 6 in [9] in our algorithms 3–5. We have demonstrated that the event processing algorithm can be distributed. Each of the processors in our algorithm works independently on separate compute nodes, connected only by the event stream infrastructure. We have provided an implementation on a commercially available compute cloud, which demonstrates 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 algorithms shown in Section 3 show, 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. The other algorithms maintain state information that grows with the number of different event types in the traces, which is significantly smaller.

References