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[1], Jana-Rebecca Rehse[2,3], and Peter Fettke[2,3]

[1] Memorial University of Newfoundland
[2] Deutsches Forschungszentrum für Künstliche Intelligenz
[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 impractial 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 indepedent 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 arbitary 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.

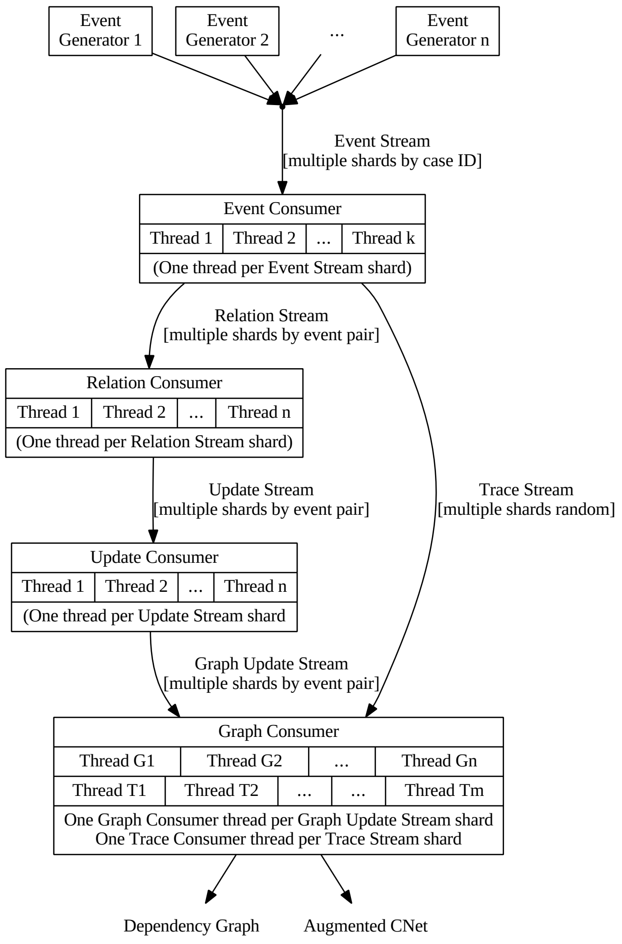
Event Generator 1
Event Generator 2
...
Event Generator n
Event Stream [multiple shards by case ID]
Event Consumer
Thread 1
Thread 2
...
Thread k
(One thread per Event Stream shard)
Relation Stream [multiple shards by event pair]
Relation Consumer
Thread 1
Thread 2
...
Thread n
(One thread per Relation Stream shard)
Update Stream [multiple shards by event pair]
Trace Stream [multiple shards random]
Update Consumer
Thread 1
Thread 2
...
Thread n
(One thread per Update Stream shard
Graph Update Stream [multiple shards by event pair]
Graph Consumer
Thread G1
Thread G2
...
Thread Gn
Thread T1
Thread T2
...
...
Thread Tm
One Graph Consumer thread per Graph Update Stream shard
One Trace Consumer thread per Trace Stream shard
Dependency Graph
Augmented CNet

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

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 $((Event1, Event2), RelationType, Count)$ where $RelationType$ indicates the type of basic relation (Def. 3 in [9]), and $Count$ indicates how many instances of each event pair $(Event1, Event2)$ 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 $(Event1, Event2)$ as key. Because the subsequent computation of the dependency measures requires information not only about the pair $(Event1, Event2)$ but also about its 'inverse' $(Event2, Event1)$, 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 $Event1, Event2, Type, Value)$ 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
  $a : CaseId \mapsto Trace$, traces for active cases
  $i : CaseId \mapsto Time$, mean interarrival times for each case
  $l : CaseId \mapsto Time$, last arrival times for each case
Function EventConsumer()
    while $true$ do
        $e \leftarrow EventStream.dequeue()$
        if $e.CaseId \notin r$ then
            if $e.CaseId \in dom(a)$ then
                $t \leftarrow a(e.CaseId)$
                $i(e.CaseId) \leftarrow \frac{i(e.CaseId) \times t.length + (e.TimeStamp - l(e.CaseId)}{t.length+1}$
                $l(e.caseID) \leftarrow e.Timestamp$
            else
                $t =$ new $Trace$
                $a \leftarrow a \cup (e.CaseId, t)$
                $i \leftarrow i \cup (e.caseID, 0)$
                $l \leftarrow l \cup (e.CaseId, e.Timestamp)$
            end
            $t \leftarrow t \cup (e.Timestamp, e.Event)$
            $relations \leftarrow computeBasicLogRelations(t)$    // Def. 3 in [9]
            $RelationStream.enqueue(relations)$
        end
        $retireTraces()$
    end
Function retireTraces()
    foreach $c \in dom(a)$ do
        if $l(c) + qPoisson(i(c), 0.999) < now$ then
            $TraceStream.enqueue(a(c))$
            $dom(a) \leftarrow dom(a) \setminus c; dom(i) \leftarrow dom(i) \setminus c; dom(l) \leftarrow dom(l) \setminus c$
            $r \leftarrow r \cup c$
        end
    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 $(Event1, Event2, UpdateOp)$ into the graph update stream where $UpdateOp$ 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:
  $c : Event \mapsto \mathbb{N}$, a map of counts ($|t|$) for each event type $t$
  $f : Event \times Event \mapsto \mathbb{N}$, a map of $| >_w |$ for each event type pair
  $l2 : Event \times Event \mapsto \mathbb{N}$, a map of $| >>_w |$ for each event type pair
  $ld : Event \times Event \mapsto \mathbb{N}$, a map of $| >>>_w |$ for each event type pair
Function $RelationConsumer()$
    while $true$ do
        $r \leftarrow RelationStream.dequeue()$
        $ep \leftarrow (r.Event1, r.Event2);\ pe \leftarrow (r.Event2, r.Event1)$
        switch $r.RelationType$ do
            case $>_w$ do
                $f(ep.Event1, ep.Event2) \leftarrow f(ep.Event1, ep.Event2) + r.Count$
                $d1 \leftarrow \frac{f(ep)-f(pe)}{f(ep)+f(pe)+1};\ d2 \leftarrow \frac{f(pe)-f(ep)}{f(pe)+f(ep)+1}$
                $DependencyStream.enque((ep.Event1, ep.Event2, >_w, d1))$
                $DependencyStream.enque((ep.Event2, ep.Event1, >_w, d2))$
            case $>>_w$ do
                $l2(ep.Event1, Event2) \leftarrow l2(ep.Event1, ep.Event2) + r.count$
                $d1 \leftarrow \frac{l2(ep)-l2(pe)}{l2(ep)+l2(pe)+1};\ d2 \leftarrow \frac{l2(pe)-l2(ep)}{l2(pe)+l2(ep)+1}$
                $DependencyStream.enque((ep.Event1, ep.Event2, >>_w, d1))$
                $DependencyStream.enque((ep.Event2, ep.Event1, >>_w, d2))$
            case $>>>_w$ do
                $ld(ep.Event1, Event2) \leftarrow ld(ep.Event1, ep.Event2) + r.count$
                $d1 \leftarrow 2\ ld(ep) - \frac{|c(ep.Event1)-c(ep.Event2)|}{c(ep.Event1)+c(ep.Event2)+1}$
                $d2 \leftarrow 2\ ld(pe) - \frac{|c(ep.Event2)-c(ep.Event1)|}{c(ep.Event2)+c(ep.Event1)+1}$
                $DependencyStream.enqueue((ep.Event1, ep.Event2, >>>_w, d1))$
                $DependencyStream.enqueue((ep.Event2, ep.Event1, >>>_w, d2))$
            otherwise do
                $c(ep.Event1) \leftarrow c(ep.Event1) + r.count$
            end
        end
    end
```

Algorithm 2: Outline of the Relation Consumer

```
Data:
  $\alpha, \sigma_{>}, \sigma_{>>}, \sigma_{>>>} \leftarrow 0.9$, initial dependency thresholds
  $\rho \leftarrow 0.05$, initial relative-to-best threshold for dependencies
  $d : Event \times Event \mapsto \mathbb{R}$, a map of direct dependencies for each event pair
  $l2 : Event \times Event \mapsto \mathbb{R}$, a map of loop-two dependencies for each event pair
  $dg : Event \times Event \mapsto 2^{\{>_w, >>_w, >>>_w\}}$, a map of edges of the partial dependency graph to the
powerset of dependency relation types
Function $UpdateConsumer()$
    while $true$ do
        $u \leftarrow UpdateStream.dequeue()$
        switch $u.Type$ do
            case $>_w$ do
                $ProcessDirectFollows(u)$
            case $>>_w$ do
                $o \leftarrow l2(u.Event1, u.Event2) \leftarrow u.Value$
                if $(u.Value > o) \wedge (u.Value \geq \sigma_{>>}) \wedge (o < \sigma_{>>})$ then
                    $addEdge(u.Event1, u.Event2, >>_w)$
                    $addEdge(u.Event2, uEvent1, >>_w)$
            case $>>>_w$ do
                if $u.Value > \sigma_{>>>}$ then
                    $addEdge(u.Event1, uEvent2, >>>_w)$
                if $u.Value < \sigma_{>>>}$ then
                    $removeEdge(u.Event1, uEvent2, >>>_w)$
        end
        $updateThresholds()$
    end
```

Algorithm 3: Outline of Update Consumer

```
Function ProcessDirectFollows(u)
    $o \leftarrow d(u.Event1, u.Event2)$
    $d(u.Event1, u.Event2) \leftarrow u.Value$
    if $u.Event1 = u.Event2$ then
        if $(o < \sigma_{l1}) \wedge (u.Value \geq \sigma_{l1})$ then
            $addEdge(u.Event1, u.Event2, >_w)$
    else
        $cout \leftarrow \max(d(u.Event1, b) | b \neq u.Event2)$
        $c \leftarrow \{b | d(u.Event1, b) = cout\}$
        if $cout = 0$ then
            $addEdge(u.Event1, u.Event2, >_w)$
        else if $(u.Value > cout)$ then
            $removeEdge(u.Event1, c, >_w)$
            $addEdge(u.Event1, u.Event2, >_w)$
        $cin \leftarrow \max(d(a, u.Event2) | a \neq u.Event1)$
        $c \leftarrow \{a | d(a, u.Event2) = cin\}$
        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 $e \in \{e | (u.Event1, e) \in dg \wedge (e, u.Event1) \in dg \wedge e \neq u.Event2\}$ do
            $f \leftarrow \max(d(e, b) | b \neq u.Event1)$
            if $f - u.Value > \rho$ then
                $removeEdge(u.Event1, u.Event2, >_w)$
        end
        foreach $e \in \{e | (e, u.Event2) \in dg \wedge (u.Event2, e) \in dg \wedge e \neq u.Event1\}$ do
            $f \leftarrow \max(d(a, e) | a \neq u.Event1)$
            if $f - u.Value > \rho$ then
                $removeEdge(u.Event1, u.Event2, >_w)$
        end
        if $d > \alpha$ then
            $addEdge(u.Event1, u.Event2, >_w)$
        foreach $e \in \{e | (u.Event1, e) \in dg \wedge e \neq u.Event2 \wedge d(u.Event1, e) - u.Value < \rho\}$ do
            $addEdge(u.Event1, u.Event2, >_w)$
        end
```

Algorithm 4: Update Consumer (part 2)

```
Function addEdge(event1, event2, dep.type)
    if $dg(event1, event2) = \varnothing$ then
        $dg(event1, event2) \leftarrow dg(event1, event2) \cup \{dep.type\}$
    else
        $dg(event1, event2) \leftarrow \{dep.type\}$
        $GraphStream.enqeue(event1, event2, ADD)$
Function removeEdge(event1, event2, dep.type)
    if $dg(event1, event2) \neq \varnothing$ then
        $dg(event1, event2) \leftarrow dg(event1, event2) \setminus \{dep.type\}$
        if $dg(event1, event2) = \varnothing$ then
            $dom(dg) \leftarrow dom(dg) \setminus (event1, event2)$
            $GraphStream.enqeue(event1, event2, REM)$
```

Algorithm 5: Update Consumer (part 3)

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.

```
Data:
  $dg \subseteq Event \times Event$, the set of edges of the dependency graph
  $CNet_{succ} : (Event, Event^n) \mapsto \mathbb{N}$, augmented CNet, successor counts
  $CNet_{pred} : (Event, Event^n) \mapsto \mathbb{N}$, augmented CNet, predecessor counts
Function $TraceConsumer()$
    while $true$ do
        $t = t_1 \dots t_n \leftarrow TraceStream.dequeue()$
        foreach $i \in 1 \dots length(t) - 1$ do
            $\mathsf{cand}_{succ} \leftarrow \varnothing$
            foreach $s|(t_i, s) \in dg$ do
                if $(\exists t_j \in t | i + 1 \leq j \leq n)$ then
                    $s_{ind} \leftarrow min(j | t_j \in t \wedge j > i \wedge t_j = s)$
                    $p \leftarrow \{a | (a, s) \in dg\}$
                    if $p \cap t_{i+1} \dots t_{s_{ind}} \neq \emptyset$ then
                        $cand_{succ} \leftarrow cand_{succ} \cap s$
            end
            $CNet_{succ}(t_i, cand_{succ}) + +$
        end
        foreach $i \in 2 \dots length(t)$ do
            $\mathsf{cand}_{pred} \leftarrow \varnothing$
            foreach $s|(s, t_i) \in dg$ do
                if $(\exists t_j \in t | 0 \leq j \leq i - 1)$ then
                    $s_{ind} \leftarrow max(j | t_j \in t \wedge j < i \wedge t_j = s)$
                    $p \leftarrow \{a | (s, a) \in dg\}$
                    if $p \cap t_{s_{ind}} \dots t_{i-1} \neq \emptyset$ then
                        $cand_{pred} \leftarrow cand_{pred} \cap s$
            end
            $CNet_{pred}(t_i, cand_{pred}) + +$
        end
    end
```

Algorithm 6: Trace Consumer

4 Implementation and Experiment

We implemented our method employing the Amazon Web Services (AWS) commercial cloud infrastructure. Source code is available[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

[1] `http://joerg.evermann.ca/software.html`

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 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 ≈ 10%.

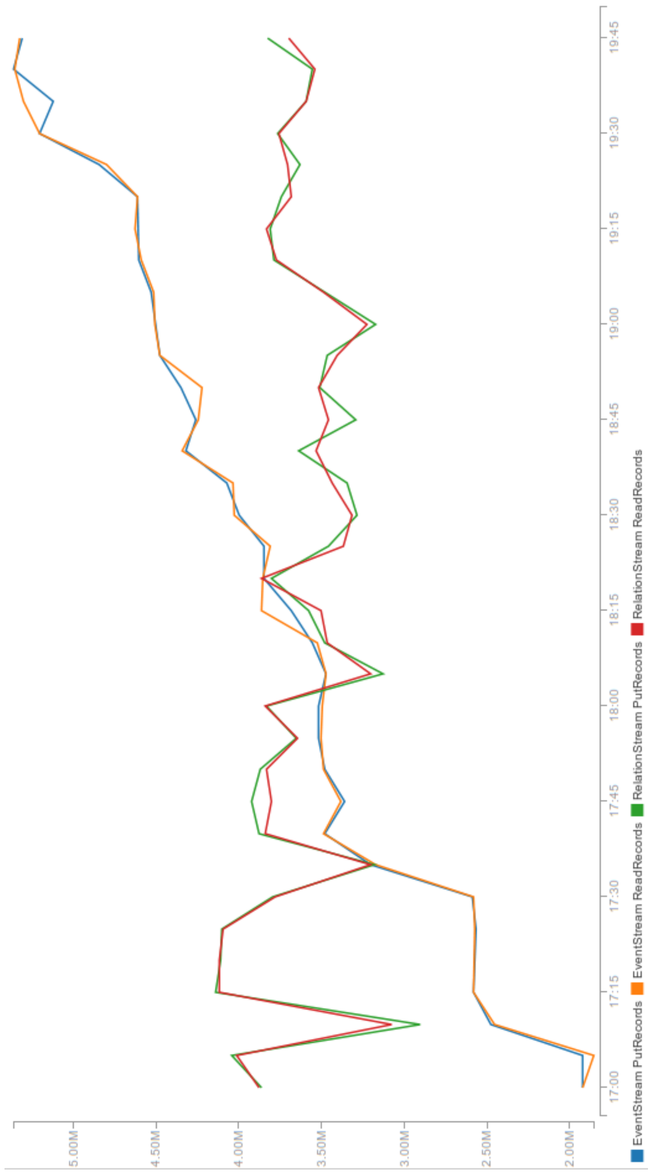

Fig. 2. Records per minute for event and relation streams, 5 minute averages

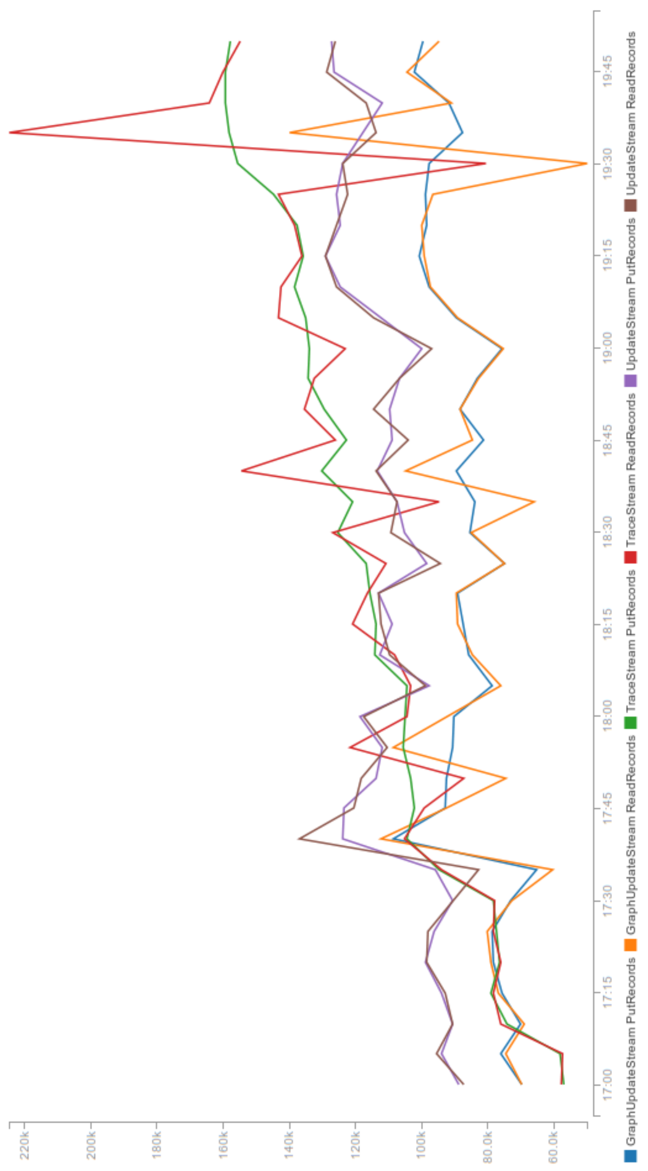

Fig. 3. Records per minute for trace, update and graph update streams, 5 minutes averages

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

1. van der Aalst, W.M.P.: Process mining: Overview and opportunities. ACM Trans. Manage. Inf. Syst. **3**(2) (July 2012) 7:1–7:17
2. van der Aalst, W., Damiani, E.: Processes meet big data: Connecting data science with process science. Services Computing, IEEE Transactions on **8**(6) (Nov 2015) 810–819
3. Reguieg, H., Benatallah, B., Motahari Nezhad, H., Toumani, F.: Event correlation analytics: Scaling process mining using mapreduce-aware event correlation discovery techniques. Services Computing, IEEE Transactions on **8**(6) (Nov 2015) 847–860
4. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Heuristics miners for streaming event data. CoRR **abs/1212.6383** (2012)
5. Burattin, A., Cimitile, M., Maggi, F., Sperduti, A.: Online discovery of declarative process models from event streams. Services Computing, IEEE Transactions on **8**(6) (Nov 2015) 833–846
6. Evermann, J., Assadipour, G.: Big data meets process mining: Implementing the alpha algorithm with map-reduce. In: Proceedings of the ACM Symposium on Applied Computing. (2014)
7. Evermann, J.: Scalable process discovery using map-reduce. Services Computing, IEEE Transactions on **PP**(99) (2014) 1–1
8. Claes, J., Poels, G.: Process mining and the prom framework: An exploratory survey. In: Business Process Management Workshops - BPM 2012. (2012) 187–198
9. Weijters, A., Ribeiro, J.: Flexible heuristics miner (FHM). In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining CIDM 2011, Paris, France. (2011)
10. Burattin, A., Sperduti, A.: PLG: A framework for the generation of business process models and their execution logs. In zur Muehlen, M., Su, J., eds.: Business Process Management Workshops. Volume 66 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg (2011) 214–219