Journal of King Saud University – Computer and Information Sciences xxx (xxxx) xxx

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Processes meet Big Data: Scaling process discovery algorithms in Big Data environment

Reguieg Hicham *, Benallal Mohamed Anis

Laboratoire SIMPA, Département d'informatique, Faculté des Mathématiques et Informatique, Université des Sciences et de la Technologie d'Oran Mohamed Boudiaf, USTO-MB, BP 1505, El Mnaouer, 31000 Oran, Algeria

A R T I C L E I N F O

Article history:
Received 14 December 2020
Revised 9 February 2021
Accepted 18 February 2021
Available online xxxx

Keywords:
Process discovery
Event logs
Big Data
Apache Spark
Distributed computing

A B S T R A C T

Process mining is one of business process management techniques which is used to extract values from process execution logs. Process discovery algorithms, like alpha and heuristic miners, are used to automatically discover/rebuild business process models from event logs. However, the performance of these techniques is limited when dealing with Big Data. To cope with this issue, we propose a distributed implementation, based on Spark framework, of the alpha and heuristic algorithms to support efficient scalable process discovery for big process data. The approach consists of distributing the CPU intensive phases, such as the construction of the causality matrix related to these algorithms. Experimental results show that the proposed algorithms speed-up and scale-up well with regard to the variation of both data size and the number of nodes in the cluster.

1. Introduction

Process mining is a discipline that aims to automate discovery, compliance and improvement of business processes (Van der Aalst, 2016). Specifically, Process discovery is a major area of interest within the field of Process mining. In fact, it consists of analysing a repository of event logs in order to infer models that reproduce the business process behaviour. Indeed, this is a computationally challenging task, since it consists of exploring a vast space of potential relationships between events over a broad and continually increasing repositories of logs.

Centralized process discovery have been widely studied in literature. As Examples of process discovery algorithms we cite α-miner (alpha) (van der Aalst et al., 2004), *Flexible Heuristic Miner* (FHM) (Weijters and Ribeiro, 2011), *Integer Linear programming* (ILP) Miner (van derWerf et al., 2009) and *Inductive Miner* (Leemans et al., 2013). The previously cited algorithms are able to produce process models from event logs. Nonetheless, these techniques are not designed to support the growing size of event log generated by large scale modern information systems (Sakr et al., 2018; van der Aalst, 2013). Velocity, variety and volume of generated event logs are often considered as main challenges of process mining solutions.

Given the distributed nature and large size of event logs, Process mining analysis has received considerable scholarly attention in the field of big data. In (Reguieg et al., 2012; Reguieg et al., 2015), authors investigated MapReduce (Dean and Ghemawat, 2008) to support event correlation discovery in scattered event logs, i.e., to identify events that belong to the same process execution. In addition, MapReduce has been used to implement a scalable α-miner and flexible heuristic Miner in (Hernandez et al., 2015; Evermann, 2016).

With the emergence of in-memory computation clusters (e.g., the Spark platform (Zaharia et al., 2010)), the limitations of MapReduce become increasingly noticeable, such as poor performance for iterative processing (Shi et al., 2015). Therefore, more and more research efforts and applications, namely machine learning (JayaLakshmi and Krishna Kishore, 2018) and data mining (Kumar and Mohbey, 2019) turned to the Spark. Spark is a fast, versatile and scalable in-memory big data computing engine. It provides a one-stop solution for most big data problems. Moreover, it integrates batch processing, real-time stream processing, interactive query, graph computing and machine learning. Despite its benefits, Spark has not been widely investigated in the field of process mining. Considering its advantages and the fact that most of the process mining algorithms are iterative, it makes sense to investigate how Spark performs in this research area.

* Corresponding author.
E-mail address: hicham.reguieg@univ-usto.dz (R. Hicham).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2021.02.008

In this paper, we introduce an efficient scalable process mining techniques, including distributed algorithms, to support process discovery from Big event logs. Relying on the well known process discovery algorithms *Alpha* (van der Aalst et al., 2004), *Alpha+* (Medeiros, 2004; Hung et al., 2020) and *Flexible Heuristic Miner* (Weijters and Ribeiro, 2011), we investigated the processing framework Apache Spark to propose a fast and scalable process discovery algorithms. The main contributions of our work can be summarized as follows:

1. We propose a distributed alternatives to extract the causality matrix from a large event log for the α and alpha+ miners.
2. We introduce a parallel computation for the frequency and dependency matrices in the *Flexible Heuristic Miner*.
3. We prove by performance tests that our proposed algorithms are efficient and scale well to large event logs.

The rest of the paper is organized as follows. Section 2 describes the background of our work. Section 3 discusses some related works. Section 4 presents our main contribution where conceptual architecture of our approach and details the implementation of our algorithms are presented. Evaluation tests performed during this study are then presented in Section 5. Finally, we conclude and outline future works in Section 6.

Table 1
A fragment of an Event log (taken from (Van der Aalst, 2016)).

Case	Event ID	Activity	Timestamp	...
1	eID1	a	ts-1	
1	eID2	b	ts-2	
1	eID3	d	ts-3	
1	eID4	e	ts-4	
1	eID5	h	ts-5	
2	eID6	a	ts-6	
2	eID7	d	ts-7	
2	eID8	C	ts-8	

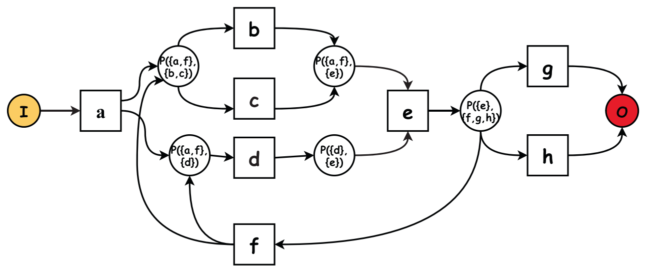


Fig. 1. A *Petri net* model discovered by the Alpha-algorithm corresponding to the footprint matrix depicted in Table 2.

2. Preliminaries

In this section, we summarize the basic concepts in process discovery.

2.1. Process event log

A process event log L (Van der Aalst, 2016), event log for short, can be seen as a relation over a relational schema $\mathscr{L}$ (CaseID, Event id, Activity, Timestamp, resources, ...), where each row presents one event and refers to an activity (*Activity-name attribute*). Moreover, events are ordered by timestamp and grouped together by CaseID to form a process execution instance (a trace). Table 1 illustrates an example of a compact event log. This event log represents the minimum requirement input for any process discovery algorithm to transform the information presented in the log into a process model.

2.2. Petri net

Petri net (Murata, 1989) is a graphical and mathematical model used for descriptions of systems. It allows for the modelling of formal business processes. *Petri net* is an oriented bipartite graph. It consists of three kinds of elements: *transitions* and *places* connected by *directed arcs*. Fig. 1 shows an example of *Petri net* representation.

2.3. Alpha algorithm

The so called α-algorithm (van der Aalst et al., 2004), is an example of a basic process discovery algorithm able to automatically infer a process model as marked *Petri net* from an event log. The algorithm consists of two major phases: ***(i)*** building the footprint matrix, ***(ii)*** constructing the process model.

2.3.1. Footprint of L

The first step consists in extracting the ordering-relation between each pair of activities (transitions) in order to capture the footprint of the event log in a matrix. Table 2 shows the footprint of the event log depicted in Table 1. For any event log L, one of the following ordering-relations holds for any pair of activities:

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

- ***Direct succession*** $(a >_L b) \iff$ it exists at least one trace in the log where a is followed by b.
- ***Causality relation*** $(a \longrightarrow_L b) \iff (a >_L b \wedge b \not>_L a)$, a is followed by b and b is never followed by a.
- ***Parallel relation*** $(a \|_L b) \iff ((a >_L b) \wedge (b >_L a))$ sometimes b follows a and sometimes a follows b.
- ***No relation*** $(a \#_L b) \iff ((a \not>_L b) \wedge (b \not>_L a))$ a is never followed by b or vice versa.

Table 2
Footprint matrix of the event log presented in Table 1.

	a	b	c	d	e	f	g	h
a	$\#_L$	$\rightarrow_L$	$\rightarrow_L$	$\rightarrow_L$	$\#_L$	$\#_L$	$\#_L$	$\#_L$
b	$\leftarrow_L$	$\#_L$	$\#_L$	$\|_L$	$\rightarrow_L$	$\leftarrow_L$	$\#_L$	$\#_L$
c	$\leftarrow_L$	$\#_L$	$\#_L$	$\|_L$	$\rightarrow_L$	$\leftarrow_L$	$\#_L$	$\#_L$
d	$\leftarrow_L$	$\|_L$	$\|_L$	$\#_L$	$\rightarrow_L$	$\leftarrow_L$	$\#_L$	$\#_L$
e	$\#_L$	$\leftarrow_L$	$\leftarrow_L$	$\leftarrow_L$	$\#_L$	$\rightarrow_L$	$\rightarrow_L$	$\rightarrow_L$
f	$\#_L$	$\rightarrow_L$	$\rightarrow_L$	$\rightarrow_L$	$\leftarrow_L$	$\#_L$	$\#_L$	$\#_L$
g	$\#_L$	$\#_L$	$\#_L$	$\#_L$	$\leftarrow_L$	$\#_L$	$\#_L$	$\#_L$
h	$\#_L$	$\#_L$	$\#_L$	$\#_L$	$\leftarrow_L$	$\#_L$	$\#_L$	$\#_L$

2.3.2. Constructing the process model

The second step applies the α-algorithm over the footprint to build a marked *Petri Net*. The α-algorithm proceeds as follows:

1. Identify the set of:
   (a) All distinct activities (e.g. T_L ={a, b, c, d, e, f, g, h})
   (b) Possible initial activities (e.g. T_i ={a})
   (c) Possible final activities (e.g. T_o ={g,h})
2. Calculate possible set of X = ({A, B}) where:
   (a) All element in A (respectively B) are independent (i.e., $a_1, a_2 \in$ A $a_1 \#_L a_2$ and $b_1, b_2 \in$ B $b_1 \#_L b_2$).
   (b) For each (a,b) $\in$ (A,B), $a \longrightarrow_L b$
   (c) A and B are not empty.

(e.g. X = {({a}, {b}), ({a}, {c}), ({a}, {d}), ({b}, {e}), ({c}, {e}) …({e}, {h}), ({a}, {b,c}) ({b,c}, {e}), ({e}, {f,g,h}), ({f}, {b,c}) … ({a,f}, {b,c}), ({a,f}, {d})})

3. From step 2, select only maximum sets (e.g. Y = {({a,f}, {b,c}), ({a,f}, {d}), ({d}, {e}), ({b,c}, {e}), ({e}, {f,g,h})}.
4. Create a place for each set in Y and add an initial and final places.
5. Draw arcs between places and activities

Example 1. Taking the event log presented in Table 1 from (Van der Aalst, 2016), the α-algorithm will generate the process model described in Petri-nets [35] as shown in Fig. 1.

2.4. Alpha+ algorithm

An extension of the α-algorithm is introduced in (Medeiros, 2004; Hung et al., 2020) under the name of α+. This step beyond the alpha algorithm can catch short loops of length 1 and 2. The alpha+ algorithm proceed as follows:

1. It identifies all one-loop activities and put them into a separate list.
2. It identifies all arcs connected to one loop activities i.e., activities that happen before and after the one-loop activities. (e.g trace = aeed, arcs = {(a,e), (e,d)})
3. It removes all one-loop activities from the original event log.
4. It applies the α algorithm on the cleaned event log.
5. Finally, it reconnects the one-loop activities and their arcs to the discovered model.

2.5. Flexible Heuristic miner

Another alternative for discovering process model is the *Flexible Heuristic Miner* (*FHM*) (Weijters and Ribeiro, 2011). *FHM* was introduced to process noisy event logs by eliminating infrequent or rare execution traces which is considered as abnormal behaviour. *FHM* consists of the following steps:

2.5.1. Extracting log-ordering relations and frequency matrices

The *FHM* defines three kind of log-based ordering relations:

Definition 2. FHM algorithm log-based ordering relations. Let T be a set of activities and L be an event log over T. Let a, b $\in T$:

- ***Direct succession:*** (a $>_L$ b) $\iff$ it exists a trace $\sigma = \{t_1, t_2, t_3, \ldots, t_n\}$ in L where a = t_i and b = t_{i+1} for i $\in$ {1, 2,…, n-1} (Also used by alpha algorithm).
- ***Loops of length 2:*** (a $\gg_L$ b) $\iff$ $\exists \sigma = \{t_1, t_2, t_3, \ldots, t_n\} \in L$ where $t_i = t_{i+2}$ = a and t_{i+1} = b for i $\in$ {1, 2,…, n-2}.
- ***Indirect succession:*** (a $\gg >_L$ b) $\iff$ $\exists \sigma = \{t_1, t_2, t_3, \ldots, t_n\} \in L$ where t_i = a and t_j = b and i < j for i,j in $\in$ {1, 2,…, n-1}. Only a…b patterns without other a's or b's between them.

Table 3
Subset log-ordering relation frequencies and dependency measures for FHM based on the event log in the Example 2.

(a)	
$\lvert a >_L b \rvert = 5$	$\lvert a >_L c \rvert = 15$
$\lvert b >_L a \rvert = 5$	$\lvert b >_L c \rvert = 10$
$\lvert c >_L b \rvert = 10$	$\lvert c >_L d \rvert = 15$
$\lvert e >_L d \rvert = 5$	$\lvert a >_L e \rvert = 5$
$\lvert a \gg_L b \rvert = 5$	$\lvert c \gg_L b \rvert = 10$
$\lvert a \gg >_L b \rvert = 15$	$\lvert a \gg >_L c \rvert = 15$
$\lvert a \gg >_L d \rvert = 15$	$\lvert b \gg >_L a \rvert = 5$
$\lvert b \gg >_L c \rvert = 15$	$\lvert b \gg >_L d \rvert = 15$
$\lvert c \gg >_L d \rvert = 15$	$\lvert e \gg >_L d \rvert = 5$
(b)	
$\lvert a \Rightarrow_L b \rvert = 0$	$\lvert a \Rightarrow_L c \rvert = 0.93$
$\lvert a \Rightarrow_L e \rvert = 0.83$	$\lvert c \Rightarrow_L d \rvert = 0.93$
$\lvert e \Rightarrow_L d \rvert = = 10$	
$\lvert a \Rightarrow_L^2 b \rvert = 0.83$	$\lvert c \Rightarrow_L^2 b \rvert = 0.9$
$\lvert a \Rightarrow_L^l b \rvert = 0.24$	$\lvert a \Rightarrow_L^l c \rvert = 0.58$
$\lvert a \Rightarrow_L^l d \rvert = 0.65$	$\lvert a \Rightarrow_L^l e \rvert = -0.96$
$\lvert b \Rightarrow_L^l a \rvert = -0.24$	$\lvert a \Rightarrow_L^l c \rvert = 0.24$
$\lvert b \Rightarrow_L^l d \rvert = 0.43$	$\lvert c \Rightarrow_L^l d \rvert = 0.43$

Example 2. Let L be an event log and L = $\{\langle a, b, a, c, d\rangle^5, \langle a, c, b, c, d\rangle^{10}, \langle a, e, d\rangle^5\}$. Consider the event log L, the frequency matrices for the log-based ordering relations are shown in Table 3a.

2.5.2. Computing dependency measures

Based on the log-ordering relations, the *FHM* computes the *dependency measures* which signify the relative frequency of the activities in the log-ordering relations. Furthermore, these measures denote the sureness that there exists a true dependency relation between two activities 'a' and 'b' or there exists a loop of length 2.

Definition 3. FHM algorithm dependency measures. Let T be a set of activities and L be an event log over T. Let a, b $\in T, |a|$ is the number of occurrences of a in L, $|a >_L b|$ is the number of occurrences of $a >_L b$ in L, (same for $|a \gg_L b|$and $|a \gg >_L b|$), then:

$$a \Rightarrow_L b = \begin{cases} \dfrac{|a >_L b| - |b >_L a|}{|a >_L b| + |b >_L a| + 1}, & if\ (a \neq b) \\[2ex] \dfrac{|a >_L b|}{|a >_L b| + 1}, & if\ (a = b) \end{cases} \tag{1}$$

$$a \Rightarrow_L^2 b = \frac{|a \gg_L b| + |b \gg_L a|}{|a \gg_L b| + |b \gg_L a| + 1} \tag{2}$$

$$a \Rightarrow_L^l b = 2\left(\frac{|a \gg >_L b| - abs(|a| - |b|)}{|a| + |b| + 1}\right) \tag{3}$$

These frequency-based dependency measures are used by the FHM algorithm to define a dependency graph which can be converted to a *Petri net*. The algorithm will take noise into account at this stage by requiring a certain frequency threshold to assure the validity of the dependence. Generally, these thresholds are set to 0.9 and dependencies lower than these thresholds are eliminated.

Example 3. Consider the event log in Example 2 and the frequencies in Table 3a, the resulted dependency measures are presented in Table 3b.

2.5.3. Discovering process model

At this stage the algorithm identifies the direct inputs and outputs of each activity to build the footprint matrix. Then, these footprint matrix containing direct relation dependencies is converted into a dependency graph. After that, the algorithm extends the graph by adding loops, 'and' and 'or' operators. Finally, indirect dependencies are inserted, and the resulted graph represents the process model discovered by the FHM algorithm.

2.6. Spark overview

Apache Spark (Zaharia et al., 2010) is an open source in-memory distributed computing framework. It uses Hadoop (Hadoop, 2020) infrastructure component such as Hadoop Distributed File System (HDFS) and Yet Another Resource Negotiator (Yarn) to manage large scale clusters. Spark has a sub-second level latency for small datasets. Specifically, Spark is more suitable than MapReduce (Dean and Ghemawat, 2008) for large datasets application. Due its various features Spark has been adopted by many research areas, such as machine learning (JayaLakshmi and Krishna Kishore, 2018) and pattern mining (Kumar and Mohbey, 2019; Sundari and Subaji, 2020), as a processing engine for solving Big Data problems.

Spark revolves around RDD which is Resilient Distributed Dataset (Zaharia et al., 2012). RDD is a fundamental data structure of Spark. It is elastic, fault tolerant, read only and distributed collection of records. Each RDD is divided into logical partitions which may be stored and computed on a different node of the cluster. These features improve performance by their locality. Spark supports two types of operations on RDD: *transformations* and *actions*. Transformation operations such as *map* and *flatMap* create new RDD from existing one. An action operation such as *reduce* runs a computation (e.g., aggregation) on the RDD and returns the results to the driver program. Similar to MapReduce, Spark computation is based on functional programming (An example of Spark running wordcount is in Appendix A.1).

3. Related works

Process discovery on a single machine have been widely studied in literature. Algorithms such as α miner (van der Aalst et al., 2004), heuristic (Weijters et al., 2006) and flexible heuristic miner (Weijters and Ribeiro, 2011), ILP miner (van derWerf et al., 2009) and inductive miner (Leemans et al., 2013) are examples of algorithms able to learn and discover process model from historical event logs. However, these techniques became unable to keep up with evolution of information systems where event logs become larger, scattered over multiple sub systems and have a variety of formats.

With the emergence of distributed data processing frameworks (e.g. MapReduce (Dean and Ghemawat, 2008) and Spark (Zaharia et al., 2010)), many researches interested in combining big data research area and process analysis. The first study that incorporates MapReduce with process mining was in (Reguieg et al., 2012). Authors proposed a technique for *event correlation discovery* in systems were event log are scattered and the relation between events is missed. They used MapReduce to identify set of rules that specify how event are grouped together to form a trace. This technique has been extended in (Reguieg et al., 2015; Cheng et al., 2017) to discover composite rules. This approach can be considered as a preprocessing step for process discovery, in situations where events are not associated with a caseID. In context of process discovery, (Evermann, 2016; Hernandez et al., 2015) have been used MapReduce to implement alpha algorithm and flexible heuristic miner. On one hand, Process discovery techniques are iterative computations and implementing alpha miner or flexible heuristic miner on MapReduce requires several MapReduce jobs. On the other hand MapReduce store, read and sort intermediate data on disk which involves more I/O on disk (Kalavri and Vlassov, 2013). Therefore, This can affect the global performance of any computations.

Authors in (Cheng et al., 2020), have been investigated Spark to introduce a technique denoted by *filter-based Hybrid Model discovery*. Their approach aims to discover hybrid process model i.e., a combination of formal and informal models.

Parallel process discovery has been also studied using High Performance Computing (HPC). In (Sahu et al., 2018), authors implemented the parallel version of alpha miner using Message Passing Interface MPI. Their approach is based on task parallelism available on the alpha algorithm. Thead-level parallelism and shared memory among cluster nodes are two major benefits when working with MPI. However, MPI has a poor reliability since it is not fault tolerant and has no common runtime for big data processing environment (Jha et al., 2014). Unlike (MPI) jobs that use thread-level parallelism, Spark jobs used to follow data-flow model (Chowdhury et al., 2014).

Big data platforms, such as Hadoop MapReduce and Apache Spark have been adopted as one stop solution for most of big data problems. However, jobs running under these frameworks require a careful parameter tunings to improve processing performances. Moreover, bad parameter tunnings may lead to poor performances or even job failures. In (Cai et al., 2019), authors introduced an adaptive framework, named mrMoulder, for automatic tuning parameters of new jobs. mrMoulder exploits a dynamically extended configuration repository to recommand near-optimal configuration for big data jobs in a short time.

The processing of a huge amount of data in a reasonable time with the use of huge computing resources will increase the energy consumption and consequently this will lead to the increase of greenhouse gas emissions and environmental impacts. Authors in (Wu et al., 2016b,a) deal with this problem and analyze the relevance among green measures and big data. In the same context, authors in (Wu et al., 2018) presented a state-of-the-art on solutions aiming to find correlations among sustainable development goals and information and communications technologies.

4. Contribution

In this section, we present the conceptual architecture of our approach and describe the implementation of the proposed algorithms based on Apache Spark framework (Zaharia et al., 2010).

4.1. Conceptual architecture

Fig. 2 illustrates an abstraction of the conceptual architecture for the proposed approach. It is organized in the following layers:

4.1.1. Data source layer

The first layer outlines event data source systems such as Business Process Management Systems (BPMS), production systems and web applications. Such systems generate continuously a wide range of process information items (events).

4.1.2. Data storage layer

Events are then integrated into data storages for further analysis. Since, event data are semi-structured, Hadoop distributed file system (HDFS) (Shvachko et al., 2010) is suitable to store such data.

4.1.3. Processing layer

Indeed, this layer represents the architecture's core. It includes parallel process discovery algorithms running over Apache Spark processing engine. The processing engine takes event logs stored in HDFS as inputs then it applies one of the proposed algorithms

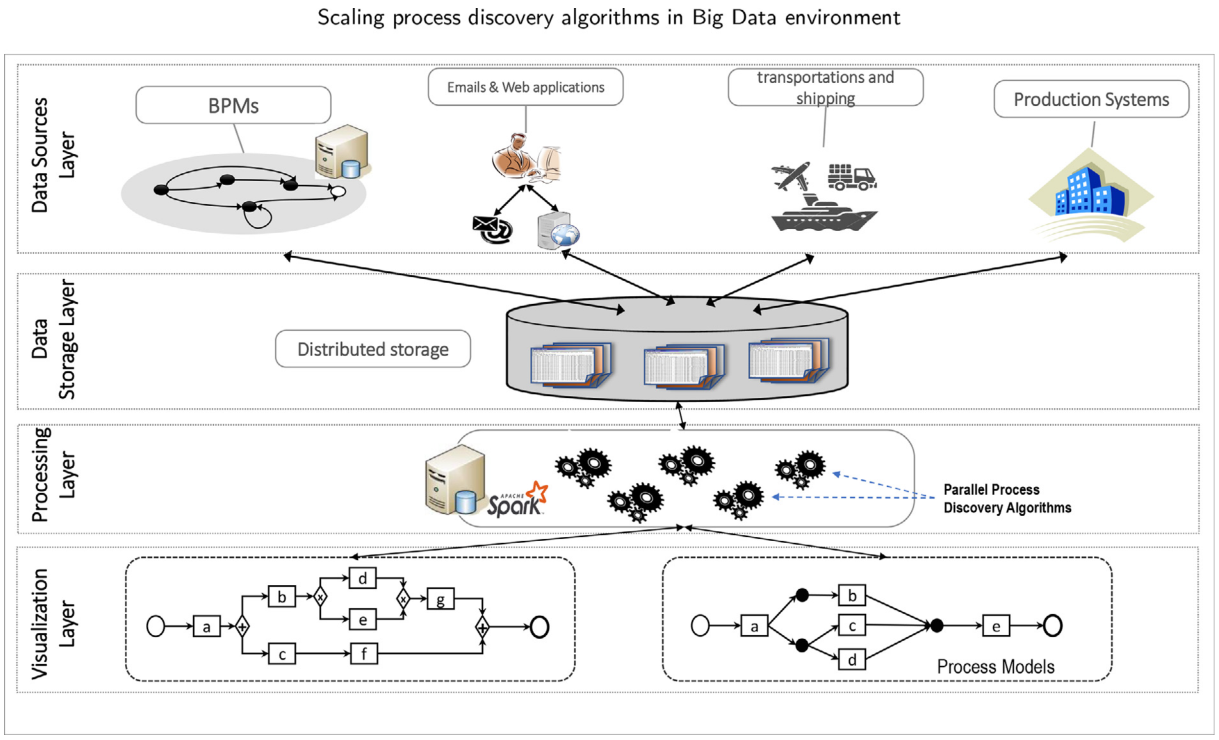

Scaling process discovery algorithms in Big Data environment
Data Sources Layer
BPMs
Emails & Web applications
transportations and shipping
Production Systems
Data Storage Layer
Distributed storage
Processing Layer
Spark
Parallel Process Discovery Algorithms
Visualization Layer
Process Models

Fig. 2. The Conceptual architecture of the proposed approach.

to scale automatic process discovery. Parallel Algorithms in this layer are the main contribution of this paper.

4.1.4. Visualization layer

This layer provides a graphical representation for the resulted models.

4.2. Parallel process discovery algorithms

The main goal of our work is to propose a scalable process discovery algorithms in order to process Big event logs. Note that the computationally-intensive task in process discovery is transforming the event log into causality matrices. This step requires the exploration of a large space of log-order relations among events.

Algorithm 1: Spark-based Alpha algorithm (SAlpha)

```
   Input: Event Log as traces
   Output: Footprint Matrix
1  io_tasks = traces.map(selectFirstAndLast)).cache()
2  inputs = io_tasks.map(selectFirst).distinct() .collect()
3  outputs = io_tasks.map(selectLast).distinct()
   .collect()
4  relations = traces.flatMap(extract_successions)
   .reduceByKey(lambda AB, r : set(AB) .union(set(r)))
5  orderingRelations =
   relations.map(compute_relations)
6  foreach pair, relation in orderingRelations do
7      a, b = pair
8      footprint[a, b] = relation
9  end
```

4.2.1. SAlpha-algorithm

The first parallel process discovery algorithm is the *Spark Based Alapha* algorithm *(SAlpha)* described in Algorithm4. It relies on the alpha miner (van der Aalst et al., 2004). The *SAlpha*, as input, requires an event log and produces a footprint matrix which is, then, transformed into a *Petri* process model.

First, the event log is loaded into HDFS and read as RDD. Here, the RDD is partitioned over nodes where each partition contains a set of traces (see Fig. 3(a) and (b)). Then, the *SAlpha* proceeds as follows:

- **Extracting initial and final activities** (line 1–3 in Algorithm1): on each RDD partition, we apply user defined function *selectFirstAndLast* over a *map* transformation to select first and last activities in each trace (line 1, map1 in Fig. 3(a)). The results are cached to be used in the next step. Then, we use selectFirst (selectLast respect.) to extract all initial activities (final activities respect.) (line 2 and 3, map2 and map3 in Fig. 3(a)). Next, a *distinct* operation is applied to eliminate redundant activities. After that, initial and final activities are returned to the driver program using *collect* action.
- **Extracting causality relations over activity pairs** (line 4–5, flatMap1 in Fig. 3(b)): at this step we compute the log based ordering relations over activity pairs. First, we extract direct succession relation ($a >_L b$). Thus, we parse each trace into (key, value) pairs in the form of (AB, $>$) and (BA, $<$), where A and B are activities and $>$ represents the direct follow relation. This phase is done in parallel at each split of the RDD by using user defined function *extract_succesion* over a *flatMap* transformation. Then, all values having the same key are aggregated together into a single pair (Key, list of values) by using *reduceByKey* action. Here, the merged pairs on each partition are first merged locally, and then shuffled over cluster nodes based on their keys, and then merged again locally (see Fig. 3(b)). For

example, pairs (AB,>), (AB,>) and (AB,<) will be reduced into (AB, {<, >}). After that, in (line 5) we use *compute_relation* function over a *map* transformation to compute the global relations ($a \longrightarrow_L b$ and $a||_L b$) of all pairs of activities. For example, (AB, {<, >}) and (CD, {>}) will be transformed into (AB, '||') and (CD, '⟶').

- **Generating footprint matrix** (line 6–8): Finally, all the aggregated pairs will be collected by the program driver to construct the footprint (causality) matrix. Empty cells in the matrix are considered as there is no follow relation between pair of activities (a $\#_L$ b). This matrix reflects an abstraction of the whole event log, and it will be transformed into a *Petri net*.

4.2.2. SAlphaplus-algorithm

The second parallel process discovery algorithm is the *Spark Based Alapha plus* algorithm denoted by *(SAlpha+)*. The implementation of the *SAlpha+* is very similar to the *SAlpha* algorithm. The main difference is that the *SAlpha+* has an extra preprocessing step which discovers short loops of length 1, and it takes into account patterns as *'ABA'* and *'BAB'* to consider loops of length 2.

Algorithm 2: Spark-based Alpha plus algorithm (SAlpha+)

```
   Input: Event Log as traces
   Output: Footprint Matrix
1  result = traces.flatMap(getLoopTaskWithArcs).filter(lambda event :event).cache()
2  OneLoopTasks = result.map.(lambda c:c[0]).distinct().collect()
3  arcs = result.map(lambda arc : arc[1]).distinct().collect() foreach task in OneLoopTasks do
4      left = map(lampda arc: arc[0], filter(lambda arc: arc[1] == task, arcs) )
5      right = map(lampda arc: arc[1], filter(lambda arc: arc[0] == task, arcs) )
6      places.append(('P(left,right)', task))
7      places.append((task, 'P(left,right)'))
8  end
9  cleanTraces = traces.map(lambda trace: filter(removeTasksWithOneLoopLength))
10 /** We apply the SAlpha algorithm on the cleaned event log (cleanTraces).**/
```

The detail of the implementation of the SAlpha+ Algorithm is depicted in Algorithm 2. We first extract all activities 'A' having pattern in the form of 'xAAy' and activities which follow and precede them. For example, a trace *'abeed'* will be returned as (e, (b, e), (e,d)) where 'e' is the one-loop activity, (b,e) is an arc from 'b' to 'e' and (e,d) is an arc from 'e' to 'd'. This task is done in parallel using *getLoopTasksWithArcs* function over a *flatMap* transformation (line 1, flatMap2 in Fig. 4). Next, one-loop activities and their successor/predecessor arcs are stored in different RDDs (line 2–3, map6 and map7 in Fig. 4). Then, we need to create a '*place*' for each one-loop activity to connect them, at the end of the algorithm, to the produced *Petri net* model (line 3–7). After that, we remove all one-loop activities from the event log. This is, also, done in parallel by filtering traces in each partition and generate new RDD free from one-loop activities (line 9). Finally, we can apply the *SAlpha* algorithm on the cleaned log to produce the footprint matrix (line 10).

4.2.3. SFHM-algorithm

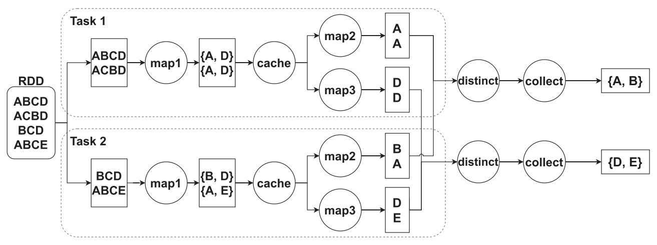
This subsection discusses the implementation of the *Spark-Based Flexible Heuristic Miner* denoted as *SFHM* (depicted in Fig. 5). *SFHM* relies on the flexible heuristic miner (Weijters and Ribeiro, 2011). Here again, we concentrate on the log based ordering relations to build dependency measure matrices.

Algorithm 3: Spark-based Flexible Heuristic Miner algorithm (SFHM)

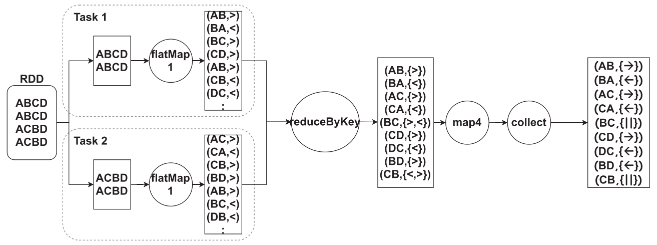
```
   Input: Event Log as traces
   Output: Dependency matrix
1  occurrences = traces.flatMap(lambda trace : map(lambda a: (a, 1))) .reduceByKey(sum) .collect()
2  directPairs = traces.flatMap(lambda trace : map(lambda a, b: (a,b),1)).reduceByKey(sum)
3  foreach pair in directPairs do
4      a, b = pair
5      directDepMat[a, b] = $\frac{pair(a,b) - pair(b,a)}{(pair(a,b) + pair(b,a) + 1}$
6  end
7  twoLLPairs = traces.flatMap(getPairsOfLengthTwo). reduceByKey(sum)
8  foreach pair in twoLLPairs do
9      a, b = pair
10     twoLLDepMat[a, b] = $\frac{pair(a,b) - pair(b,a)}{pair(a,b) + pair(b,a) + 1}$
11 end
12 indirectPairs = traces.flatMap(getIndirectPairs). reduceByKey(). map.(lambda pair : (pair[0], sum(pair[1])))
13 foreach pair in indirectPairs do
14     a, b = pair
15     longDMat[a, b] = $2(\frac{pair(a,b) - abs(|a| - |b|)}{|a| + |b| + 1})$
16 end
```

Algorithm 3 describes the implementation of the *SFHM* algorithm. It proceed as follows:

- **Compute the frequency of activities** (line 1, flatMap1 in Fig. 5): First, we compute the number of occurrences of each activity in the event log as follows: (1) using a *map* transformation, iterate over all activities in the partition and assign value 1 to each activity to generate pairs in the form of (A, 1), then (2) perform a reduceByKey function to sum together all values for each activity. For example, L = {abc, bc} will give us (a,1)(b,2)(c,2). Since, the number of unique activity is very small, the results are collected into the program driver.
- **Compute direct succession relation** a $>_L$ b **frequency** (line 2, flatMap2 in Fig. 5): similar to the last step, we iterate over each trace and produce pairs in the form of (AB, 1) by using *map* transformation. Then, we sum the number of all values by using a reduceByKey action. For example, L = {abc, bc} will give us (ab,1)(bc,2).
- **Compute dependency measure** ($a \Rightarrow_L b$) (line 3–5): we apply the formula 1 in Section 2.5 to compute dependency measure ($a \Rightarrow_L b$). All candidates lower the user defined threshold are eliminated.



(a) Extracting Initial and Final activities.



(b) Extracting log based ordering relations

Fig. 3. Data-flow of SAlpha algorithm.

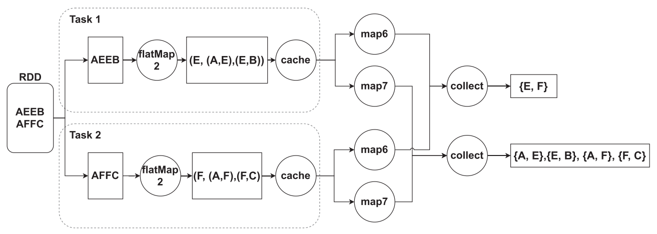


Fig. 4. SAlpha+: Extracting one length loop activities.

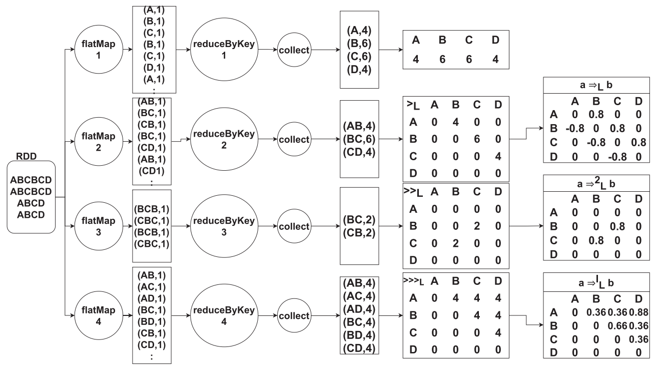


Fig. 5. Data flow of SFHM algorithm.

- **Compute the** $(a \gg_L b)$ **relation frequency and the dependency measure** $(a \Rightarrow^2_L b)$ (line 7–10, flatMap3 in Fig. 5): using a *map* transformation, we iterate over all traces looking for patterns like 'ABA' and generate pairs in the form of (AB, 1). Next, we sum all the values in a *reduceByKey* action. For example, L = {abacd, abacd, aede} will result (ab,2)(ed,1). After that, in the program driver, we apply formula 2 to compute dependency measure $(a \Rightarrow^2_L b)$ and eliminate candidates that do not satisfy the defined threshold.
- **Compute the** $(a \gg >_L b)$ **relation frequency and the dependency measure** $(a \Rightarrow^l_L b)$ (line 12–15, flatMap4 in Fig. 5): the final step, we iterate over traces and generate all possible combination of activities as pairs (AB, 1) (e.g. for a trace σ = 'abcd' we generate pairs (ab,1) (ac,1) (ad,1) (bc,1) (bd,1) (cd,1)). Then, by using the *reduceByKey* action, we compute the frequency of each pair of activities. For example, L = {abcd, abef} will give us (ab, 2) (ac, 1) (ad, 1) ...(ef,1). All the statistics are then collected to the node master. After that, it applies formula 3 to calculate $(a \Rightarrow^l_L b)$ dependency measures. Candidates that do not satisfy the threshold are pruned.

5. Evaluation

In this section, performance and scalability of the proposed algorithms are evaluated through an experimental study.

5.1. Experimental framework

We have implemented the *SAlpha*, *SAlpha +* and *SFHM* algorithms using *Python* over Spark (Zaharia et al., 2010).

Environment. We conducted our performance tests on a cluster of 13 physical nodes, one master and 12 slaves (workers). Nodes are connected using 1 Gb/s Ethernet CISCO 2960-X switch. The master node has an intel i7-3370 CPU running at 3.4 GHz with 8 cores, 16 Gb ram and 100 Gb of disk. Each slave node has an intel I5-4460 CPU running at 2.8 Ghz with four cores, 8 Gb of ram and 80 Gb of free space disk. The operating system is Linux Ubuntu server 18.04 LTS and the software stack consists of Spark version 3.0.0, Hadoop 3.2.1, Scala version 2.12, Python version 3.7, Java JDK version 1.8.0_251 and Graphviz version 2.40.1 used for model visualization. Note that non-virtualised salve nodes used in our environment are equivalent and nearly approximates cluster instance a1.xlarge in Amazon EC2 instances. Therefore, our approach can be easily deployed on a real cloud computing (Armbrust et al., 2009) environment.

Datasets. we performed our experiment tests on *compensation requests*[1] event log taken from (Van der Aalst, 2016). It contains 8 distinct activities, 6 traces and the maximum number of events per trace is 13. In order to evaluate the scalability and speed performance of the algorithms, this event log is synthetically replicated in order to evaluate speed-up and scale-up of the algorithms. Table 4 shows the size, number of traces and number of events in the replicated files.

Setup. To enable a better exploit of the cluster resources, we configured Spark as follows: *(i)* the slave worker uses 6.7 Gb and 4 CPU cores, *(ii)* the master does not perform any parallel computation nor storing data, it runs only program driver and Hadoop/ Spark master daemons (Master, NodeManager, ResourceManger and NameNode).

Throughout all the tests, event logs are stored and read from the HDFS (Shvachko et al., 2010) and the outputs (discovered models) are written to the master disk as *dot* and *png* file format. Running time is measured from the job submission to the job completion. Finally, we carried out our experiments on a cluster of one master and 12 worker nodes.

5.2. Experimental results

In this section, we report our experimental results. First, we present the quality measures for the discovered models. Then, we discuss the running time and scalability performance of the algorithms.

5.3. Quality of the discovered model

Fig. 6 illustrates the process model obtained from executing our algorithms on the *compensation requests* event log. To measure the

[1] https://doi.org/10.17632/xct7k6w98k.1

Table 4
Summary of size, number of traces and events in log used in the experiments.

Size (in Gb)	#Cases	#events
2	5,382,630	37,678,164
4	10,671,792	74,702,544
8	21,228,858	148,602,006
16	42,342,996	296,400,972
32	84,155,646	589,089,522
64	167,183,604	1,170,285,228
128	330,768,352	2,094,866,224

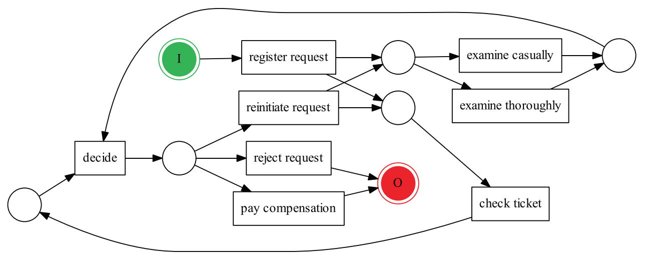


Fig. 6. Process model discovered from *request compensation* dataset using the proposed algorithms.

Table 5
Conformance checking for the proposed algorithms.

Approach	*fitness*	*precision*	*generalisation*
Original	1	0.83730	0.83214
Proposed	1	0.83730	0.83214

quality of the discovered models, we used three quality metrics, ***(i)*** *fitness*: the model is able to reproduce the event log, ***(ii)*** *precision*: the discovered model should be precise and allows only for minimal new behaviour than seen in the log, ***(iii)*** *generalisation*: the ability of the discovered model to produce future behaviour of the process (for more details about conformance checking metrics reader may refer to (Adriansyah et al., 2015)).

From Table 5, we notice that our algorithms can discover models that have the same model quality as the original algorithms. Note that the main gaol in this paper is to scale process discovery algorithms in big data environment without affecting the quality metrics, i.e, *fitness*, *precision* and *generalisation* remain stable.

5.4. Efficiency

Runtime. In order to evaluate the runtime evolution performance of the proposed algorithms, We performed our experiments by using various size of input event log.

As the first test we have executed the 3 algorithms on the *compensation requests* dataset using the full capacity of our cluster (i.e., 48 cores, $\tilde{8}0$ Gb of RAM and 1 Gb/s Ethernet). As input log size, we begin with 2 Gb, equivalent to a log having *5M* traces and more than *37M* events, then we doubled the log size at each step until 128 Gb (see Table 4). The x-axis represents the input data sizes and the y-axis represents the running time in minutes. Furthermore, we should recall that the discovered process models of the proposed algorithms were the same as those discovered by the original algorithms *Alpha* (van der Aalst et al., 2004) and *FHM* (Weijters and Ribeiro, 2011).

Fig. 7 compares the results obtained from executing the *SAlpha*, *SAlpha+* and *SFHM* algorithms. First, we observe that the required

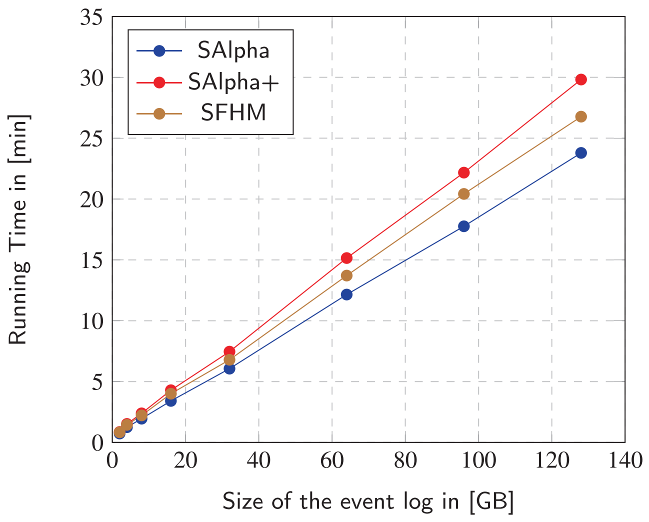


Fig. 7. Running time of the SAlpha, SAlpha+ and SFHM algorithms on different input event log size.

Table 6
Average gradient and estimated number of shuffles of each algorithm.

	SAlpha	SAlpha+	SFHM
Avg.gradient	0.19	0.24	0.21
Shuffles	7	9	8

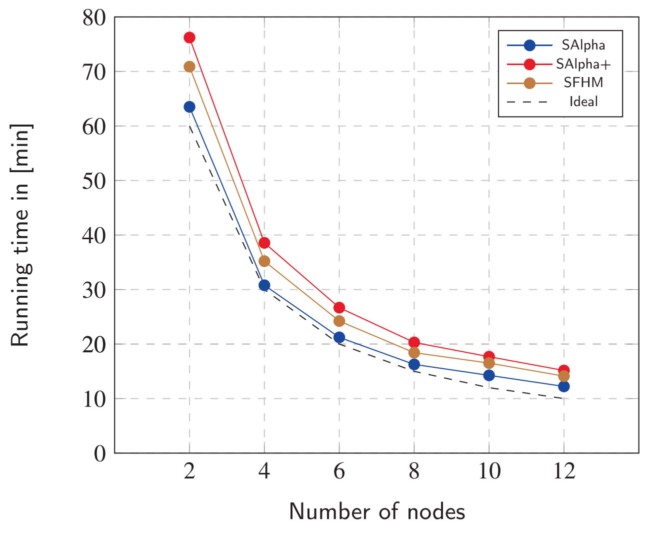
time for processing the largest dataset (128 Gb) is in a range of 30 min ($TSAlpha^{128}$ = 23.79 min, $T^{128}_{SAlpha+}$ = 29.82 min T^{128}_{SFHM} = 26.77 min). Secondly, we identify a linear evolution of the running time of the 3 algorithms with respect to the event log size used. In addition, although the data size is multiplied by a factor of 2 at each step, the average[2] increase in execution time was by a factor of 1.79, 1.81 and 1.79 for the *SAlpha*, *SAlpha+* and *SFHM* algorithms respectively. Moreover, the average gradient of the running time was *0.19*, *0.24* and *0.21* for the *SAlpha*, *SAlpha+* and *SFHM* algorithms respectively. This means that in one minute, *SAlpha*, *SAlpha+* and *SFHM* algorithms process approximately 5, 4 and 5 Giga Bytes of data.

Another key observation in Fig. 7 is the slight difference between the execution time of the algorithms. We observe that *SAlpha* was faster than *SAlpha+* and *SFHM*. This is mainly due to:

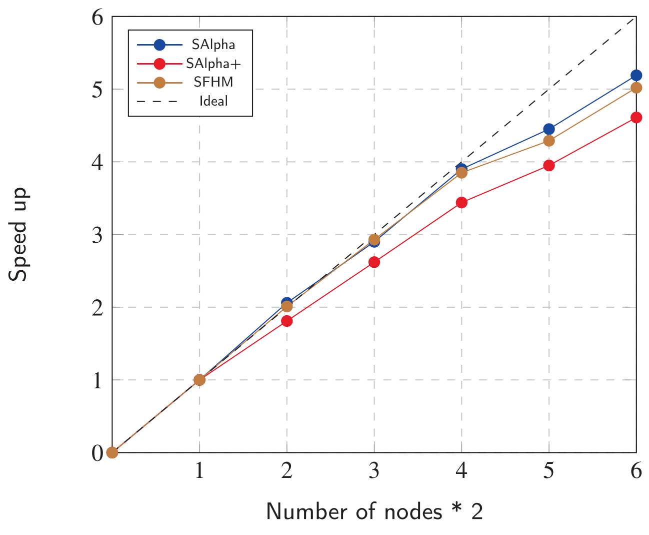
1. The *SAlpha+* performs an extra step over *SAlpha* algorithm. It extracts one-length-loop activities and then scans the entire event log to remove loop patterns. This task affects deeply the performance of the algorithm.
2. The *SFHM* algorithm computes three dependency matrices rather than one footprint matrix in the *SAlpha* algorithm.

A second factor which impacts the performance of the algorithms is the number of data *shuffles*. A shuffle is an expensive operation because it involves grouping, partitioning and transferring data among nodes over network. It is triggered by RDD actions such as collect(), distinct(), reduceByKey() and groupByKey(). Table 6 shows the estimated number of shuffles made in each algorithm. All that being said, *SAlpha+* and *SFHM* algorithms are able to discover complex model than the *SAlpha* algorithm.

[2] The average increase could be computed as a = $avg(\sum_{i=2}^{6} \frac{t_i - t_{i-1}}{t_{i-1}})$

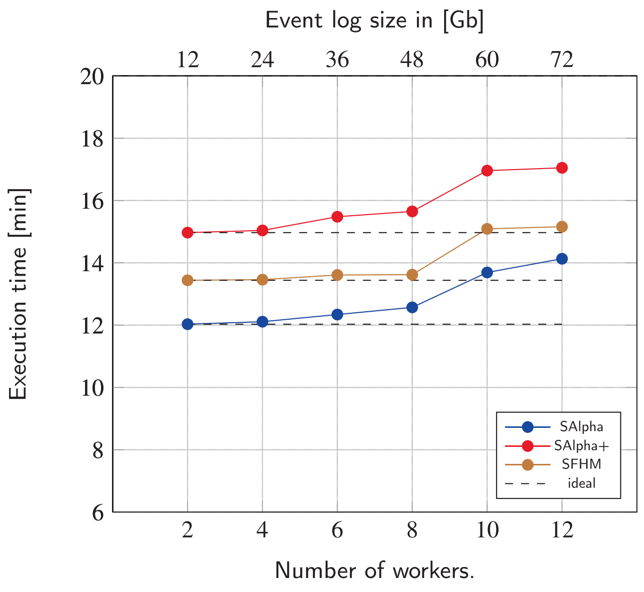


(a) Varying cluster configuration

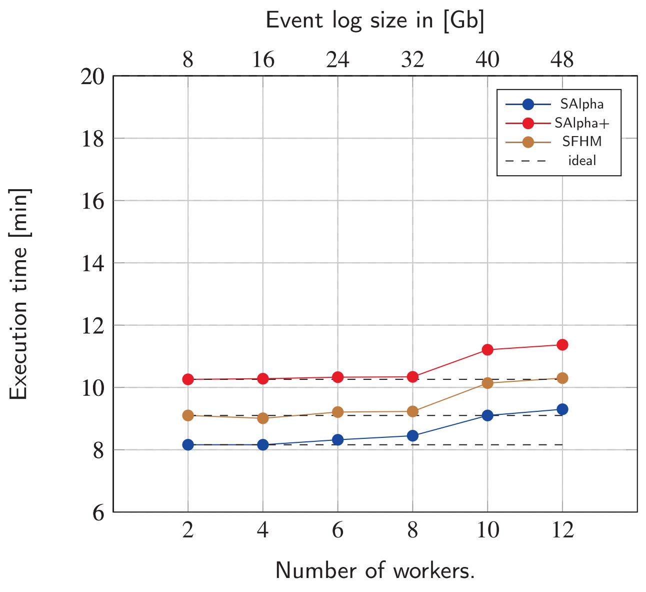


(b) Algorithms speed up.

Fig. 8. Running time and relative running time of the algorithms on 64 Gb of event log and on different cluster configurations (speed-up).



(a)



(b)

Fig. 9. Running time on different cluster and event log configurations (scale-up).

5.5. Scalability

We carried out two different test to validate the speed-up and scale-up of our algorithms. In the first test we fixed the size of the input event log and we varied the number of nodes in the cluster. In the second test we varied both size of the input event log and the number of nodes with the same factor.

5.5.1. Speed up

In order to measure the speed up of the approach. We applied the algorithms on a fixed size of the input event log to *64* Gb (about 1.170 billion of events) and we increased the number of workers by adding 2 nodes at each step. The ideal time can be obtained by $t_i = \frac{time_1}{i}$ where i $\in \{1,2,3,4,5,6\}$.

In Fig. 8(a), we plot the running time of the algorithms on a fixed size of input log as we increased the cluster size from 2 to 12 workers. Mainly, it can be seen that as the number of workers increased, the running time of all the algorithms remarkably decreased following approximately the same pattern as the ideal time. We can observe that running time decrease faster until configuration with 8 nodes. Then, it decreases slowly. We suppose the reason is that the amount of the memory in the cluster with 8 nodes get closer to the size of the input log file and beyond configuration 10 the cluster becomes able to handle all the workload in-memory.[3] This is proved by the insignificant running time improvement between configurations with 10 and 12 nodes. This note appears well in the Fig. 8(b).

$$relativeTime_i = \frac{time_i}{time_1}, \ i \in \{1,2,3,4,5,6\} \tag{4}$$

[3] A key feature of Spark is the in-memory computation.

Table 7
Comparing the running time of the proposed spark based algorithms with the original centralized algorithms.

Approach	# Cases	Alpha	Alpha+	FHM
Original	2.65M	177s	185s	235s
Proposed	5.3M	44s	52s	50s
Improvement	**x0.5**	**x4**	**x3.5**	**x4.7**

Fig. 8(b) shows the same results as in Fig. 8(a), but reported on a ”relative scale” i.e., for each cluster configuration, we compute the ratio between running time of the smallest configuration and the current one (as shown in formula 4). We can see that all the algorithms speed-up significantly until configuration 4 (8 nodes) then it slightly deviates. A relevant information captured in Fig. 8 is the overall speed up. So that when we increased the size of the cluster 6 times we got a time acceleration of *5.2*, *4.61* and *5.02* for the *SAlpha SAlpha+* and *SFHM* respectively.

As information derived from this experiments, We believe that doubling resources may decrease the processing time. However, it may affect the overall speed up of the algorithms. Furthermore, in a paid environment such as public cloud computing, supplying more resources may involve additional fees (Kllapi et al., 2011).

5.5.2. Scale up

To test the algorithms scale-up,[4] we increased the cluster size and the input log size by the same factor. An algorithm performs a perfect scale-up if the running time in all configurations remained constant.

Fig. 9(a) reports the running time for event log increased from 12 Gb to 72 Gb, on a cluster with 2 to 12 nodes, respectively (12 Gb for each 2 workers). First, we observe that the overall running time of the 3 algorithms has changed slightly. The difference between the first and the last configuration was very negligible (about 2 min for all algorithms). A main reason for this slight loss of performance is the network bandwidth, since it remained constant in all configuration (1 Gb/s). We believe that this loss of performance could be reduced if we are able to increase the bandwidth performance in each configuration.

A second key observation in Fig. 9(a) is between configuration 8 and 10. We note an palpable increase in running time for all algorithms. Since resources (RAM and CPU) are increased proportionally with the size of the input log we can have two factors that may cause this rise in running time. Either, the amount of data shuffled has exceeded the available network bandwidth (size between 48 Gb and 60 Gb) and the network slows down, or the queue in *ShuffleBlockFetchIterator* (Kay, 2020) is overloaded by many threads fetching data over network which involves an extra time to satisfy all fetching threads.

In order to approve our assumptions, we repeated the test but this time we decreased the size of the input log to 8 Gb for each 2 workers.

Looking at Fig. 9(b), it is apparent that the running time of all algorithms behave as that in Fig. 9(b). We observe that it always occurs between configuration 8 and 10. Thus, we have eliminated the hypothesis that says the amount of the shuffled data may cause this phenomenon.

5.6. Proposed approach vs. original centralized algorithms

In order to highlight the effectiveness of our proposed algorithms, we compare our results reported in this paper with the original algorithm results obtained from *Prom* execution (van Dongen et al., 2005).

Table 7 shows a comparison between the running time (in seconds) of the proposed algorithms, in this paper, and the original centralized algorithms using the second dataset (*pay compensation* event log). We, clearly, observe the benefits of using Spark framework. Although the data size is twice larger, our approach outperforms the original one and is 4 times faster. Moreover, centralized algorithms suffer from scalability problems since they are not designed to handle very large event logs. Furthermore, centralized approach does not support horizontal scalability, i.e., it is limited to the capacity of one machine.

5.7. Summary

In summary, we have performed experiments to evaluate, most important factors that covers issues for evaluating distributed approaches, both scalability and speed up of our algorithms. The experiments have shown that the proposed algorithms scale well either by varying input event log sizes, cluster resource configurations, or both of them. In addition, we presented the efficiency of using distributed approaches against centralized single node methods for process model discovery.

6. Conclusion

Process discovery is a key task for process mining. It consists of analysing process historical data to infer process behaviour. In this paper, we addressed the problem of process discovery for big event logs using the Apache Spark framework. Spark offers an in-memory scalable model for distributed computation among cluster of nodes. The choice of Apache Spark as a computing engine is quite natural for two reasons: (1) modern information system become increasingly federated and continually growing in size and complexity which implies recording a large variety of historical data, and (2) the iterative nature of computation in process discovery algorithms. We presented a parallel implementation of the well known process discovery α-miner (van der Aalst et al., 2004) and the *Flexible Heuristic Miner* (Weijters and Ribeiro, 2011) in a distributed environment (i.e., over Spark). Our performance tests have demonstrated that the proposed algorithms are able to process big event log efficiently. In addition, the scalability experimentation have shown that the algorithms scale very well when increasing resources and maintain their performance when increasing event log size and resources in the cluster with the same workload.

As a future works, we plan to extend our approach to support real time stream of events by using Spark-streaming with Apache Kafka (Kreps et al., 2011). First tests were promising and our algorithm were able to update the models instantly (each 1 s). Moreover, Apache flume can be integrated as a pre-processing framework in order to efficiently aggregate event logs to the former architecture (Kafka + Spark-streaming). Finally, we plan to incorporate our architecture with an end-to-end business mining tool such as Prom.

[4] **Scale-up** is the ability to keep the same performance levels (response time) when both workload (event log size) and resources (CPU, memory and network bandwidth) increase proportionally.

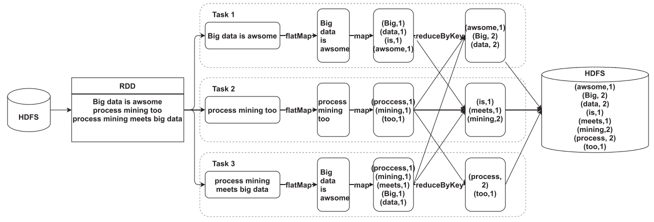

Task 1
Big data is awsome
flatMap
Big data is awsome
map
(Big,1) (data,1) (is,1) (awsome,1)
reduceByKey
(awsome,1) (Big, 2) (data, 2)
HDFS
RDD
Big data is awsome
process mining too
process mining meets big data
Task 2
process mining too
flatMap
process mining too
map
(proccess,1) (mining,1) (too,1)
(is,1) (meets,1) (mining,2)
HDFS
(awsome,1) (Big, 2) (data, 2) (is,1) (meets,1) (mining,2) (process, 2) (too,1)
Task 3
process mining meets big data
flatMap
Big data is awsome
map
(proccess,1) (mining,1) (meets,1) (Big,1) (data,1)
reduceByKey
(process, 2) (too,1)

Fig. 10. WordCount example data flow on Spark framework.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

We would like to thank all those who contribute directly or indirectly to this work.

Appendix A

A.1. WordCount on Apache Spark

Algorithm 4. Spark WordCount code.

```
1: text  = sc.textFile("hdfs://.../inputFile.txt")
2: counts  = text.flatMap(lambda line: line.split(" "))
      .map(lambda word: (word, 1))
      .reduceByKey(sum)
3: counts.saveAsTextFile("hdfs://.../ouputFile.txt")
4: sum = lambda a, b: a + b
```

Example 4. Fig. 10 illustrates the data flow for the WordCount example described in Algorithm 4. The example counts words in the file (*inputFile.txt*). The file is stored in HDFS and read as RDD. The RDD is partitioned across three tasks. Each task parse a line into words using *flatMap* transformation. Then, each word is assigned with a value 1 and mapped as pair in the form of (word, 1). After that, a *reduceByKey* action is applied to aggregate values within the same key. A user defined function sum is applied on the aggregated values to return the final aggregated count. Finally, the results are stored into HDFS.

References

Van der Aalst, W., 2016. Process Mining: Data Science in Action. Springer Berlin Heidelberg, New York, NY.

van der Aalst, W., Weijters, T., Maruster, L., 2004. Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16, 1128–1142. https://doi.org/10.1109/TKDE.2004.47. URL: http://ieeexplore.ieee.org/document/1316839/.

van der Aalst, W.M.P., 2013. A general divide and conquer approach for process mining. In: 2013 Federated Conference on Computer Science and Information Systems, pp. 1–10.

Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P., 2015. Measuring precision of modeled behavior 13, 37–67. URL: https://doi.org/10.1007/s10257-014-0234-7, doi: 10.1007/s10257-014-0234-7.

Armbrust, Michael, Fox, A., Griffith, Rean, Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, Ariel, Stoica, Ion, Zaharia, Matei, 2009. Above the Clouds: A Berkeley View of Cloud Computing.

Cai, L., Qi, Y., Wei, W., Wu, J., Li, J., 2019. mrMoulder: A recommendation-based adaptive parameter tuning approach for big data processing platform 93, 570–582. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17318526, doi: 10.1016/j.future.2018.05.080.

Cheng, L., van Dongen, B.F., van der Aalst, W.M., 2020. Scalable discovery of hybrid process models in a cloud computing environment. IEEE Transactions on Services Computing 13, 368–380. https://doi.org/10.1109/TSC.2019.2906203. URL: https://ieeexplore.ieee.org/document/8669858/.

Cheng, L., Van Dongen, B.F., Van Der Aalst, W.M.P., 2017. Efficient event correlation over distributed systems. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, Madrid. pp. 1–10. URL: https://ieeexplore.ieee.org/document/7973683/, doi: 10.1109/CCGRID.2017.94.

Chowdhury, M., Zhong, Y., Stoica, I., 2014. Efficient coflow scheduling with Varys. In: Proceedings of the 2014 ACM Conference on SIGCOMM. Association for Computing Machinery, New York, NY, USA, pp. 443–454. https://doi.org/10.1145/2619239.2626315. URL: https://doi.org/10.1145/2619239.2626315.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM 51, 107–113. https://doi.org/10.1145/1327452.1327492. URL:https://dl.acm.org/doi/10.1145/1327452.1327492.

van derWerf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A., 2009. Process discovery using integer linear programming. Fundamenta Informaticae 94, 387–412.

van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P., 2005. The prom framework: a new era in process mining tool support. In: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets. Springer-Verlag, Berlin, Heidelberg, pp. 444–454. https://doi.org/10.1007/11494744_25. URL: https://doi.org/10.1007/11494744_25.

Evermann, J., 2016. Scalable process discovery using map-reduce. IEEE Transactions on Services Computing 9, 469–481. https://doi.org/10.1109/TSC.2014.2367525. URL: http://ieeexplore.ieee.org/document/6948229/.

Hadoop, A. Apache Hadoop. URL: https://hadoop.apache.org/ (accessed on 2020-12-11).

Hernandez, S., Ezpeleta, J., Zelst, S.v., Aalst, W.M.P.v.d., 2015. Assessing process discovery scalability in data intensive environments. In: 2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC), IEEE, Limassol. pp. 99–104. URL: http://ieeexplore.ieee.org/document/7406336/, doi: 10.1109/BDC.2015.31.

Hung, C., Amit, P., Manach, S., Alexy, G., BPM-project: The alpha plus algorithm for process mining. URL: https://docs.google.com/document/d/1JtuECbGZ3DusNpmBZhXeq8R_UPCRU5V7NG8GL17h1aA/pub#h.axfqwsh2c5y (accessed on 2020-12-10).

JayaLakshmi, A.N.M., Krishna Kishore, K.V., 2018. Performance evaluation of DNN with other machine learning techniques in a cluster using Apache Spark and MLlib. Journal of King Saud University – Computer and Information Sciences (in press). URL: http://www.sciencedirect.com/science/article/pii/S131915781830212X, doi: 10.1016/j.jksuci.2018.09.022.

Jha, S., Qiu, J., Luckow, A., Mantha, P., Fox, G.C., 2014. A tale of two data-intensive paradigms: applications, abstractions, and architectures, in: 2014 IEEE International Congress on Big Data, pp. 645–652. doi: 10.1109/BigData.Congress.2014.137. iSSN: 2379-7703.

Kalavri, V., Vlassov, V., 2013. MapReduce: limitations, optimizations and open issues. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, Melbourne, Australia, pp. 1031–1038. URL:http://ieeexplore.ieee.org/document/6680946/, doi: 10.1109/TrustCom.2013.126.

Kay, O. Shuffle Internals – Spark – Apache Software Foundation. URL: https://cwiki.apache.org/confluence/display/SPARK/Shuffle+Internals. (accessed on 2020-12-10).

Kllapi, H., Sitaridi, E., Tsangaris, M.M., Ioannidis, Y., 2011. Schedule optimization for data processing flows on the cloud. In: Proceedings of the 2011 International Conference on Management of Data – SIGMOD '11. ACM Press, Athens, Greece, p. 289. https://doi.org/10.1145/1989323.1989355. URL: http://portal.acm.org/citation.cfm?doid=1989323.1989355.

Kreps, J., Narkhede, N., Rao, J., 2011. Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7.

Kumar, S., Mohbey, K.K., 2019. A review on big data based parallel and distributed approaches of pattern mining. Journal of King Saud University – Computer and Information Sciences (in press). URL: http://www.sciencedirect.com/science/article/pii/S131915781930905X, doi: 10.1016/j.jksuci.2019.09.006.

Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P., 2013. Discovering block-structured process models from event logs – a constructive approach. In: Proceedings of the 34th international conference on Application and Theory of Petri Nets and Concurrency. Springer-Verlag, Berlin, Heidelberg, pp. 311–329. https://doi.org/10.1007/978-3-642-38697-8_17. URL:https://doi.org/10.1007/978-3-642-38697-8_17.

Medeiros, A.d., 2004. Process mining: extending the [alpha]-algorithm to mine short loops. Beta, Research School for Operations Management and Logistics, Eindhoven. OCLC: 777757595.

Murata, T., 1989. Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77, 541–580. https://doi.org/10.1109/5.24143. URL:http://ieeexplore.ieee.org/document/24143/.

Regueig, H., Benatallah, B., Nezhad, H.R.M., Toumani, F., 2015. Event correlation analytics: scaling process mining using mapreduce-aware event correlation discovery techniques. IEEE Transactions on Services Computing 8, 847–860. https://doi.org/10.1109/TSC.2015.2476463. URL: http://ieeexplore.ieee.org/document/7239631/.

Regueig, H., Toumani, F., Motahari-Nezhad, H.R., Benatallah, B., 2012. Using mapreduce to scale events correlation discovery for business processes mining. In: Proceedings of the 10th International Conference on Business Process Management. Springer-Verlag, Berlin, Heidelberg, pp. 279–284. https://doi.org/10.1007/978-3-642-32885-522. URL: https://doi.org/10.1007/978-3-642-32885-5_22.

Sahu, M., Chakraborty, R., Nayak, G.K., 2018. A task-level parallelism approach for process discovery. International Journal of Engineering & Technology 7, 2446. https://doi.org/10.14419/ijet.v7i4.14748. URL: https://www.sciencepubco.com/index.php/ijet/article/view/14748.

Sakr, S., Maamar, Z., Awad, A., Benatallah, B., Van Der Aalst, W.M.P., 2018. Business process analytics and big data systems: a roadmap to bridge the gap. IEEE Access 6, 77308–77320. https://doi.org/10.1109/ACCESS.2018.2881759. URL: https://ieeexplore.ieee.org/document/8537879/.

Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Özcan, F., 2015. Clash of the titans: MapReduce vs. Spark for large scale data analytics. Proceedings of the VLDB Endowment 8, 2110–2121. https://doi.org/10.14778/2831360.2831365. URL:https://dl.acm.org/doi/10.14778/2831360.2831365.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The Hadoop Distributed File System. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, Incline Village, NV, USA, pp. 1–10. URL:http://ieeexplore.ieee.org/document/5496972/, doi: 10.1109/MSST.2010.5496972.

Sundari, P.S., Subaji, M., 2020. An improved hidden behavioral pattern mining approach to enhance the performance of recommendation system in a big data environment. Journal of King Saud University - Computer and Information Sciences (in press). URL: http://www.sciencedirect.com/science/article/pii/S1319157820304730, doi: 10.1016/j.jksuci.2020.09.010.

Weijters, A., Aalst, W., Medeiros, A., 2006. Process Mining with the Heuristics Miner-algorithm, vol. 166. Publication Title: Cirp Annals-manufacturing Technology – CIRP ANN-MANUF TECHNOL.

Weijters, A., Ribeiro, J., 2011. Flexible Heuristics Miner (FHM). In: 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM). IEEE, Paris, France, pp. 310–317. https://doi.org/10.1109/CIDM.2011.5949453. URL: http://ieeexplore.ieee.org/document/5949453/.

Wu, J., Guo, S., Huang, H., Liu, W., Xiang, Y., 2018. Information and communications technologies for sustainable development goals: state-of-the-art, needs and perspectives. IEEE Communications Surveys Tutorials 20, 2389–2406. https://doi.org/10.1109/COMST.2018.2812301.

Wu, J., Guo, S., Li, J., Zeng, D., 2016a. Big data meet green challenges: Big data toward green applications. IEEE Systems Journal 10, 888–900. https://doi.org/10.1109/JSYST.2016.2550530.

Wu, J., Guo, S., Li, J., Zeng, D., 2016b. Big data meet green challenges: Greening big data. IEEE Systems Journal 10, 873–887. https://doi.org/10.1109/JSYST.2016.2550538.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I., 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, USA, p. 2.

Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., 2010. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. USENIX Association, USA, p. 10.