Event Correlation Analytics: Scaling Process Mining Using Mapreduce-Aware Event Correlation Discovery Techniques

Hicham Reguieg, Boualem Benatallah, Hamid R Motahari Nezhad, Farouk Toumani

Event Correlation Analytics: Scaling Process Mining Using Mapreduce-Aware Event Correlation Discovery Techniques

Hicham Reguieg, Boualem Benatallah,
Hamid R. Motahari Nezhad, *Senior Member, IEEE*, and Farouk Toumani

Abstract—This paper introduces a scalable process event analysis approach, including parallel algorithms, to support efficient event correlation for big process data. It proposes a two-stages approach for finding potential event relationships, and their verification over big event datasets using MapReduce framework. We report on the experimental results, which show the scalability of the proposed methods, and also on the comparative analysis of the approach with traditional non-parallel approaches in terms of time and cost complexity.

Index Terms—Process mining, correlation discovery, event analytics, mapReduce, distributed computing

1 Introduction

Process analysis and improvement is a key endeavor of any enterprise. A supporting evidence is the increased number of process execution analysis and mining tools available and used today [1], [2], [3]. Model-based analysis is a key technique used to understand process execution landscape and reason about enterprise productivity factors (e.g., time, cost, and quality). However, this is becoming more difficult as processes in the enterprise span over a number of heterogeneous, distributed systems and services spread across various business functions [3]. Consequently, the information related to process execution may be scattered across multiple data sources, and in many cases, the knowledge about how this information is related to each other and to the overall business process execution may not be available or kept updated. Identifying relationships among process-related information (e.g., process-related events) allows discovering process instances and eventually process models that simplify understanding operational processes. We refer to the problem of identifying relationships among process-related events (process events, for short) as *event correlation*. Event correlation has received a notable attention from researchers and practitioners in many application domains including process discovery, monitoring, analysis, browsing and querying [3], [4], [5], [6], [7], [8].

Event correlation consists of discovering a set of correlation conditions that specify how events are related to each other in order to form a (process instance). A correlation condition describes rules that allow to group together events that are generated as the result of the execution of the same process instance. Event correlation discovery comprises of the process of analysing events in various data sources in order to identify event types, exploring different combinations of the attributes of event types for candidate correlation conditions, and verifying whether a candidate correlation condition leads to forming process instances in the context of the overall business process. This task is computationally intensive. It requires the exploration of a huge space of possible correlation conditions among the attributes of different event types, over potentially a very large and evolving event logs [3].

The typical approach for dealing with a large and unstructured dataset is employing parallel algorithms over a cluster of nodes in order to efficiently process the splitted workload [9]. However, in case of correlation discovery over a large dataset the problem is challenging even if a large computational cluster is available. Parallel data processing relies on data distribution and replication for efficient query execution. Partitioning event logs to identify correlation rules, which are used to determine relationships among events in order to identify end-to-end process instances, is a challenging task due to the large size of datasets, the high number of candidate *correlation* rules (also called *correlation* conditions), and the need for efficiently sharing the intermediate computation results among various parallel jobs in various nodes for maximum performance achievement.

Recently, distributed computing technologies have been widely adopted in order to improve system performance in terms of scalability and dependability. As a promising framework, MapReduce has been emerged for processing huge amounts of data on a multitude of machines in a cluster. It provides a simple programming framework that enables harnessing the power of very large data centers, while hiding low level programming details related to parallelization, fault tolerance, and load balancing. It should be noted that distributed parallel computing is however not a trademark of the MapReduce approach but can indeed be realized

- *H. Reguieg and F. Toumani are with the LIMOS, CNRS, Blaise Pascal Univ., France. E-mail: reguieg.hicham@gmail.com, ftoumani@isima.fr.*
- *B. Benatallah ia with the UNSW, Sydney, Australia. E-mail: boualem.benatallah@gmail.com.*
- *H.R. Motahari Nezhad is with the IBM, Almaden, San Jose, CA. E-mail: motahari.hamid@gmail.com.*

Manuscript received 24 Feb. 2014; revised 25 July 2015; accepted 14 Aug. 2015. Date of publication 0 . 0000; date of current version 0 . 0000.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2015.2476463

using other techniques e.g., in memory based technologies. The arguments in favor of using MapReduce for event correlation discovery are: (i) it provides a simple way to implement massive parallelism on a large number of commodity low-end servers [11], while freeing the programmers from the task of tackling the difficulty of traditional parallel programming, (ii) Log files are usually heterogeneous in the sense that they come in a variety of forms. The heterogeneity issue is more easily handled using MapReduce since no predefined schema is imposed on the input data. Finally, MapReduce is suitable, specifically, when high memory machines are not available.

To the best of our knowledge, our prior preliminary work [10] is the first work that introduces parallel, MapReduce-based, algorithms for efficient and scalable discovery of correlation conditions from big process data, and for the purpose of process mining. This is a unique problem posing a number of interesting challenges which makes MapReduce-based algorithms a suitable choice. In this paper, we propose new MapReduce-based techniques to scale correlation rules discovery for the purpose of process mining, extending our previous work in [10] in the following directions: (i) improved and novel algorithms for atomic correlation conditions discovery, (ii) novel algorithms for composite correlation conditions discovery, (iii) complexity analysis of the proposed algorithms, and (iv) extensive experiments. The main difficulties encountered when designing our approach are related to log partitioning and redistribution in order to generate efficient parallel computations. The solutions proposed in this paper cater for wide range of typical (both simple and composite) correlation conditions in process mining. We believe that the proposed solutions chart an effective new paradigm to make correlation condition discovery reusable and scalable. Effective process views exploration and events processing pipelines will work in tandem with iterative MapReduce-based analytics to scale correlation condition discovery analyses. In summary, the main contributions of the paper are as follows:

- We develop multi-pass algorithms to perform correlation condition discovery computations using a scalable parallel shared-nothing data processing platform. More specifically, the proposed approach: (i) introduces efficient methods to partition an events log (aggregated from different data sources) across map-reduce cluster nodes in order to balance the workload related to atomic condition computations while reducing data transfers, and (ii) uses algorithms that are optimal w.r.t. I/O cost and hence are very effective in situations where the size of data to be processed is much larger than the size of the memory available at the processing node.
- We introduce two strategies to perform a MapReduce-based level-wise-like algorithms to explore the space of candidate composite conditions. (i) *Single-pass strategy*, we use the notion of *partitioning conditions* for partition vertically the lattice of candidate composite conditions, and (ii) *Multi-pass strategy*, we partition the lattice horizontally and process each level in a distinct MapReduce job.
- We present experimental results that show the scale-up and speed-up of the algorithms with regard to variation of both data sizes and number of compute nodes. The experiments show that the overhead introduced by MapReduce is negligible compared to the global gain in performance and scalability.

The rest of this paper is organized as follows. Section 2 gives an overview on the event correlation approach used in this paper and introduces the main MapReduce concepts. Section 3 deals with the problem of computing atomic correlation conditions. Section 4 describes the problem of discovering computing composite (conjunctive and disjunctive) correlation conditions along with a set of efficient algorithms for composite correlation condition discovery. Complexity analysis of the proposed algorithms is given at Section 5 while experimental results are presented in Section 6. We discuss related work in Section 7 and draw some future research directions in Section 8.

2 Background

2.1 Overview on Event Correlation Discovery

Let us assume all the event data related to process execution over distributed systems and services are compiled in a process events log. Such a log may contain the payload of messages that are exchanged among systems and services in the course of process execution, and therefore, we may refer to it as a process messages log, interchangeably. In the spirit of [3], [10], we define a process messages log as follows:

Process messages log. A process messages log L, can be viewed as a relation over a relational schema $\mathcal{L}$ (id, $A_1, A_2, \ldots, A_n$), where $\mathrm{U} = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes used in messages and *id* is a special attribute denoting message identifier. Let $X \subseteq \mathrm{U}$, we note by $\pi_X(L)$ the relation corresponding to the projection of L on the attributes of X. Elements of L are called messages (we use *message* and *event* interchangeably). For a message $m \in L$, we denote by $m.A_i$ the value of the attribute A_i in the message m and by $m.id$ the message *id*.

In addition, some of the log attributes are supposed to determine if two given messages belong to the same conversation. This attributes are called *correlator attributes*, and the functions defined over them as *correlation conditions* or *correlation rules*. Correlated messages are identified using a *correlation condition*.

Correlation condition. A *correlation condition*, denoted by $\psi(m_l.A_i, m_p.A_j)$, is a boolean predicate over attributes A_i and A_j of respectively the two messages m_l and m_p. If the condition $\psi(m_l.A_i, m_p.A_j)$ returns t*rue* then we say that m_l and m_p are correlated through the attributes A_i and A_j. We refer to a correlation condition of the form $\psi(m_l.A, m_p.A)$, defined over the same attribute A, as a *key-based* correlation condition. We use *reference-based* correlation conditions, otherwise.

In the sequel, we note such an atomic condition ψ_{A_i,A_j}. A *conjunctive* (respectively, *disjunctive*) condition consists of conjunction (respectively, disjunction) of atomic conditions. In [3], algorithms and heuristics are proposed to identify correlated event sets that lead to discovering potentially interesting process views. The following two criteria and measures have been proposed to select

relevant conditions. First, globally unique keys are not correlator. Two main observations can be made at this stage: (i) an attribute is a possible correlator only if it contains values that are not globally unique, i.e., they can be found in other messages, and (ii) attributes having unique values or attributes with very small domains (e.g., Boolean) are not interesting. The following measures are proposed to capture these properties: $distinct\_ratio(A_i) = \frac{distinct(A_i)}{nonNull(A_i)}$ and $shared\_ratio(\psi) = \frac{|distinct(A_i) \cap distinct(A_j)|}{max|distinct(A_i),distinct(A_j)|}$.

Given a threshold α, the *distinct_ratio* is used to prune conditions defined over the same attribute A_i (i.e., conditions having $distinct\_ratio(A_i) < \alpha$) while the *shared_ratio* is used to prune conditions over two distinct attributes A_i and A_j (i.e., conditions with $shared\_ratio(\psi) < \alpha$). The threshold α can be user provided or computed using information categorical attributes [3].

As a second criteria, a correlation condition ψ is considered not interesting if it partition the log into a high number of small instances or a few number of long instances. To capture this property, the following measure is defined and used: $PI\_ratio(\psi) = \frac{|PI_\psi|}{nonNull(\psi)}$.

where $|PI_\psi|$ denotes the number of process instances identified by the condition ψ and $nonNull(\psi)$ denotes the number of messages for which attributes A_i and A_j of condition ψ are not null. The ratio $PI\_ratio(\psi)$ enables to reason about the number of instances. A threshold β is then used to select interesting conditions as the ones having a $PI\_ratio > \beta$. For example, to select instances that have at least a length of 2, the threshold β should be set to 0.5. This criteria is referred to as *imbalancedPI*.

Based on the measures described above, we present in the following sections, MapReduce-based algorithms to discover interesting correlation conditions and associated process instances from an events log.

2.2 Overview on MapReduce Framework

MapReduce is a new programming model to facilitate the development of parallel computations on large clusters of inexpensive commodity machines [11]. MapReduce provides a simple programming constructs to perform a computation over an input file f through two primitives: a map and a $reduce$ functions. MapReduce operates exclusively on $\langle key, value\rangle$ pairs and produces as output a set of $\langle key, value\rangle$ pairs. A map function takes as input a data set in form of a set of key-value pairs, and for every pair $\langle k, v\rangle$ of the input returns zero or more intermediate key-value pairs $\langle k', v'\rangle$. The map outputs are then processed by $reduce$ function. A $reduce$ function takes as input key-list a pair $\langle k', list(v')\rangle$, where k' is an intermediate key and $list(v')$ is the list of all the intermediate values associated with k', and returns as final result zero or more key-value pairs $\langle k'', v''\rangle$. Several instantiations of the map and $reduce$ functions can operate simultaneously. Note that while map executions do not need any coordination, a given reduce execution requires all the intermediate values associated with a same intermediate key k' (i.e., for a given intermediate key k', all the pairs $\langle k', v'\rangle$ produced by the different map tasks **must be** processed by the same $reduce$ task).

3 Discovering Atomic Correlation Conditions Using MapReduce

We consider a two-stage approach to discover correlation conditions from an event log. The first stage is devoted to the computation of simple correlation rules (called atomic conditions). The second stage is devoted to the discovery of composite correlation conditions (conjunctive and disjunctive conditions).

In this section, we focus on the first stage. Given an events log L, our aim is to discover the interesting atomic correlation conditions and associated process instances. One of the main issues to cope with, is to decide how data and computations should be partitioned, replicated and distributed, in order to efficiently execute the operations entailed by this task. Indeed, there is no unique optimal solution to such a problem since several parameters (e.g., characteristics of the data to be analyzed, physical characteristics of the cluster, such as the bandwidth, memory and cpu at each node, etc.) are involved and may influence the global performance. This is why we propose three algorithms, namely *Sorted Values Centric* (SVC), *Hashed Values Centric* (HVC) and *Per-split Correlated Messages* (PSCM), to handle the problem of discovering *atomic correlation conditions*. All the three algorithms are based on the following general idea.

First, we generate all possible candidate conditions and partition the data across the network by hashing on the candidate name (e.g., $A_i = A_j$). Then, we process each candidate condition ψ_{A_i,A_j} by a single Reduce function and, thus, each candidate can be handled separately and in parallel with the others. Then, interesting correlation conditions are stored into files to be fed to the next step (candidate composite conditions discovery). *SVC* relies on one MapReduce job, it sorts the intermediate data to efficiently compute the correlated message buffer denoted by $\mathcal{CMB}$. However, it involves a large intermediate data size. *HVC* relies on one MapReduce job, has a low intermediate data size, but requires several iterations to compute correlated messages. Finally, *PSCM* relies on two MapReduce jobs. The first step computes the correlated message buffer in parallel, where the second step groups the correlated messages and deduces the process instances. Based on their distinctive features, it is possible to identify suitable situations for each algorithm. For example, *SVC* is adequate for situations where the discovered process instances are numerous and short (having a low number of messages) while *HVC* is more suitable when the discovered process instances are less numerous and long. Finally, *PSCM* is suitable for larger datasets and also for the case of events correlated by key attribute-based conditions. In the sequel, we present the data structure and then we describe in more details the proposed algorithms.

3.1 The Correlated Message Buffer ($\mathcal{CMB}$)

To facilitate correlation computation, we define two types of data structures. The first data structure is similar to inverted index in relational databases, used to index values of the same column (for key-based condition). The second data structure can be obtained by performing a join between two

TABLE 1
Example of CMB Data Structures

Log L		
msg-id	A_i	A_j
m_1	C_2	C_1
m_2	C_2	C_2
m_3	C_1	C_1
m_4	C_1	C_2
m_5	C_3	C_4

(a) *condition* $A_1 = A_1$

Val	$\rightsquigarrow$	$IdSet$
C1	$\rightsquigarrow$	$\{m_3, m_4\}$
C2	$\rightsquigarrow$	$\{m_1, m_2\}$
C3	$\rightsquigarrow$	$\{m_5\}$

(b) *condition* $A_1 = A_2$

Val	$\rightsquigarrow$	$IdSet1$	$IdSet2$
C1	$\rightsquigarrow$	$\{m_3, m_4\}$	$\{m_1, m_3\}$
C2	$\rightsquigarrow$	$\{m_1, m_2\}$	$\{m_2, m_4\}$

inverted indices on the value part (for reference-based condition). The description of this data structure is as follows:

- The first data structure is devoted to *key-based* conditions. This data structure is defined as $T_1 : [val, \{IdSet\}]$, where *val* is a given value of the attribute forming the condition (e.g., A_i) and $\{IdSet\}$ represents the set of messages having val as value in A_i i.e., $\{\{m_x.id\} | m_x.A_i = val\}$. T_1 is an array used to store all the distinct values of a given attribute A_i. This data structure is used to calculate statistics such as: number of distinct values of an attribute, number of correlated messages with a given value. The previous metrics are needed in the pruning phase. Table 1a shows the indexed values of A_1 (respect, A_2). We refer to this table as $\mathcal{CMB}$ (Correlated Message Buffer).
- The second data structure, defined as $T_2 : [$val, IdSet1, IdSet2$]$, is devoted to *reference-based* condition. It is obtained by joining two $\mathcal{CMB}$s on the value part. Values that do not appear in both indexes are discarded. Table 1b shows a shared index of attributes A_1 and A_2, where values $C3$ and $C4$ are discarded.

3.2 Sorted Values Centric Algorithm

The first algorithm devoted to computing atomic conditions is Sorted Values Centric algorithm, depicted in Algorithm 1 (Map) and Algorithm 2(Reduce). It relies on one MapReduce job. The sorted values centric algorithm, as input, requires a log L over the relational schema $\mathcal{L}$ $(A_1, A_2, \ldots, A_n, id)$ and the user provided thresholds α and β. The Map reads a split of the log and, from the set of attributes in $\mathcal{L}$, generates the set of all possible candidate atomic correlation conditions ψ_{A_i,A_j} for two attribute A_i and A_j. This is achieved by computing the cross product $\mathcal{L} \times \mathcal{L}$ (line 2 to 6 of Algorithm 1). For each message, it extracts the values corresponding to A_i and A_j (attributes forming the condition ψ_{A_i,A_j}). In order, to keep track of the origin of each value, the Map tags the values by their original attribute name and message-id (line 5 and 6 of Algorithm 1). Then, it outputs the conditions name and the tagged values as $(key + value, value)$ pairs (lines 7 and 8 of Algorithm 1).

The Map function ensures that : (i) a given pair of attributes A_i and A_j is allocated to only one reducer, and (ii) a given reducer, in charge of the attributes A_i and A_j, will receive all the values of these attributes appearing in L (i.e., the values of the projections $\pi_{A_i}(L)$ and $\pi_{A_j}(L)$ are tagged and sent to the same reducer).

Algorithm 1. Sorted Values Centric Map Function

Input: $\mathcal{K}$: unused, $\mathcal{V}$: a record from the log file
Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V} : \pi_{A_i,A_j}(\mathcal{L})$
1 **begin**
2 **foreach** $A_i \in \mathcal{V}$ **do**
3 **foreach** $A_j \in \mathcal{V}$ **do**
4 condition $\leftarrow$ "$A_i = A_j$" ;
5 $Value_i \leftarrow \{\mathcal{V}.A_i - A_i - \mathcal{V}.id\}$;
6 $Value_j \leftarrow \{\mathcal{V}.A_j - A_j - \mathcal{V}.id\}$;
7 output ({condition - $Value_i$ }, $Value_i$);
8 output ({condition - $Value_j$ }, $Value_j$);
9 **end**
10 `Partitioner` ($\mathcal{K}$: *MapOutputKey*, $\mathcal{V}$: *MapOutputValue*, $\mathcal{N}$: *numPartitions*)
11 **begin**
12 `/** We Hash Partition only the Outputed Key part **/`
13 **return** *Hash(MapOutputKey.OutputKey) % numPartitions*;
14 **end**

Algorithm 2. Sorted Values Centric Reduce Function

Input: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: a Sorted list of Map-Output Values
Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: *PI* set Of discovered instances
1 `Reduce_Configure`
2 $|\mathcal{L}| \leftarrow$ count_rows_log();
3 $\alpha \leftarrow$ getUserThreshold();
4 $\beta \leftarrow$ getUserThreshold();
5 **begin**
6 $\mathcal{CMB} \leftarrow$ build_correlated_message_buffer($\mathcal{V}$) ;
7 *shared_ratio*($\mathcal{K}$) $\leftarrow \frac{|\mathcal{CMB}|}{|\mathcal{L}|}$;
8 **if** *shared_ratio*($\mathcal{K}$) $< \alpha$ **then**
9 $PI_\psi \leftarrow$ compute_instances($\mathcal{CMB}$);
10 **if** ψ *has ImbalancedPI*(PI, β) **then**
11 output($\mathcal{K}$, *PI*) ;
12 **end**

Note that, during the shuffle and sort phase, MapReduce sorts and groups intermediate key-value pairs by their keys. However, it is very convenient for our purposes to also sort the intermediated values since, as detailed below, the computations inside the reducer will take benefits from such operations. Therefore, instead of implementing an additional secondary sorting within the reducer, we used the value-to-key conversion design pattern [12], which is known to provide a scalable solution for secondary sorting. This is achieved by moving intermediate values into the intermediate keys, during the map phase, to form composite keys (line 7 and 8 of algorithm 1), which enables the execution components to handle the sorting. In addition, the partitioning function is customized, to take into account only the origin key-part for hashing and partitioning of data. Hence, values with the same key are still assigned to the same reducer. The merging function at the reducer is also customized to group data w.r.t the original key. The reduce-input are sorted in ascending order and grouped by value, tag and id. Table 2 shows an example of a log file L and an excerpt of the outputs corresponding to the pair of attributes A_i, A_j produced by two mappers that have processed respectively a split made of the first five messages (respectively, the last five messages) of the log L. Once the Reduce (Algorithm 2) collects all the data, it proceeds in three steps,

TABLE 2
Example of a Log and the Outputs of Two Mappers

Log L		
msg-id	A_i	A_j
m_1	C_3	C_4
m_2	C_2	C_2
m_3	C_1	C_2
m_4	C_1	C_1
m_5	C_2	C_1
m_6	C_3	C_3
m_7	C_4	C_3
m_8	C_3	C_4
m_9	C_4	C_3
m_{10}	C_1	C_2

Mapper 1 outputs			
composite-key	val	tag	Id
A_i=A_j+(C_1,A_i,m_3)	C_1	A_i	m_3
A_i=A_j +(C_1,A_i,m_4)	C_1	A_i	m_4
A_i=A_j +(C_1,A_j,m_4)	C_1	A_j	m_4
...	...	...	...

Mapper 2 outputs			
composite-key	val	tag	Id
A_i=A_j +(C_1,A_i,m_{10})	C_1	A_i	m_{10}
A_i=A_j +(C_2,A_j,m_{10})	C_2	A_j	m_{10}
A_i=A_j +(C_3,A_i,m_6)	C_3	A_i	m_6
...	...	...	...

namely building the correlated message buffer (line 6 of Algorithm 2), pruning non-interesting conditions based on non-repeating value criterion (line 7 to 8 of Algorithm 2), and Computing the process instances associated to the condition (line 9 of Algorithm 2). These steps are explained bellow. First we recall that, correlated messages denoted by R_ψ are defined as $R_\psi = \{(\{x, y\})/\forall x \in A_i, \forall y \in A_j\colon\ x.val = y.val\}$. Now regarding the first step, building correlated message buffer works as follows: since the input of the Reduce are sorted and grouped, only one iteration is needed to build $\mathcal{CMB}$ in the case of reference-base conditions. Moreover, message's-ids from A_i appear before those of A_j (assuming that $i < j$). So, for each new distinct value V which appears in the reduce-input, the *Reduce* creates a temporary entry in $\mathcal{CMB}$, with V as *val*. Then, it buffers ids from A_i into $IdSet1$ then those from A_j into $IdSet2$. In case of values having empty $IdSet$ (1 or 2), i.e., none pair of messages $(x, y) \in (A_i, A_j)$ satisfies $x.val = y.val = V$, then V is discarded. For key-based conditions, a new entry of $\mathcal{CMB}$ is created for each new distinct value V in the input of the reduce, and all messages satisfying $x \in A_i,\ and\ x.val = V$ are buffered to the corresponding IdSet.

Example 1. Using as input the buffer in Table 2, the buffer $\mathcal{CMB}$ produced by the function Build Correlated $Message\ Buffer$ is as follows:

- $[C_1, \{m_3, m_4, m_{10}\}, \{m_4, m_5\}]$.
- $[C_2, \{m_2, m_5\}, \{m_2, m_3, m_{10}\}]$.
- $[C_3, \{m_1, m_6, m_8\}, \{m_6, m_7, m_9\}]$.
- $[C_4, \{m_7, m_9\}, \{m_1, m_8\}]$.

Since, the input of the Reduce are sorted, then values in column val and messages in IdSets are also sorted. In the first row, $C1$ is a value which appears in both column A_i and A_j, where messages having $C1$ as value in A_i are $\{m_3,\ m_4, m_{10}\}$ and those having $C1$ in A_j are $\{m_4,\ m_5\}$.

After the $\mathcal{CMB}$ is created, the algorithm starts the second step, i.e., pruning non-interesting conditions based on non-repeating value criterion. The *shared_ratio* can be computed as the ratio of the number of distinct values (number of rows in $\mathcal{CMB}$) with regard to the size of L, $shared\_ratio(\psi_{ij}) = \frac{|\mathcal{CMB}|}{|L|}$. In case of *key-based* conditions, $|\mathcal{CMB}|$ represents the number of distinct values present in the corresponding attribute. In the other case (*reference-based* conditions), $|\mathcal{CMB}|$ represents the number of shared distinct values between A_i and A_j. Next, candidates that do not satisfy $shared\_ratio(\psi_{ij}) < 0.2$ ($\alpha = 0.2$) are pruned (line 8 of algorithm 2).

The third step of the algorithm, computing Instances, is achieved by the compute-instances function. This function, called only in the case of *reference-based* conditions, applies a DFS (Depth-First Search) based algorithm to explore the $\mathcal{CMB}$. It is in charge of grouping together the messages correlated by a condition ψ_{A_i,A_j} in order to form individual process instances. It takes as input a buffer $\mathcal{CMB}$ associated with A_i, A_j. Then, the computation achieved by compute-instances is based on the observation that two messages $m1$ and $m2$ that appear in $\mathcal{CMB}$ are correlated by the condition ψ_{A_i,A_j} if and only if one of the following conditions is satisfied:

(i) the messages $m1$ and $m2$ appear in a same row of $\mathcal{CMB}$. We state this condition more precisely as follows: $m1$ and $m2$ are correlated by ψ_{A_i,A_j} if there exist an integer i such that $m1 \in \mathcal{CMB}\ [i].idset_1$ and $m2 \in \mathcal{CMB}\ [i].idset_2$. Indeed, in this case we have by construction of $\mathcal{CMB}$ that $m1.A_i = m2.A_j = CMB[i].val$. Therefore, we can extend this observation to deduce that the elements of each $\mathcal{CMB}\ [i].idset_1 \cup \mathcal{CMB}\ [i].idset_2$, for $i \in [1, |\mathcal{CMB}\ |]$, are correlated by the condition ψ_{A_i,A_j} and hence belong to the same process instance.

(ii) the messages $m1$ and $m2$ appear in two sets of $\mathcal{CMB}$ that have a non-empty intersection. More formally: there exists $i, j \in [1, |\mathcal{CMB}\ |]$ such that $m1 \in \mathcal{CMB}\ [i].idset_1$, $m2 \in \mathcal{CMB}\ [j].idset_2$ and $\mathcal{CMB}\ [i].idset_1 \cap \mathcal{CMB}\ [j].idset_2 \neq \emptyset$. Indeed, let m be in such an intersection then m is correlated with $m1$ (because both m and $m1$ belongs to $\mathcal{CMB}\ [i].idset_1$) and m is correlated with $m2$ (because both m and $m2$ belongs to $\mathcal{CMB}\ [j].idset_2$). Hence, by using transitivity of the correlation relation we conclude that m, $m1$ and $m2$ belong the same process instance. getNextUnvisited Neighbor() function is used to check this property. Since, message-ids are sorted, a one pass algorithm is applied to check for intersection by performing $2 \times (|idset1| + |idset_2|)$ operations

(iii) $(m1, m2)$ belongs to the transitive closure of the correlation relation computed using (i) and (ii).

The function compute-instances can be viewed as a computation of the connected components of an undirected bipartite graph. The vertices of such a graph are the sets appearing in the columns $idset_1$ and $idset_2$ of $\mathcal{CMB}$ and the edges are constructed as follows: let $i, j \in [1, |\mathcal{CMB}\ |]$, then there is an edge between $\mathcal{CMB}\ [i].idset_1$ and $\mathcal{CMB}\ [j].idset_2$ if: $i = j$ (condition (i) above) , or $\mathcal{CMB}\ [i].idset_1 \cap \mathcal{CMB}\ [j].idset_2 \neq \emptyset$ (condition (ii) above). The condition (iii) is achieved by the computation of the connected components of this graph. Each connected component corresponds to a discovered process instance.

However, in the case of *key-based* conditions, each set of correlated messages corresponding to a distinct value forms a process instance. In other words, each vertex in the graph forms a connected component. Fig. 1 depicts the graph corresponding to the buffer $\mathcal{CMB}$ of the previous example. We can observe that there are two connected components of this graph. The associated discovered process instances

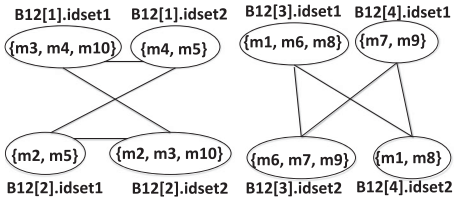

Fig. 1. Bipartite graph of $\mathcal{CMB}$ with two connected components.

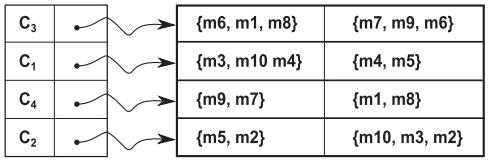

Fig. 2. Correlated messages hash buffer.

are: Instance 1 $= \{m_2, m_3, m_4, m_5, m_{10}\}$, and Instance 2 $= \{m_1, m_6, m_7, m_8, m_9\}$.

Finally, using user provided threshold β, the non interesting instances are pruned (line 10 of algorithm 2). By computing the PI_ratio of each discovered atomic condition ψ, i.e., the ratio of the number of instances associated to ψ to the number of messages for which the attributes A_i and A_2 of condition ψ are defined. The PI_ratio is compared with β and the conditions that do not satisfy the criteria are pruned. In our case, $PI\_ratio = 0.2 < 0.5$. Therefore, the condition "$A_i = A_j$" is considered as interesting. For more details about using thresholds and pruning criteria, we refer to [3].

3.3 Hashed Values Centric Algorithm

In order to avoid generating and transferring intermediate data with duplicated values, as in *SVC* algorithm, we propose the **H**ashed **V**alues **C**entric algorithm. *HVC* processes each candidate atomic correlation condition independently. Also, it can be implemented in a single MapReduce job. The Map generates the set of all possible candidate atomic conditions from the set of attributes in $\mathcal{L}$. It ensures that each pair will be allocated to the corresponding Reduce by assigning a single key to each pair. The main difference w.r.t *SVC* lies in the keys used to distinguish the target Reducers. Unlike *SVC* algorithm, in *HVC* composite keys are not used. Therefore, the map-output data size is not duplicated and, also, not sorted. Table 3 shows an example of the outputs of the log $\mathcal{L}$ of Table 2 processed by two mappers of the *HVC*. Once the *Reduce* receives it corresponding data, it proceeds as follow: build the correlated message hash buffer, pruning non-interesting candidates, and computing process instances. These steps are described below.

The first step, build correlated message hash buffer, aims at grouping together message-ids having same values, and separate those coming from A_i from those coming from A_j in different sets (IdSet1 to A_i and IdSet2 to A_j). Since, the input of the Reduce are not sorted, a hash table is used to store the correlated messages. As a consequence, several iterations are required to achieve this task. Moreover, an additional step is needed to clean the buffer by deleting entries with empty IdSets. This step is less effective than the equivalent one in the *SVC* algorithm.

TABLE 3
Outputs, w.r.t. to (A_1, A_2), of Two Mappers

Mapper 1 outputs			
key	val	tag	Id
A_i=A_j	C_3	i	m_1
A_i=A_j	C_1	i	m_3
A_i=A_j	C_4	j	m_1
A_i=A_j	C_2	j	m_2
...	...	...	...

Mapper 2 outputs			
key	val	tag	Id
A_i=A_j	C_3	i	m_6
A_i=A_j	C_2	j	m_{10}
A_i=A_j	C_3	j	m_6
A_i=A_j	C_3	i	m_8
...	...	...	...

For example, using as input the buffer at Table 3, the buffer $\mathcal{CMB}$ produced by the function Build Correlated Message hash Buffer is depicted at Fig. 2. Several iterations are required to fill the buffer.

The second step of the reduce function of *HVC*, pruning non-interesting candidates, is devoted to the elimination of the candidate conditions that do not satisfy the criterion $shared\_ratio(\psi) < \alpha$. The last step, compute Instances uses a depth-first search-like algorithm to compute the transitive closure of the $\mathcal{CMB}$ and derives the process instances associated to each candidate condition.An algorithm, similar to Algorithm 2 but with few critical changes, is used to compute the discovered instances. The main modification is applied to getNextUnvisited Neighbor() function. Since data are not sorted, checking the intersection requires, in worst case, $2 \times |idSet1| \times |idSet2|$ operations. Using this property we reduce the number of operations to $2 \times (|idSet1| + |idSet2|)$. Finally, conditions are pruned using the *Imbalanced_PI criterion*.

3.4 Per-Split Correlated Messages Algorithm

One problem with sorted values centric and Hashed Values Centric algorithms is the existence of non-correlating values. Therefore, not every message from A_i will be correlated with one or more messages from A_j for a given condition ψ_{A_i,A_j}. Such non-correlating values are identified only after building $\mathcal{CMB}$ which lead to some overhead to eliminate such values. To cope with this problem, we propose the per-split correlated messages algorithm which distributes the computation of the $\mathcal{CMB}$ over processing nodes and anticipates such non-correlating values before constructing the process instances. The per-split correlated message is a two-phase algorithm, each phase corresponding to a separate MapReduce job. This algorithm is introduced to parallelize the computation of the $\mathcal{CMB}$. In case of referenced-based conditions, not all values are included into the intersection of the distinct values of A_i and A_j. This property is stated more formally as { $\exists \mathcal{V}$, $\mathcal{V} \in distinct(A_i) \cup distinct(A_j)$ and $\mathcal{V} \notin distinct(A_i) \cap distinct(A_j)$}. In this case, $\mathcal{V}$ should be ignored and all messages having this value should be discarded. This is done by the phase one of the algorithm, which can be seen as a pre-processing step.

At the phase 1, the Map function of the algorithm adds the attribute values to the output key (the key refers to the condition name). By this way, the map function ensures that all messages from the same condition having the same value will be allocated to a single Reduce. Each Reducer produces a single row of the $\mathcal{CMB}$ for each condition. It fills the buffer row by putting message-ids from A_i into IdSet1 (resp. A_j into IdSet2). Rows with empty IdSet1 or empty IdSet2 are ignored. This step is similar to a standard SQL

query (self-join of log $\mathcal{L}$) but in our case the join is computed for each pair of attributes (A_i, A_j) of the log $\mathcal{L}$.

At the phase 2 of the *PSCM* algorithm, the **Map** is the identity function which returns as an output-key the condition name and as output-value the row of the $\mathcal{CMB}$. The **Reduce** function receives all rows of the same condition. First, it groups them into a single buffer $\mathcal{CMB}$, and then applies the compute_instance() function to compute the set of instances associated to each condition. Finally, non-interesting conditions are pruned based on *Imbalanced_PI criterion*.

4 Discovering Composite Correlation Conditions Using MapReduce

Composite conditions are computed from the set of atomic correlation condition discovered in the previous step, using conjunction operator (respectively disjunction operator). This step is clearly complex and challenging since the search space, made of the candidate composite conditions, is very huge. For example, for a set n of atomic conditions, the number of candidate conjunctive conditions is 2^n. The verification of each candidate requires the execution of some computation-intensive operations such as counting, join and intersection over a large volume of data. To cope with the scalability issue, we designed two MapReduce-Based algorithms, namely Single-Pass Composite Condition Discovery algorithms and Muti-Pass composite Conditions Discovering algorithms, to compute composite conditions from a given set of atomic ones. The distinctive features of each algorithm are the following: To achieve this work, the Single-Pass Composite Condition Discovery algorithms use a unique MapReduce job. This algorithm splits the space of candidates *vertically* in such a way that the subsets can be processed in parallel, each one by a unique reducer. The Muti-Pass composite Conditions Discovering algorithm however operates using several passes, each passe being a separate MapReduce job. This algorithm uses a horizontal partitioning of the computation space, each fragment corresponding to a given level in the lattice of candidates. As explained below, each algorithm is more adequate in specific situations.

For space reasons, in the sequel, we focus the presentation on the problem of computing conjunctive conditions. The computation of disjunctive conditions is achieved using similar algorithms, which are detailed in [13].

4.1 Single-Pass Composite Condition Discovery Algorithms

Usually messages in logs are not only correlated by a single atomic condition. Indeed, several conditions can correlate messages and partition the logs into relevant instances. This case can be viewed as composite keys in relational databases, where multiple attributes are used to identify rows. For instance, messages can be correlated using values of attributes CustomerId (ψ_1: $m_i.CustomerId = m_j.CustomerId$) and **OrderID** ($\psi_2$: $m_i.OrderID = m_j.OrderID$). In this case, we note $\psi_{1\wedge 2} = \psi_1 \wedge \psi_2$.

A *Conjunctive correlation condition* consists of conjunction of at least two atomic conditions. It has the following form: $\Phi = \psi_1 \wedge \psi_2 \wedge \ldots \psi_n$. Where, ψ_is, $1 \leq i \leq n$ are atomic conditions. Conjunctive conditions are computed using conjunctive operator on atomic conditions: let ψ_1 and ψ_2 be two atomic conditions elicited during the previous step, then the goal is to compute the process instances associated to the condition $\psi_1 \wedge \psi_2$, noted $\psi_{1\wedge 2}$. More specifically, given a set AC of atomic conditions, the goal is to identify the set of *minimal* conjunctive conditions that partition the log into interesting process instances. As explained in [3], such a task can be achieved using a levelwise-like approach [14] where, roughly speaking, each level is determined by the length, in terms of number of conjuncts, of the considered conditions. Starting from atomic conditions (level 1), the discovery process consists in two main parts: (i) generating candidate conditions of level l from candidates of level $l-1$, and (ii) pruning non interesting conditions. At each level, the process instances associated with each generated candidate condition are computed and used to prune, if any, the considered candidate condition.

Example 2. Consider as an example a set of atomic condition $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$. The candidate conditions at each level are shown below:

- Level 1 : $\psi_1, \psi_2, \psi_3, \psi_4, \psi_5$
- Level 2 : $\psi_{1\wedge 2}, \psi_{1\wedge 3}, \psi_{1\wedge 4}, \psi_{1\wedge 5}, \psi_{2\wedge 3}, \psi_{2\wedge 4}, \psi_{2\wedge 5}, \psi_{3\wedge 4}, \psi_{3\wedge 5}, \psi_{4\wedge 5}$
- Level 3 : $\psi_{1\wedge 2\wedge 3}, \psi_{1\wedge 2\wedge 4}, \psi_{1\wedge 2\wedge 5}, \psi_{1\wedge 3\wedge 4}, \psi_{1\wedge 3\wedge 5}, \psi_{1\wedge 4\wedge 5}, \psi_{2\wedge 3\wedge 4}, \psi_{2\wedge 3\wedge 5}, \psi_{2\wedge 4\wedge 5}, \psi_{3\wedge 4\wedge 5}$
- Level 4 : $\psi_{1\wedge 2\wedge 3\wedge 4}, \psi_{1\wedge 2\wedge 3\wedge 5}, \psi_{1\wedge 2\wedge 4\wedge 5}, \psi_{1,3\wedge 4\wedge 5}, \psi_{2\wedge 3\wedge 4\wedge 5}$
- Level 5 : $\psi_{1\wedge 2\wedge 3\wedge 4\wedge 5}$

To cast the algorithm Level-wise into a MapReduce framework, the main issue to deal with is how to distribute the candidates among reducers such that the generation and pruning computations are effectively parallelized. We propose to partition the space of candidates in such a way that an element of the partition can be handled by a unique reducer. This enables to avoid multiple MapReduce steps in order to compute conjunctive conditions. Henceforth, each element of the partition is called a *chunk*.

We proceed as follows to compute the partitions. Let AC be a set of n atomic conditions and let $PC = \{\psi_1, \ldots, \psi_l\} \subseteq AC$ be a subset of AC containing l atomic conditions, hereafter called the partitioning conditions. The main idea is to define partition of the space of candidate conditions with respect to the presence or absence of partitioning conditions. We annotate a chunk with a condition ψ_i to indicate that this chunk is made of candidates that contain the subscript i and with $\bar{\psi}_i$ to indicate that the chunk is made of candidates that do not contain such a subscript. Consequently, given AC and PC defined as previously, the partition of the space of candidates ($\mathcal{P}$) using PC is obtained as follows: $\mathcal{P} = \{\psi_1, \bar{\psi}_1\} \times \cdots \times \{\psi_l, \bar{\psi}_l\}$. Each element $(\phi_1, \ldots, \phi_n) \in \mathcal{P}$, with $\phi_1 \in \{\psi_1, \bar{\psi}_1\}$, for $i \in [1, l]$, forms a partition of the space of candidate conditions.

Example 3. Continuing with the previous example with $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$ and assuming that $PC = \{\psi_1, \psi_2\}$, we obtain four possible chunks corresponding to the columns of Table 4. For example, the chunk $(\psi_{\mathbf{1}}, \psi_{\mathbf{2}})$ contains the candidates with subscripts 1 and 2. Each column in Table 4 can be processed separately by a given reducer.

TABLE 4
Partitioned Candidates Space

	(ψ_1, ψ_2)	$(\psi_1, \bar{\psi_2})$	$(\bar{\psi_1}, \psi_2)$	$(\bar{\psi_1}, \bar{\psi_2})$
Level 1		ψ_1	ψ_2	ψ_3, ψ_4, ψ_5
Level 2	$\psi_{1\wedge2}$	$\psi_{1\wedge3}, \psi_{1\wedge4}, \psi_{1\wedge5}$	$\psi_{2\wedge3}, \psi_{2\wedge4}, \psi_{2\wedge5}$	$\psi_{3\wedge4}, \psi_{3\wedge5}, \psi_{4\wedge5}$
Level 3	$\psi_{1\wedge2\wedge3}, \psi_{1\wedge2\wedge4}, \psi_{1\wedge2\wedge5}$	$\psi_{1\wedge3\wedge4}, \psi_{1\wedge3\wedge5}, \psi_{1\wedge4\wedge5}$	$\psi_{2\wedge3\wedge4}, \psi_{2\wedge3\wedge5}, \psi_{2\wedge4\wedge5}$	$\psi_{3\wedge4\wedge5}$
...	...	...	...	...

Note that the obtained chunks are balanced (i.e., they have the same number of candidate conditions) and they form a partition of the initial space of candidates. Indeed, in Table 4, if we also consider that $\emptyset \in (\bar{\psi_1}, \bar{\psi_2})$, then every chunk will have eight candidates. It is also worth noting that each chunk can be treated separately from the others in order to compute the corresponding interesting conditions. The algorithm that enables to compute conjunctive conditions, called Conjunctive-MR, takes as input a set of atomic conditions (*AC*) and iteratively generates candidates of higher level based on candidates in lower level, then it prunes non interesting ones. A classical candidate generation procedure, e.g., see the *Apriori* algorithm [15], computes candidates at level l by a *self-join* on level $l-1$. For example, if both $\psi_{1\wedge2}$ and $\psi_{1\wedge3}$ appear at level 2, than the candidate $\psi_{1\wedge2\wedge3}$ will be generated at level 3. The Map function is used to partition the space of candidates. For a given condition ψ_x, it proceeds as follows:

1) Checks the presence of ψ_x in *PC*.
2) ψ_x is sent to the corresponding reducers, i.e., sent to each reducer in charge of processing a chunk that contains ψ_x, if $\psi_x \in PC$.
3) ψ_x is sent to all reducers, otherwise.

At the Reduce side, (subset of) the partitioning conditions (*PC*) are buffered separately from the remainder conditions. Note that the atomic conditions that do not belong to partitioning condition set are referenced as RC, (i.e., $RC = AC \setminus PC$). Then, each reducer first computes the conjunction of the partitioning conditions presented in its corresponding chunk, i.e., it computes ψ_{PC}, where $\psi_{PC} = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_l$. After that, a *level-wise* algorithm is applied to explore the space of candidate conjunctive conditions built on ψ_{PC}. For example, a given reducer in charge of a chunk with $PC = \{\psi_1, \psi_2, \psi_3\}$ and $RC = \{\psi_4, \psi_5\}$, then it computes ψ_{123}, after that it applies the level-wise algorithm to compute $\psi_{123\wedge4}$, $\psi_{123\wedge5}$ and $\psi_{123\wedge4\wedge5}$. At each iteration of the level-wise the reducers proceed as follows:

(i) *Computing process instances associated to ψ_{PC} and ψ_x (where $\psi_x \in RC$).* The first step computes the messages correlated by $\psi_{PC\wedge x}$. Indeed, this operation relies on the following property: two messages m and m' are correlated by the conjunctive condition ψ_{12} if they are correlated by both ψ_1 and ψ_2 (i.e., $< m, m' > \in \mathcal{CMB}_{\psi_1} \cap \mathcal{CMB}_{\psi_2}$).

(ii) *Pruning candidate conjunctive conditions.* In this step, non interesting conditions are pruned using *ImbalancedPI* criterion. We check if the condition $PI\_ratio(\psi_{PC\wedge x}) < \beta$ is satisfied or not. Using conjunctive operator implies a new criteria that can be applied to prune non interesting conditions. The first criterion is referred as *notMon(ψ)*, it is used to check the monotony of the number and the length of instances with respect to the conjunctive operator. It states that, conjunctive conditions that do not increase the number of discovered instances and do not decrease the length of the already discovered instances is considered as non interesting and therefore pruned. Secondly, if the set of correlated messages of psi_x is included in that of ψ_{PC} (or vice versa), then, the condition $\psi_{PC\wedge x}$ is discarded.

(iii) *Generating candidate conditions.* Candidate conjunctive conditions of $level_l$ are formed using non-pruned from $level_{l-1}$. In the levelness algorithm, candidate conjunctive conditions in the first level are computed by conjunction of condition $\psi_P C$ with each condition from RC. From the second level to higher levels, conjunctive conditions are computed by joining conditions from the previous level with those from RC (e.g., $level_1 = \{\psi_{13}, \psi_{14}\}, RC = \{\psi_3, \psi_4\}$ then $level_2 = \{\psi_{134}\}$). It should be noted that redundancy is eliminated, since $\psi_{123} = \psi_{213} = \psi_{231}$ only ψ_{123} is computed. For example, using the algorithm Conjunctive-MR with inputs as $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$, enables to generate the whole space of candidates described in Table 4. An exceptional case needs a different processing occurs when the reducer chunk does not contain any partitioning condition (i.e., $PC = \emptyset$, e.g., the fourth column in Table 4). The candidate conjunctive conditions in the first level are directly computed from RC and the algorithm follows the same behavior as previous algorithm from the step *(ii)*.

Single-pass composite conditions discovery algorithm provides an efficient strategy to partition, evenly, the space of candidate composite conditions across nodes. In addition, it requires only a single MapReduce job. Therefore, the overhead due to the scheduling and reading data multiple time is reduced. Also, it can be easily implemented and tested. However, the algorithm may suffer from some problems. One potential problem with *single-pass discovery* algorithm is that nodes may be overloaded, especially at the Reduce side where the large part of computations takes place. In some situations, nodes may not handle a large number of candidates having long process instances. To deal with this issue we propose a multi-pass algorithm for composite candidate condition discovery to overcome the problem of overloaded nodes.

4.2 Muti-Pass Composite Conditions Discovering Algorithms

To enhance the performance of the algorithm presented previously and minimize the workload of each node, we introduce a multi-pass algorithm to discover candidate composite conditions. Besides generating and processing candidates, the aim of the algorithm is to minimize the workload allocated to each node by adopting a new strategy of partitioning and splitting the space of candidates.

The multi-pass algorithm relies on several passes (each pass is a MapReduce job). Taking the set of atomic

conditions discovered in previous algorithms as input, every pass of the algorithm is devoted to carry out candidates composite conditions presented in a single level, in the lattice, independently from the others. Based on this strategy any node in the cluster may not be overloaded since it will process only a single candidate condition at each level. Besides that, candidate composite conditions retained in a step (except the last level) are combined to generate the set of candidate of high level and, thus, used as the inputs of the next pass.

The algorithm partitions the lattice horizontally, i.e by levels. It discovers relevant candidate composite conditions presented in $level_k$ in $iteration_k$. Each level is distinguished by the number of atomic conditions merged together (e.g., $\psi_{1,2,3}$ is in $level_3$). Also, it is handled by a single MapReduce job. The first iteration (job) of the algorithm combines the set of $candidate\ atomic\ conditions$, discovered in the previous stage, to generate conditions of $level_2$ then select interesting candidates to be fed to the next iteration. Afterwards, every $iteration_k$ generates the candidate of $level_k$ from the selected candidates, those that are not pruned, of $iteration_{k-1}$. For a given iteration k and before the Map functions start their execution, an initialization function is called to load from the *DistributedCache* the non-pruned candidate conditions from iteration $k-1$ excepting the first iteration which loads the *candidates atomic conditions*. Then, it combines these candidates to generates a set of new candidates. Thereafter, it applies the *associativity* criterion to clean the list from non-interesting candidate. The Map function then retrieves the *correlated message buffers* ($\mathcal{CMB}_\psi$) from HDFS. Next, from each $\mathcal{CMB}_\psi$, it extracts the condition name and uses it to probes the list of keys built in the initialization function for testing whether any key contains the condition name. Hereafter, the Map function produces the key-value pair (*key*, $\mathcal{CMB}_\psi$) for all keys that has the condition name as part.

In the Reduce function, each reducer will receive a single candidate ψ as key, which corresponds to the candidate that will be processed by this reducer. Associated with that key the set of values are two conditions of $level_{k-1}$ such as the combination of their name produces the key. Before, computing instances, the reducer checks whether the conditions satisfies the *non Inclusion* and *Trivial Union* criteria. If so, a *DFS*-like algorithm is applied to discover the process instances involved by the composite conditions. After that, the reducers verifies whether new candidate condition induces a new interesting process instances by carrying out the *monotonicity* and *imbalanced_PI* criteria. If the condition survives the criteria, then the reducer outputs the key-value pairs (ψ, $\mathcal{CMB}_\psi$).

Finally, selected candidate names, in iteration k, are stored into the *DistributedCache* and used to generate candidates for the next iteration and the computed process instances are stored into the HDFS.

5 Complexity Analysis

As explained in previous sections, event correlation discovery requires the exploration of a large space of candidate correlation conditions. This space is made of: *(i)* atomic conditions, with a total number of $a = (k^2 + k)/2$ candidate atomic conditions (i.e., possible pairs of attributes, where k is the number of attributes in the log), and *(ii)* conjunctive and disjunctive correlation conditions. As an upper bound, the number of conjunctive conditions can reach $c = 2^a - 1$ while the upper bound regarding the number of disjunctive conditions is $d = 2^{a+c} - 1$. Therefore, a major issue that makes the correlation discovery a computationally-intensive task comes from the combinatorial explosion of the number of candidate conditions to explore. As shown above, this number depends on the total number of attributes in the log. The number of conjunctive conditions grows exponentially w.r.t. to the number of attributes while the number of disjunctive conditions is double-exponential. Indeed, the relationship between the size of the input dataset and the size of the intermediary results follows the same trend. For example, let us consider an event log file of $1\ Million$ of events with a total number of 50 attributes. Then, if the size of the log file is 1 GB, in worst case the total size of candidate atomic conditions is roughly around 50 GB, while the size of the candidate conjunctive conditions is in the order of magnitude of 2^a GB, with an upper bound of $a = 1{,}250$ (which is in this case larger than $10^{370}\ PB$). Even though, we can expect that in practical cases the effective number of candidate atomic conditions to be much lower than the theoretical upper bound due to the used pruning criteria, the number of candidate conjunctive and disjunctive conditions remains in general very large. For example, continuing with our input log file of 1 GB, if we assume that the number of candidate atomic conditions have been drastically pruned to obtain $a = 20$ (instead of the upper bound $a = 1{,}250$), we still have in this case a total size of candidate conjunctive conditions around 20 TB.

Turning our attention to the complexity of the algorithms presented in this paper, and unlike the result mentioned in [3] which is in $\mathcal{O}(N.|\mathcal{L}|^2)$ for the non MapReduce approach, our algorithms are parallel and hence candidate conditions are processed independently from each other. Since, the main computation is done by reduce function, we omit the complexity of the map function. For each candidate correlation conditions, the reduce function receives a list of the corresponding message values. Therefore, the time complexity to explore all the space of correlation conditions is $\mathcal{O}(p)$ where p is different from one algorithm to another: it consists of the sum of (i) the time complexity of computing correlated messages, and (ii) the time complexity of computing instances.

- In the case of sorted data, the worst case time complexity of building correlated message buffer ($\mathcal{CMB}$) is in $\mathcal{O}(|\mathcal{L}|)$, because there is no need to compare message values with others. The worst case of computing instances is in $\mathcal{O}(d.s)$ where d is the number of distinct values of $A_i \cap A_j$ (the number of entries in the $\mathcal{CMB}$) and s is the size of the largest IdSet.
- In the case of non sorted data, time complexity to compute correlated messages buffer is in $\mathcal{O}(\mathcal{L}^2)$ while computing instances is in $\mathcal{O}(d.s^2)$.

In both algorithms, we assume that reduce functions is able to load $\mathcal{CMB}$ in memory. Therefore, the space complexity is in $\mathcal{O}(|\mathcal{L}|)$ for *key-based* conditions and in $\mathcal{O}(|\mathcal{L}|^2)$ for *reference-based* conditions.

Single-pass composite condition: the time complexity of the reducer in *SPCC* algorithm depends on the number of

candidate atomic conditions assigned to that reducer. Considering that the *partitioning condition size* parameter is equal to r, which leads to a number of reducers equal to 2^r, makes in the worst case, the number of conjunctive conditions hold by a single reducer equal to $\lceil 2^a/2^r \rceil$ and, as a consequence, time complexity in $\mathcal{O}(\lceil 2^a/2^r \rceil . d.s^2)$. In addition, space complexity of the reducers is the sum of the $\mathcal{CMB}$s of all its corresponding candidate conditions.

Multi-pass composite condition: since reducers in *mpcc* algorithm hold only one candidate composite condition, the time complexity is equals to $\mathcal{O}(d.s^2)$ and space complexity is the size of two $\mathcal{CMB}$s. However, the *mpcc* algorithm requires at most $\mathcal{M}$-1 rounds, where $\mathcal{M}$ is the depth of the lattice.

6 Experimental Evaluation

In this section, we report the result of the evaluation of our algorithms, and their performance based on the execution time, and the scale-up and speed-up performance (increasing the amout of data to be processed, and the computing power that is available), which are classical metrics for evaluating the performance of MapReduce-based approaches.

6.1 Environment

We ran experiments on a cluster of five virtual machines. Each machine has one AMD Opteron processor 6234 2.40 GHz with four cores, 4 GB of RAM, and 50 GB of Hard disks. Thus the cluster consists of 20 cores and five disks.

Indeed, the key point is to look at MapReduce as a parallel/distributed computating/programming platform, and to Hadoop as a way to implement it over a cluster and mainly based on disks. While, we can use in-memory architecture to realize the same/similar parallel methods.

6.2 DataSets

We run our algorithms on two different datasets:

- **SCM.** This dataset is the interaction log of SCM [3] business service, developed based on the supply chain management scenario provided by WS-I (the Web Service Interoperability organization). There are eight web services realizing this business service. The interaction log of Web services with clients was collected using a real-world commercial logging system for Web services, HP SOA Manager. This dataset has 19 attributes and 4,050 messages, each corresponding to an activity invocation.
- **Robostrike.** One month collected datasets from a multi-player on-line game service named *Robostrike* (http://www.Robostrike.com/). Players exchange XML messages with game server containing several activities that can be performed during a game session. This dataset has 18 attributes and more than 1.8 million messages.

Increasing dataset size. To evaluate our event correlation algorithms for computing candidate atomic correlation conditions on large datasets, we increase the size of the SCM dataset size while maintaining its data behaviour and distribution. We maintain the number of interesting candidate atomic correlation conditions discovered in the original size. We wanted the number of discovered process instances for each correlation conditions to increase linearly with regard to the size of the increased data. Increasing the dataset size by replicating the original data would only preserve the cardinality of the discovered instances, and may blow-up their sizes. In addition, the original interesting conditions may not satisfy interestingness criteria. Therefore, we scan the whole dataset and for each new event record, we generate clone event records by adding a suffix to its value, and assign a unique identifier to each new event record.

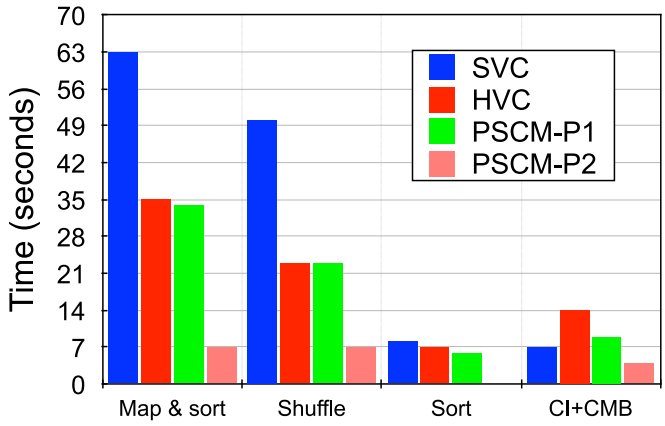

Fig. 3. Time breakdown SCM×100.

In our experiments, and for SCM dataset, we increased the data size by factors of 100, 500 and 1,000. We refer to the enlarged data set as "SCM" $\times\, n$ where $n \in \{100, 500, 1{,}000\}$ represents the increase factor. For our two dataset with XML event payload (SCM and Robostrike), before starting experiments, we extracted all attributes of events and their values from the XML content associated to events using an ETL-like preprocessing step and represented the extracted attribute-value pairs as event tuples in csv files. These files are then grouped into buckets with different size (e.g., RS10K contains one millions events of RobotStrike dataset). Next, we feed them into the hadoop distributed file system for each experiment.

For all the experiments, we used the attribute names to form correlation conditions (e.g., $A_1 = A_2$), assuming A_1 and A_2 are attribute names. We set the lower bound threshold and upper bound threshold used in *non-repeating values* criterion are 0.01 and 0.8 respectively. We define, also, 0.5 as a threshold to prune correlation conditions based on *Imbalanced_PI* criterion.

6.3 Atomic Correlation Condition Discovery Algorithms

As the first experiment, we study the main differences between the three algorithms (Sorted Values Centric, Hashed Values and Per-Split Correlated messages algorithms) for atomic correlation condition discovery. We run the algorithms for three different sizes of SCM× n (where n ∈ {100, 500, 1,000}). We report the total execution time into two main steps (Map and Reduce). Furthermore, the reduce is, also, subdivided into three substeps (shuffle, sort and CMB+CI). $\mathcal{CMB}$ and *CI* denote respectively computing *correlated message buffer* and *compute instances*.

Figs. 3 and 4 show the execution time proportion of each step on the 5-nodes cluster for the dataset "SCM" $\times$ 1,000

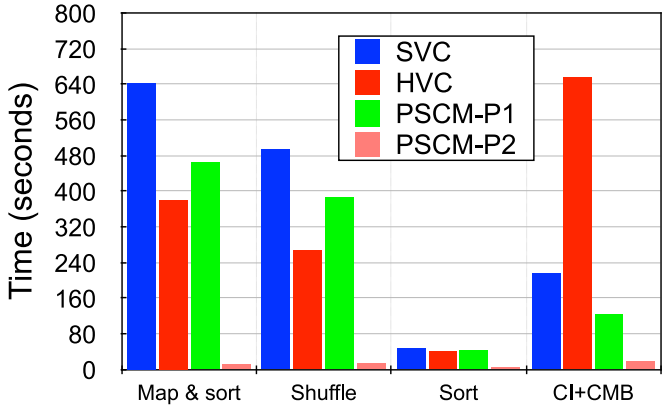

Fig. 4. Time breakdown SCM×1,000.

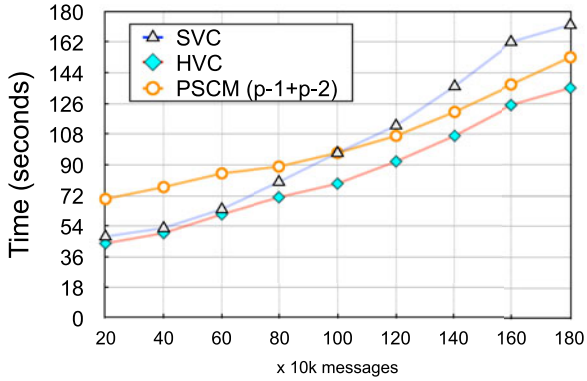

Fig. 6. Execution times on different data size (RobotStrike).

(the results corresponding to $\times 500$ are omitted for space reasons and can be found [13]). Per-Split Correlated Messages consists of two MR jobs, the first phase is denoted as *PSCM-p1* and the second as *PSCM-p2*.

Starting with the *map* phase, we observe that *SVC* is always the most expensive algorithm. This is because the *SVC*'s map-outputs size is twice as those of *HVC* and *PSCM*. This fact implies moving a large amount of data over network during the shuffle phase. A significant difference between *HVC*'s map and *PSCM-p1*'map can be observed in Fig. 4. This is caused by the difference in the map-selectivity in each algorithm. In other words, the *PSCM-p1*'s map step produces a larger number of keys than *HVC*'map. This difference may not be seen in case of small data sizes. Moving to the *reduce* phase, the *shuffle* and *sort* phases are directly affected by the map-output data. Therefore, we do not observe significant changes in the performance (*SVC* is always the worst). On the other hand, during *CMB+CI* phases, *SVC* shows a better performance than *HVC* and *PSCM*, as denoted in Fig. 3. This is because *SVC*'Reducers handle a sorted data. However, in Fig. 4, *PSCM* was the best because it divides this step into two stages. Finally, Fig. 5 shows the total execution time of the three algorithms. For n equals to 100 and 500, *HVC* is the best algorithm. This is because *SVC* is less efficient due to the large size of intermediate-step data, and *PSCM* has an additional overhead due to the need for another MapReduce step. Whereas, for larger dataset sizes, *PSCM* is the best. This means that the framework overheads became negligible when the size of the workload increases.

In order to evaluate the scale up and speed up performance of the algorithms, we conducted two experiments. In the first, we keep the number of nodes fixed and vary the size of the input data (Robostrike). In the second experiment, we keep the input data size fixed (Robostrike and SCM×500) and vary the number of processing nodes.

Fig. 6 presents the execution time observed from executing the three algorithms on *RobotStrike* dataset. We start with 200 k messages as input log size then add 200 k in the next steps until 1,800 k. As the data size increased the execution time of the three algorithms increase linearly. The *x*-axis shows the size of input data in terms of number of messages. The *y*-axis shows the elapsed time in second.

In Fig. 6, moving from the left to right, we observe that as the size of data increases, the execution times of the three algorithms increase linearly. The main observations in Fig. 6 are: (1) *Hashed Values Centric* algorithm, denoted as *HVC*, is the best algorithm for all data sizes. (2) *Sorted Values Centric* algorithm, denoted as *SVC*, has approximately the same performance as *HVC* until 800 thousands messages where its execution time increases significantly. This is because the cost of sorting and shuffling data to nodes over network started to show up. *SVC* is the worst algorithm from 1,000 to 1,800 k messages. Finally, (3) *Per-Split Correlated Messages* algorithm is the worst algorithm for small sizes of data and this is mainly due to the overhead involved in writing and reading data in/from HDFS during the first stage and the cost of scheduling new tasks. But, when the data size gets bigger ($> 1{,}000\ k\ messages$), *PSCM* outperforms *SVC* because dividing the work into two stages becomes an advantage as there is no need for processing a large amount of data.

Moreover, we observed that for all the three algorithms, the amount of intermediate-data transferred over the network between nodes in the shuffle phase is increasing linearly. *SVC* has the largest amount of data transferred over network. This is caused due to the need for adding values to the key-part to be sorted in the map-outputs. *HVC* and *PSCM-p1* have the same data transfer size, since they do not replicate the values. *PSCM-p2* has the smallest intermediate data size, this is due to eliminating non-used messages in *PSCM-p2*.

Robostrike dataset: In Fig. 7, we plot the execution time of the three algorithms on the same Robostrike data size,

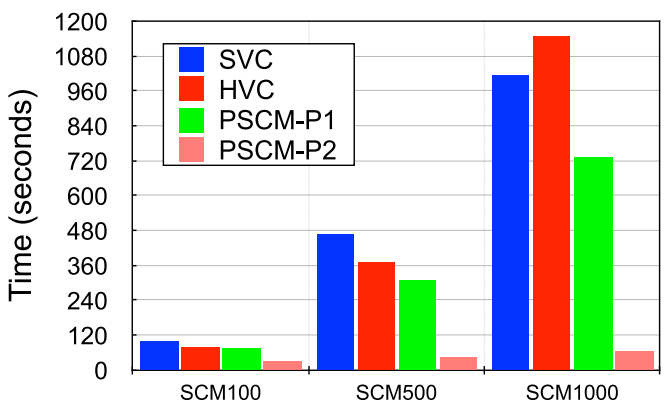

Fig. 5. Total Run time on SCM×n datasets.

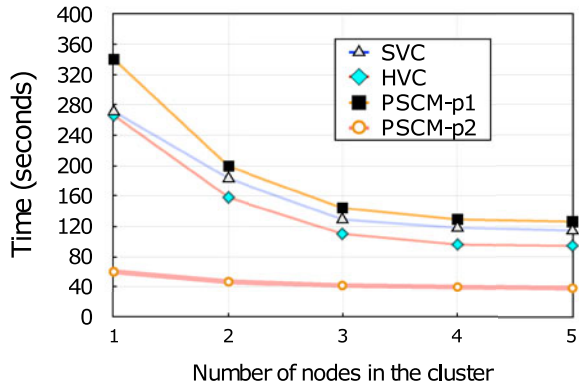

Fig. 7. Execution time of the algorithms for Robostrike data set on different cluster sizes.

varying only the size of the cluster from 1 to 5 nodes. We can see that the *HVC* is the best for all the settings from 1 to 5 nodes (as in previous graphs). The *PSCM-p1* scales faster because of the balance in the number of map-output groups (keys) over nodes. Unlike in *SVC* and *HVC*, the routing keys in *PSCM* are numerous and therefore require the same number of reduce instances. In addition, the reducers in *PSCM-p1* will receive only message ids of a single condition, and messages that have the same value for that condition. On the other hand, *PSCM-p2* has the smallest speed up. The main reason for the poor speed up of *PSCM-p2* is due to: *(1)* the low size of input data (already eliminated by *PSCM-p1*), *(2)* it only groups already computed correlated messages and applies the pruning. In general, the execution time of the three algorithms decrease as the number of nodes increases, and therefore showing a positive speed up performance.

Fig. 8 shows the same result as in Fig. 7 but plotted on *"relative scale"*. In other words, we plot the ratio between the execution time of the current cluster size and the smallest size of the cluster. For example, for the 4-node cluster, we plot the ratio between the execution time on the 1-node cluster and the execution time on the 4-node cluster. We can see that the fastest algorithm is *HVC*. Also, we can see that *PSCM-p1* surpassed *HVC* in 2-node cluster but then worsen. This fact is due to cost of scheduling tasks and partitioning large number of groups. However, all the three algorithms have approximately the same speed up curves and scale well.

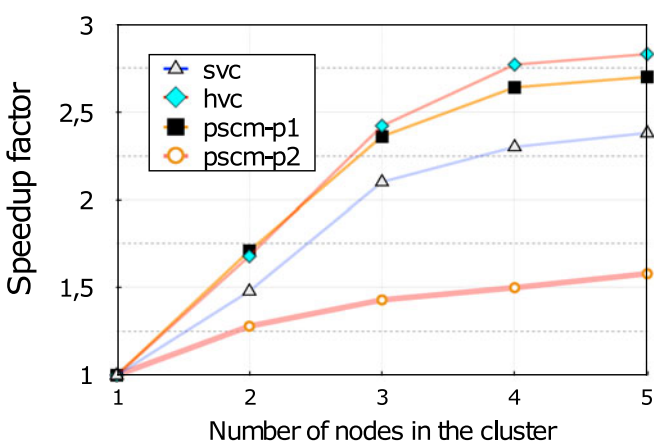

Fig. 8. Relative execution time of the algorithms for Robostrike data set on different cluster sizes.

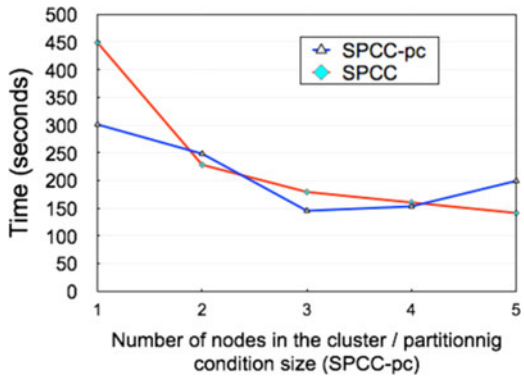

Fig. 9. Execution time of the *SPCC* algorithm for RoboStrike dataset on different cluster sizes (resp. partitioning condition sizes).

SCM×500 dataset. The execution times of the three algorithms on SCM×500 data set on different cluster sizes follow the same pattern as in Fig. 7. *HVC* achieves the best performance. We observed, also, that the *SVC*'s execution time decreases linearly as the cluster size increases. *PSCM-p1*'s execution time is the worst for all cluster sizes. However, its speed up scales faster than *HVC* and *SVC*.

6.4 Composite Correlation Condition Discovery Algorithms

Single-pass conjunctive conditions algorithm. We ran the *single-pass conjunctive conditions* algorithm on a fixed size of *RobotStrike* dataset and varied the number of nodes from 1 to 5. We kept the number of *partitioning conditions* fixed to 3. Consequently, the number of Reduce tasks required to achieve the computations is 2^3. Fig. 10 shows the evolution of the relative execution time w.r.t. the number of nodes. The execution time decreases as the number of nodes increases. The breaking point can be observed moving from configuration with 1-node to configuration with 2-nodes, where the execution time decreases, approximately, by half. This is because the two nodes receive the same workload. Adding more nodes decreases the execution time with a factor of 0.4. This is due to two reasons: (i) nodes do not have the same workload (e.g., with 3-nodes configuration, two nodes receive three tasks and one receives two tasks), (ii) some tasks take more time in execution than the others, depending on the amount of candidates that are pruned before computing their process instances.

In the second experiment, we fix the number of nodes to 5 and vary the number of partitioning conditions from 1 to

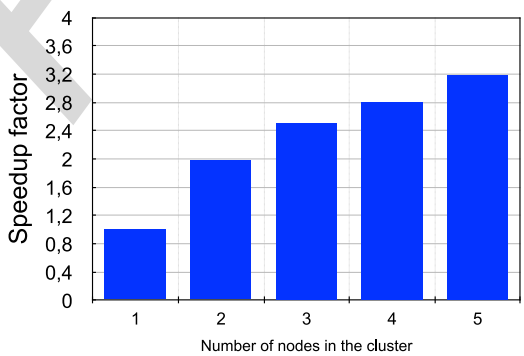

Fig. 10. Relative execution time of the *SPCC* algorithm for RobotStrike dataset set on different cluster sizes.

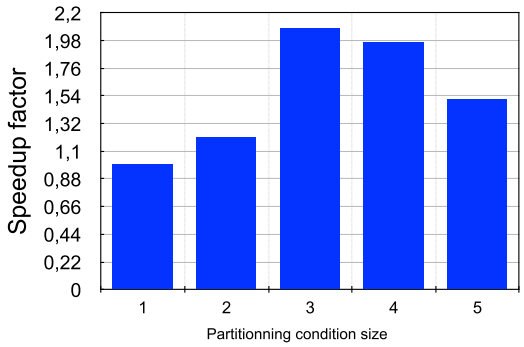

Fig. 11. Relative execution time of the *SPDC* algorithm for RobotStrike dataset set on 5-nodes cluster with different *partitioning conditions* sizes.

5. Fig. 11 shows the relative execution time of different configurations. We start with *partitioning conditions* size (p) equal to 1. This implies 2^1 Reduce tasks. Therefore, only two nodes were working where the other three nodes are idle. We observe that the speed-up factor increases at the beginning then it decreases comparing to the first configuration. In configuration 2 and 3 (in Fig. 9 SPCC-PC graph), the execution time decreases because each node processes at most two Reduce tasks. Also, all nodes receive a piece of workload (no idle nodes). In configuration 4 and 5, increasing p spawns more Reduce tasks. However, it involves larger intermediate data sizes and scheduling a huge number of tasks. These overheads affect the performance of the algorithm and increase the execution time.

Based on this evaluation, we conclude that a good value of p w.r.t. the number of nodes should be a factor of two (e.g., if we have 10 nodes, then cluster sizes p should be equal to 4, and one node coordinating the operation).

6.5 MapReduce versus Centralized Approach

Table 5 shows a brief comparison between the centralized approaches reported in [3] and the approach proposed in this paper using the RoboStrike dataset. We observe the benefits of using a MapReduce approach. Although the data set size is 120 times larger, MapReduce approach outperforms the centralized one. Moreover, centralized approaches suffer from a scalability problem since they cannot handle hundreds of thousands or millions of messages.

In summary, we have conducted experiments to study, most important factors that covers issues for evaluating distributed approaches, both scalability and speed up of our algorithms. Also, we showed the effectiveness of using a parallel method with respect to serial, single node method for event correlation. Moreover, we compared how different event data partitioning and routing impacts the performance. By consequent, we concluded that: a weak or badly designed partitioning may lead to cumulate workloads on some nodes where others are free. Therefore, the fact of using MapReduce does not automatically grants a linear increase in the performance, i.e., adding more nodes will not increase the performance of the job and may even do not improve the performance much.

7 Related Work

Event correlation discovery received a notable attention from researchers and practitioners in many application domains including process discovery, monitoring, analysis and browsing and querying [3], [4], [5], [6], [7], [8]. However, to the best of our knowledge, this is the first work that introduces parallel, MapReduce-based, algorithms for efficient and scalable discovery of correlation conditions from big process data, and for the purpose of process mining. In the following, we discuss related work in three categories.

TABLE 5
Centralized versus MapReduce Approaches

Approach	# of msg.	AC Exec. time	CC Exec. time
Centralized	15 k	$>$ 300 sec	$>$ 120 sec
MR appr.	1,800 K	HVC $<$ 140 sec	
	1,800 K	SVC $<$ 180 sec	
	1,800 K	PSCM $<$ 160 sec	spcc = 150 sec

Correlation rules in business processes. Barros et al. [16] characterized the problem of event correlation in the business processes, where they identified three classes of correlation patterns as function-based (key-based), chain-based and aggregation functions. The proposed correlation patterns identify how events in business processes could be grouped into instances of the execution of business processes. This work does not investigate the problem of correlation condition discovery but focuses on characterizing the problem.

Authors in [6] propose an approach to discover the correlation between message pairs (e.g., *Purchase Order* and *Shipping* message pair) from the log of service interactions. Their approach is based on identifying the conversation identifiers within exchanged messages. They used the term *semantic correlation* to describe how these (key-based) identifiers correlate messages across service interactions. This work is limited to identifying pairwise relationships based on a serial as opposed to a parallel approach. In our earlier work [3], we were first in presenting a method for identifying and discovering event correlation conditions for the purpose of process instance discovery, and eventually process discovery. An issue with investigating only the event pair-level correlation analysis is that events may have shared values on some attributes but may not be part of any instances (which can be used to simply group the messages based the shared values), and process instances that are formed with combination of correlation conditions or chains for them are not discovered. In [7], authors present a saticatical-based approach for the discovery of simple, non-composite correlation rules on business process event data. The presented approach is a serial algorithm, and no parallel or scalability issues have been investigated.

Data correlation in databases, query optimization and search. Brown and Hass [4] present a data-driven technique, called BHUNT, that uses a "Bump Hunting" technique for automatically discovering fuzzy (soft) hidden relationships between pairs of numerical attributes in relational databases, and incorporates this knowledge into an optimizer in the form of algebraic constraints. In [5], the authors introduce CORrelation Detection via Sampling (CORDS), a data-driven technique for automatically discovering correlations and soft functional dependencies between database columns. These approaches are complementary to ours and

focus on identifying data quality issues based on the discovered relationships and constraints on the data.

Finally, we should mention that distributed and parallel computing is a dynamic research area, advancing at a fast pace including memory-based approaches as the latest wave (e.g., Spark). This is still however an evolving space due to the strong assumption that the whole data must entirely fit in the available memory.

8 Conclusion and Future Work

We presented the first generic framework and techniques for systematically endowing process discovery and analysis tasks with efficient and scalable MapReduce-based algorithms. We showed how to efficiently deal with problems such as partitioning, replication, and multiple inputs by manipulating the keys used to route the data between nodes of a MapReduce cluster. Our future work in this area will focus on the exploration of alternative data structures, for example, distributed non-relational database as Hbase to integrate and store event related data. Such kind of data structure provides advantages in distributed cloud storage systems as tables are always sorted by their key and thus can be easily distributed horizontally over several machines. Finally, it would be interesting to experimentally compare our approach w.r.t. emerging new technologies, such as in-memory based solutions, and to investigate their extensions to scenarios where all data cannot fit in the transient memory to complement with fast non-transient data storage devices such as flash.

Acknowledgments

This work has been partially supported by the Labex IMobS3.

References

[1] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, Sep. 2004.
[2] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Softw. Eng. Methodol.*, vol. 7, pp. 215–249, 1998.
[3] H. R. M. Nezhad, R. Saint-Paul, F. Casati, and B. Benatallah, "Event correlation for process discovery from web service interaction logs," *VLDB J.*, vol. 20, no. 3, pp. 417–444, 2011.
[4] P. Brown and P. Hass, "BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data," in *Proc. 29th Int. Conf. Very Large Databases*, 2003, pp. 668–679.
[5] L. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga, "Cords: Automatic generation of correlation statistics in DB2," in *Proc. 30th Int. Conf. Very Large Databases*, 2004, pp. 1341–1344.
[6] W. D. Pauw, R. Hoch, and Y. Huang, "Discovering conversations in web services using semantic correlation analysis," in *Proc. IEEE 19th Int. Conf. Web Services*, 2007, pp. 639–646.
[7] S. Rozsnyai, A. Slominski, and G. Lakshmanan, "Discovering event correlation rules for semi-structured business processes," in *Proc. 5th ACM Int. Conf. Distrib. Event-Based Syst.*, 2011, pp. 75–86.
[8] R. S. Barga, and H. Caituiro-Monge, "Event correlation and pattern detection in CEDR," in *Proc. Int. Conf. Current Trends Database Technol.*, 2006, pp. 919–930.
[9] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using MapReduce," *ACM Comput. Surv.*, vol. 46, 2014.
[10] H. Reguieg, F. Toumani, H. Motahari-Nezhad, and B. Benatallah, "Using mapreduce to scale events correlation discovery for business processes mining," in *Proc. 10th Int. Conf. Bus. Process Manage.*, 2012, 279–284.
[11] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010.
[12] J. Lin and C. Dyer, "Data-intensive text processing with MapReduce," in *Proc. HLT*, 2009.
[13] H. Reguieg, "Using mapreduce to scale event correlation discovery for process mining," Ph.D. dissertation, Blaise Pascal Univ., Aubière, France, 2014.
[14] H. Mannila and H. Toivonen, "Levelwise search and borders of theories in knowledgediscovery," *Data Min. Knowl. Discov.*, vol. 1, pp. 241–258, 1997.
[15] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th Int. Conf. Very Large Databases*, 1994, pp. 487–499.
[16] A. Barros, G. Decker, M. Dumas, and F. Weber, "Correlation patterns in service-oriented architectures," in *Proc. Int. Conf. Fundamental Approaches Softw. Eng.*, 2007, pp. 245–259.

Hicham Reguieg is an assistant professor of computer science at the USTO, Algeria. His research interests include events correlation discovery, process mining and web services.

Boualem Benatallah is a scientia professor of computer science and engineering at UNSW Australia. His main research interests include service-based programming, process discovery, end users analytics, cloud services, and crowd sourcing.

Hamid R. Motahari Nezhad is a research staff member, and data analytics research lead at IBM Almaden Research Center. His research interest include data analytics, services computing, cognitive computing and its applications in the area of business process and case management. He is a senior member of the IEEE.

Farouk Toumani is a full professor of computer science at UBP, France. His research interests lie in the areas services oriented computing, business processes, and big data.

Queries to the Author

Q1. Please provide page range in Refs. [9] and [12].
Q2. There are 16 references in source file and 17 in authors PDF. We have followed the source file.
Q3. Please provide photos of authors.