Efficient Event Correlation over Distributed Systems

Long Cheng, Boudewijn F. van Dongen and Wil M.P. van der Aalst
Department Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
L.Cheng@tue.nl B.F.v.Dongen@tue.nl W.M.P.v.d.Aalst@tue.nl

***Abstract*—Event correlation is a cornerstone for process discovery over event logs crossing multiple data sources. The computed correlation rules and process instances will greatly help us to unleash the power of process mining. However, exploring all possible event correlations over a log could be time consuming, especially when the log is large. State-of-the-art methods based on MapReduce designed to handle this challenge have offered significant performance improvements over standalone implementations. However, all existing techniques are still based on a conventional generating-and-pruning scheme. Therefore, event partitioning across multiple machines is often inefficient. In this paper, following the principle of filtering-and-verification, we propose a new algorithm, called RF-GraP, which provides a more efficient correlation over distributed systems. We present the detailed implementation of our approach and conduct a quantitative evaluation using the Spark platform. Experimental results demonstrate that the proposed method is indeed efficient. Compared to the state-of-the-art, we are able to achieve significant performance speedups with obviously less network communication.**

***Keywords*-event correlation; process mining; service computing; data partitioning; big data; data-intensive computing**

I. Introduction

Modern IT systems collect and store large mounts of event data. For instance, ERP systems log business transaction events, and high-tech systems such as X-ray machines record software and hardware events [1]. Such "historical event data" can be used to extract non-trivial knowledge and interesting insights using current mining techniques [2], [3]. As one of the key tasks in such scenarios, *event correlation*, has received notable attention from researchers and practitioners in various domains.

A typical example is the domain of process mining [3], which includes three main types of analysis. *Process discovery* is concerned with automatically extracting a process model from an event log; *conformance checking* measures how well the behaviour recorded in an event log fits a given process model and vice versa; and *process enhancement* focuses improving an existing model using information about the actual process recorded in an event log [4]. After correlating events, one is then able to unleash the power of process mining, i.e., discover process models providing novel insights, check compliance with respect to some normative model, and analyze the bottlenecks and other performance hazards in operational processes.

Event correlation mainly comprises the operation of discovering a set of correlation rules and on that basis to group process events into process instances [5]. To illustrate this, an example of a simplified log is shown in Table I, in which there are four events and each with four attributes[1]. Assume that we have worked out a correlation rule ψ_{A_1,A_1} (i.e., events having the same value on the attribute A_1) from the log already, then we will be able to get several process instances such as $\langle e_1, e_2\rangle$, $\langle e_3\rangle$ and $\langle e_4\rangle$ very easily. We will give the details of this processing in Section II.

TABLE I: An example of a simplified log

Event	A_1	A_2	A_3	A_4	A_5
e_1	C_2	C_1	C_2	C_5	C_6
e_2	C_2	C_2	C_4	C_4	C_6
e_3	C_1	C_1	C_2	C_5	C_7
e_4	C_3	C_2	C_4	C_4	C_6

Nowadays, event logs recorded for processes executed in highly variable and heterogeneous contexts become common in modern information systems. This makes discovering correlation rules and forming process instances in the context of an overall business process very hard [5]. The reason is that events collected from multiple data sources could lack structures and contexts. In fact, correlating events is also computationally expensive, as it requires the exploration of a huge space of possible correlation conditions among the attributes of different event types [6]. This means that an efficient execution of event correlations will be challenging and also desirable in the presence of large logs. In fact, we can all witness that datasets containing event data are growing in size [3].

Significant progress has been made to realize high performance event correlation systems. For example, [6] uses database queries to compute all the correlated event pairs for all the potential correlation rules and then prunes the non-interesting ones by applying non-interestingness criteria. The approach has been shown to be very efficient as demonstrated by the experimental results reported in [6]. As the amount of event log data continues to scale, the method will be not suitable for process entire data sets on a single machine, due to the hardware limitations (e.g., CPU and memory). To handle this issue, state-of-the-art work research [5] uses a MapReduce-based method for event correlations over distributed platforms. The evaluations presented in [5] have shown that the approach can offer significant performance improvements over the implementation of [6]. Nevertheless,

[1]To simplify our presentation, we do not consider the attribute A_5 (gray) in the log for all of our examples, unless otherwise specified.

the proposed technique [5] is still based on a conventional *generating-and-pruning* scheme. In this case, the number of generated candidate correlation rules will be large, and as a result, the partitioning of log events across multiple machines will be inefficient, in terms of time cost on both computation and network communication.

In this work, following the *filtering-and-verification* principle, we propose a new approach, called *RF-GraP* (**R**ule **F**iltering and **Gra**ph **P**artitioning), which aims at a more efficient event correlation using distributed platforms. We provide the detailed design and conducted a performance evaluation of our method. The main contributions of this paper can be summarized as follows:

- We introduce the state-of-the-art event correlation approach and analyze its possible performance issues in processing large event logs over distributed systems.
- We adopt the principle of filtering-and-verification for efficient correlation analytics over large logs, by incorporating light-weight filter units into candidate correlation rules, which can be utilized to prune large numbers of non-interesting rules before verification.
- To further improve our performance, in the verification phase, we model existing correlation rules as a graph and introduce a graph partitioning approach to decompose the potentially correlated events into chunks by exploring efficient data locality assignment.
- We describe the implementation of our approach and report on experiments using Spark [7]. The results demonstrate that we can achieve significant performance improvements over the state-of-the-art method in [5]. For example, for a log with 1 million events and 38 attributes, our algorithm performs 4.5× faster with 43× less network traffic.

The remainder of this paper is organized as follows. In Section II, we introduce preliminaries including the state-of-the-art techniques on event correlation. We present the detailed design and implementation of our approach in Section III. We report on the evaluation of our approach in Section IV. We discuss related work in Section V and summarize our findings in Section VI.

II. PRELIMINARIES

In this section, we introduce the preliminaries of event correlations and also discuss possible performance issues that current approaches suffer from.

A. Event Correlations

In this work, we build on the standard process mining notations as defined in [5], [6].

Definition 2.1: [6] A *process event log* is a set of events $\mathcal{L} = \{e_1, e_2, ..., e_l\}$, where each event e is represented by a tuple $e_i \in A_1 \times A_2 \times ... \times A_k$, and attributes $A_1, ..., A_k$ represent the union of all the attributes contained in all events. An event typically contains only a subset of these attributes and will therefore have many of its attributes undefined, which will be marked with the value *null*.

For example, the event log in Table I has four events and four attributes. For an event $e \in \mathcal{L}$, we denote the value of the attribute A_i in the event e by $e.A_i$ in the following.

Definition 2.2: [5], [6] A defined *correlation condition (or rule)*, is denoted by $\psi(e_x.A_i, e_y.A_j)$, over attributes A_i and A_j of respectively the two events e_x and e_y. If the condition $\psi(e_x.A_i, e_y.A_j)$ returns true then we say that e_x and e_y are correlated through the attributes A_i and A_j.

There are two kinds of correlation conditions: atomic correlation and multiple correlation conditions (e.g., conjunctive conditions and disjunctive conditions). As the former case is the most basic condition, we will focus on the atomic correlation conditions in this work.

Definition 2.3: [6] An *atomic correlation condition* ψ_{A_i,A_j} specifies that two events are correlated if they have the same value on two of their attributes A_i and A_j, namely, there exists $e_x.A_i = e_y.A_j$ (or $e_x.A_j = e_y.A_i$).

The atomic correlation condition contains two types of correlations: *key-based correlation* and *reference-based correlation*. For the former correlation, a unique value is used in each event to directly identify a instance to which it belongs. For the later one, events of an instance are correlated using a reference with a previous event in the instance. Namely, ψ_{A_i,A_j} is a key-based correlation only when $i = j$, otherwise, it is a reference-based correlation. For example, the correlation rule ψ_{A_1,A_1} as described in Section I is a key-based condition, and the condition ψ_{A_1,A_2} is a reference-based correlation rule. In our presentation, sometimes we will also use $A_i = A_j$ to replace the condition ψ_{A_i,A_j} in this paper[2].

Definition 2.4: [6] A *process instance* p is a sequence of events corresponding to a subset of events of the log $\mathcal{L}$.

For instance, the sequence of $\langle e_1, e_2 \rangle$ in Section I is a process instance.

B. Correlation Discovery

Similar to the approach in [5], [6], we identify the interesting correlation conditions on the basis of our observation that some are non-interesting [8].

The first observation is based on the distribution of attribution values over all the events. For a key-based condition, if the value domain of an attribute A_i is very small (e.g., Boolean), then the condition $A_i = A_i$ will be not interesting [6]. In the meantime, for a reference-based case, if most of the values of an attribute can not be found in another attribute, then the two attributes will be not correlated. These two cases can be measured by the properties $distinct\_ratio(A_i)$ and $shared\_ratio(\psi_{A_i,A_j})$, as defined in Equation (1.1) and (1.2) respectively. Generally, a threshold α is used to prune the non-interesting conditions. Namely, a correlation condition will be interesting if its computed property value is greater than α.

$$distinct\_ratio(A_i) = \frac{|distinct(A_i)|}{|nonNull(A_i)|} \quad (1.1)$$

$$shared\_ratio(\psi_{A_i,A_j}) = \frac{|distinct(A_i) \cap distinct(A_j)|}{Max\{|distinct(A_i)|, |distinct(A_j)|\}} \ (i \neq j) \quad (1.2)$$

[2]Note that the condition $A_i = A_j$ will be the same as $A_j = A_i$.

The second observation is based on the number of discovered process instances. Namely, a correlation condition will be not interesting if it partitions a log into large number of short instances or small number of long instances. This can be measured by the property $PI\_ratio(\psi)$ as described in Equation (2). There, $|PI_\psi|$ means the number of discovered process instances over the condition ψ, and $nonNull(\psi)$ denotes the number of events for which attributes A_i and A_j of the condition ψ are not null. A threshold β is used to select the interesting conditions which meet the condition $PI\_ratio(\psi) \leq \beta$.

$$PI\_ratio(\psi) = \frac{|PI_\psi|}{nonNull(\psi)} \quad (2)$$

To efficiently compute the values of above properties for a given log, a data structure (i.e., a table) named *Correlated Message Buffer* (CMB) has been proposed in [5]. For each potential correlation condition ψ, a CMB can aggregate all the events into *event sets* according to the values of their attributes in ψ. For example, Figure 1 shows two such buffers based on the correlation conditions $e_x.A_1 = e_y.A_1$ and $e_x.A_1 = e_y.A_2$, respectively. Using the two tables, we can quickly get that the value of $distinct\_ratio(A_1)$ is $3/4$ and $shared\_ratio(\psi)$ is $2/3$. Moreover, we can also discover the potential interesting process instances and consequently calculate the value of $PI\_ratio(\psi)$. For the key-based condition in Figure 1(a), the instances are the aggregated event sets, i.e., there are three instances there. For a reference-based case, process instances are computed by applying the DFS (Depth-First Search) algorithm over a bipartite graph [9], in which the two disjoint vertex sets U and V are represented by the aggregated event sets in a CMB. For the case in Figure 1(b), we have that $U = \{\{e_3\}, \{e_1, e_2\}\}$ and $V = \{\{e_1, e_3\}, \{e_2, e_4\}\}$. Using DFS, we can get a path $\{e_3\} \rightarrow \{e_1, e_3\} \rightarrow \{e_1, e_2\} \rightarrow \{e_2, e_4\}$ from the graph, thus the discovered process will be $\langle e_1, e_2, e_3, e_4\rangle$. As there is only one process instance for the condition $A_1 = A_2$, the value of its PI_ratio will be $1/4$. If we set $\alpha = 0.5$ and $\beta = 0.5$, then the condition $A_1 = A_2$ is an interesting rule and $A_1 = A_1$ is not. We will show in our later evaluation in Section IV-B, correlations over such a criterion can indeed discover interesting rules from real logs.

C. Current Approaches

Several systems/techniques have been developed to facilitate correlations over event logs in the past years [10]. In the following, we mainly focus on two typical approaches with regard to efficient computation of event correlations. One is based on relational databases systems [6] and the other uses the MapReduce framework [5].

Motahari-Nezhad et al. [6] introduce an approach to perform standard SQL queries over a standalone database system to retrieve the CMB related information. They first generate all the potential atomic correlation conditions based on the attributes of an event log, and then they implement each condition as a self-join over the log table. For example, correlations over the condition $A_1 = A_2$ for the log in Table I are done by joining the table itself, on its columns A_1 and A_2. Obviously, this approach can use all advantages of current database systems on query/join executions (e.g., using indexes) and thus it could be very efficient when computing event correlations. However, join operations are always time-consuming and considered as the performance bottlenecks of current data processing systems [11]. When the number of attributes of a log is large, large number of self-join executions generated by the potential correlation rules will undoubtedly lead to a poor correlation performance. Moreover, when the number of events is large, the intermediate results (e.g., aggregated event sets) generated by query executions would be huge. This will impact the performance of subsequet operations, i.e., computing process instances, due to the associated memory and computing cost. In such scenarios, the approach presented in [6] will not be an ideal solution for large event logs processing (e.g., including millions of events or more).

Val		EventSet
C_1	$\rightarrow$	e_3
C_2	$\rightarrow$	e_1, e_2
C_3	$\rightarrow$	e_4

(a) condition $A_1 = A_1$

Val		EventSet1	EventSet2
C_1	$\rightarrow$	e_3	e_1, e_3
C_2	$\rightarrow$	e_1, e_2	e_2, e_4

(b) condition $A_1 = A_2$

Fig. 1: Two generated CMBs based on applying two correlation conditions over the log in Table I.

key	(val, tag, event)
$A_1 = A_1$	(C_2,1,e_1), (C_2,1,e_1)
$A_1 = A_2$	(C_2,1,e_1), (C_1,2,e_1)
$A_1 = A_3$	(C_2,1,e_1), (C_2,3,e_1)
$A_1 = A_4$	(C_2,1,e_1), (C_5,4,e_1)
$A_2 = A_2$	(C_1,2,e_1),...
$A_2 = A_3$	(C_1,2,e_1),...
$A_2 = A_4$	(C_1,2,e_1),...
$A_3 = A_3$	(C_2,3,e_1),...
$A_3 = A_4$	(C_2,3,e_1),...
$A_4 = A_4$	(C_5,4,e_1),...

(a) message pairs for event e_1

key	(val, tag, event)
$A_1 = A_2$	(C_2,1,e_2), (C_2,2,e_2)

(b) $A_1 = A_2$ for event e_2

key	(val, tag, event)
$A_1 = A_2$	(C_1,1,e_3), (C_1,2,e_3)

(c) $A_1 = A_2$ for event e_3

key	(val, tag, event)
$A_1 = A_2$	(C_3,1,e_4), (C_2,2,e_4)

(d) $A_1 = A_2$ for event e_4

Fig. 2: Generated message pairs by applying candidate correlation conditions on each event as listed in Table I.

To handle big event logs, Reguieg et al. [5] have proposed an algorithm called *Hashed Values Centric*[3] (refereed to as HVC in the following) over the popular used MapReduce framework [12]. They focus on how to efficiently partition events and consequently maximize the correlation performance by using the advantages of parallel implementations. The main workflow of HVC can be divided into the following three phases: (1) based on all the potential correlation rules, message pairs[4] in the form of *(rule, (value, tag, event))* are generated over all the events on each computing node. The *value* in the pairs means the attribute value and the *tag* is the index number i for the attribute A_i. For example, as illustrated in Figure 2, for a given rule $A_1 = A_2$, two pairs will be generated for each event; (2) all the generated message pairs are redistributed to

[3]The authors have also proposed two additional methods in their work. However, the HVC approach generally performs the best in their experimental results. Therefore, we only focus on HVC in this work.

[4]As the term *message* in some papers means *event*. To avoid confusion, throughout this work, when we say *message* or *pair* or *message pair*, we mean the newly generated data which to be transferred over networks.

all the nodes based on the hash values of their correlation rules. For a two-node computing system, assume that the events e_1 and e_2 are located in node 1, e_3 and e_4 are in node 2 and the hash value of $A_1 = A_2$ is 1, then all the generated 8 message pairs will be transferred to the node 1; and (3) local CMBs are built based on the received message pairs and the entire correlation analytics will be terminated when all the CMBs have been processed. For instance, the CMB illustrated by Figure 1(b) will be built by the 8 message pairs received on node 1 in the second phase. With constructed CMBs on each node and the criteria described previously, we can then compute for the interesting conditions and process instances.

Compared to the method [6], the high-level implementation of [5] is more general, as it can be run on ordinary clusters and file systems. Moreover, experimental results have shown that [5] can outperform [6] under various conditions. As such, we consider the method [5] as the state-of-the-art technique. Since each phase of HVC as described above can be parallelized across nodes, the scheme offers the potential for scalability and thus can be applied on large logs. However, the HVC algorithm is still within the *generating-and-pruning* scheme. Similar as [6], all its operations are based on **all** the potential correlation rules. Namely, the number of generated message pairs in HVC could be very huge, and this will make the network communication and computing expensive in a distributed environment.

Figure 2(a) illustrates the problem: we have 10 candidate correlation rules for the log in Table I, and two message pairs are generated for each event and for each rule. This means that for a very small log with only 4 events, $10 \times 2 \times 4 = 80$ message pairs will be generated by HVC, before subsequent operations, i.e., data redistribution, event aggregation and process instance computation, can be performed. More important, the number of the generated pairs will increase with increasing the number of events and number of attributes, which will degrade the performance of [6] in the presence of large logs. In the following, we will show how the performance of event correlations can be further improved by using a more advanced scheme.

III. Our Approach

In this section, we present the detailed design and implementation of our RF-GraP approach. Moreover, we also discuss about its performance advantages by the comparison with the state-of-the-art method.

A. Overview of Design

Similar to the approach in [5], we utilize a distributed method to discover interesting rules and process instances over an input log. Nevertheless, to achieve a higher performance, we mainly focus on how to efficiently reduce the *number of transferred message pairs* over networks rather than how to parallelize correlation workloads. The reason is that the metric gives the insight into cost on computation and network communication. For example, the larger the number of message pairs a node receives, the greater the associated workload could be. To achieve our target, we introduce a new approach, RF-GraP, following the *filtering-and-verification* scheme. In the filter phase, we incorporate a light-weight filter unit into all possible correlation rules to prune large number of non-interesting rules without significantly increasing processing time. For the rest candidate rules, in the verification phase, we use a graph partitioning approach to decompose the potentially correlated events into chunks by exploring data partitioning and locality assignment. In general, we divide our approach into the following six steps:

- *Step 1.* Model all the possible correlation conditions of a log as a correlation graph.
- *Step 2.* Simplify the correlation graph by filtering out potential interesting rules.
- *Step 3.* Partition the simplified correlation graph by exploring data locality.
- *Step 4.* Generate message pairs based on the partitioned result and then transfer them to the responsible nodes.
- *Step 5.* Build local data structures (e.g., CMBs) based on the allocated message pairs at each node.
- *Step 6.* Compute the process instances and calculate the values of $PI\_ratio(\psi)$ to get the final outputs.

The graph modeling process in Step 1 is relatively simple, and the methods to compute process instances in Step 6 have been extensively described in [5], [6] as well as our Section II. Therefore, in the following, we will focus on the details of rule filtering, data partitioning and placement as described in Steps 2-4. Moreover, we will also propose a more efficient data structure rather than the CMB [5] for process instance computing as mentioned in Step 5.

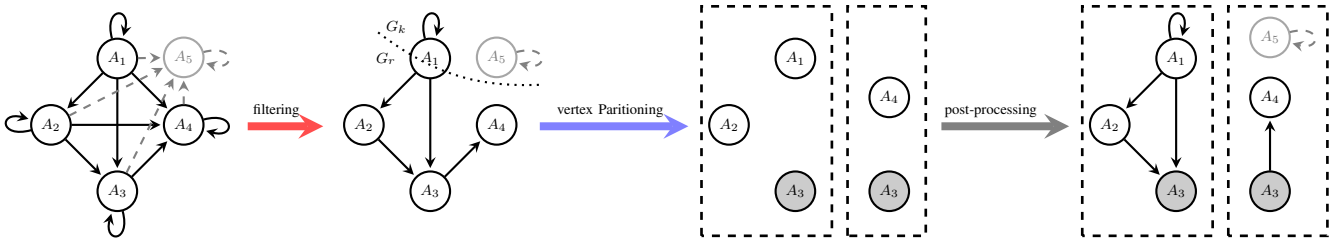
B. Implementation of RF-GraP

The detailed implementation of our approach is shown in Algorithm 1. Again, log $\mathcal{L}_1$ in Table I is used as an example. We assume that there are two nodes[5], the first two events e_1 and e_2 are located in node 1 and the other two are in node 2.

1) Rule Filtering: Instead of handling each correlation rule independently like in current approaches, before our filtering process, we treat the set of rules as a graph. We model all the potential correlation rules as a *correlation graph* $G_c = (\mathcal{V}, \mathcal{E})$, where the vertexes $\mathcal{V}$ are the attributes in a log $\mathcal{L}$, i.e., $\mathcal{V} = \{A_1, ...A_n\}$, and $\mathcal{E} = \{(A_i, A_j) \in \mathcal{V} \times \mathcal{V} | i \leq j\}$ is the set of edges. Intuitively, each edge between two vertexes in G_c indicates a potential correlation condition, and the graph G_c shows all the possible conditions of $\mathcal{L}$. For example, the graph in Figure 3(a) is the correlation graph of $\mathcal{L}_1$.

We apply the first criterion as introduced in Section II-B as a filter in our approach. For each attribute A_i in log $\mathcal{L}$, it is easy to get the frequency information of its values (lines 4-5 in the algorithm). Specially, in a distributed environment, we can get such information very quickly using an approach similar as the *WordCount* implementation. For example, for

[5]Note that, we focus on explaining the high-level implementation in a distributed environment here. In this context, a node here refers to a computing unit (e.g., an execution core in Spark).



(a) a correlation graph G_c (b) the simplified graph G_s (c) partitioning result of graph (d) solution for data assignment

Fig. 3: The filtering and graph partitioning as well as the assignment solution of message pairs in our approach.

the attribute A_1 in $\mathcal{L}_1$, the value C_2 appears 2 times on node 1 and 0 times on node 2, therefore we know that it appears 2 times in attribute A_1. With such statistic information for all the attribute values, we can then examine whether the value of $distinct\_ratio(A_i)$ or $shared\_ratio(\psi)$ for each correlation rule meets the threshold α (lines 6-18). Consequently, we can easily prune part of correlation rules (i.e., the non-interesting ones) and filter out the potentially interested ones ψ_f before we generate any message pairs. This way, the original correlation graph G_c can be transformed to a simplified graph G_s, by removing the edges not meeting the threshold. For $\mathcal{L}_1$, if $\alpha = 0.7$, then we will get its simplified correlation graph as Figure 3(b).

2) Graph Partitioning: Similar as [5], we create message pairs based on correlation rules and then partition them to facilitate the parallel execution. Nevertheless, we only generate pairs for the filtered rules. Moreover, to achieve better performance, we adopt graph partitioning in our approach rather than the hash partitioning in [5]. The reason is the hash-based partitioning could bring in redundant message pairs that need to be transferred over networks. For example, the hash values of the rules $A_1 = A_2$ and $A_1 = A_3$ in Figure 2(a) are different, therefore the tuple $(C_2, 1, e1)$ will be transferred two times rather than one.

The simplified graph G_s is composed of two parts, the subgraph G_r from the reference-based conditions ψ_r, and G_k from the key-based ψ_k. We divide our graph partitioning method into two steps, while considering both the network communication and load balancing. (1) We divide vertexes in G_r into disjoint partitions using the most popular used vertex partitioning, and assign the vertexes to partitions without destructing their connections in G_r (lines 19-20). As shown in Figure 3(c), we could get into the situation where the vertexes A_1, A_2 and A_3 reside in one partition and A_4 is in another partition for our input graph G_r. Since there is a connection between A_3 and A_4, A_3 needs to be replicated. (2) For a vertex partitioning, it is very possible that large number of vertexes closed to each other will be located to a same partition, and consequently results in load balancing problems. To improve this, we assign the element in G_k to all the partitioned subgraphs to balance the workloads. As shown in Algorithm 1 lines 21-23, if a vertex in G_k has not been assigned in (1), then it will be assigned to a subgraph with the minimal edges (i.e., workloads for computing process instances). For example, the vertex A_5 as shown in Figure 3 will be assigned to the second partition rather than the first one.

Following the previous example with the rules $A_1 = A_2$

Algorithm 1 Implementation of RF-GraP

Input: input log $\mathcal{L}$ and parameters α, β
Output: interesting rules ψ_I and process instances PI

Main procedure:
1: ψ_f = `filtering`($\mathcal{L}$, α)
2: $\{\psi_I, PI\}$ = `verification`($\mathcal{L}$, β, ψ_f)
3: **return** $\{\psi_I, PI\}$

Procedure `filtering`($\mathcal{L}$, α):
4: Compute the statistic information $(A_i, val, freq)$ for sublogs on each node and redistribute the tuples based A_i
5: Construct $K = list(A_i, map(val, freq))$ at each node, by aggregating the values of A_i and val respectively
6: **for each** $K_i \in K$ **do** // key-based conditions
7: $dis\_rat(i) = S_i.map.size / \sum S_i.map.freq$
8: **if** $dis\_rat(i) \geq \alpha$ **then**
9: add a rule ψ_k $A_i = A_i$ in ψ_f
10: **end if**
11: **end for**
12: Collect K from each node to construct $R = list(A_i, set(val))$
13: **for each pair** $(R_i, R_j) \in R$ **do** // reference-based conditions
14: $sh\_rat(i,j) = |R_i.set \cap R_j.set| / max\{|R_i.set|, |R_j.set|\}$
15: **if** $sh\_rat(i,j) \geq \alpha$ **then**
16: add a rule ψ_r $A_i = A_j$ in ψ_f
17: **end if**
18: **end for**

Procedure `verification`($\mathcal{L}$, β, ψ_f):
19: Build correlation graph G_r based on all ψ_r in ψ_f
20: Partition graph G_r into disjoint parts, and copy a vertex $\mathcal{V}_x \in G_{ri}$ to G_{rj}, if there exist $x < y$, $\mathcal{V}_y \in G_{rj}$ and $\mathcal{V}_x \times \mathcal{V}_y \neq \emptyset$
21: **for each** $\psi \in \psi_k$, *if*$(\psi.A \notin G_{ri}, \forall i)$ **do**
22: Assign ψ to a subgraph $G_{rj}, \exists j$ to balance the edges
23: **end for**
24: **for** $e \in \mathcal{L}_i$, $A_j \in \mathcal{L}$ on each node i **do**
25: **if** $A_j \in G_{rk}$ **then**
26: Generate message pairs $(k, (e.A_j, j, e))$
27: **end if**
28: **end for**
29: Redistribute message pairs based on the value of k to all nodes
30: Build a C^{2}MB with received tuples on each node
31: **for** $\mathcal{E} \in G_{ri}$ on each node i **do**
32: Compute $\{\psi_I, PI\}$ with C^{2}MB, β and the rule from $\mathcal{E}$
33: **end for**

key	(val, tag, event)
A_1	(C_2,1,e_1)
A_2	(C_1,2,e_1)
A_3	(C_3,3,e_1)
...	...

(a) generated pairs for e_1

Val		EventSet1	E-Set2	E-Set3
C_1	$\rightarrow$	e_3	e_1, e_3	$\emptyset$
C_2	$\rightarrow$	e_1, e_2	e_2, e_4	e_1, e_3
C_3	$\rightarrow$	e_4	$\emptyset$	$\emptyset$
C_4	$\rightarrow$	$\emptyset$	$\emptyset$	e_2, e_4

(b) the proposed C^{2}MB data structure

Fig. 4: The form of the generated message pairs, and the data structure used for event correlations in our approach.

and $A_1 = A_3$: these two rules are assigned the same node within our approach, therefore the tuple $(C_2, 1, e1)$ will be transferred only one time rather than two then. This means that using a graph-based partitioning, we can efficiently reduce the redundant message paris to be transferred over networks. In fact, vertex partitioning of graphs is a well-studied problem in computer science, therefore we can leverage existing code to do the partitioning for us. In our first step, we use the most popular used METIS partitioner [13] for this purpose. We input the correlation graph G_r as an *undirected* graph to METIS, specify the desired number of partitions, and METIS outputs partitions of vertexes that are pairwise disjoint. Normally, we set the number of computing nodes as the input parameter to METIS. However, when G_r is very small, e.g., the number of edges is smaller than the underlying nodes, then unloaded cases will happen when using METIS. To remedy this problem, we simply partition the whole graph G_s, with a method similar as the second step of our partitioning scheme, to balance the workloads on each node.

3) Message Pair Generating: After the partitioning of the simplified graph G_s, we can then generate message pairs and transfer them to remote nodes according to the graph partitioning results. Different from [5], we generate message pairs based on *attributes* rather than *rules* in our approach (lines 24-28). Namely, an attribute is used as the key in each pair, and its responsible value is in the form of *(val, tag, event)*. Therefore, for a given event and attribute, only one message will be generated. An example of such processing over the event e_1 is demonstrated as Figure 4(a). After redistributing all the message pairs, we can then rebuild the responsible correlation rules at each node if required. For example, the pairs with the keys A_1 and A_2 received at a node can be used to build the CMB with the correlation rule $A_1 = A_2$.

4) Efficient Data Structure: As mentioned above, we can build CMBs based on the received message pairs and the assigned correlation rules on each node. However, such independent buffers could lead to additional cost on memory consumption due to redundancies. For example, the event set $\{e_1, e_2\}$ in Figure 1 appears in both the CMBs based on $A_1 = A_1$ and $A_1 = A_2$. It should be noted that when the number of events is large, such kind of redundancy would greatly impact the correlation performance or even break the analytics. To improve this problem, we propose a new data structure called *Comprehensive Correlation Message Buffer* (C^{2}MB) as shown as Figure 4(b). We store all the received message pairs in a single buffer on each node, and on this basis to compute all the required values in the event correlations (lines 30-33). For

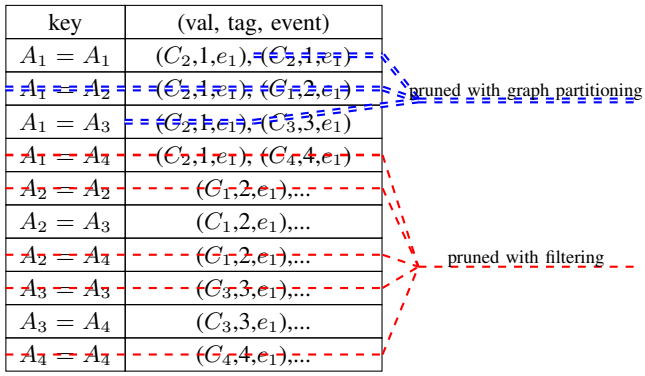


Fig. 5: The reduction process of generated message pairs for event e_1 compared with the state-of-the-art.

instance, when computing the $shared\_ratio(\psi)$ of $A_1 = A_2$, the value of $|distinct(A_1) \cap distinct(A_2)|$ will be number of rows, in each of which both the sets in the first and second columns are not empty. All these operations can be done in parallel at each node, and the whole correlation process will be terminated until all the nodes have finish their jobs.

C. Comparison with Current Approaches

Compared to current approaches [5], [6], in general, our design follows the filtering-and-verification principle rather than the conventional one based on generating-and-pruning. This way we are able to prune a lot potential correlation rules and reduce large number of generated message pairs, as not all the possible correlation rules will contribute to the final outputs. Moreover, since correlation rules have overlaps on log attributes, message pairs generated on that basis would bring in redundant cost on data transferring and computing. Instead of a hash partitioning in [5], we have used a more advanced graph partitioning to refine this. Namely, our approach focuses on using the strategies of **R**ule **F**iltering and **Gra**ph **P**artitioning to improve the correlation performance. That is also the reason why we call our approach as RF-GraP.

Our method is designed specifically for distributed platforms, therefore, we will be able to handle large event logs. In comparison, the standalone implementation [6] will meet great challenges in this aspect as we mentioned previously. Though the state-of-the-art [5] is also able to process large logs, as the differences mentioned above, our approach can highly reduce the network communication cost, which is always critical for the performance of a distributed application [14]. To highlight this advantage, we use an example as shown in Figure 5. It can be observed that, in our approach, many generated message pairs can be discarded before being transferred over networks. Moreover, as we have described, the proposed C^{2}MB data structure can also save a lot of memory.

Compared with the workflow in [5], our method has additional operations like filtering and graph partitioning, we believe that all these operations will be light weight. There are two reasons for this: (1) filtering can be done by a very simple statistic-based job; and (2) the METIS partitioner [13]

is able to partition large graph in a short time, and actually the correlation graph of an event log is typically small. As we will show in the next section, our approach is indeed very efficient, and outperforms the state-of-the-art [5].

IV. Experimental Evaluation

In this section, we present an experimental evaluation of our approach and compare it with the state-of-the-art [5].

A. Experimental Framework

As described in Section II, our approach is built based on the event correlation as defined in [5], [6]. Since the results reported in [5], [6] do not provide any information about the *quality* of discovered correlation conditions, we first conduct a qualitative evaluation looking at the acutal results. Namely, in the context of process mining [3], we check whether the cases of a log can be rediscovered by hiding its actually case identifiers.

In terms of detailed performance evaluation, since the work in [5] does not provide any source code information, in the interest of a fair comparison, we have implemented both the RF-GraP and the HVC approach using Scala over the Spark platform [7]. The source code we used for the performance comparison in this section are available at https://github.com/longcheng11/ECA

1) Platform: We evaluate our approach over a cluster. Each node we used has 4 CPU cores running at 2.80 GHz with 32GB of RAM and nodes are connected by Infiniband. The operating system is Linux kernel version 2.6.32-279 and the software stack consists of Spark version 2.0.0, Hadoop version 2.7.3, Scala version 2.11.4 and Java version 1.7.0_25. Moreover, we use Metis version 5.1.0 to partition a correlation graph as we have described.

2) Datasets: We measure the *quality* of discovered rules over a real log we collected from a commercial corporation. As shown as in Figure 6(a), the pre-possessed log (referred to as X) we used has 16 attributes. The 8-th attribute (i.e., `DCN`) is used as the case id in our real process mining analytics. Therefore, the rule $e_x.A_8 = e_y.A_8$ should be the ideal condition in our event correlations. As an example of the log, the third column of table in Figure 6(a) shows the detailed values of the first event in our test data.

We run our *performance* tests over different datasets based on an event log extracted from the SCM business service [6]. This log has been used as a benchmark dataset for performance evaluations in the work [5], [6] as well. The original event log has 19 attributes and 4,050 events. To evaluate our approach in large-scale cases, we use a same approach as [5] to increase both the number of events and attributes while maintaining the data behavior and distribution of the log. As a default, we increased the number of events and attributes by a factor of 250 and 2, respectively. Namely, there are around 1 million events and 38 attributes in the default dataset.

0	Event_ID	0
1	Start_End	start
2	ITE_ID	40842566
3	TAS_ID	3
4	TAS_NAME	OCRFlowValve
5	STACKNAME	STACK_9_201510300837
6	BATCHNAME	BATCH_4_15103003596
7	BATCHNAME_2	BATCH_4
8	DCN	151030808370
9	ITS_STATUS	2
10	STATUS	A
11	ITS_DTSTART	2015-11-01 21:15:09.040
12	IMAGECOUNT	12
13	PAGECOUNT	12
14	PROCESSDATE	2015-10-30 00:00:00
15	USE_LOGIN	MrAuto
16	KEYLOCATION	Cebu

(a) attribute names and example values

correlated	e_1	e_2	e_3	e_4
e_1	1	1	0	0
e_2	1	1	0	0
e_3	0	0	1	1
e_4	0	0	1	1

(b) matrix for the ideal condition

correlated	e_1	e_2	e_3	e_4
e_1	1	1	1	1
e_2	1	1	1	1
e_3	1	1	1	1
e_4	1	1	1	1

(c) matrix for the rule $A_1 = A_2$

Fig. 6: The attribute names of log X we used in our quality validation, and example correlation matrices.

3) Setup: We set the following system parameters for Spark: *spark_worker_memory* and *spark_executor_memory* are set to 30GB and *spark_worker_cores* is to 4. Recall that there are two application parameters: (1) α, namely the threshold for $distinct\_ratio(A_i)$ and $shared\_ratio(\psi)$; and (2) β, namely the threshold of $PI\_ratio(\psi)$. Because the power of our filtering operation will rely on the value the first parameter, to examine the performance difference with different α, we will vary its value in our tests, from 1% to 10%. In contrast, the value of β only impacts final output and not the correlation cost on computing and network communication. Therefore, we just fix its value in all performance tests. Like in [5], we set its value to 0.5. In all our experiments, the operations of input file reading and final result output are both on the HDFS system. We measure runtime as the elapsed time from job submission to the job being reported as finished. Because we want to focus on the runtime performance of each correlation implementation, we only record the number of the final outputs, rather than materializing them. As a default, we implement our tests using 9 nodes, composed one master node and 8 worker nodes (i.e., 32 cores).

B. Quality of Event Correlations

We use *accuracy* over *correlation matrices* to measure the quality of discovered rules[6]. An example of a correlation matrix is shown in Figure 6(b). There, for the log $\mathcal{L}_1$ in Table I, we can build a 4×4 matrix. If we know that, for an ideal condition (i.e., a known case id for a business process [3]), the events e_1 and e_2 are correlated, and e_3 and e_4 are correlated, then we can mark the correlated events with 1, otherwise 0, in the matrix. The accuracy of a discovered rule is the fraction of marked values in its correlation matrix that are correct, compared to the ideal one. For example, with the correlation rule $A_1 = A_2$ over $\mathcal{L}_1$, we discover the process instance $\langle e_1, e_2, e_3, e_4 \rangle$. Note that all the four events are correlated to each other, and we have its correlation matrix shown as Figure 6(c). By comparing it with the ideal one we can derive that its accuracy is 0.5. We calculate the accuracy for all discovered rules and then sort their values. In such scenarios, the higher a value is, the better the rule will be.

[6]We only conduct an initial evalution here, intead of detailed analysis on the real quality of process mining results.

condition	score
(8,8)	1.0
(6,6)	0.9995673
(2,2)	0.9995673

(a) $\alpha = 0.01$ and $\beta = 0.5$

condition	score
(8,8)	1.0
(6,6)	0.9995673
(2,2)	0.9995673

(b) $\alpha = 0.001$ and $\beta = 0.5$

condition	score
(8,8)	1.0
(6,6)	0.9995673
(2,2)	0.9995673

(c) $\alpha = 0.01$ and $\beta = 0.8$

condition	score
(8,8)	1.0
(6,6)	0.9995673
(2,2)	0.9995673

(d) $\alpha = 0.001$ and $\beta = 0.8$

Fig. 7: Correlation accuracy over the log X.

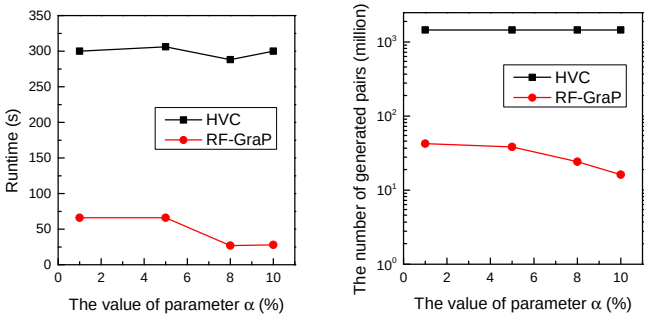
We choose the first 10000 events from the collected X log for the quality evaluation. The reason we did not choose more events, is that the correlation matrices would consume excessive amount of memory. For example, for 1 million events and each event id is an integer, a correlation matrix will consume 4TB memory (i.e., 4Byte$\times 10^6 \times 10^6$). We vary the values of α and β, and the results are shown in Figure 7. It can be seen that the results for the discovered rules and their scores are the same for all the cases, even with a very small $\alpha = 0.001$ and a big $\beta = 0.8$. Moreover, for each condition, the score of the rule `(8,8)` is 1, which is consistent with our expectation that the correlation over the attribute `DCN` is actually the ideal condition. Moreover, we see that accuracy of the rules `(6,6)` and `(2,2)` is very close to 1, which means that the 6-*th* and 2-*th* attribute of the log can be treated as case identifiers in the condition of missing the attribute `DCN`. Consider the hierarchy of the log, this is reasonable, since the `ITE_ID` and `BATCHNAME` are the upper level attributes of events. It is likely that events correlated on `DCN` will be correlated based on their values in `ITE_ID` and `BATCHNAME`. These initial results suggest that the correlation approach we adopted in this work is also applicable in real-life settings.

C. Performance Results

Next, we present our performance results including runtime, network communication as well as scalability as below.

1) Efficiency: We evaluate the efficiency of our approach by comparing its runtime with the HVC approach [5]. Moreover, to better compare the two algorithms, we also measure the *number of generated message pairs* in their implementations. As described previously, this metric represents the potential cost on network communication and computing in a correlation execution.

We execute each approach with varying the values of the application parameter α, from 1% up to 10%, over the default data. The results for the runtime and the number of generated message pairs (referred to as *gmp*) are presented in Figure 8. Figure 8(a) shows that our algorithm performs much faster then the HVC algorithm in all the cases, and is able to achieve a speedup of $4.5 - 10.7\times$. When varying the value of α, the runtime of the HVC method is generally the same. In our approach, it decreases when increasing the value of α. This is reasonable, as HVC always generates message pairs for all the possible correlation rules, i.e., the number of *gmp* is a fixed value, making its cost on network communication and



(a) runtime efficiency (b) # message pairs (log-scale)

Fig. 8: The efficiency of each algorithm over the default dataset, with varying the values α (using 32 cores).

α	1%			10%		
Scale	0.5	1	2	0.5	1	2
Speedup	5.1	4.5	> 6.6	8.0	10.7	> 44.7

(a) varying the scale factor of number of events

α	1%			10%		
Scale	0.5	1	2	0.5	1	2
Speedup	1.6	4.5	> 50.7	5.2	10.7	> 61.3

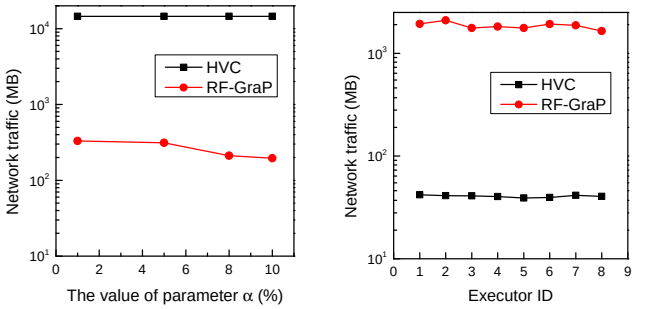
(b) varying the scale factor of number of attributes

Fig. 9: Speedup achieved by RF-GraP over HVC with varying the scale factors over the default data (with 32 cores).

computation constant. In comparison, our approach only needs to process message pairs over the filtered rules, and the power of the filtering is more effective when increasing the value of α. This can also be observed in Figure 8(b). There, the results demonstrate that the number of *gmp* of our approach is much less than Hash, and this number decreases obviously with the increase of α.

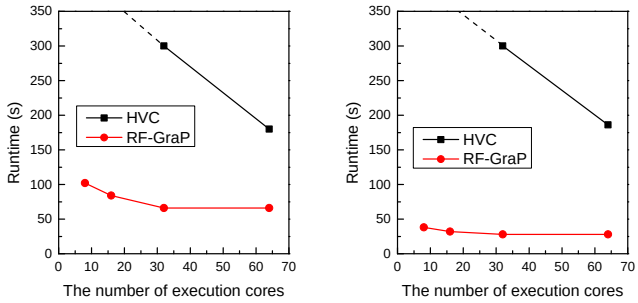
2) Cardinality Experiments: To see how the performance changes with increasing the number of events and attributes in a log, we have done the following two tests based on the default dataset: (1) We fix the attributes of the log, to 38, and vary the number of events from 0.5 million to 2 million; and (2) We fix the number of events to 1 millions, and vary the number of attributes from 19 to 76. We also vary the threshold α for each case. As the trends of the results are very similar to each other, we only report the conditions with $\alpha = 1\%$ and 10% here.

In our tests, several runs of the HVC approach aborted unexpectedly while raising the exception *job aborted due to stage failure* in Spark. Upon further analysis, we think the possible reason for this is that the number of generated message pairs is too large in HVC. This results in out-of-memory problems or the size of a partition of a RDD [7] exceeding its maximum value. For this condition, we just record the runtime of HVC from a job submission until the job is aborted, and mark the achieved speedup with a symbol "$>$" in our results. The results are presented in Figure 9(a) and



(a) general network traffic (b) retrieved data of each executor

Fig. 10: The network communication of each algorithm when using 8 workers (log-scale).



(a) with $\alpha = 1\%$ (b) with $\alpha = 10\%$

Fig. 11: Scalability of our algorithms by varying the number of executors.

Figure 9(b) respectively. It can be seen that our approach can always achieve a significant speedup over the HVC method, and the speedup is becoming more obviously with increasing either the number of events and attributes, highlighting its advantages on processing large event logs.

3) Network Communication: Performance regarding communication cost is evaluated by recording the metric *Shuffle Read*, as provided by Spark. It records the data in bytes read from remote executors (machines) but not the data read locally. This means that this metric indicates the data transferred around the underlying networks during the correlation implementations. The results by varying threshold α over the default dataset with 8 workers (32 cores) are shown in Figure 10(a). It can be seen that our approach transfers about $43 - 74$ times less data than HVC. Moreover, to analyze the load balancing properties of each algorithm, we also measure the *Shuffle Read* at each executor. As shown in Figure 10(b), the transferred data is generally evenly redistributed over all the executors, implying that there are no hot spot in both the algorithms.

4) Scalability: We test the scalability (scale-out) of our approach by varying number of slaves (workers) over the default dataset, from 8 cores (2 nodes) to 64 cores. The test results for our algorithm and the HVC approach are shown in Figure 11. There, for simplicity, we also only report the cases with $\alpha = 1\%$ and 10%. As the submitted jobs of HVC are aborted when using 2 and 4 worker nodes, we just add an artificial dashed line to demonstrate its potential scalability. We can see that the runtime of both the algorithms decreases with increasing the number of cores under different α. This means that both RF-GraP and HVC generally scale well with the number of workers. Moreover, we can see that the benefit of adding more workers (i.e., the scaled speedup) decreases for our approach, though the runtime becomes lower. We attribute this to the overhead caused by underlying platform and statistic-based operations in our approach. Upon closer inspection of the results, we can observe the achieved runtime speedups under small α is higher than that under large α. The reason is that the transferred data is very small for the case with a large α and this results in the network and computing workloads are comparably small for the underlying platform in our approach. In general, our algorithm always performs much faster than HVC, even when using a small number of cores. All of these indicate that our method will be more suitable for large event log processing than the HVC approach.

V. Related Work

Correlating events is a challenging problem in both the fields of process mining and service mining [15]. Up to now, various approaches have been proposed on identifying process-related events and discovering event correlations [6], [10], [16], [17], [18], [19], [20], [21]. For example, Pourmirza et al. [10] propose a linear programming based method for mining process models over a log that contains no case identifiers. Poggi et al. [16] introduce a methodology to classify and transform clicked URLs into events and consequently extract business models from web logs. Moreover, Georgakopoulos et al. [21] analyze how events in business processes could be grouped into instances of the execution of business processes, and Motahari-Nezhad et al. [6] investigate the problem of correlation condition discovery. We have shown that these approaches (e.g., [6]) are truly efficient in our quality experiments. However, most of these approaches focus on high-level methods for event correlations, and few of them have ever considered the detailed performance or scalability issues in the presence of big data. In comparison, our approach provides a much better scalable way to deal with huge event logs.

In fact, quantity of available event data from current information systems is increasing in an unprecedented pace, and this has posed new challenges for current mining techniques [22]. To efficient identify and discover event correlation conditions for the purpose of process instance discovery over large event logs, the state-of-the-art approach [5] uses MapReduce leveraging distributed environments. One of its main targets is to investigate how data and computations should be partitioned and distributed over underlying nodes, so that the correlation tasks can be efficiently computed in parallel. Although the approach in [5] has achieved obvious performance improvements over the standalone implementation [6], we have shown in our experimental results that our approach can perform much faster in various situations.

To support parallel and distributed applications, various parallel programming languages and paradigms have been

developed by the high performance computing community (e.g., MPI [23]) and large scale data-analytics community (e.g., MapReduce [12] and Spark [7]). It is obvious that implementing an application using different languages and systems mentioned above would lead to different application execution times. Regardless, the technique used for parallel execution is actually more important than the language/paradigm used for implementations [24]. For example, the generating-and-pruning based correlations [5] would always generate huge number of message pairs and lead to heavy network communication, in the presence of large event logs, irrespective of the underlying implementation systems. In contrast to this, our filtering-and-verification based approach is able to highly reduce the responsible cost, as it can prune large numbers of noninteresting rules before actually generating message pairs. It should be noticed again that network communication is critical for high-level applications in terms of performance improvements, since data transfers could account for more than 50% of job completion time in large data applications [14].

We have adopted a graph-based method for data partitioning to facilitate the parallel implementation of event correlations, which is motivated by studies on graph query answering in the data management domain. Usually, with an efficient graph partitioning, graph queries are able to be answered by subgraph isomorphism [25]. Different from this, we focus on reducing generated message pairs based on the correlation graph generated by the potentially interesting correlation conditions. In our prototype, we adopt the commonly used vertex partitioning and then manually balance the workloads on each node. In fact, we can use more advanced techniques, such as balanced graph partitioning [26], to further improve the robustness of our approach in different distributed environments.

VI. Conclusion

In this paper, we discussed the importance of event correlation and revealed possible performance issues of current approaches in the presence of large event logs. Based on this analysis, we have introduced a new approach, called RFGraP, which aims a much more efficient event correlation while leveraging state-of-the-art distributed systems. We have described the detailed design and implementation of our approach, and conducted an experimental evaluation which included a comparison with a competing approach using the Spark platform [7]. Our experimental results have shown that the proposed algorithm is highly efficient and can achieve significant speedups over the state-of-the-art, and with much less network communication.

Acknowledgments. This work is supported by the NWO DELIBIDA research program.

References

[1] S. J. Leemans, D. Fahland, and W. van der Aalst, "Scalable process discovery with guarantees," in *Proc. 16th International Conference on Enterprise, Business-Process and Information Systems Modeling*, 2015, pp. 85–101.

[2] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[3] W. van der Aalst, *Process Mining: Data Science in Action*. Springer, 2016.

[4] S. Hernández, J. Ezpeleta, S. J. v. Zelst, and W. van der Aalst, "Assessing process discovery scalability in data intensive environments," in *2015 IEEE/ACM International Symposium on Big Data Computing*, 2015, pp. 99–104.

[5] H. Reguieg, B. Benatallah, H. R. Motahari-Nezhad, and F. Toumani, "Event correlation analytics: Scaling process mining using MapReduce-aware event correlation discovery techniques," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 847–860, 2015.

[6] H. R. Motahari-Nezhad, R. Saint-Paul, F. Casati, and B. Benatallah, "Event correlation for process discovery from web service interaction logs," *The VLDB Journal*, vol. 20, no. 3, pp. 417–444, 2011.

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012, pp. 15–28.

[8] S. Sahar, "Interestingness via what is not interesting," in *SIGKDD*, 1999, pp. 332–336.

[9] J. L. Gross and J. Yellen, *Handbook of graph theory*. CRC press, 2004.

[10] S. Pourmirza, R. Dijkman, and P. Grefen, "Correlation mining: mining process orchestrations without case identifiers," in *ICSOC*, 2015, pp. 237–252.

[11] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and skew-resistant parallel joins in shared-nothing systems," in *CIKM*, 2014, pp. 1399–1408.

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[15] W. van der Aalst, "Service mining: Using process mining to discover, check, and improve service behavior," *IEEE Transactions on Services Computing*, vol. 6, no. 4, pp. 525–535, 2013.

[16] N. Poggi, V. Muthusamy, D. Carrera, and R. Khalaf, "Business process mining from e-commerce web logs," in *BPM*, 2013, pp. 65–80.

[17] R. Engel, W. Krathu, M. Zapletal, C. Pichler, R. J. C. Bose, W. van der Aalst, H. Werthner, and C. Huemer, "Analyzing inter-organizational business processes," *Information Systems and e-Business Management*, pp. 1–36, 2015.

[18] K. Musaraj, T. Yoshida, F. Daniel, M.-S. Hacid, F. Casati, and B. Benatallah, "Message correlation and web service protocol mining from inaccurate logs," in *ICWS*, 2010, pp. 259–266.

[19] S. Rozsnyai, A. Slominski, and G. T. Lakshmanan, "Discovering event correlation rules for semi-structured business processes," in *DEBS*, 2011, pp. 75–86.

[20] D. R. Ferreira and D. Gillblad, "Discovering process models from unlabelled event logs," in *BPM*, 2009, pp. 143–158.

[21] D. Georgakopoulos, M. Hornick, and A. Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure," *Distributed and parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.

[22] W. van der Aalst, "A general divide and conquer approach for process mining," in *Proc. 2013 Federated Conference on Computer Science and Information Systems*, 2013, pp. 1–10.

[23] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[24] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Fast compression of large semantic web data using X10," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2603–2617, Sept 2016.

[25] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[26] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.