Using Mapreduce to scale events correlation discovery for business processes mining

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

HP Laboratories
HPL-2012-170

Keyword(s):

business processes; Event Correlation; map reduce

Abstract:

The volume of data related to business process execution is increasing significantly in the enterprise. Many of data sources include events related to the execution of the same processes in various systems or applications. Event correlation is the task of analyzing a repository of event logs in order to find out the set of events that belong to the same business process execution instance. This is a key step in the discovery of business processes from event execution logs. Event correlation is a computationally-intensive task in the sense that it requires a deep analysis of very large and growing repositories of event logs, and exploration of various possible relationships among the events. In this paper, we present a scalable data analysis technique to support efficient event correlation for mining business processes. We propose a two-stages approach to compute correlation conditions and their entailed process instances from event logs using MapReduce framework. The experimental results show that the algorithm scales well to large datasets.

External Posting Date: August 7, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: August 7, 2012 [Fulltext]

Using Mapreduce to scale events correlation discovery for business processes mining

H. Reguieg[1], F. Toumani[1], H.R. Motahari-Nezhad[3] and B. Benatallah[2]

[1] LIMOS, CNRS, Blaise Pascal University,
Clermont-Ferrand, France
{reguieg,ftoumani}@isima.fr
[2] CSE, UNSW, Sydney, Australia
boualem@cse.unsw.edu.au
[3] HP Labs, Palo Alto, USA
hamid.motahari@hp.com

Abstract. The volume of data related to business process execution is increasing significantly in the enterprise. Many of data sources include events related to the execution of the same processes in various systems or applications. Event correlation is the task of analyzing a repository of event logs in order to find out the set of events that belong to the same business process execution instance. This is a key step in the discovery of business processes from event execution logs. Event correlation is a computationally-intensive task in the sense that it requires a deep analysis of very large and growing repositories of event logs, and exploration of various possible relationships among the events. In this paper, we present a scalable data analysis technique to support efficient event correlation for mining business processes. We propose a two-stages approach to compute correlation conditions and their entailed process instances from event logs using MapReduce framework. The experimental results show that the algorithm scales well to large datasets.

1 Introduction

Business process discovery, a kind of process mining technique, allows for extracting information from event logs, e.g. the audit trails of a workflow management system or the transaction logs of an enterprise application, to infer an explicit representation of intra- and/or inter-organizational business processes [14,9,13]. There are several attractive application areas for business process discovery in a wide variety of domains, e.g., healthcare, governments, banking, insurance, education, transport, etc. Process discovery allows organizations to gain insights into their operational processes, ensure compliance with standard processes, and improve processes in general.

Due to its importance, business process discovery has recently received wide attention from practitioners and researchers [14,8,9,13]. As a key-step in process discovery, *event correlation discovery* consists in analyzing event logs or interactions among processes entities in order to find relationships between events that belong to the same business process execution instance [2,8,9]. However, this is a computaionally-intensive task [9] as it involves the exploration of a huge space of possible relationships among events over very large and continuously growing event repositories. In particular, this is challenging for two main reasons: (i) Event correlation discovery is in essence a data-intensive task. It consists of various repetitive data-intensive computations(e.g., aggregation of events, intersection and join, computing transitive closures, etc) on a sheer large amount of data. (ii) *Big data is a fact of the modern world.* Modern infrastructures supporting large scale enterprise applications record more and more information about the history of business processes. According to a recent Gartner survey[4], the volume of digital business data to be stored is

[4] http://www.gartner.com/it/page.jsp?id=1460213.

growing at a rate of 40 percent to 60 percent each year. This trend is particularly supported by the unprecedented computation opportunities offered by the emerging service-oriented cloud computing infrastructures, which open new possibilities for process mining in cross-organizational and/or multi-tenants settings [12].

In this paper, we investigate the application of modern large scale data analysis techniques, and in particular MapReduce framework, to support efficient event correlation discovery in process mining activities. Mapreduce [3] has emerged recently as a promising approach 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 using other techniques e.g., general purpose parallel DBMS or specific parallel algorithms [10]. The arguments in favor of using MapReduce for event correlation discovery are: (i) Mapreduce provides a simple way to implement massive parallelism on a large number of commodity low-end servers (i.e., the *scaling out* approach), while freeing the programmers from the task of tackling the difficulty of traditional parallel programming. (ii) *"Component failures are endemic to very large clusters of distributed computers"* [5]. The event correlation discovery task can be very time consuming and therefore failure recovery solutions that require restarting the discovery process from scratch are indeed inadequate. Mapreduce handles failures at a fine-grained level by re-executing only the failed job on some other nodes in the network, and (iii) log files are usually heterogeneous in the sense that they comes in a variety of forms. The heterogeneity issue is more easily handled using Mapreduce since no predefined schema is imposed on the input data.

We propose a two-stages approach to compute correlation conditions and their entailed process instances from event logs. The first stage is devoted to the computation of simple correlation conditions (called atomic conditions) and their associated process instances. The second stage devoted to composite correlation conditions (conjunctive and disjunctive conditions) and associated process instances. Each stage is implemented as a map and reduce step. The main difficulties encountered when designing our approach are related to log partitioning and redistribution in order to generate efficient parallel computations. The main contributions of the paper are:

- We introduce an efficient method to partition an events log across map-reduce cluster nodes in order to balance the workload related to atomic conditions computation while reducing data transfers.
- We introduce an efficient solution to compute process instances corresponding to correlation conditions in a scalable parallel shared-nothing data processing platform. Our approach relies on a vertical partitioning of the space of candidate conditions in a way that each partition can be processed autonomously without need of synchronization.
- We develop one-pass algorithms to perform conditions discovery computations at the reducer nodes. Such algorithms 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 present experimental results that show that the algorithms scale well to large datasets. The experiments show that the overhead introduced by the mapreduce approach is small compared to the global gain in performance and scalability.

The 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 useful to understand the approach we propose to scale the discovery of correlation conditions and process instances. Section 3 presents a new algorithm to compute interesting process instances by

analyzing process events logs. Experimental results are presented at section 4. We discuss related work and conclude in section 5.

2 Preliminaries

2.1 Overview on event correlation discovery

Our work in this paper uses the event correlation discovery approach proposed in [9] as a basis. In [9], we analyzed web service interaction logs in order to identify correlation between events and then exploit such a knowledge to build views that represent resulting processes. The *event correlation discovery problem* is defined as the problem of finding relationships between events that belong to the same process execution instance. Events correlation discovery is a fundamental step in any process discovery method. Figure 1 depicts the global approach proposed in [8]. The approach consists of three main steps: (i) finding correlation between events (transform messages in the log into a set of process instances), (ii) using an algorithm for process mining to discover the process model for each set of the discovered process instances, and (iii) organizing the process views into a process map. Our aim is

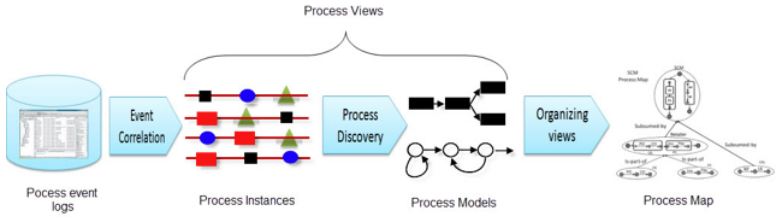

Fig. 1. A process mining approach

to investigate the implementation of this approach in a MapReduce Framework [4]. We consider as input a web services interaction log L. A log L can be viewed as a relation over a relational schema $\mathcal{L}(id, A_1, \ldots, A_n)$, where $U = \{A_1, \ldots, A_n\}$ is a set of attributes used in messages parameters and id a special attribute denoting message identifiers. Let $X \subseteq U$, 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. A message m is a tuple over the schema $\mathcal{L}$. Since typically a message $m \in L$ contains only a subset of attributes of U, therefore, m amy have several undefined attributes in L (i.e., having a *null* value). 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. Correlated messages are identified using *correlation conditions*. A correlation condition, denoted by $\psi(m_l.A_i, m_p.A_j)$, is a boolean predicate over the attributes A_i and A_j of respectively the two messages m_l and m_p. The condition $\psi(m_l.A_i, m_p.A_j)$ returns true if m_l and m_p are correlated through the attributes A_i and A_j and returns false otherwise. Conditions of the form $m_l.A_i = m_p.A_j$ are called *atomic conditions*. 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 [9], algorithms and heuristics are proposed to identify correlated event sets that lead to discovering potentially interesting process views. The main idea is to identify interesting correlation conditions based on what eliminating is not interesting. The following criteria and measures have been proposed to select relevant conditions:

– 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:

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

$$shared\_ratio(\psi) = \frac{|distinct(A_i) \cap distinct(A_j)|}{max|distinct(A_i), distinct(A_j)|}$$

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 [9].

– 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 section 3, a MapReduce-based algorithm to discover interesting correlation conditions and associated process instances from an events log.

2.2 Overview on the MapReduce framework

MapReduce is a new programming model to facilitate the development of scalable parallel computations on large server clusters [3]. A MapReduce framework provides a simple programming constructs to perform a computation over an input file f through two primitives: a *map* and a *reduce* functions. It 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). Map and reduce functions can be implemented using any general-purpose programming language. Typically, MapReduce programs are executed on clusters of several nodes and both their inputs and outputs are files in a distributed file system.

The execution workflow of a MapReduce step is as follows: the input file is split into several units (called a split) of, typically, 16 to 64 MegaBytes per unit. Each node hosting

a map task, called a mapper, reads the content of the corresponding input split from a distributed file system. The mapper then converts the content of its input split into a sequence of key-value pairs and calls the user-defined *Map* function for each $\langle k, v\rangle$ pair. The produced intermediate pairs $\langle k', v'\rangle$ are buffered in memory. Periodically, the buffered intermediate key-value pairs are stored in r local intermediate files, called segment files, where r is the number of reducer nodes. The partitioning of data into r regions is achieved by a partitioning function which ensures that pairs with the same key are always allocated to the same segment file. In each partition, the data items are sorted by keys. The sorted chunks are stored in (persistent) local storage. On the completion of a map task, the reducers (i.e., nodes executing the reduction function), will pull over their corresponding segments. When a reducer has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort is used. The reducer then merges the data to produce for each intermediate key k' a single pair $\langle k', list(v')\rangle$. Each reducer iterates over the sorted intermediate data and passes each pair $\langle k', list(v')\rangle$ to the user's *reduce* function. Finally, each reducer writes its final results to the distributed file system.

3 Event Correlation discovery using elastic MapReduce

In this section, we describe our approach for computing correlation conditions, and the process instances entailed by these conditions, from event logs. We first present an algorithm, named Split-Attributes, that executes a map and a reduce step to compute atomic correlation conditions and their associated process instances. Then, as a second map-reduce step, we present a MapReduce levelwise-based algorithm to compute conjunctive conditions and associated instances. Disjunctive correlation conditions discovery, not detailed here for space reasons, can be processed following a similar approach.

3.1 The Split-Attributes algorithm

Given an events log L, the aim of the Split-Attributes is to compute the interesting atomic correlation conditions as well as the process instances entailed by these conditions. One of the main issues to cope with this task, is to decide how data and computations should be partitioned in order to efficiently execute the operations entailed by this task. The main idea behind Split-Attributes is to use several instantiations of the reduce functions in order to process in parallel candidate conditions ψ_{A_i,A_j} associated to a couple of attributes A_i, A_j. Hence, a map phase will hash-partitions the content of the events log across the network based on keys in such a way that every pair of attributes A_i, A_j (with $i, j \in [1, n]$) of L will be allocated to a unique reducer (i.e., identified by a unique key value). To achieve this task, the map function will be in charge of projecting the log L on each pair A_i, A_j (with $i, j \in [1, n]$), of its attributes, and then assigning a same unique key to the projected record of A_i, A_j. A given reducer, in charge of a pair of attributes A_i, A_j, will execute the necessary computations to verify the relevance of the atomic condition ψ_{A_i,A_j} and eventually computes the associated process instances. We explain below in more details the map and reduce functions of Split-Attributes (c.f., algorithm 1).

The inputs of the Split-Attributes algorithm are made of a log L over a relational schema $\mathcal{L}(A_1, \ldots, A_n)$ and the user provided tresholds α and β. The map function takes as input a key-value pair (k, v) with $k = null$ and $v = m$ a message in L and for each pair of values $m.A_i$ and $m.A_j$ of m, with $i, j \in [1, n]$, will emit two intermediate key-value pairs (k'_{ij}, v'_i) and (k'_{ij}, v'_j) where:

- k'_{ij} is a unique key identifying a couple (i, j), and

Algorithm 1: Split-Attributes

```
   Input: a message log L, α, β
   Output: set of interesting atomic conditions
1  begin
2      Map (key: null , m: a message in L) ;
3      xkey := 1 ;
4      for i = 1 to k do
5          for j = i to k do
6              output(xkey, i-m.A_i-m.id) ;
7              output(xkey, j-m.A_j-m.id) ;
8              xkey := xkey + 1 ;
9      Reduce (k', V=list(v')) ;
10     B_i, B_j : buffers of type struct1;
11     B_ij : buffer of type struct2;
       /* Variables used to store the number of distinct values          */
12     δ_i, δ_j, δ_ij ;
13     (δ_i, δ_i) ← compute-distinct(B_i, B_j, V) ;
14     (B_ij, δ_ij) ← compute-intersect(B_i, B_j);
       /* Only shared_ratio is computed since if i = j its equal to the
          distinct_ratio                                                 */
15     shared_ratio(ψ) ← δ_ij / max|δ_i, δ_j| ;
16     if shared_ratio(ψ) > α then
17         (PI_ψ, | PI_ψ |) ← compute-instance(B_ij);
18         if ψ has_not_imbalancedPI(Based on PI_ψ) then
19             output(ψ, PI_ψ) ;
```

– each value v'_l = *l*-$m.A_l$-$m.id$, with $l \in \{i, j\}$, corresponds to the message value $m.A_l$ tagged with both the attribute position l, as a prefix, and the message identifier $m.id$, as a postfix. In the sequel, given an intermediate value v' = *l*-$m.A_l$-$m.id$, we use the notation $v'_l.tag$, $v'_l.val$, and $v'_l.id$, to respectively access to the attribute position l, the message value $m.A_l$ and the message identifier $m.id$ in v'.

Example 1. Table 1 shows an example of a log file L and the outputs corresponding the pair of attributes A_1, A_2 produced by two mappers that have processed respectively a split made of the first half (respectively, the second half) of the log L.

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 $A_j(L)$ are tagged and sent to the same reducer). Each reducer is then in charge of processing the received data related to the attributes A_i and A_j in order to verify the interestingness of the atomic condition ψ_{A_i,A_j} and eventually compute the associated process instances. To efficiently achieve this task, the reduce function uses the following data strutcture types:

– Type struct1:[val, idset], an array of records. Each record is made of a value *val* and a set of message id *idset*. Let T be a data structure of type *struct*1 and i be an integer. Then we use $T[i]$ to access to the record at row i of T and, respectively, $T[i].val$ and $T[i].idset$ to respectively access to the value and the set of ids of the record $T[i]$. We also use $|T|$ to refer to the size (i.e., number of records) of T.
– Type struct2:[val, idset1, idset2], similar to struct1 but with a record structure containing two sets of message identifiers idset1 and idset2.

Log L		
message-id	A_1	A_2
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			
key	tag	val	Id
k1	*1*	C_1	m_3
k1	*1*	C_1	m_4
k1	*1*	C_2	m_2
k1	*1*	C_2	m_5
k1	*1*	C_3	m_1
k1	*2*	C_1	m_4
k1	*2*	C_1	m_5
k1	*2*	C_2	m_2
k1	*2*	C_2	m_3
k1	*2*	C_4	m_1

Mapper 2 outputs			
key	tag	val	Id
k2	*1*	C_1	m_{10}
k2	*1*	C_3	m_6
k2	*1*	C_3	m_8
k2	*1*	C_4	m_7
k2	*1*	C_4	m_9
k2	*2*	C_2	m_{10}
k2	*2*	C_3	m_6
k2	*2*	C_3	m_7
k2	*2*	C_3	m_9
k2	*2*	C_4	m_8

Table 1. Example of a log and the outputs, w.r.t. to (A_1, A_2), of two mappers.

Once a reducer node has received all intermediate data (k'_{ij}, v'_i) and (k'_{ij}, v'_j) related to the attributes A_i and A_j, it merges the data to produce a single pair $\langle k'_{ij}, list(v')\rangle$. Note that, during the shuffle and sort phase, MapReduce sorts intermediate key-value pairs by the keys but not by the values (i.e., values in $list(v')$ may be arbitrarily ordered). 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 an order. Therefore, instead of implementing an additional secondary sorting within the reducer, we used the *value-to-key conversion* design pattern [7], 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, and then we let the execution framework handle the sorting. Therefore, a given reducer will process as input a pair $\langle k'_{ij}, list(v')\rangle$ where $list(v')$ is sorted by the intermediate values. Moreover, since the values are prefixed by the attribute position, then if $i < j$, the values of the attribute A_i will appear in $list(v')$ before the values of A_j. A reduce function then takes as input an intermediate key-list of values pair $\langle k'_{ij}, list(v')\rangle$ and then successively calls compute-distinct and compute-intersect functions. The compute-distinct function (c.f., algorithm 2) achieves two main computations:

Algorithm 2: compute-distinct

Input: B_i, B_j : buffers of type struct1; V : list of intermediate values
Output: (δ_i, δ_j) : number of distinct values of respectively A_i and A_j

```
1  begin
2      δ1, δ2 ← 0;
3      xval1, xval2 ← null;
4      for each v ∈ V do
5          if v.val ≠ xval_{v.tag} then
6              δ_{v.tag} ← δ_{v.tag} + 1;
7              B_{v.tag}[δ_{v.tag}].val ← v.val;
8              B_{v.tag}[δ_{v.tag}].idset ← B_{v.tag}[c_{v.tag}].idset ∪ {v.id};
9              xval_{v.tag} ← v.val;
10         else
11             B_{v.tag}[δ_{v.tag}].idset ← B_{v.tag}[δ_{v.tag}].idset ∪ {v.id} ;
12     return (δ_i, δ_j);
```

– it fills the buffers B_i and B_j as follows: each buffer B_l, with $l \in \{i, j\}$, will contain exactly one record for each distinct message value having a prefix tag equal to l with the val cell of this record set to the message value and its $idset$ cell set to the set of message id having this value (lines 4 to 11 of algorithm 2).
– when reading $list(v')$ to fill the buffers B_i and B_j, the function compute-distinct uses the variable δ_i, respectively δ_j, to count the number of distinct values in B_i, respectively B_j, with prefix tag i, respectively with prefix tag j (line 6 of algorithm 2).

Note that, since the lists $list(v')$ of intermediate values are sorted, the compute-distinct is executed in only one pass (i.e., it needs to scan $list(v')$ only once). This makes compute-distinct optimal w.r.t. disk I/O operations, which is particularly interesting in the situations where $list(v')$ is too large to fit entirely in main memory. Finally, it is worth noting that the buffers B_i and B_j computed by compute-distinct are sorted by the message values (i.e., the column val).

Example 2. Table 2 and 3 show respectively the contents of buffers B_1 and B_2 after the execution of compute-distinct function. The computed numbers of distinct values are: $\delta_1 = \delta_2 = 4$.

val	**idset**
C_1	{ m_3, m_4, m_{10}}
C_2	{ m_2, m_5}
C_3	{ m_1, m_6, m_8}
C_4	{ m_7, m_9}

Table 2. Buffer B_1.

val	**idset**
C_1	{ m_4, m_5 }
C_2	{ m_2, m_3, m_{10}}
C_3	{ m_6, m_7, m_9}
C_4	{ m_1, m_8}

Table 3. Buffer B_2.

The compute-intersect function takes as input two buffer B_1 and B_2, corresponding respectively to the buffers B_i and B_j filled by compute-distinct, and fills a buffer B_{12} with the intersection of B_1 and B_2. This buffer will contain the message values that belongs both to B_1 and B_2 and records for each such value the corresponding set of message identifiers from B_1 (the cells $idset_1$) and from B_2 (the cells $idset_2$). While reading the buffers B_1 and B_2, compute-intersect uses the variable δ_{12} to count the number of message values common to these two buffers. Finally, the function compute-distinct returns as a result the buffer B_{12} and δ_{12}. Also, note that compute-intersect assumes that the inputs B_1 and B_2 are sorted by the message values and hence it computes the intersection in only one pass (i.e., one scan of B_1 and one scan of B_2). Moreover, it produces a buffer B_{12} in which each set $idset_1$ (respectively, $idset_2$) is ordered by message identifiers.

Example 3. Using as input the buffers B_1 and B_2 of the previous example, the buffer B_{12} produced by the function compute-intersect is depicted at table 4 and the computed number of distinct values is: $\delta_{12} = 4$.

At this stage, the algorithm has all the needed information to compute the shared ratio of the considered condition ψ (line 15 of algorithm 1). Then, if the correlation condition ψ is considered as interesting (i.e., its shared ratio is above the threshold α), its entailed process instances are computed using the function compute-instance (line 17 of algorithm 1).

The compute-instances function 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 B_{12} associated with a couple of attributes A_i, A_j. We recall that a buffer B_{12} produced by the function compute-intersect contains in its column val the set of values v common to the attributes A_i and A_j , and for each such value v, records in the cell $idset_1$ (respectively, $idset_2$), the set of message identifiers $m.id$ such that $m.A_i = v$ (respectively, $m.A_j = v$). Then, the computation achieved by compute-instances is based on the observation that two

Algorithm 3: compute-intersect

Input: B_1, B_2
Output: B_{12}, δ_{12}

```
1  begin
2      δ12 ← 0 ;
3      tmp1, tmp2, tmp12 ← 1 ;
4      while (tmp1 ≤ |B1|) ∧ (tmp2 ≤ |B2|) do
5          if B1[tmp1].val = B2[tmp2].val then
6              δ12 ← δ12 + 1;
7              B12[tmp12].val ← B1[tmp1].val;
8              B12[tmp12].idset1 ← B1[tmp1].idset;
9              B12[tmp12].idset2 ← B2[tmp2].idset;
10             tmp12 ← tmp12 + 1 ;
11             tmp1 ← tmp1 + 1 ;
12             tmp2 ← tmp2 + 1 ;
13         else if B1[tmp1].val < B2[tmp2].val then
14             tmp1 ← tmp1 + 1
15         else
16             tmp2 ← tmp2 + 1
17     return (B12, δ12);
```

val	$\mathbf{idset_1}$	$\mathbf{idset_2}$
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	$\{ m_7, m_9\}$	$\{ m_1, m_8\}$

Table 4. Buffer B_{12}.

messages $m1$ and $m2$ that appear in B_{12} 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 B_{12}. 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 $m1 \in B_{12}[i].idset_1$ and $m2 \in B_{12}[i].idset_2$. Indeed, in this case we have by construction of B_{12} that $m1.A_i = m2.A_j = B_{12}[i].val$. Therefore, we can extend this observation to deduce that the elements of each $B_{12}[i].idset_1 \cup B_{12}[i].idset_2$, for $i \in [1, |B_{12}|]$, 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 B_{12} that have a non empty intersection. More formally: there exists $i, j \in [1, |B_{12}|]$ such that $m1 \in B_{12}[i].idset_1$, $m2 \in B_{12}[i].idset_2$ and $B_{12}[i].idset_1 \cap B_{12}[i].idset_2 \neq \emptyset$. Indeed, let m be in such an intersection than m is correlated with $m1$ (because both m and $m1$ belongs to $B_{12}[i].idset_1$) and m is correlated with $m2$ (because both m and $m2$ belongs to $B_{12}[j].idset_2$). Hence, using transitivity of the correlation relation we conclude that m, $m1$ and $m2$ belong the same process instance.

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

The function compute-instances can be better 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 B_{12} and the edges are constructed as follows: let $i, j \in [1, |B_{12}|]$, then there is an edge between $B_{12}[i].idset_1$ and $B_{12}[j].idset_2$ if: $i = j$ (condition (i) above) , or $B_{12}[i].idset_1 \cap B_{12}[j].idset_2 \neq \emptyset$ (condition (ii) above). The

Algorithm 4: compute-instances

```
   Input: B12
   Output: PIψ, |PIψ|
1  begin
2      I, tmpset, visited ← ∅ ;
3      j ← 0;
5      for tmp1 in B12 \ visited do
6          I ← {tmp1.idset1, tmp1.idset2} ;
7          j ← j + 1;
8          I ← I ∪ ccomp1(tmp1.idset1) ;
9          visited ← visited ∪ I ;
10         PIψ ← PIψ ∪ {I};
11     |PIψ| ← j;
12     return PIψ, |PIψ|;
```

Algorithm 5: ccomp1

```
   Input: tmp1.idset1
   Output: Itemp
1  begin
2      tempset, Itemp ← ∅;
3      for tmp2 ∈ B12 ∧ tmp2.ind ∉ visited do
4          if tmp2.ind = tmp1.ind then
5              visited ← tmp2.ind;
6              Itemp ← {tmp2.idset1, tmp2.idset2}∪ ccomp2(tmp2.idset2) ;
7          else
8              tmpset ← tmp1.idset1 ∩ tmp2.idset2;
9              if tmpset ≠ ∅ then
10                 Itemp ← Itemp ∪ {tmp2.idset1, tmp2.idset2}∪ ccomp2(tmp2.idset2);
11     return Itemp;
```

condition (iii) is achieved by the computation of the connected components of this graph. Each component corresponds to a discovered process instance.

Example 4. Figure 2 depicts the graph corresponding the buffer B_{12} of table 4. We can observe that there are two connected components of this graph. The associated discovered process instances are the following:

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

Note that compute-instances computes the instances on the fly (i.e., without materializing the whole graph). This is achieved by making use of to the recursive functions ccomp1 (c.f., algorithms 5) and ccomp2 (c.f., algorithms 6) that implements a depth-first search of connected components starting from the node (set) $B_{12}[1].idset_1$.

Finally, using user provided tresholds α and β, the non interesting instances are pruned (line 16 to 19 of algorithm Split-Attributes). Due to space limitation, the pruning procedure is ommitted and the reader may refer to [11] for the detailed description. It computes the PI_ratio of each discovered atomic condition ψ, i.e., the ratio of the number of instances entailed by ψ to the number of messages for which the attributes A_i and A_j of condition ψ are defined. The PI_ratio is compared with tresholds α and β and the conditions that do not satisfy the criteria are pruned.

Algorithm 6: ccomp2

Input: $tmp1.idset_2$
Output: $Itemp$

```
1  begin
2      tempset, Itemp ← ∅;
3      for tmp1 ∈ B12 ∧ tmp1.ind ∉ visited do
4          if tmp2.ind = tmp1.ind then
5              visited ← tmp1.ind;
6              Itemp ← {tmp1.idset1, tmp1.idset2}∪ ccomp1(tmp1.idset1) ;
7          else
8              tmpset ← tmp1.idset1 ∩ tmp2.idset2;
9              if tmpset ≠ ∅ then
10                 Itemp ← Itemp ∪ {tmp1.idset1, tmp1.idset2}∪ ccomp1(tmp1.idset1);
11     return Itemp;
```

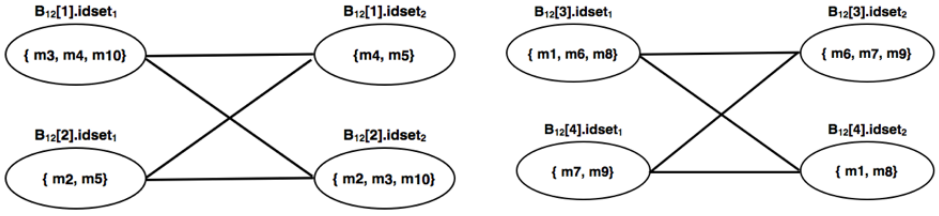

Fig. 2. Bipartite graph of B_{12} with two connected components.

3.2 Computing conjunctive 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 entailed by the condition $\psi_1 \wedge \psi_2$, noted ψ_{12}. More generally, given a set AC of atomic conditions, the goal is to identify the set of *minimal* atomic conditions that partition the log into interesting process instances. As explained in [9], such a task can be achieved using a levelwise-like approach where, roughly speaking, each level is determined by the length, in terms of number of conjuncts, of the considered conditions. Starting from atmic conditions (level 1), the discovery process consists in two main parts: (i) generating candidate conditions of level k from candidates of level $k-1$, 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 5. Consider as an example a set of atomic condition $CA = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$. The candidate conditions at each level are shown at table 5.

The high level structure of the algorithm that enables to compute conjunctive conditions, called computeConjunct, is given at algorithm 7. The algorithm takes as input a set AC of atomic conditions and recursively generate candidates of higher level (line 7) and then prune non interesting ones (line 9). A classical candidate generation procedure, e.g., see the *Apriori* algorithm [1], computes candidates at level k by a *self-join* on level k - 1. For example, if both $\psi_{1,2}$ and $\psi_{1,3}$ appear at level 2, than the candidate $\psi_{1,2,3}$ will be generated at level 3.

Level 1	$\psi_1, \psi_2, \psi_3, \psi_4, \psi_5$
Level 2	$\psi_{1,2}$, $\psi_{1,3}$, $\psi_{1,4}$, $\psi_{1,5}$, $\psi_{2,3}$, $\psi_{2,4}$, $\psi_{2,5}$, $\psi_{3,4}$, $\psi_{3,5}$, $\psi_{4,5}$
Level 3	$\psi_{1,2,3}$, $\psi_{1,2,4}$, $\psi_{1,2,5}$, $\psi_{1,3,4}$, $\psi_{1,3,5}$, $\psi_{1,4,5}$, $\psi_{2,3,4}$, $\psi_{2,3,5}$, $\psi_{2,4,5}$, $\psi_{3,4,5}$
Level 4	$\psi_{1,2,3,4}$, $\psi_{1,2,3,5}$, $\psi_{1,2,4,5}$, $\psi_{1,3,4,5}$, $\psi_{2,3,4,5}$
Level 6	$\psi_{1,2,3,4,5}$

Table 5. Candidates space.

Algorithm 7: computeConjunct

```
   Input: AC
   Output: CC
1  begin
2      CC ← ∅;
3      cand ← AC;
4      while cand ≠ ∅ do
5          CC ← CC ∪ cand;
7          cand ← genCandidates(cand);
9          cand ← pruneCandidates(cand);
10     return CC;
```

We use a different candidate generation procedure that, as explained below, is more suitable for parallelization. The procedure, called genCandidates, is presented at algorithm 9. It takes as input a set CC of conjunctive conditions, corresponding to the conditions generated at level k, and a set AC of atomic conditions. Then it generates candidate conditions for level $k+1$ by combining each condition $c \in CC$ with each condition $c_a \in CA$. Let us first introduce some notation. Atomic conditions are indexed by integer subscripts (e.g., we note ψ_1, $\psi_2, \ldots$) while conjunctive conditions are indexed by ordered sequence of integers (e.g., we note ψ_{12}, $\psi_{145} \ldots$). Given, a condition $c \in CC$, we use the function $last(c)$ to return the last subscript of c (e.g., $last(\psi_{145}) = 5$). Given $c \in CC$ and $ca \in$ CA, the procedure genCandidates produces a new candidate condition using the merge operator, noted $\otimes$, with consists in adding the subscript of c_a to the subscript sequence of c (e.g., $\psi_{145} \otimes \psi_7 = \psi_{1457}$).

Example 6. Using the procedure genCandidates in the recursive algorithm computeConjunct (c.f., algorithm 7) with as inputs $CC = \emptyset$ and $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$, and omitting the pruning procedure, enables to generate the whole space of candidates described at table 5.

Generated candidate conditions are pruned using algorithm 8. For each candidate conjunctive conditions $\psi_{12} = \psi_1 \wedge \psi_2$, the algorithm starts by computing the messages correlated by this conditions. This is achieved by the function conj-interesect (line 3 of the algorithm 8). The function conj-interesect (for algorithmic details of this algorithm, see [11]) relies on the following property: two messages m and m' are correlated by the conjunctive condition ψ_{12} iff they are correlated by both ψ_1 and ψ_2. This means that there exists two integers i and j such that either: (i) $m \in B_1[i].idset_1 \cap B_2[j].idset_1$ and $m' \in B_1[i].idset_2 \cap B_2[j].idset_2$, or (ii) $m \in B_1[i].idset_1 \cap B_2[j].idset_2$ and $m' \in B_1[i].idset_2 \cap B_2[j].idset_1$. The function conj-interesect takes as input two buffers B_1 and B_2 (associated respectively to conditions ψ_1 and ψ_2), and returns a buffer B_{12}, associated with the conjunctive condition ψ_{12}. The buffer B_{12} is obtained by computing the intersections corresponding to cases (i) and (ii) above such that for each record $r \in B_{12}$, the messages in $r.idset_1$ are correlated with the messages of $r.idset_2$ by the condition ψ_{12}. Candidate conjunctive conditions are then pruned using the *distinct-ratio* and *shared-ratio* criteria (line 5 to 9 of algorithm 8). In addition, conjunctive conditions that do not increase the number of instances w.r.t. the number of instances already discovered by their respective conjuncts are pruned (last disjunct in the condition of line 8).

Algorithm 8: pruneCandidates

Input: CC
Output: CC

begin
  foreach *condition* $\psi_{1,2} \in CC$ *generated from* ψ_1 *and* ψ_2 **do**
    $(B_{1,2}, \delta_{12}) \leftarrow$ conj-intersect(B_1, B_2) ;
    shared_ratio$(\psi_{1,2}) \leftarrow \dfrac{\delta_{12}}{max|\delta_1, \delta_2|}$;
    if *shared_ratio*$(\psi) > \alpha$ **then**
      $(PI_{\psi_{1,2}}, |PI_{\psi_{1,2}}|) \leftarrow$ compute-instance(B_{12}) ;
      PI_ratio$(\psi_{1,2}) \leftarrow \dfrac{|\ PI_{\psi_{1,2}}\ |}{nonNull(\psi_{1,2})}$;
      `/* Pruning the non interesting conditions */`
      if
      $(PI\_ratio(\psi_{1,2}) \geq \beta) \vee (PI\_ratio(\psi_{1,2}) \geq \alpha) \vee (|PI_{\psi_{1,2}}| \leq Max(|PI_{\psi_1}|, |PI_{\psi_2}|))$
      then
        $CC \leftarrow CC \setminus \{\psi_{1,2}\}$;
  return CC;

Algorithm 9: genCandidates

Input: CC, AC
Output: *cand*

begin
  $cand \leftarrow \emptyset$;
  for $c \in CC$ **do**
    $c_{last} \leftarrow last(cc)$;
    for $c_a \in |AC|$ **do**
      if $c_a > c_{last}$ **then**
        $c \leftarrow c \otimes c_a$;
        $cand \leftarrow cand + [c]$;
  return *cand*;

To cast the algorithm computeConjunct 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 a given partition can be handled by a unique reducer. This enables to avoid multiple mapreduce steps in order to compute conjunctive conditions. We proceed as follows to compute the partitions. Let AC be a set of n atomic conditions and let $P = \{\psi_1, \ldots, \psi_l\} \subseteq AC$ 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 partition with a condition ψ_i to indicate that this partition is made of candidates that contain the subscript i and with $\bar{\psi}_i$ to indicate that the partition is made of candidates that do not contain such a subscript. Consequently, given AC and P defined as previously, the set $\mathcal{P}$ of partitions of the space of candidates using P is obtained as follows: $\mathcal{P} = \{\psi_1, \bar{\psi}_1\} \times \ldots \times \{\psi_l, \bar{\psi}_l\}$. Each element $(\phi_1, \ldots, \phi_n) \in \mathcal{P}$, with $\phi_i \in \{\psi_i, \bar{\psi}_i\}$, for $i \in [1, l]$, forms a partition of the space of candidate conditions.

Example 7. Continuing with the previous example and assuming that $P = \{\psi_1, \psi_2\}$, we obtain four possible partitions:

- $(\psi_{\mathbf{1}}, \psi_{\mathbf{2}})$: contains the candidates with subscripts 1 and 2,
- $(\psi_{\mathbf{1}}, \bar{\psi}_{\mathbf{2}})$: contains the candidates with subscript 1 but without 2,
- $(\bar{\psi}_{\mathbf{1}}, \psi_{\mathbf{2}})$: contains the candidates with subscript 2 but without 1,
- $(\bar{\psi}_{\mathbf{1}}, \bar{\psi}_{\mathbf{2}})$: contains the candidates without the subscripts 1 and 2,

Table 6 shows the obtained partitions of the space of candidate conditions of table 5 partitioned using $P = \{\psi_1, \psi_2\}$. Each column of the table contains a partition of the space of candidates that can be processed separately by a given reducer.

	$(\psi_{\mathbf{1}}, \psi_{\mathbf{2}})$	$(\psi_{\mathbf{1}}, \bar{\psi}_{\mathbf{2}})$	$(\bar{\psi}_{\mathbf{1}}, \psi_{\mathbf{2}})$	$(\bar{\psi}_{\mathbf{1}}, \bar{\psi}_{\mathbf{2}})$
Level 1		ψ_1	ψ_2	ψ_3, ψ_4, ψ_5
Level 2	$\psi_{1,2}$	$\psi_{13}, \psi_{1,4}, \psi_{1,5}$	$\psi_{2,3}, \psi_{2,4}, \psi_{2,5}$	$\psi_{3,4}, \psi_{3,5}, \psi_{4,5}$
Level 3	$\psi_{1,2,3}, \psi_{1,2,4}, \psi_{1,2,5}$	$\psi_{1,3,4}, \psi_{1,3,5}, \psi_{1,4,5}$	$\psi_{2,3,4}, \psi_{2,3,5}, \psi_{2,4,5}$	$\psi_{3,4,5}$
Level 4	$\psi_{1,2,3,4}, \psi_{1,2,3,5}, \psi_{1,2,4,5}$	$\psi_{1,3,4,5}$	$\psi_{2,3,4,5}$	
Level 6	$\psi_{1,2,3,4,5}$			

Table 6. Partitioned candidates space.

Note that the obtained partitions are balanced (i.e., they have the same number of candidate conditions[5]) and they form a partition of the initial space of candidates. It is also worth noting that each partition can be treated separately from the others in order to compute the corresponding interesting conditions.

Algorithm 10: Levelwise-mapreduce

Input: CA, $P = \{\psi_1, \ldots, \psi_l\}$
Output: set of interesting atomic conditions

```
1 begin
2     Map (key: null , c_a: an atomic condition in CA) ;
3         P ← {ψ_1, ψ̄_1} × ... × {ψ_l, ψ̄_l} ;
4     for p ∈ P do
5         if c_A ∈ p then
6             output(p, c_a) ;
      /* (k', V = list(v')) is an intermediate key-list of values pair   */
7     Reduce (k', V) ;
8         CC ← computeConjunct(V);
9         output(CC) ;
```

The obtained mapreduce based levelwise algorithm is presented at algorithm 10. It takes as input a set CA of atomic conditions and a set P, subset of CA of partitioning conditions. Then, the map function generate the different partitions and send each conditions, and its associated data, to the corresponding reducer. The reduce function calls the computeConjunct procedure. We recall that this procedure, described by algorithm 7, consist mainly in the generation and pruning of the candidate conditions as described above.

[5] In table 6, if we also consider that $\emptyset \in (\bar{\psi}_1, \bar{\psi}_2)$, then every partition will have 8 candidates.

4 Preliminary experimental evaluation

The focus of our preliminary experiments was on the evaluation of the overhead due to the use of the MapReduce framework. Our experiments have been executed using Apache Hadoop running on 5 virtual machines, one master and four nodes, interconnected with a Gigabit Ethernet network and having CPU running at 2.493 GHz with 0.5 GB of RAM for the master and 1 GB for nodes and 60 to 80 Bytes/s for I/O bandwidth. We considered two categories of data: (ii) real world datasets, noted DS-0, corresponding to an interaction log of a the supply chain management scenario provided by the Web Service Interoperability organization (http://www.ws-i.org.), and (ii) randomy generated datasets DS-1 to DS-4 with different log sizes. The results of experiments are shown at table 7. Table 8 shows the

Dataset	**# msg**	**Map 1**	**Reduce 1**	**Map 2**	**Reduce 2**
DS-0	6050	3	13	1	8
DS-1	5000	1	3	1	30
DS-2	10 000	2	3	1	12
DS-3	50 000	2	5	4	44
DS-4	100 000	3	9	17	15

Table 7. Experimental results on real and synthetic datasets (average time in sec).

overhead due to the *shuffle* and *sort* operations of MapReduce which remains small w.r.t. the global gain in performance and scalability. Two main observations can be derived from these results: (i) they suggest that the extension of the proposed approach with additional map-reduce steps is potentially interesting since it will increase the level of parallelization, and hence improving scalability, without impacting too negatively the global performances, and (ii) the implementation of the reducer 2 can be improved drastically using compact data structures and specialized Apriori-like algorithms developed in the data minig field.

Dataset	**Shuffle 1**	**Sort 1**	**Shuffle 2**	**Sort 2**	**Total**
DS-0	8	1	6	1	16
DS-1	7	0	6	1	14
DS-2	6	1	6	1	14
DS-3	6	1	8	1	16
DS-4	7	1	7	1	16

Table 8. MapReduce overhead (in sec).

5 Conclusions and Future Work

Since its introduction in [3], MapReduce has been used in several application domains such as data management [7], data analysis [6,10], text processing [7], etc. To the best of our knowledge, this is the first work that uses MapReduce in processes discovery.

In this paper, we studied the problem of event correlation discovery using the MapReduce framework. We proposed a two-stages approach to compute correlation conditions and their corresponding process instances from service interactions event logs. We described efficient methods to partition an events log across cluster nodes in order to balance the workload related to atomic and conjunctive conditions computation while reducing data transfer. First experimental results show that the overhead introduced by the Mapreduce approach is small compared to gain in performance and scalability. This suggests future research work on investigation of additional implementation strategies, e.g., by increasing the number of MapReduce steps or using different partitioning techniques that may increase

the overhead while improving the scalability of the proposed approach. Moreover, our implementation can benefit from existing works devoted to the design of parallel levelwise algorithms in the data mining area [15]. While existing solutions, primarily designed for distributed memory parallel architectures with message-passing based inter-processor communications, cannot be used as it is in our context, we believe that many interesting ideas can be borrowed from them in order to improve the implementation of the conjunctive and disjunctive correlation conditions discovery in the MapReduce framework.

References

1. Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Chile*, pages 487–499, 1994.
2. Alistair Barros, Gero Decker, Marlon Dumas, and Franz Weber. Correlation patterns in service-oriented architectures. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 245–259, 2007.
3. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, 2004.
4. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
5. Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
6. Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM.
7. Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.
8. Hamid Motahari, Regis Saint-Paul, Boualem Benatallah, and Fabio Casati. Protocol discovery from web service interaction logs. In *IEEE ICDE 07*, April 2007.
9. Hamid R. Motahari Nezhad, Régis Saint-Paul, Fabio Casati, and Boualem Benatallah. Event correlation for process discovery from web service interaction logs. *VLDB J.*, 20(3):417–444, 2011.
10. Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178, 2009.
11. H. Reguieg, F. Toumani, H.R. Motahari-Nezhad, and B. Benatallah. Using mapreduce to scale events correlation discovery for business processes mining. *Hewlett Packard Laboratories Technical Report*, 2012.
12. W Van Der Aalst. Configurable services in the cloud: supporting variability while enabling cross-organizational process mining. *On the Move to Meaningful Internet Systems OTM 2010*, pages 8–25, 2010.
13. W M P Van Der Aalst. Process mining: Discovery, conformance and enhancement of business processes. *Media*, 136(2):352, 2011.
14. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, November 2003.
15. M.J. Zaki and C-T. Ho, editors. *Large-Scale Parallel Data Mining*. Springer, 2000.