Using MapReduce to scale event correlation discovery for process mining

Hicham Reguieg

N° d'ordre: D. U : 2438
E D S P I C: 644

UNIVERSITÉ Blaise Pascal Clermont-Ferrand II

ÉCOLE DOCTORALE SCIENCES POUR L'INGENIEUR DE CLERMONT FERRAND

T H È S E

Présentée par

M. Hicham REGUIEG

soutenue le

19 Fèvrier 2014

en vue de l'obtention du grade de

DOCTEUR de l'UNIVERSITÉ BLAISE PASCAL

Spécialité : Informatique

Arrêté du 07 août 2006

Titre :

Using MapReduce To Scale Event Correlation Discovery For Process Mining.

MEMBRES du JURY :

Mme Daniela Grigori	Professeur	Université Paris Dauphine	(***Rapporteur***)
M. Samir Tata	Professeur	TELECOM SudParis	(***Rapporteur***)
Mme Vargas-Solar Genoveva	CR	CNRS	
M. Laurent D'orazio	Maître de conférence	Université Blaise Pascal	
M. Farouk Toumani	Professeur	Université Blaise Pascal	(***Directeur de thèse***)

N° d'ordre: D. U : 2438
E D S P I C: 644

UNIVERSITÉ Blaise Pascal Clermont-Ferrand II

U.F.R. de Sciences

ÉCOLE DOCTORALE
SCIENENCES POUR L'INGENIEUR DE CLERMONT FERRAND

T H È S E

Présentée par

M. Hicham REGUIEG

soutenue le

19 Fèvrier 2014

en vue de l'obtention du grade de

DOCTEUR de l'UNIVERSITÉ BLAISE PASCAL

Spécialité : Base des données et Systems d'informations

Titre :

Using MapReduce To Scale Event Correlation Discovery For Process Mining.

MEMBRES du JURY :

Mme Daniela Grigori	Professeur	Université Paris Dauphine	(***Rapporteur***)
M. Samir Tata	Professeur	TELECOM SudParis	(***Rapporteur***)
Mme Vargas-Solar Genoveva	CR	CNRS	
M. Laurent D'orazio	Maître de conférence	Université Blaise Pascal	
M. Farouk Toumani	Professeur	Université Blaise Pascal	(***Directeur de thèse***)

Contents

Chapter 1

Introduction

Carried by the impressive development of new communication technologies, business processes (BPs) are becoming more and more central to the operation of modern information systems. On one hand, the success of most organisations hinges on the quality and efficiency of services provided to customers. On the other hand, organizations have to cope with the new economic model that requires the ability to adopt to changes of the market. A continuous business process improvement is essential for companies to keep up with the market needs. The nineties were the decade of the revolution of "Processes": implementation of information systems around process automation has begun to revolutionize the enterprises architectures. The focus of the process improvement was on automation [29, 101, 44], in other words human involvement was reduced by using *workflow management systems* (WFMS) and other middleware technologies. Moreover, using such technologies provides a good system integration and automated enactment of operational business processes. In addition, automated support provides the ability to observe and collect events related to process execution. As a result, it enables an opportunity to build a data source for analysis.

Recently, process analysis has received a wide attention for the purpose of process improvement. Hence, understanding information system behaviour and the processes and services they support become a priority in large-scale companies. This is illustrated by the increased number of process execution analysing tools and techniques available today [99, 31, 72]. The aim of such tools and techniques is to extract value from recorded data sources [98]. Nowadays, the wide-scale automation has led the business processes to be implemented over several (heterogeneous) systems. By consequent, the information related to the process execution may be scattered across multiple data sources, and in many cases, the knowledge about how this informations is related to each other and to the overall business process of the enterprise, is missing. In this case, the issue of identifying such a kind of relationship arise, in other words how to *correlate* informations (events related to process execution) in order to extract a knowledge about

the operational processes. The problem of *correlation discovery* can be defined as the problem of finding out rules (informations) that allow to group together recorded events that belong to the same process execution (*process instance*).

Due to its importance *correlation discovery* has received a wide attention from researchers and practitioners [27, 54, 55, 79, 87, 86, 74, 73, 20], from several application domains such as: process discovery, monitoring, analysis and browsing and querying. Correlation discovery consists of analysing a repository of event logs in order to find out the set of events that belong to the same business process execution instance. However, this is a computationally-intensive task [74] 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 task is challenging for two main reasons:

- *Correlation discovery* is in essence a computation-intensive task. It consists of various repetitive data-intensive computations (e.g., aggregation of events, intersection and join, computing transitive closures, and so on) on a sheer large amount of data.
- *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. Usually, the recorded data may not fit in one machine. According to a recent Gartner survey[1], the volume of digital business data to be stored is growing at a rate of 40 percent to 60 percent each year.

Applications with a large and unstructured data set usually employ parallel algorithms over a cluster of nodes in order to efficiently split the workload. When dealing with a very large amount of data, detecting relationships between events and identifying correlation rules become a challenging problem, even if a large computational cluster is available. Parallel data processing relies on data distribution and replication for efficient query execution. Partitioning event logs to identify correlation rules, used to determine relationships over events in order to isolate end-to-end process instances, is a challenging task due to the large size of datasets and the high number of candidate *correlation* rules (also called *correlation* conditions).

In this thesis, we investigate the application of modern large scale data analysis techniques, and in particular MapReduce [33] framework, to support efficient event correlation

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

discovery in process mining activities. MapReduce 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 [80]. The arguments in favor of using MapReduce for event correlation discovery are:

- 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,
- *"Component failures are endemic to very large clusters of distributed computers"* [50]. 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,
- Log files are usually heterogeneous in the sense that they come in a variety of forms. The heterogeneity issue is more easily handled using MapReduce since no predefined schema is imposed on the input data.

In this thesis, we rest on the *event correlation discovery* approach proposed by Motahari et al. in [74] to propose a two-stages approach for discovering correlation rules and their entailed process instances from event logs using MapReduce. The first stage is devoted to the computation of simple correlation rules (called atomic conditions) and their associated process instances. The second stage is devoted to composite correlation conditions (conjunctive and disjunctive conditions) and associated process instances. Composite correlation conditions are built by combining *atomic* conditions using $\{\wedge, \vee\}$ operators. For each stage, we provide a variety of algorithms. First, for discovering *atomic correlation conditions*, we propose *Sorted Values Centric* (SVC), *Hashed Values Centric* (HVC) and *Per-split Correlated Messages* (PSCM) algorithms. However, each algorithm has distinct properties. *SVC* relies on one MapReduce step, it sorts the intermediate values to efficiently compute the correlated message buffer denoted by $\mathcal{CMB}$. However,

it involves a large intermediate data size, because it uses the *value-to-key* pattern to impose an order on the values[2]. *HVC* relies on one MapReduce step, has a low intermediate data size, but requires several iterations to compute correlated messages. Finally, *PSCM* relies on two MapReduce steps. The first step computes the correlated message buffer in parallel, where the second step groups together the correlated messages and deduces the process instances. Secondly, *Single and Multi-pass Composite Condition Discovery* algorithms aim to build the *composite* candidate correlation condition space (lattice) and retains only significant ones. The two algorithms devoted to compute candidate composite correlation conditions, adopt different partitioning strategies, where (i) *single-pass* algorithm partitions the lattice vertically based on *partitioning conditions* and it performs the computation in one MapReduce job. (ii) *multi-pass* algorithm horizontally partitions the lattice by levels and processes each level in a MapReduce job. 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 thesis are:

- We introduce efficient methods to partition an events log across map-reduce cluster nodes in order to balance the workload related to atomic condition computations 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/multi-pass algorithms to perform condition 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 introduce two strategies to perform a MapReduce-based level-wise-like algorithms to explore the space of candidate composite conditions.

[2]The MapReduce framework sorts the records by key before they reach the reducers. The order that the values appear is not even stable from one run to the next, since they come from different map tasks, which may finish at different times from run to run.

  - *Single-pass strategy*, we use the notion of *partitioning conditions* for partition vertically the lattice of candidate composite conditions.
  - *Multi-pass strategy*, we partition the lattice horizontally and process each level in a distinct MapReduce job.

- We present experimental results that show the scale-up and speed-up of the algorithms with regard to variation of both data sizes and number of nodes. The experiments show that the overhead introduced by MapReduce is negligible compared to the global gain in performance and scalability.

The rest of the thesis is organized as follows: In Chapter 2 we describe the event correlation discovery problem statement and we discuss related works. Also, we describe the shared-nothing parallel data processing framework MapReduce and we provide a cost model to measure MapReduce Programs. Then, in Chapter 3 we detail the approach of event correlation discovery proposed by Motahari et al, this approach is considered as the basis of our work. Next, In Chapter 4 we present the algorithms dedicated to discover *atomic* correlation conditions. In Chapter 5 we present algorithms devoted to discover *composite* candidate correlation conditions. Finally we conclude in Chapter 6.

Chapter 2

Background

Contents

2.1 Introduction

In this chapter we describe the background of our work. First, in Section 2.2, we present an overview of *business process management*. Next, Section 2.3 focuses on one particular process mining task: *process discovery*. We present the process of extraction/collecting event logs from heterogeneous data sources, and we give an example on discovering business model using the so called α-algorithm [100]. In Section 2.4, we describe the problem of event correlation discovery. Then, in section 2.5 we discuss few existing application scenarios for event correlation discovery. Finally, in Section 2.6 we present the large-scale data processing framework MapReduce and we propose a cost model for estimating MapReduce-based algorithms.

2.2 Business Process Management

A business process is defined as a set of coordinated tasks and activities more/less related, collectively realizing a business objective. A business process can be entirely executed within a single organization or may span multiple organizations [44, 62, 101, 1]. A business process can combine automatic and manual activities.

Example 1 *Figure 2.1 represents a simple ordering business process using BPMN [76]. The process is made of two 'roles', namely a buyer and a sender, and several activities that work as follows:*

1. *The buyer sends an order request (message) with ordering information to the seller by executing the activity Place Order.*
2. *On receiving the message, the seller process starts. It extracts information about the buyer from the request message. Then, it checks the order by executing Check Order activity. After that, it sends the invoice.*
3. *Then buyer settle the received invoice.*
4. *When the seller receives the payment it ships the products.*
5. *Finally, the buyer receives the products, and the process is completed.*

Usually, a business process is associated with a data-flow, that defines how data evolves between process activities, and a control-flow, that defines the business logic of

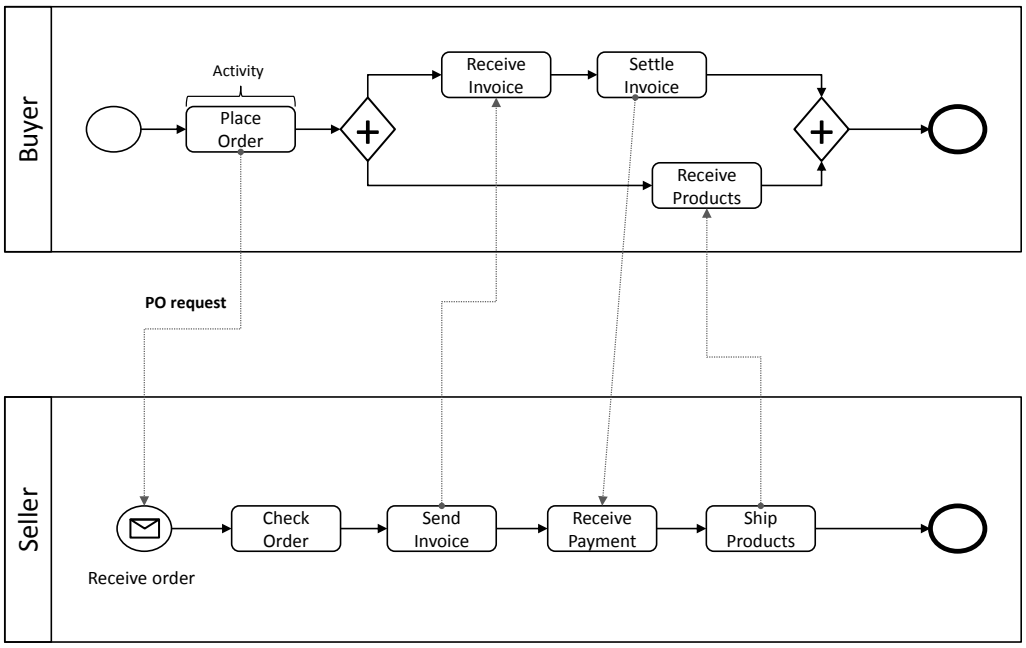

Figure 2.1: Simple ordering business process.

the process. The control-flow guides the execution of the activities, i.e, shows the order in which activities should be executed.

Business process management (BPM) covers concepts, methods, techniques and software to support activities such as design, administration, configuration, enactment and analysis of operational processes involving humans, organizations, applications, documents and other sources of information [104, 101]. Once the business process is explicitly defined as well as its activities and the constraints between them, it can be a subject to enactment, analysis and improvements.

Figure 2.2 shows a *business process* lifecycle. It consists of 4 phases: *design* phase, *configuration* phase, *enactment* phase and *evaluation* phase. These phases, organized in circular structure showing their logical relationship, are explained below.

- *Design phase*: This phase is interested by the design of the business process and its logic. It identifies the activities to be synchronised, and their logical order. Also, it

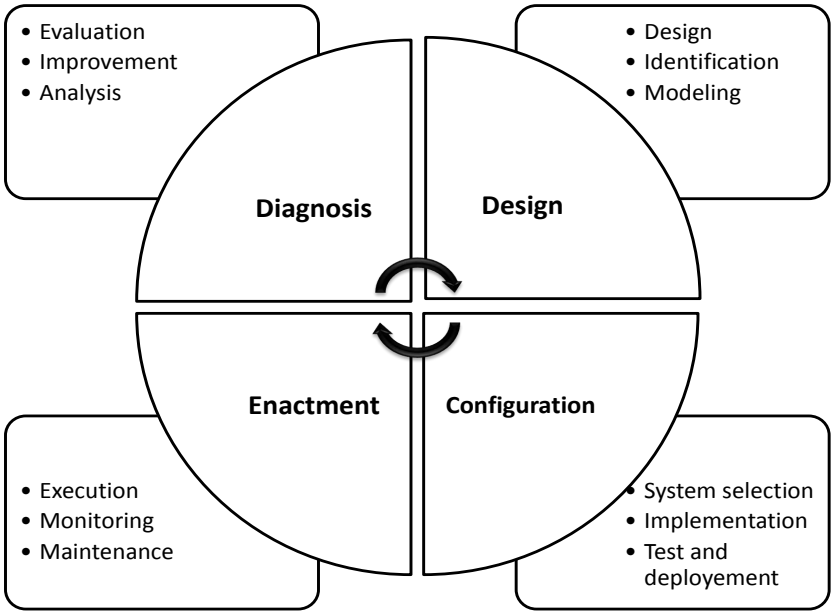

Figure 2.2: Business process lifecycle.

defines the roles to be assigned to these activities. A graphical representation of the business process is provided in order to facilitate communication between different stockholders. The business process management notation standard (BPMN) can be used to describe the processes in this phase [76].

- *Configuration phase*: Once the design phase is completed, the business process needs to be implemented. To do so, the abstract descriptions of the activities are implemented on a dedicated business process management system (BPMS) or workflow management system (WFMS), using software and procedures like filing a form, JAVA or SQL programmes.

- *Enactment phase*: During this phase, the (BPMS) controls the execution of the activities according to the flow previously established. It provides accurate informations on the status of an execution of the business process (process instance). Usually, the history of the execution of the process are recorded into log files.

- *Diagnosis phase:* This phase aims at analysing qualitative and quantitative effec-

tiveness of the business process model already deployed. Techniques and methods such as process mining are used to improve the process model and its implementation.

To support the business process lifecycle, *BPMS* were introduced as an extension of *workflow management systems*(*WFMS*). *BPMSs* focus more on the diagnosis phase of the *BPM* lifecycle, i.e., monitoring, tracking, analysing and prediction of business processes [46, 101]. A *BPMS* is defined as: *"a system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants, and where required, invoke the use of IT tools and applications"* [53, 105].

2.3 Process Mining

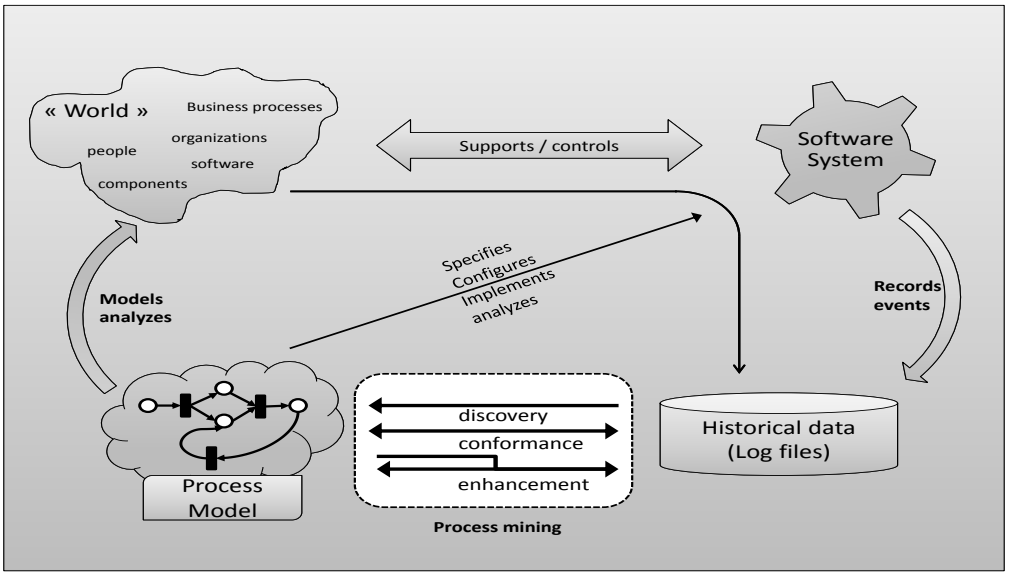

Figure 2.3: Positioning of the process mining in the business process lifecycle [8].

In modern enterprise, business processes are rarely supported by a single centralized workflow management system. Indeed, many existing processes span over multiple heterogeneous systems. Thereby, an accurate specification (formal description) of the process is not always available or may change to adapt the new enterprise requirements and services. Therefore, understanding, analysing and improving business processes become a

challenging task. As a business process management technique, process mining allows to deal with this issue. Process mining is a relatively new research discipline that combines machine learning and data mining on one hand and process modelling and analysis on the other hand [8]. The process mining aims to support the evolution of the re-engineering process [45].

The main goals of process mining technique is to extract knowledge from historical log files already recorded at the *enactment* phase by most of today's *WFMS*. This knowledge is used for various gaols such as process discovery, improving the quality of the process by detecting deviations in the process model. Figure 2.3 shows the position of process mining in the business process lifecycle. Note that any step of the three steps of process mining cannot be performed without a presence of a correct historical data. In the sequel we describe how data are gathered from multiple data sources to be analyzed from a process-oriented perspective i.e., for the purpose of process discovery.

2.3.1 Getting Data

In Figure 2.4, we present three layers (steps) that enables to understand the process of collecting process related event-data [86, 74].

Data sources Layer. The first layer represents event processing source systems such as *BPMS*, documents management system, *ERP* systems that capture and maintain information related to process executions. Such systems produce a wide range of *process information items* (e.g., a raw in a database, an event in a log file, a SOAP message exchanged between services, an email).

Data integration Layer. At this step, activities and resources associated to process execution are captured by tapping of message exchanges [39] (e.g. SOAP message) and recording read and write actions [8]. Such event-data come in different format (columns in relational databases, XML, CSV files, ...) and with various structures (relational schema, XSD, ...). Therefore, additional efforts are needed to collect, unify and store relevant data in a single data storage. ETL techniques [82, 91] are used to *extract* data from existing sources, and *transform* it to fit operational need and finally *load* it into a data warehouse or a relational database.

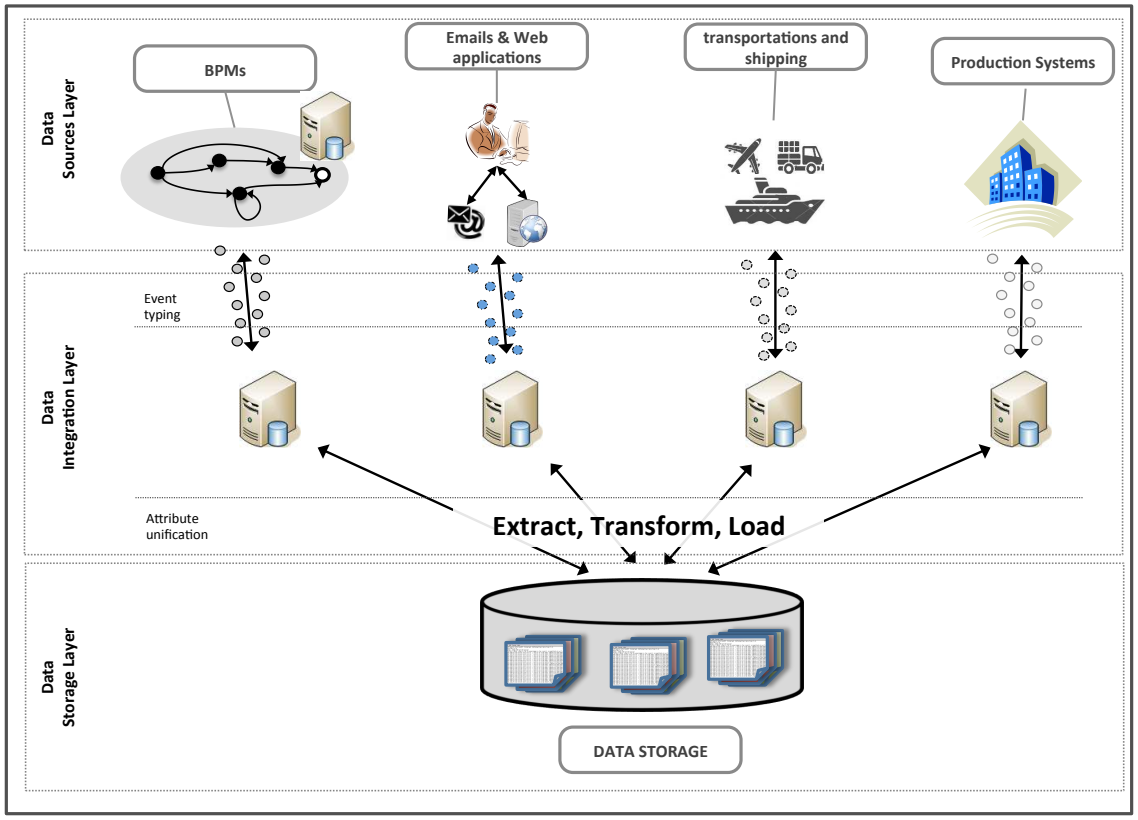

Figure 2.4: Getting data from heterogeneous data sources.

Data storage layer. Events extracted in previous layer are used to populate data storage (eg. data warehouse, relational database) for further analysis following the *store every thing, discover later* paradigm. Indeed, many analysis techniques depend on the analyser interest viewpoint. Consider for example data in a hospital. One may be interested by the discovery of patient flows. However, another one may also be interested in optimizing the workflow within the radiology department. Answering both questions requires the availability of informations related to both processes executions. Therefore, it is important to store as much data as possible.

In the following, we describe the process discovery and we give an example of process discovery algorithms.

Caseid	eventid	properties				
		timestamp	activity	resource	cost	...
1	35654423	30-12-2010:11.02	Register request	Pete	50	a
	35654424	31-12-2010:10.06	Examine thoroughly	Sue	400	b
	35654425	05-01-2011:15.12	Check ticket	Mike	100	d
	35654426	06-01-2011:11.18	Decide	Sara	200	e
	35654427	07-01-2011:14.24	Reject request	Pete	200	h
2	35654483	30-12-2010:11.32	Register request	Mike	50	a
	35654485	30-12-2010:12.12	Check ticket	Mike	100	d
	35654487	30-12-2010:14.16	Examine casually	Pete	400	c
	35654488	05-01-2011:11.22	Decide	Sara	200	e
	35654489	08-01-2011:12.05	Pay compensation	Ellen	200	g
3	35654521	30-12-2010:14.32	Register request	Pete	50	a
	35654522	30-12-2010:15.06	Examine casually	Mike	400	c
	35654524	30-12-2010:16.34	Check ticket	Ellen	100	d
	35654525	06-01-2011:09.18	Decide	Sara	200	e
	35654526	06-01-2011:12.18	Reinitiate request	Sara	200	f
	35654527	06-01-2011:13.06	Examine thoroughly	Sean	400	b
	35654530	08-01-2011:11.43	Check ticket	Pete	100	d
	35654531	09-01-2011:09.55	Decide	Sara	200	e
	35654533	15-01-2011:10.45	Pay compensation	Ellen	200	g

Table 2.1: A fragment of an event log: each line corresponds to an event.

2.3.2 Process Discovery

Business process discovery, also known as process mining, allows for extracting information from event logs, e.g. from 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 [99, 74, 98]. 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. The so called α-algorithm [100], is an example of a naive process mining algorithm able to handle such a task. This

algorithm is based on the following assumption:

- *Safety:* Any event presented in the log should refer to both a process execution (case) and an activity. Event within a process execution are ordered. Generally events are ordered by timestamps.

- *Completeness:* The process instances should cover all the possible executions of the process, i.e, each activity in the initial process model should appear at least one time on an event in the log.

This is the minimum requirement for any process mining algorithm to transform the information presented in the log into a process model (see Table 2.1).

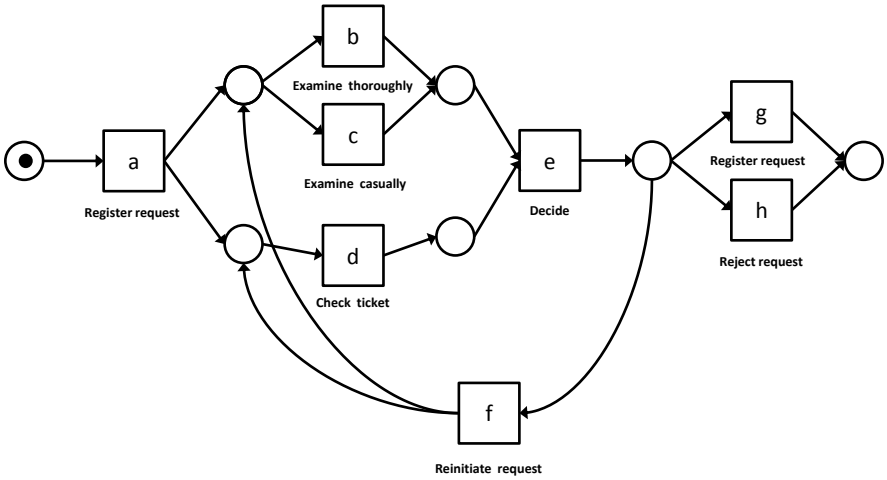

Figure 2.5: Process model discovered by $\alpha-$algorithm based on the process instances presented in the log depicted in Table 2.1.

Example 2 *Taking the event log presented in Table 2.1, the $\alpha-$ algorithm will generate the process model described in Petri-nets [35] as shown in Figure 2.5 (cf, [100, 98]) .*

2.3.3 Correlation Discovery, a Key Step For Process Discovery

Due to its importance, business process discovery has recently received a wide attention from practitioners and researchers [99, 72, 74, 98]. As a key-step in process discovery, *event correlation discovery* consists in analysing event logs or interactions among processes entities in order to find out relationships between events that belong to the same

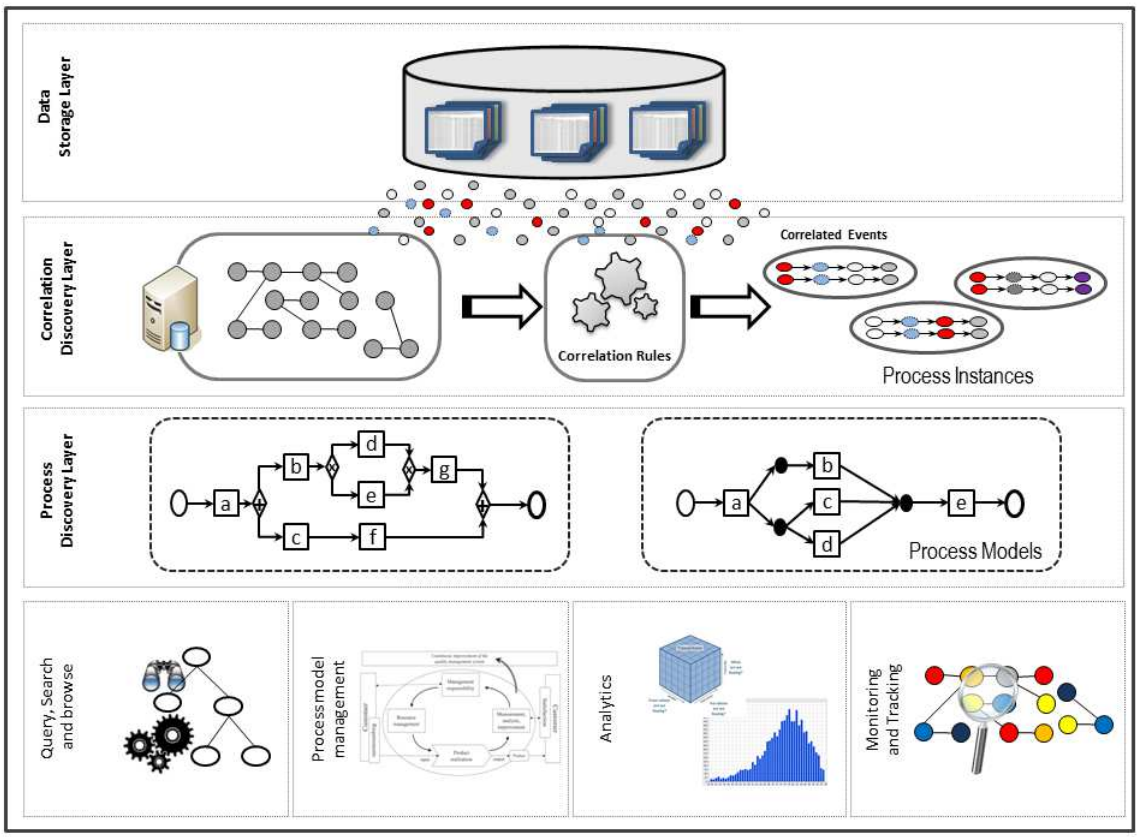

Figure 2.6: Event Correlation, Process discovery and its fields of application.

business process execution instance [21, 72, 74]. *Correlation discovery* involves the exploration of a huge space of possible relationships among events over very large and continuously growing event repositories.

Figure 2.6 depicts where the event correlation discovery step is incorporated with respect to an end-to-end system devoted to different business process management applications such as process discovery, analysis, monitoring and querying. In general, such a system consists of the following parts (represented as layers):

Correlation Discovery Layer: as defined previously, a correlation discovery algorithm (i) takes events presented in the storage system as input, (ii) deduces correlation between events by computing statistics on attribute combination (correlation conditions), (iii) groups together events correlated by the already *correlation conditions* discovered in the previous step to form process instances (e.g, a purchase order process).

Process Discovery Layer: historical traces of process instances discovered in the Correlation Discovery step are provided as input for mining algorithms (e.g the $\alpha-$algorithms). Process discovery algorithms require process instances executions from which they derive the process model.

Application fields Layer: this layer represents the various fields of application of correlated events. Events correlated during execution-time might be used in monitoring applications, or earlier alert-system to detect exceptional situations. Another application is querying and browsing the historical traces, a correlation rule may induce graphs of relationships that can be used to speed up querying and browsing events. Correlations are particularly useful for features that require interaction, analysis and exploration of events.

In this thesis we focus on the problem of identifying event-attributes that correlate events presented in the log files and, by consequent, lead to isolate process execution traces.

2.4 Event Correlation Discovery Problem

The First challenging step to achieve a process mining and/or process analysis approaches involves correlation of event generated by heterogeneous and distributed systems. In other words, identifying the set of events that belong to the same process or service execution (also called *case*).

In large-scale modern enterprise, event data are widely scattered over several tables or even a set of heterogeneous systems. Therefore, identifiers that relates event to process instance or to each other become extremely hard to track [86, 38, 42]. Answering questions such: *how events and their instances could be grouped ?. how to relate a response to the original request, in case of message exchange ?* becomes harder compared to a centralized business process, where all process are implemented using a single central *WFMS* [46].

In this thesis, we consider that all processes related data are stored in a centralized storage such as data warehouse, relational database. We call such document as event logs. In the sequel we introduce a formal definition of an event log document.

2.4.1 Event logs

An event log can be extracted from data warehouse (as seen in section 2.3.1), captured from web-service interaction or generated by *WFMS* or *BPMS* during process execution *enactment phase*. Various ways are used to log [39] and capture [8] messages exchanged between interacting services.

In the case of services based business processes, services exchange messages to achieve an objective, e.g, register a client request, booking a hotel and/or a flight, on-line payment, sending invoice, archiving the request. The order in which these messages appear form a *conversation* that achieve a single business goal. Taking an example of booking a hotel and flight service, several, *conversation* may be running, at any given time, corresponding to multiple customers interacting with a given service. In the spirit of [74, 85], we define in this thesis web service interaction log as follows:

Process message log. A process message log L, can be viewed as a relation over a relational schema $\mathcal{L}$ (id, A_1, A_2, ..., A_n), where U = $\{A_1, A_2, \ldots, A_n\}$ is a set of attributes used in messages parameters and *id* is a special attribute denoting message identifier. 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[1]. For a message $m \in L$, we denote by $m.A_i$ the value of the attribute A_i in the message m and by $m.id$ the message *id*.

In the case of web services interactions, messages are structured (XML) documents (of different types, and therefore with different schema) organized in sections. A preprocessing ETL-like is required to extract items from the XML documents and load them as event tuples in a relation L over the schema $\mathcal{L}$. The set of attributes $A_1 \times A_2 \times \ldots \times A_n$ represent different message attributes that belong to the XML document. Since typically a message $m \in L$ contains only a subset of attributes of U, therefore, m may have several undefined attributes in L (i.e., *null* values). In addition, some of these attributes are supposed to determine if two given messages belong to the same conversation. This attributes are called *correlator attributes*, and the functions defined over them as *correlation conditions* or *correlation rules*.

[1] We use *message* and *event* interchangeably.

2.4.2 Correlation Condition

Correlated messages are identified using a *correlation conditions* (also called *rules*). A correlation condition (*correlation rule*) is defined below.

Correlation condition. A *correlation condition*, denoted by $\psi(m_l.A_i, m_p.A_j)$, is a boolean predicate over attributes A_i and A_j of respectively the two messages m_l and m_p. 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 return false otherwise.

The condition form depends on the specific domain in which this condition is defined. For example, a condition of the form $m_l.A_i = m_p.A_j$ specifies the equality relationship between attributes A_i and A_j in (m_l, m_p). A condition having this form is an *atomic condition* or *atomic rule*. A conjunctive (respectively, *disjunctive*) condition consists of conjunction (respectively, disjunction) of atomic conditions.

A correlation condition groups messages in the service interaction logs L into a collection of conversations[2] $c_1, c_2, \ldots,$. Each c_i is a sequence of messages (events). Henceforth we use the term process instance to express a conversation. A process instance is defined as follows:

Process Instance. *A process instance 'pi' (instance for short) is a sequence of messages $\langle m_1, m_2, \ldots, m_k \rangle$ corresponding to a subset of messages of the log L. For a given message $m_x \in pi$, it exist at least one message $m_y \in pi$ and $x \neq y$. where m_x is directly correlated with m_y, i.e, $\psi(m_x, m_y)$ holds* [73, 74].

2.5 Related Works

In this section, we present some existing works used *correlation discovery* from either relational data or process event log files for various objectives such as, speed up queries processing by providing an optimal query plan, discovering hard and/or soft functional dependencies, discovering events related to a process execution instance for process discovery purpose, ..., etc. Despite the fact that each of the presented works has a different purpose, they share some steps as generating candidates and pruning non-relevant

[2] *conversation, instance and trace* all these terms have the same meaning which is a complete process execution.

candidate using heuristics and measures. At the end of the section, we discuss these approaches.

2.5.1 BHUNT [27]

In [27], the authors present a data-driven technique called BHUNT that uses a "Bump Hunting" techniques for automatically discovering fuzzy (soft) hidden relationships between pairs of numerical attributes in relational databases, and incorporates this knowledge (the relationship between attributes) into an optimizer in the form of algebraic constraints. Such constraints can be exploited in various ways as data mining and for improving query processing performance.

OrderID	Shipdate
A1	1990-01-01
A2	1990-03-05
A3	1990-09-26
A4	1991-06-13
A5	1992-05-30
A6	1993-01-11
A7	1993-02-04

orders

OrderID	deleverydate
A1	1990-01-03
A2	1990-03-07
A3	1990-10-10
A4	1991-06-15
A5	1992-06-30
A6	1993-02-13
A7	1993-02-19

deleveries

Figure 2.7: Two tables in sales database.

First, BHUNT detects the set of candidates column value pairs that might satisfy an algebraic constraint. Then, pruning heuristics obtained by exploiting the system catalogue and data samples are used to eliminate useless candidates. To specify the relationship between column value pairs the authors define the *algebraic constraint* as 5-tuple

$$AC = (a_1, a_2, P, \oplus, I)$$

where a_1, a_2 are two numerical attributes satisfying $a_1 \oplus a_2 \in I$, and $\oplus$ is an algebraic operator such as $\{+, -, \times, \div\}$, I represents a subset of the real numbers, P is the pairing rule or the predicate. As example, consider a sales database containing two relational tables *orders* and *deliveries* as shown in Figure 2.7. An example of an algebraic constraint, that BHUNT can identify and may not be revealed in a casual inspection, is specified by taking a_1 as $deliveries.deliverydate$, a_2 as $orders.shipdate$, $\oplus$ is the subtraction operator and P is the predicate

$$orders.OrderID = deliveries.OrderID$$

and

$$I = \{2,\ 3,\ 4\} \cup \{14,\ 15\} \cup \{30,\ 31\}$$

This latter represents the delivery time of three different shipping methods. It can be obtained by applying statistical histogramming technique to the data in the two tables, i.e, by computing the subtraction between the *deliverydate* and *shipdate* of each order and plot a histogram of the resulting data points. In this example, the data may qualify the following predicate.

(deliverydate BETWEEN shipdate + 2 days AND shipdate + 4 days)
OR (deliverydate BETWEEN shipdate + 14 days AND shipdate + 15 days)
OR (deliverydate BETWEEN shipdate + 30 days AND shipdate + 31 days)

The three clauses in the predicate represents the three "bumps" in the histogram 2.8.

Figure 2.8: Histogram of shipping delays

BHUNT finds and exploits hidden fuzzy algebraic constraints. It proceeds as follow:

- Generating candidates $C = (a_1, a_2, P, \oplus)$. This is achieved by searching for the key columns and then finding columns related to the key columns via inclusion dependency.
- For each generated candidate, it builds the algebraic constraint (i.e., construct the intervals $I_1,\ I_2,\ \ldots,\ I_k$) by employing statistical histograming, segmentation, or clustering techniques to a sample of data values. Since most of the constraints are *fuzzy*, some "exception" records may not satisfy the constraints.

- Identifiying the most effective set of constraincts, and create "exception tables" that holds exception records.
- Finally, the query plan is modified to incorporta the constraints. The RDBMS optimizer uses the constraints to speed up the query processing by finding new more effective access paths. The results are combined with results of executing the original query on the (small) exception tables.

In addition, BHUNT uses some heuristics to prune non-interesting candidate generated in step one, e.g., P is of the form R.a=S.b, and the number of rows in either R or S does not satisfy the following measure:

$$\frac{\#rows(a)}{\#distinctValues(a)} \leq 1 - \varepsilon$$

where ε a user pre-specified parameter.

2.5.2 CORDS [54, 55]

In [54], the authors introduce CORDS (CORrelation Detection via Sampling), a data-driven technique for automatically discovering correlations and soft functional dependencies between database columns. It provides a dependency graph to improve the performance of query optimizer. This tool is built upon BHUNT [27], previously presented. CORDS enumerates candidate column pairs searching for interesting and useful correlations, and pruning unpromising candidate using a set of heuristics. CORDS applies a chi-squared analysis to a sample of column values in order to identify correlation between attributes (categorical and numerical) and an analysis of the number of distinct values to detect functional dependencies. A correlation rule in the context in this work is relation ship between two columns such as for instance a join between two tables over two attributes.

CORDS exploits column pairs to identify functional dependencies and statistical correlations. It proceeds as follows:

- Generating candidate of the form $C = (a_1, a_2, P)$, where a_1, a_2 are attributes and P is the pairing rule that specifies how a_1 values get paired with which a_2 values to form correlated values.
  - First, it generates all candidates having a trivial pairing rule (when the columns lie on the same table).

  - Then, it searches for non-trivial pairing rule (when the columns are in separate tables.).

- Pruning unpromising candidates using a set of heuristics to reduce the search space.
- Finally, detecting correlations by applying a chi-squared analysis on data samples.

2.5.3 DePauw et al. [79]

The problem of discovering conversation in web services is raised in [79]. DePauw et al. proposed an approach to discover the correlation between message pairs (e.g., *Purchase Order* and *Shipping* message pair) from the log of service interactions. Their approach is based on identifying the conversation identifiers within exchanged messages. They used the term *semantic correlation* to describe how theses identifiers correlate messages across different activities execution.

Starting from a set of all exchanged messages, DePauw et al. propose the following steps to discover conversation identifiers:

- **From XML to value tables:** First, messages are grouped by their full message name and for each group a schema is derived based on the content of the messages. A value table is created for each schema, where each row represent one message. The term *path* is used to refer to the location of an element or attribute.
- **Finding candidate correlation identifiers:** At this step, *path* pairs used as correlation identifiers are determined based on the following criteria. (i) The first path should have unique values and *(ii)* the second path should have either unique values or a large number of distinct values. To catch this criteria, the authors define the indexability α_p of a path as

$$\alpha_p = \frac{Card_p}{Pop_p}$$

  where $Card_p$ represents the number of distinct values in message type p and Pop_p is the number of data element presented in p. Next, **highly indexable paths**, these are paths with an index higher then $\alpha_p > 95\%$, and **Mappable paths**, these are paths having an interesting number of matching values with highly indexable paths, are identified.

- **Finding correlation between correlation identifiers:** At this step, correlation identifiers identified in the previous step are classified as highly indexable and mappable paths. Then, pairs from different schemas are tested to find those produce an important match between their values. Hence, A pair of paths is considered as interesting semantic correlation if a significant overlap exists between the values set of the first and the second paths. Next, a causal relationship based on timestamps is assigned to each two matched values to determine the origin path and the destination paths.

- **Finding correlation between schemas:** finally, two schemas are correlated if at least a path from one schema is correlated with a paths from the second schema.

As conclusion the proposed approach [79] can reveal correlation between pairs of messages. However, it does not provide information on how messages are related at the instance and process level.

2.5.4 Event Cloud [87]

For the purpose of exploring and searching for event within a repositories for historical events, the authors in [87] introduced Event cloud. Event cloud is an approach for searching business event captured by event-based system. This approach uses correlation sets, a defined relationships between events, to extend the search scope. Where, this correlation set is based on the conformance between elements of events. The key focus on this work is on the index based ranking system which support three different searching scopes. Each ranking level, presented in the following, reflects the type and the depth of relationships between events.

- **Rank 1 search:** The first rank search considers events autonomous, in other words it does not consider any correlation between events.

- **Rank 2 search:** The second rank extends the searching scope by considering direct correlation between events.

- **Rank 3 search:** The last rank goes deeper and allows for searching for indirect correlated events.

To manage the search (rank 1, 2 and 3) the authors developed full-text indexes on event and their correlations and proposed an architecture for preparing them (c.f, [87]).

2.5.5 Rozsnyai et al. [86]

In [86], the authors addressed the problem of automatically discovering correlation rules from various data sources. The discovered correlation rules are used to determine relationship between events and isolate end-to-end process instances. The algorithm presented is similar to previous work of DePauw et al. [79], where the focus is on determining correlation between two type of attributes highly indexable and mappable. However, the authors propose new measures to identify such attributes and to prune non-interesting candidate. The proposed algorithms consists of the, following, three main stages:

- *Data Pre-Processing.* The first step of the correlation discovery algorithm consists of loading and integrating data from sources (XML files) into data store (e.g;, data warehouse, cloud storage etc.).
- *Statistics Calculation.* Inverted indexes are created for each event attribute, and various statistics are calculated such as cardinality, attribute data type, number of instance in which the attribute is involved and the average attributes length. These statistics are stored persistently as map tables.
- *Determining Correlation Candidates.* At this stage, candidate correlation pairs are determined with a certain confidence score based on the following three parameters:
    1. *Difference set.* A difference set determines the difference between all attribute pairs (A, B), where A is a *indexable attribute set* and B is *Mappable attribute set*. It is assigned a weight of 60%.
    2. *Difference between average attribute length.* If the difference between attribute lengths is important this may lead to a poor relationship (weight of 20%).
    3. *Levenshtein Distance.* The authors assume that a good candidate pairs may have a similar or, at least, comparable names (weight of 20%).

An additional feature proposed by the authors is the *Aggregation nodes*. This consist of combining correlation rules to represent certain aspect of an application or the interest viewpoint of the user.

2.5.6 Barros et al. [21]

In this paper, Barros et al. studied a set of correlation patterns in Web service workflows where three classes of correlation patterns are identified as function-based, chain-based

and aggregation functions. The proposed correlation patterns are used as means to group atomic message events into conversations and processes. However, the authors did not provide an (semi)automated approach for event correlation.

2.5.7 Discussion

The works presented above can be classified according to their objectives into two categories. The first category includes, **BHUNT** [27] and **CORDS** [54]. In these works, correlation is used for the purpose of query optimization; by providing constraints to improve the performance of the query optimizer. Both of these works are based on sampling techniques to discover correlation between *pairs* of column tables in relational databases. The second category comprises the remaining works. In these works, correlation is used in the context of business process discovery. These works, initially, focus on identifying correlation over the message pairs only. So, they discover the correlation between message pairs, and not conversations (the entire business process execution). Later on, these approaches have been extended to find a chain of messages. So, a conversation (a collection of messages that are connected using a reference-based model or a mix of all modes) cannot be discovered. In addition, these approaches focus on message-level connections, which can be misleading. Indeed a lot of messages may have the same values on some attribute but may not be forming any conversation. Motahari et al. introduced in [74], an approach that focuses on the judgement of whether correlation is relevant at the conversation level (whether it makes a good set of conversations). This approach is used as the basis of our works. A detailed description of this approach is presented in the next chapter.

2.6 MapReduce Programming Model

Recently, data-intensive computing frameworks have been received a great attention from both industry [48, 24, 40, 32, 34, 43, 95] and the academia [22, 23, 26, 28, 30, 37, 57, 60, 75, 108, 113]. Recently, a powerful trend introduced by google [34] has gained a significant popularity. This trend relies around the *MapReduce* framework. Furthermore, in hadoop [48], the popular open-source implementation of *MapReduce*, the parallel computation is expressed by implementing two interfaces Map and Reduce. A high-level query language are built on top of hadoop for solving a complex problems [24, 77, 95].

MapReduce is a new programming model used to facilitate the development of scalable parallel computations on large server clusters [33]. 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 a key-list as pair $\langle k', list(v') \rangle$, where k' is an intermediate key and $list(v')$ is the list of all the intermediate values to be 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 (e.g., Hadoop Distributed File System (HDFS)).

2.6.1 MapReduce Execution Overview

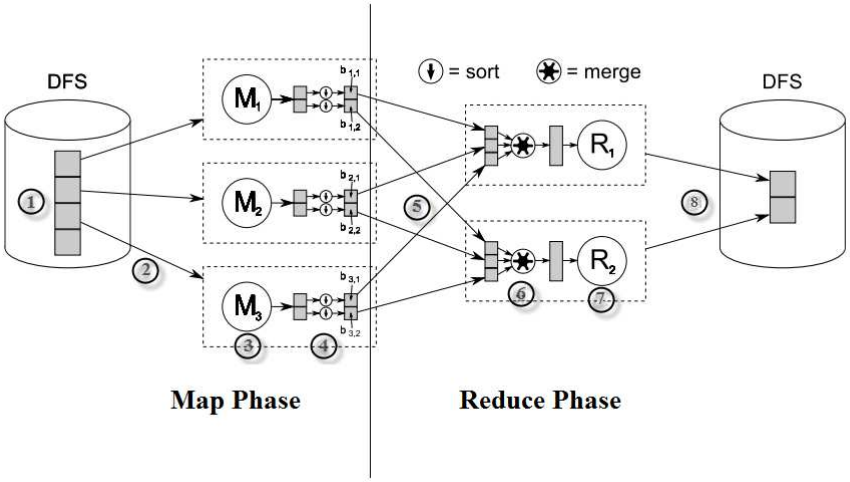

Figure 2.9: MapReduce execution Overview

Figure 2.9 shows an execution workflow of a MapReduce program. The different tasks in Figure 2.9 are numbered as a means of identifying the tasks in the following description. The execution workflow is made of two main phases:

- **Map phase**, which contain the following steps:

  (1) the input file is splitted into several pieces of, typically, 16 to 64 MegaBytes per pieces. Each such piece is called a split or *chunk*.

  (2) each node hosting a map task, called a mapper, reads the content of the corresponding input split from the distributed file system.

  (3) each mapper 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.

  (4) periodically, the buffered intermediate key-value pairs are written to 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 written to (persistent) local storage.

- The **reduce phase**, made of the following steps:

  (5) on the completion of a map task, the reducers (i.e., nodes executing the reduce function), will pull over their corresponding segments.

  (6) 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 $\langle k', list(v') \rangle$.

  (7) each reducer iterates over the sorted intermediate data and passes each pair $\langle k', list(v') \rangle$ to the user's *reduce* function.

  (8) each reducer writes its final results to the distributed file system.

As mentioned previously, our goal is to exploit such a framework to implement efficiently event correlation discovery approach.

2.6.2 Cost Model for MapReduce Programs

Bases on previous works [75, 51], we introduce a cost model that incorporates two metrics: (i) time complexity: represents the time complexity of algorithms used in both Map and Reduce functions, (ii) estimation of the overheads provided by MapReduce framework during job execution.

We consider the following measures to estimate the performance of MapReduce programs:

- I/O cost: time required to read/write data from local disks.
- Network transfer cost.
- CPU cost. We include here main-memory and cache access times as well as operation execution time.

It is worth noting that while it is usual to consider I/O and network transfer costs in (distributed) query optimization area, estimation of CPU cost is less usual and more problematic. Some works, e.g., in the area of main-memory databases, have addressed this problem. With the emergence of hierarchical memory system (small but fast cache memories organized in cascading between CPU and the main memory make such cost estimation problem more complex), access latency varies significantly and the assumption of main memory access is uniform (or covered by CPU) does not hold any more . The main approach to estimate such cost is to estimate the cache misses between each cache level and the level higher [67]. Table 2.3 given below shows the costs associated with each task of the workflow of Figure 2.9. We will show later how to compute for each proposed algorithm the cost associated with each task as well as the global cost of the algorithm.

We use the following parameters. We consider MapReduce Job J processed using m map tasks and r reduce tasks. Let $|M|$ be the average number of map-output records, and $|R|$ be the average number of a reduce-input records. The total number of intermediate records $|D| = |M| * m = |R| * r$. The sort buffer size is B[3]. The threshold for the accounting and serialization buffers is Q[4].

[3] io.sort.mb: The cumulative size of the serialization and accounting buffers storing records emitted from the map, in megabytes.

[4] io.sort.spill.percent: When this percentage of either buffer has filled, their contents will be spilled to disk in the background.

Task (Figure 2.9)	Description	Cost
2	Reading chunks from HDFS (I/O cost)	T_{read}
3	Execution of the map function (CPU cost)	T_{map}
4	Partitioning and sorting data locally (I/O + CPU costs)	T_{sort_map}

Table 2.2: Map phase.

Task (Figure 2.9)	Description	Cost
5	Reading data from mappers node (data transfer cost)	T_{tr}
6	Merge (I/O + CPU costs)	T_{sort_reduce}
7	Reduce execution (I/O + CPU cost)	T_{reduce}
8	Writing the final results to HDFS (I/O cost)	

Table 2.3: Reduce phase.

The total cost of executing a job is the sum of the cost T_{read} to read the data, the cost T_{map} to execute the map function, the cost T_{sort} to do the sorting and copying at the map and reduce nodes, the cost T_{tr} of transferring data between nodes, and the cost T_{reduce} to execute reduce function.

$$T(J) = T_{read}(J) + T_{map}(J) + T_{sort}(J) + T_{tr}(J) + T_{reduce}(J)$$

where:

$$T_{read}(J) = C_r * |Split|$$

- C_r is the cost of reading/writing a record remotely (from HDFS).
- $|Split|$ is the number of records in the split (input file).

A record emitted from a map will be serialized into a buffer and meta-data will be stored into accounting buffers. When either the serialization buffer or the metadata exceed a threshold, the contents of the buffers will be sorted and written (spilled) to disk. When the map is finished, any remaining records are written to disk and all on-disk segments are merged into a single file. The cost of the map task execution is:

$$T_{map}(J) = C_u * O_m * |Split|$$

- C_u is the cost to execute one operation in the map function.
- O_m the complexity of the map function (number of operations)
- $|Split|$ is The number of records (messages) in a Split.

$$T_{sort\_map}(J) = C_l * (spillsize * 2(|spills| + MergeSpillsPasses(|Spills|, Factor))).$$

MergeSpillsPasses($|Spills|$, Factor) =

$$\begin{cases} 0 & \text{,if } |Spills| = 1. \\ 1 & \text{,if } |Spills| \leq Factor. \\ 2 + \frac{|Spills| - |SpillsFirstPass|}{Factor} & \text{,if } |Spills| \leq Factor^2. \end{cases}$$

SpillsFirstPass =

$$\begin{cases} |Spills| & \text{,if } |Spills| < Factor. \\ Factor & \text{,if } (|Spills| - 1) \mod (Factor - 1). \\ (|Spills| - 1) \mod (Factor - 1) + 1 & \text{,otherwise.} \end{cases}$$

$$T_{sort}(J) = T_{sort\_map}(J) + T_{sort\_reduce}(J).$$

- $|Spills|$ is the number of spill to disk and it equals to:

$$|Spills| = \frac{|M|}{B * Q * P * 2^{16}}$$

- $spillSize$ is the size the spilled file.
- $Factor$: specifies the number of segments on disk to be merged at the same time. If the number of files exceeds this limit, the merge will proceed in several passes. - C_l is the cost of reading/writing data locally.
- $MergeSpillsPasses(Spills, Factor)$ is the number passes to sort $|M|$ records.

- P: The ratio of serialization to accounting space can be adjusted. Each serialized record requires 16 bytes of accounting information in addition to its serialized size to effect the sort. This percentage of space allocated from B affects the probability of a spill to disk being caused by either exhaustion of the serialization buffer or the accounting space.

Each reduce fetches the output assigned to it by the partitioner via HTTP into memory and periodically merges these outputs to disk.

$$T_{sort\_reduce}(J) = C_l * (|R|(\lceil 2 \log_{Factor} m \rceil)).$$

- At the reduce side, it starts with m sorted runs. $\lceil \log_{Factor} m \rceil$ is the number of passes to merge the m runs.

$$T_{tr}(J) = C_{tr} * D.$$

- C_{tr} is the cost of transferring data between nodes.

$$T_{reduce}(J) = C_u * O_r.$$

- C_u is the cost to execute one operation in the reduce function (the same as in the map task)
- O_r the complexity of the reduce function (number of operations).

C_u depends on the cpu capacity of the computing node. As example, Amazon web services (AWS) [2] provide a flexibility to choose from a number of different node types to meet the computing power needs. Each instance provides a predictable amount of dedicated computing capacity and is charged per instance-hour consumed. (for more information see [3]). Furthermore, a monetary cost model can be introduced to complement this latter. Such cost model might be effectively used to analyse running algorithms on cloud resources w.r.t economical dimension.

2.6.3 Disucussion

MapReduce was originally proposed to execute very large matrix-vector and matrix-matrix multiplications as are needed in the calculation of PageRank [78]. However, MapReduce model has been shown suitable for performing large scale data analysis including:

- Query processing: join algorithms and algebra operations [25, 83], set-similarity and fuzzy joins [102], Transitive Closure and Recursive queries [11, 10] ,
- Data-mining: Social Network Analysis [92], frequent itemset mining [65].
- Analytic processing: Social Network Analysis [66],

To the present day, MapReduce has only been exploited to perform scalable analysis on large size of data from data-oriented perspective. Although there is a plethora of approaches and tools devoted to process mining analysis, to the best of our knowledge, none of these approaches exploited MapReduce framework to analyse (event) data from a process-oriented perspective. Indeed, the continuous growth of the process related data makes existing process analysis approaches face the scalability issue. Therefore, the need for scalable algorithms for process analysis become a requirement and a subject of our researches. Hence, in this thesis we propose a scalable/distributed MapReduce-based approach for event correlation discovery, a key step for business process discovery approach.

Chapter 3

Process Space

Contents

3.1 Introduction

In this chapter we describe in details the approach of correlation discovery *Process Space* introduced by Motahari et al. in [74]. This approach is used as a basis of our work for developing MapReduce algorithms for event correlation discovery. The chapter is organized as follows: in section 3.2 we present the correlation patterns investigated in this approach. Next, in section 3.3 we present an overview of the approach. Then, in section 3.4 we present the heuristics used to select relevant candidates. Finally, we discuss the approach in the summary

3.2 Correlation Condition Patterns

In this section we present the correlation condition patterns used in [74, 73] for event correlation in web services.

(a)

	UserID	uSessionID
m_1	u100	
m_2	u200	
m_3	u300	u100
m_4	u400	u200
m_5	u500	u300
m_6	u600	u400
m_7	u700	u500
m_8	u800	u600

(b)

	LoginID	GameID
m_1	C1	P1
m_2	C2	P2
m_3	C2	P1
m_4	C1	P2
m_5	C2	P2
m_6	C1	P2
m_7	C2	P1
m_8	C1	P1

Table 3.1: a snapshot of example log.

3.2.1 Key-Based Correlation.

Process-related standard proposals for web services such as BPEL, WS-conversation, WS-coordination, WS-CDL [16], or industrial software such as IBM WebSphere Process Manager [6] use methods to correlate events related to the execution of a business process. These methods are characterized by the fact that all messages in a single process instance share the same value for one or more attribute(s). These attributes are called

the *correlator attribute(s)*. Indeed, a correlator attribute could be present even if these standards are not used. For example, in RoboStrike[1] game process, the events could be correlated by the **loginID**, or by the pair ⟨**UserID**, **LoginID**⟩. On this basis **Key-Based Correlation** pattern is defined as follows:

Key-Based Correlation. *"One or a set of unique identifiers are assigned to an event and all events with at least one common identifier are grouped together. A process instance identifier or a conversation identifier is attached to each event. Identifiers can be single values or compositions of several values."* [21]. The Key-Based correlation condition has the following form $\psi(m_x.A_i, m_y.A_j) : m_x.A_i = m_y.A_i$.

The identifier(s)[2] is called the *key attribute*. Contrary to the concept of key in relational databases, the value of the key is not unique per tuple but unique per process instance. Furthermore, there is no prior information about which events form the same process instance in the log.

Example 3 *Considering the condition* ψ*:* $m_x.LoginID = m_y.LoginID$ *in Table 3.1(b) the process instances entailed by this condition are* $PI_\psi = \{\langle m_1, m_4, m_6, m_8\rangle, \langle m_2, m_3, m_5, m_7\rangle\}$.

3.2.2 Reference-Based Correlation

Similar to the concept of *foreign-key* in traditional relational databases, an attribute event in the log may share the same value with a different attribute in another event. For example, a response message is mostly correlated with the request message. As a second example, messages related to a purchase activity (service) may contain the **LoginID** attribute which references the customers registered at the *registration activity (service)*. Furthermore, the shared value used for reference correlation between pair of messages may not be the same for the entire process instance. Then, the **reference-based** correlation can be defined as follows:

(Reference-Based Correlation). *"Two events are correlated, if the second event (in chronological order) contains a reference to the first event. This means that if*

[1] http://www.robostrike.com/

[2] These identifiers are different from those defined in 2.4.1. The former defines the process instance and the latter is similar to the key in relational databases which defines the tuple (event).

there is some way of extracting a datum from the second event (by applying a function) that is equal to another datum contained in the first event. This datum therefore acts as a message identifier, and the second message refers to this message identifier in some way" [21]. The **reference-based** correlation condition has the following form: $\psi(m_x.A_i, m_y.A_j) : m_x.A_i = m_y.A_j$ and $i \neq j$.

Example 4 *For the Condition* ψ*:* $m_x.UserID = m_y.uSessionID$ *in Table 3.1(a), the process instances entailed by this condition are* $PI_\psi = \{\langle m_1, m_3, m_5, m_7\rangle, \langle m_2, m_4, m_6, m_8\rangle\}$.

Each of **key-based** and **reference-based** correlation methods express equality of attribute value to pair of events. Hence, both of them belong to the same correlation condition family, which is referred as *atomic correlation condition*. It is defined as follows:

Definition 3.2.1 *(Atomic correlation conditin). Two messages (events)* (m_x, m_y) *are correlated using an atomic correlation condition* ψ *if and only if they share the same value on two attributes* A_i *and* A_j*, in other words* $\psi(m_x.A_i, m_y.A_j) : m_x.A_i = m_y.A_j$*. If* $i = j$ *the* ψ *is a* ***key-based*** *condition, otherwise it is a* ***reference-base*** *condition.*

In the following we present in details the correlation discovery approach introduced by Motahari et al. in [74].

3.3 Semi-Automated Discovery of Correlation Conditions

Figure 3.1 depicts the correlation condition discovery process. The main steps of this approach are given below:

(i) **Candidate attribute selection.** This step is also called attribute profiling step. It consists in removing non relevant attributes (e.g., attributes having Boolean values).

(ii) **Atomic condition discovery.** Attributes selected in the last step are combined, in pairs, to form atomic conditions.

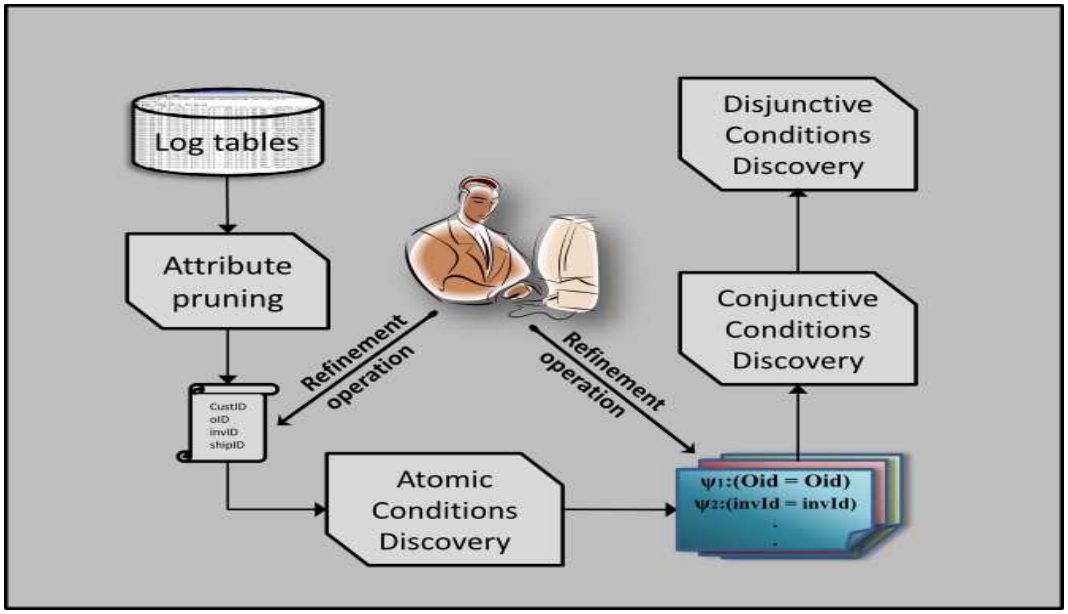

Figure 3.1: Event Correlation Discovery Process

(iii) **Composite condition discovery.** *Conjunction* (respectively. *Disjunction*) of atomic condition is considered to build composite conditions. A level-wise [68] technique is adopted to explore the space of correlation conditions.

The proposed algorithm for correlation discovery follows the same steps as *Apriori* algorithm [13]. The first step, computing *atomic conditions*, is similar to computing first order itemsets (of size 1), and the second step, computing *composite conditions*, is similar to computing itemsets of larger sizes. Besides this, each step of the algorithm has two phases: (i) generating candidate correlation, (ii) pruning candidate conditions. A set of properties and heuristics defined based on general statistical characteristics of conversations in the logs, are used to prune the search space of non-relevant conditions. More details of these properties and criteria are discussed in next sections. The output of the algorithm is the set of interesting conditions. Interesting conditions are considered as the conditions that partition the log L into a set of interesting process instances. Next section will explain how the correlation condition partition the log.

3.3.1 Partitioning the log

Motahari et *al.* represented the relationship between messages in the log as an undirected graph $G_\psi = (V, E)$. Where V is the set of messages in the log L, and E

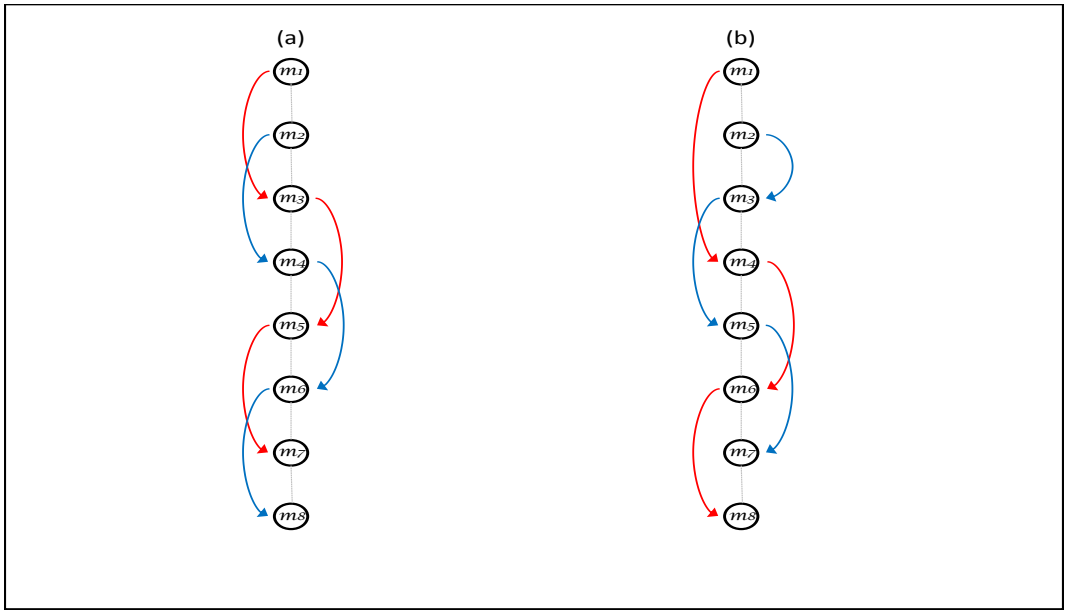

Figure 3.2: Correlated message Graph.

is defined as $E = \{(m_x, m_y) \in L^2 | \psi(m_x, m_y)\ is\ true\}$, i.e, there is an edge between m_x and m_y if and only if $\psi(m_x, m_y)$ holds. E is called the set of correlated message pairs based on condition ψ. Henceforth, R_ψ is used to express a set of correlated message pairs. Figure 3.2(a) shows the graph representing the set of correlated message pairs $R_\psi = \{\langle m_1, m_3\rangle, \langle m_3, m_5\rangle, \langle m_5, m_7\rangle, \langle m_2, m_4\rangle, \langle m_4, m_6\rangle, \langle m_6, m_8\rangle\}$ of the condition $\psi : m_x.UserID = m_y.uSessionID$ presented in Table 3.1(a). and Figure 3.2(b) depicts a second graph representing the set of correlated message pairs $R_\psi = \{\langle m_1, m_4\rangle, \langle m_4, m_6\rangle, \langle m_6, m_8\rangle, \langle m_2, m_3\rangle, \langle m_3, m_5\rangle, \langle m_5, m_7\rangle\}$ of the condition $\psi : m_x.LoginID = m_y.LoginID$ in Table 3.1(b).

A correlation condition (reference-based, key-based or composite) is used to partition the log into a set of process instances $PI_\psi(L) = \{pi_1, pi_2, \ldots\}$ such that:

- For a given message m_x in a process instance that belong to an instance entailed by a correlation condition ψ, then it should exist, at least, another message m_y such that the pair (m_x, m_y) belongs to R_ψ.

- A given message m_x cannot be part of more than one process instance. More formally:

$$\begin{cases} \forall pi \in PI_\psi(L), & m_x \in pi \Leftrightarrow \exists m_y, (m_x, m_y) \in R_\psi \vee (m_y, m_x) \in R_\psi \\ \forall pi_i, pi_j \in PI_\psi(L), & i \neq j \Leftrightarrow pi_i \cap pi_j = \emptyset \end{cases}$$

Hence, discovering process instances PI_ψ can be formulated as finding the set of connected components[3] in an undirected graph (G_ψ). For example, the set of process instances of the graph (a) in Figure 3.2 are $PI_\psi = \{\langle m_1, m_4, m_6, m_8\rangle, \langle m_2, m_3, m_5, m_7\rangle\}$. Several existing algorithms [14, 56] deal with the problem of finding the connected components such as *depth-first search* and *breadth-first search*. Such algorithms take a graph as input and returns the maximal set of connected components in the graph. The connected components represent the set of process instances.

To summarize the partition PI_ψ, the author defined a set of metrics as follows:

- $AvgLen(PI_\psi), shortInst(PI_\psi)$ and $LongInst(PI_\psi)$ represent the average, the shortest instance and the longest instance length.
- $|PI_\psi|$ represents the cardinality of PI_ψ, i.e, the number of instances.

3.4 Candidate Attributes Selection

Obviously not all attributes present in the log can be considered as *correlator attributes*. For instance, a *timestamp* attribute will not partition the log into relevant instances. Attributes selection techniques and heuristics are discussed in the next section.

3.4.1 Characteristics of Correlator attributes

Similar to primary key (respectively, foreign key) in relational databases, attributes used in key-based pattern (respectively, reference-based pattern) correlation play the role of identifiers attributes[4]. Based on these similarities a correlator attribute can be characterized as follows:

- Nominal domain: a correlator attribute does not contain a floating point value or a long free text.
- Distinct values: a correlator attribute value should not have a small domain w.r.t to the dataset size (e.g., boolean).
- The correlator should have repeated values in the log, where the same value should appear at least in two messages.

[3] In databases, this problem is called also computing the transitive closure of a query [12, 96].
[4] Porcess instance identifier and not tuple identifier

3.4.2 Attributes Pruning

From the previous characteristics, an attribute may not be considered if it has the following characteristics:

- Attributes having a float type or either a long free text are excluded,
- Attributes with a small domain such as *boolean, color or sex* are eliminated.

3.4.3 Atomic Condition Discovery

In this section, we present the approach for discovering candidate atomic correlation condition proposed by Motahari et al. in [74].

3.4.4 Candidate Atomic Condition Generation

In the first step, candidate atomic conditions (key-based and reference-based) of form $\psi : m_x.A_i = m_y.A_j, 1 \leqslant 1 \leqslant j \leqslant k$[5] are generated based on equality relationship between pairs of attributes for each message pair $(m_x, m_y) \in L^2$. Then, for a given pair of attributes, if the candidate correlation condition holds for a large subset of the data set then it is considered as interesting and is selected, otherwise this condition is pruned.

Interesting conditions are defined as the conditions that lead to an interesting partitioning of the log, i.e, an interesting condition should enable to rebuild a set of process instances from the log. Criteria and measures are defined based on (i) the properties of the attributes forming the conditions, and (ii) the statistics about the resulting process instances.

3.4.5 Atomic Condition Pruning

The main idea to identify interesting correlation conditions is based on eliminating what is not interesting [88]. The following criteria and measures have been proposed to select relevant conditions:

- Globally unique keys are not correlators. 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

[5] k is the number of attributes present in the log L.

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)*: for *key-based* conditions on attribute A_i, this ratio represents the number of distinct values of an attribute A_i with regard to the number of non-null values in A_i,

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

- *shared_ ratio(A_i, A_j)*: for *reference-based* conditions over two attributes A_i and A_j, this ratio corresponds to the number of common distinct values between attribute A_i and A_j, with regard to the maximum number of non-null values of A_i or A_j,

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

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[6] can be user provided or computed using information categorical attributes [74].

- 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 criterion is referred to as *imbalancedPI*.

[6] usually $\alpha \leq 0.01$

3.4.6 Composite Condition Discovery

Algorithm 1: Composite Correlation Condition Discovery Algorithm

Input: AC: Atomic Conditions.

Output: CC: Conjunctive conditions, DC: Disjunctive conditions.

```
1 begin
2     CC ← computeConjunctiveConditions(AC) ;
3     DC ← computeDisjunctiveConditions(AC, CC) ;
4 return {CC ∪ DC} ;
```

Correlation conditions may not be only atomic, a higher level of conditions can be built using *conjunctive* ($\wedge$) or *disjunctive* ($\vee$) operators. These conditions are called *composite conditions*, where the *conjunctive* ($\wedge$) operator is used when multiple *attributes* are defined together as correlators, and *disjunctive* ($\vee$) operator is used to correlate messages that are not correlated with the same correlation condition.

The following steps are used to discover candidate composite correlation conditions:

1. First, the set of candidate *conjunctive* conditions are built. This is performed by generating all the possible combinations of *atomic* conditions and prune evident non-candidate conditions (line 2 of algorithm 1). This is similar to the steps of generating and pruning itemsets of more than two items in *Apriori* algorithm.

2. Similar to the previous step, candidate *disjunctive* conditions are generated by a disjunction of both *atomic* and *conjunctive* conditions already discovered in the last steps (line 3 of algorithm 1).

3.4.6.1 Conjunctive Conditions

As in relational databases, where multiple keys are used to identify the same tuple, a conjunction of several attributes may also identify a conversation (process instance), in other words, more than one attribute is used to correlate messages of the same instance. For example, conditions $\psi_1 : m_x.LoginID = m_y.LoginID$ and $\psi_2 : m_x.GameID = m_y.GameID$ can be combined using ($\wedge$) operator to form: $R_{\psi_1 \wedge \psi_2} = R_{\psi_{1\wedge 2}} = \{\langle m_1, m_8\rangle, \langle m_2, m_5\rangle, \langle m_3, m_7\rangle, \langle m_4, m_6\rangle\}$.

A conjunction is performed to define intersection relation defined by two or more atomic conditions. For ψ_1 and ψ_2 two atomic conditions, the conjunctive condition $\psi_{1\wedge 2} =$

$\psi_1 \wedge \psi_2$ is defined as follows:

$$(m_x, m_y) \in \psi_{1\wedge 2} \Leftrightarrow (m_x, m_y) \in R_{\psi_1} \wedge (m_x, m_y) \in R_{\psi_2}$$

$$\Leftrightarrow (m_x, m_y) \in R_{\psi_1} \cap R_{\psi_2}$$

This means that m_x and m_y share the same values for attributes of the conditions ψ_1 and ψ_2. Hence, the pair (m_x,m_y) belongs to the intersection of the set of correlated message pairs R_{ψ_1} with R_{ψ_2}. However, $PI_{\psi_{1\wedge 2}} \neq PI_{\psi_1} \cap PI_{\psi_2}$.

To generate all the possible candidate *conjunctive* conditions, a level-wise approach is adopted [68]. Assuming that a is the number of *atomic* conditions inferred from the last step, then, the number of possible conjunction is $2^a - (a + 1)$. For example, let $AC = \{\psi_1, \psi_2, \psi_3\}$, then the set of *conjunctive* condition is $CC = \{(\psi_1 \wedge \psi_2), (\psi_1 \wedge \psi_3), (\psi_2 \wedge \psi_3), (\psi_1 \wedge \psi_2 \wedge \psi_3)\}$. Figure 3.3 shows a lattice generated by 3 *atomic conditions*, we observe that the number of candidate *conjunctive conditions* equals to $2^3 - (3 + 1) = 8 - 4 = 4$.

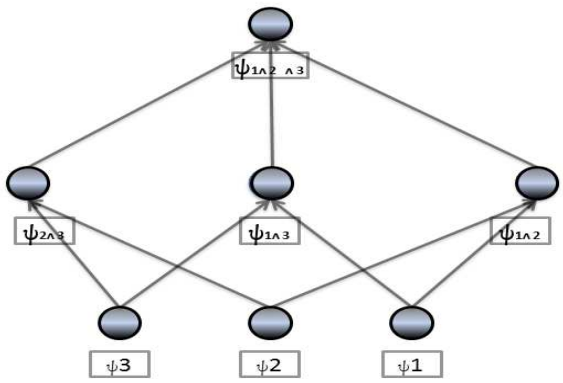

Figure 3.3: Lattice generated by 3 atomic conditions.

Situation like, $PI_{\psi_{1\wedge 2\wedge 3}} = PI_{\psi_{1\wedge 2}}$, signifies that $\psi_{1\wedge 2}$ and $\psi_{1\wedge 2\wedge 3}$ partition the log into the same process instances. In this case, it suffices to compute the *minimal conjunctive condition*, which is defined as follows:

Minimal conjunctive condition. *"A conjunctive condition ψ is minimal if no other conjunctive condition formed using fewer conjunction of atomic conditions partition the*

log into the same set of instances"[74].

In the following we present the criteria and measures proposed by *Motahari* et *al.* to eliminate non-interesting candidate conjunctive conditions.

Eliminating non-interesting conjunctive conditions in generation phase It is preferred to anticipate obvious non-interesting candidate conditions and eliminate them to reduce the exploration space. Thus, the following criteria are introduced to deal with this issue.

- *Attribute definition constraints:* given two *atomic* conditions ψ_1 and ψ_2 defined on attributes (A_{i_1}, A_{j_1}) and (A_{i_2}, A_{j_2}) respectively, the conjunctive condition formed by ψ_1 and ψ_2 has the form: $\psi_{1\wedge 2}$: $m_x.A_{i_1} = m_y.A_{j_1} \wedge m_x.A_{i_2} = m_y.A_{j_2}$. Here, attributes of ψ_2 should be defined whenever the attributes of ψ_1 are defined. This implies that A_{i_1} (respect. A_{j_1}) is defined if only if A_{i_2} (respect. A_{j_2}) is defined. Thus, conjunctive conditions are applied only for attributes that satisfy this constraint. Otherwise, the *conjunctive* condition can be safely eliminated.

- *Inclusion property:* in case of $R_{\psi_1} \subseteq R_{\psi_2}$, this means that the set of correlated message pairs of ψ_1 are included in (or equal to) the set of correlated message pairs of ψ_2 then $R_{\psi_{1\wedge 2}} = R_{\psi_1}$. Therefore,$\psi_{1\wedge 2}$ is not minimal and then discarded.

Eliminating non-interesting conjunctive conditions in pruning phase At this step, non-interesting *conjunctive* conditions are identified and pruned based on the following criteria:

- *ImblancedPI criterion:* the $PI_{ratio}(\psi_{1\wedge 2})$ is computed, and compared to threshold β. If, $PI_{ratio}(\psi_{1\wedge 2}) < \beta$ is satisfied then the condition is interesting, otherwise it is pruned.

- *Monotonic property:* An important property is the *monotonicity* w.r.t to the *conjunctive* operator. Using conjunction operator will reduce the length of the instances (number of messages in each instance) but increased their total number. To consider a conjunctive condition $\psi_{1\wedge 2}$ as interesting the following properties must be satisfied.

$$\begin{cases} shortInst(PI_{\psi_{1\wedge 2}}) \leq min(shortInst(PI_{\psi_1}), shortInst(PI_{\psi_2})) \\ |PI_{\psi_{1\wedge 2}}| \geq max(|PI_{\psi_1}|, |PI_{\psi_2})| \end{cases}$$

In other words, the number of instances discovered by conjunctive condition $\psi_{1\wedge 2}$ is expected to be greater than those of ψ_1 and ψ_2, and the length of instances decreases.

The candidate *conjunctive* conditions that satisfy all the above criteria are retained and used with initial *atomic* conditions as input for the second type of composite conditions (*disjunctive* conditions). One should note that for any non-interesting *conjunctive* condition ψ, higher (or larger) conditions built on ψ are also considered as non-interesting.

3.4.6.2 Disjunctive Conditions

Messages	**Service**	**InvId**	**PayId**
m_1	invoice	i1	
m_2	invoice	i2	
m_3	Pay	i1	P1
m_4	Pay	i2	P2
m_5	Ship		P2
m_6	Ship		P1

Table 3.2: a snapshot of example log.

As we said previously, not all messages are correlated within the same correlation condition. Indeed, a message m_x in a conversation may refer to another message m_y. For instance, when a payment message refers to an invoice number, and shipping message refers to a payment number (see Table 3.2). So, to correlate this messages a disjunction between $\psi_1 : m_x.InvID = m_y.InvID$ and $\psi_2 : m_x.PayID = m_y.PayID$ should be performed. Here, the *disjunctive* condition $\psi_{1\vee 2} = \psi_1 \vee \psi_2$ is defined as follows:

$$(m_x, m_y) \in \psi_{1\vee 2} \Leftrightarrow (m_x, m_y) \in R_{\psi_1} \vee (m_x, m_y) \in R_{\psi_2}$$

$$\Leftrightarrow (m_x, m_y) \in R_{\psi_1} \cup R_{\psi_2}$$

Where ψ_1 and ψ_2 are either *atomic* or *conjunctive* conditions.

Similar to *conjunctive* conditions, case like $PI_{\psi_{1\vee 2\vee 3}} = PI_{\psi_{1\vee 2}}$ may occur at this step also. Therefore, only *minimal disjunctive conditions* are considered.

Minimal disjunctive condition. *"A disjunctive condition ψ is minimal if no other disjunctive condition using fewer disjunction of atomic conditions partitions the log into the same set of instances"*[74].

Discovery of *disjunctive* conditions is also carried out by a level-wise approach. Each level consists of two phases: *candidate generation* and *candidate pruning*. Criteria are introduced to evaluate the interestingness of candidate *disjunctive* condition.

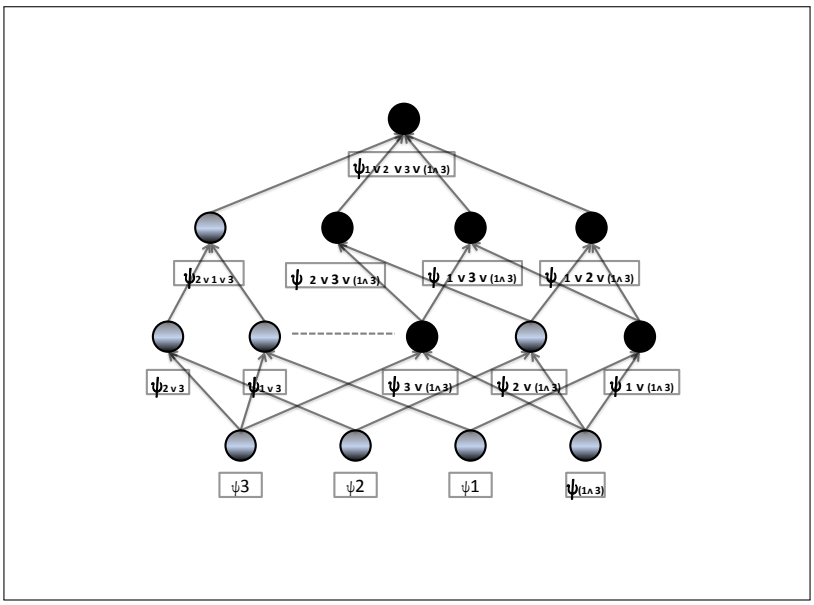

Figure 3.4: Lattice generated by 3 atomic conditions and one conjunctive condition.

Eliminating non-interesting disjunctive conditions in generation phase Non-interesting candidate *disjunctive* conditions are identified based on the following criteria:

- *Associativity of conjunction and disjunction:* conditions that associate conjunction and disjunction of the same atomic conditions are identified as not needed to be computed, since it can be simplified into a condition that has already been considered. For example, $\psi_{1\wedge 3} \vee \psi_3$ is equivalent to the condition ψ_3 (see Figure 3.4).

- *Inclusion property:* Same as *inclusion property* in *conjunctive* conditions, if a condition ψ_1 is included in ψ_2, in other words the set of correlated message pairs R_{ψ_1} is included in R_{ψ_2}, then, $\psi_{1\vee 2}$ is equivalent to ψ_2. Hence, this condition is discarded.

Eliminating non-interesting disjunctive conditions in pruning phase The following criteria are used to prune non-interesting *disjunctive* conditions after the computation operation:

- *ImblancedPI criterion:* the number of process instances entailed by a *disjunctive* condition is compared to the threshold α. This criterion refers to check whether the condition partition the log into a small number of long instances. if, $PI_{ratio}(\psi) \geq \alpha$ is not satisfied the condition is eliminated.

- *Trivial union:* given a *disjunctive* condition $\psi_{1\vee 2}$, if any process instance from ψ_1 does not connect with another instance from ψ_2 then the disjunction results a *trivial union* of instances in ψ_1 and ψ_2. To catch this property we use the following measure: $|\psi_{1\vee 2}| = |\psi_1| + |\psi_2|$, if this measure returns true, then this condition is pruned.

- *Monotonic property:* unlike *conjunctive* condition, we expect that the number of resulted process instances of the *disjunctive* condition to be less than those of the condition built on, and their length increases. The following measures catch this property:

$$\begin{cases} shortInst(PI_{\psi_{1\vee 2}}) \;\geq\; max(LongInst(PI_{\psi_1}), LongInst(PI_{\psi_2})) \\ |PI_{\psi_{1\vee 2}}| \leq |PI_{\psi_1}| + |PI_{\psi_2}| \end{cases}$$

Theoretically, if the number of inferred *atomic* conditions is ac, then there is $cc = 2^{ac} - (ac + 1)$ possible *conjunctive* conditions. As, we use *conjunctive* conditions with *atomic* condition as input for *disjunctive* condition discovery process, the number of possible *disjunctive* conditions is then equal to $dc = 2^{ac+cc} - (ac + cc + 1)$.

3.5 Summary

In this chapter we presented the approach of event correlation discovery proposed by Motahari et al in [74]. This approach covers a large number of correlation patterns that may occur in interactions of web services and provides a set of heuristics to identify relevant correlation conditions that lead to interesting process instances. Moreover, this approach takes into account the process instance characteristics to improve the result

quality. However, it requires correlating relatively a large number of messages. For example, long instances correlated with *key-based* conditions, the set of correlated messages R_ψ becomes large since it includes all correlated message pairs (if s_{max} is the size of the longest instance, then the size of the correlated message pair only for this instance is $(s_{max})^2$). Moreover, it requires exploring and processing a very large number of candidates (if the number of the discovered atomic conditions is *'a'* then the theoretical number of composite correlation conditions is 2^a). Based on these issues, we present MapReduce-based algorithms for event correlation discovery in the upcoming chapters.

Chapter 4

Discovering Atomic Conditions

Contents

4.1 Introduction

In this chapter, we focus on candidate atomic condition discovery problem. We use MapReduce framework as the parallel data processing paradigm. The main contributions of this chapter are as follows:

- We describe efficient solutions for discovering candidate atomic conditions by exploiting MapReduce framework. We show how to efficiently deal with problems such as partitioning, replication, and multiple inputs by manipulating the keys used to route the data between nodes of MapReduce cluster.

- We provide an adequate data structure similar to inverted index in order to decrease the memory usage.

- We introduce techniques to compute the set of correlated messages by optimizing memory space.

- We provide both one-pass and multi-pass algorithms for conditions discovery computations. 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 introduce an efficient solution to compute process instances based on depth-first-search-like algorithm 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.

The rest of the chapter is structured as follows. In section 4.2 we present a family of MapReduce event correlation discovery algorithms as well as the data structure they use, while in section 4.3 we discuss extensions of the proposed algorithms to handle limited memory case in MapReduce. A complexity and cost-model based analysis are presented in section 4.4. Finally, a performance evaluation is presented in section 5.4 and we summarize the chapter in section 5.5.

4.2 Atomic Condition Discovery Algorithms

One should recall that the fundamentals of parallelizing any algorithm in the MapReduce framework is to design Map and Reduce functions.

Given an events log L, the aim of our algorithms is to discover the interesting atomic correlation conditions and compute the process instances entailed by these conditions. One of the main issues to cope with, is to decide how data and computations should be partitioned, replicated and distributed, in order to efficiently execute the operations entailed by this task. The main idea of our approach is the following. First, we generate all possible candidate conditions and partition the data across the network by hashing on the candidate name (e.g., $A_i = A_j$). Then, we process each candidate condition ψ_{A_i,A_j} by a single Reduce function and, thus, each candidate can be handled separately and in parallel with the others. Then, interesting correlation conditions are retained and written into files to be fed to the next step (candidate composite conditions discovery). Table 4.1 shows a general description of the proposed algorithms: *Sorted Values Centric*, *Hashed Values Centric* and *Per-Split Correlated Messages* denoted respectively by *SVC*, *HVC* and *PSCM*. Based on this distinct features, we can distinguish suitable situations for each algorithm. *SVC* can be suitable for situations where the discovered process instances are numerous and short (having a low number of messages). Where, *HVC* is suitable when the discovered process instances are less numerous and long. Finally, *PSCM* is suitable for larger datasets and the case of events correlated by key based conditions.

On one hand, each algorithm has a distinct features that make it suitable in specific situations. *SVC* relies on one MapReduce job, it sorts the intermediate data to efficiently compute the correlated message buffer denoted by $\mathcal{CMB}$. However, it involves a large intermediate data size. *HVC* relies on one MapReduce job, has a low intermediate data size, but requires several iterations to compute correlated messages. Finally, *PSCM* relies on two MapReduce jobs. The first step computes the correlated message buffer in parallel, where the second step groups the correlated messages and deduces the process instances.

On the other hand, the algorithms have some shared parts, as the inputs, the data structure used, some pieces of functionalities and the outputs. In the remaining sections we present the data structure as well as the algorithms devoted to deal with the problem of atomic correlation condition discovery.

Algorithm	# MR steps	Map output	Reduce input	Computing $\mathcal{CMB}$
SVC	one job	(ψ+(val+tag+id), val+tag+id)	sorted	one iteration
HVC	one job	(ψ, val+tag+id)	non sorted	several iterations
PSCM	two jobs	1^{st} job: (ψ+val, tag+id) 2^{nd} job: (ψ, single row)	non sorted non sorted	single row one iteration

Table 4.1: A general description of the proposed algorithms.

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

To facilitate correlation computation we define two types of data structures. The first data structure is similar to inverted index in relational databases [89, 111, 112], used to index values of the same column (for key-based condition). The second, data structure can be obtained by performing a join between two inverted index on the value part (for reference-based condition). The description of this data structure is as follows:

- The first data structure we introduce is devoted to *key-based* conditions. This data structure is defined as $T_1 : [val, \{IdSet\}]$, where val is a given value of the attribute forming the condition (e.g.,A_i) and $\{IdSet\}$ represents the set of messages having val as value in A_i i.e, $\{\{m_x.id\}|m_x.A_i = val\}$. T_1 is an array used to store all the distinct values of a given attribute A_i. This data structure is used to calculate statistics such as: number of distinct values of an attribute, number of correlated messages with a given value. The previous metrics are needed in the pruning phase (see section 3.4.5). Table 4.2(a) (respect, 4.2(b)) shows the indexed values of A_1 (respect, A_2). We refer to this table as $\mathcal{CMB}$ (Correlated Message Buffer). Similar data structure are used in [90, 86].

- The second data structure is devoted to *reference-based* condition. It is defined as T_2:[val, {IdSet1}, {IdSet2}]. It is obtained by joining two $\mathcal{CMB}$s on the value part. Values that does not appear in both indexes are discarded. Table 4.2(c) shows a shared index of attributes A_1 and A_2, where values $C3$ and $C4$ are discarded.

Example 5 *Taking the log present in Table 3.1, the messages correlated by the condition ψ:$m_x.LoginID = m_y.LoginID$ are $R_\psi = \{\langle m_1, m_4\rangle, \langle m_1, m_6\rangle, \langle m_1, m_8\rangle, \langle m_4, m_6\rangle, \langle m_4, m_8\rangle, \langle m_6, m_8\rangle, \langle m_2, m_3\rangle, \langle m_2, m_5\rangle, \langle m_2, m_7\rangle, \langle m_3, m_5\rangle, \langle m_3, m_7\rangle, \langle m_5, m_7\rangle\}$. Such a set is represented in a condensed form using the $\mathcal{CMB}$ data structure as, T_1 :*

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

(a) *attribute* A_1

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

(b) *attribute* A_2

Val	$\rightsquigarrow$	$IdSet$
C1	$\rightsquigarrow$	$\{m_1, m_3\}$
C2	$\rightsquigarrow$	$\{m_2, m_4\}$
C4	$\rightsquigarrow$	$\{m_5\}$

(c) *attributes* A_1-A_2

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

Table 4.2: Example of CMB data structures

$[C1|\langle m_1, m_4, m_6, m_8\rangle], [C2|\langle m_2, m_3, m_5, m_7\rangle]$. *It is worth noting that process instances can be directly deduced from* T_1*, i.e, computing transitivity is not needed. Noting that,* $[C1|\langle m_1, m_4, m_6, m_8\rangle]$ *represents a* $\mathcal{CMB}$ *row.*

Example 5 shows the effectiveness of the use of data structure T_1 in saving memory and computation of transitive closure to find process instances. Below, we explain each algorithm in more details.

4.2.2 Sorted Values Centric Algorithm

The first algorithm devoted to compute atomic conditions is Sorted Values Centric (SVC) algorithm depicted in algorithms 2 (Map) and 3(Reduce). It relies on one MapReduce job. The sorted values centric algorithm, as input, requires a log L over the relational schema $\mathcal{L}$ $(A_1, A_2, \ldots, A_n, id)$ and the user provided thresholds α and β. The Map reads a split of the log and, from the set of attributes in $\mathcal{L}$, generates the set of all possible candidate atomic correlation conditions ψ_{A_i,A_j} for two attribute A_i and A_j. This is achieved by computing the cross product $\mathcal{L} \times \mathcal{L}$ (line 2 to 6 of algorithm 2). For each message, it extracts the values corresponding to A_i and A_j (attributes forming the condition ψ_{A_i,A_j}). In order, to keep track of the origin of each value, the Map tags the values by their original attribute name and message-id (line 5 and 6 of algorithm 2). Then, it outputs the conditions name and the tagged values as $(key + value, value)$ pairs (lines 7 and 8 of algorithm 2).

The Map function ensures that : (i) a given pair of attributes A_i and A_j is allocated

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

Algorithm 2: Sorted Values Centric map function.

Input: $\mathcal{K}$: unused, $\mathcal{V}$: a record from the log file
Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V} : \pi_{A_i,A_j}(\mathcal{L})$

```
1  begin
2      foreach A_i ∈ V do
3          foreach A_j ∈ V do
4              condition ← "A_i = A_j" ;
5              Value_i ← {V.A_i − A_i − V.id} ;
6              Value_j ← {V.A_j − A_j − V.id} ;
7              output ({condition - Value_i }, Value_i);
8              output ({condition - Value_j }, Value_j);
9  Partitioner ( K : MapOutputKey, V : MapOutputValue, N : numPartitions )
10 begin
11     /** We Hash Partition only the Outputed Key part **/
12     return Hash(MapOutputKey.OutputKey) % numPartitions ;
```

Note that, during the shuffle and sort phase, MapReduce sorts and groups intermediate key-value pairs by their keys. However, it is very convenient for our purposes to also sort the intermediated values since, as detailed below, the computations inside the reducer will take benefits from such operations. Therefore, instead of implementing an additional secondary sorting within the reducer, we used the value-to-key conversion design pattern [64], which is known to provide a scalable solution for secondary sorting. This is achieved by moving intermediate values into the intermediate keys, during the map phase, to form composite keys (line 7 and 8 of algorithm 2), then we let the execution framework handle the sorting [106]. In addition, the partitioning function is customized, to take into account only the origin key-part for hashing and partitioning data. Hence, values with the same key are still assigned to the same reducer. The merging function at the reduce is also customized to group data w.r.t the original key. The reduce-input are sorted in ascending order and grouped by value, tag and id. Table 4.3 shows an example of a log file L and the outputs corresponding to the pair of attributes A_i, A_j produced

Algorithm 3: Sorted Values Centric reduce function.

Input: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: a Sorted list of Map-Output Values

Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: *PI* set Of discovered instances

```
1  Reduce_Configure
2  |L| ← count_rows_log();
3  α ← getUserThreshold();
4  β ← getUserThreshold();
5  begin
6      CMB ← build_correlated_message_buffer(V) ;
7      shared_ratio(K) ← |CMB| / |L| ;
8      if shared_ratio(K) < α then
9          PI_ψ ← compute_instances(CMB);
10         if ψ has ImbalancedPI(PI, β) then
11             output(K, PI) ;
```

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

Mapper 1 outputs			
key	val	tag	Id
A_j=A_j	C_1	A_i	m_3
A_j=A_j	C_1	A_i	m_4
A_j=A_j	C_1	A_j	m_4
A_j=A_j	C_1	A_j	m_5
A_j=A_j	C_2	A_i	m_2
A_j=A_j	C_2	A_i	m_5
A_j=A_j	C_2	A_j	m_2
A_j=A_j	C_2	A_j	m_3
A_j=A_j	C_3	A_i	m_1
A_j=A_j	C_4	A_j	m_1

Mapper 2 outputs			
key	val	tag	Id
A_j=A_j	C_1	A_i	m_{10}
A_j=A_j	C_2	A_j	m_{10}
A_j=A_j	C_3	A_i	m_6
A_j=A_j	C_3	A_i	m_8
A_j=A_j	C_3	A_j	m_6
A_j=A_j	C_3	A_j	m_7
A_j=A_j	C_3	A_j	m_9
A_j=A_j	C_4	A_i	m_7
A_j=A_j	C_4	A_i	m_9
A_j=A_j	C_4	A_j	m_8

Table 4.3: Example of a log and the outputs, w.r.t. to (A_i, A_j), of two mappers.

by two mappers that have processed respectively a split made of the first five messages (respectively, the last five messages) of the log L. Once the Reduce (algorithm 3) collects all the data, it proceeds as follows:

1. Building the correlated message buffer (line 6 of algorithm 3).

2. Pruning non-interesting conditions based on non-repeating value criterion (line 7

to 8 of algorithm 3).

3. Computing/finding the process instances entailed by the condition (line 9 of algorithm 3).

This steps are explained bellow.

Building Correlated Message Buffer : Case of (reference-base conditions). Recall that, correlated messages denoted by R_ψ are defined as $R_\psi = \{(\{x, y\})/\forall x \in A_i, \forall y \in A_j: x.val = y.val\}$. Since the input of the Reduce are sorted and grouped, only one iteration is needed to build $\mathcal{CMB}$. Moreover, message's-ids from A_i appear before those of A_j (suppose that $i < j$). So, for each new distinct value V which appears in the reduce-input, the *Reduce* creates a temporary entry in $\mathcal{CMB}$, with V as *val*. Then, it buffers ids from A_i into $IdSet1$ then those from A_j into $IdSet2$. In case of values having empty $IdSet$ (1 or 2), i.e., none pair of messages $(x, y) \in (A_i, A_j)$ satisfies $x.val = y.val = V$, then V is discarded. For key-based conditions, a new entry of $\mathcal{CMB}$ is created for each new distinct value V in the input of the reduce, and all messages satisfying $x \in A_i$, *and* $x.val = V$ are buffered to the corresponding IdSet.

val	**idset$_1$**	**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 4.4: Buffer $\mathcal{CMB}$.

Example 6 *Using as input the buffer in Table 4.3, the buffer $\mathcal{CMB}$ produced by the function Build Correlated Message Buffer is depicted in Table 4.4. Since, the input of the Reduce are sorted, then values in column val and messages in IdSets are also sorted. In the first row, $C1$ is a value which appear in both column A_i and A_j, where messages having $C1$ as value in A_i are $\{m_3, m_4, m_{10}\}$ and those having $C1$ in A_j are $\{m_4, m_5\}$.*

Pruning non-interesting conditions based on non-repeating values criterion: After the $\mathcal{CMB}$ is created, the *shared_ratio* can be computed as the ratio of the number of distinct values (number of rows in $\mathcal{CMB}$) with regard to the size of L, $shared\_ratio(\psi_{ij}) = \frac{|\mathcal{CMB}|}{|L|}$. In case of *key-based* conditions, $|\mathcal{CMB}|$ represents the

number of distinct values present in the corresponding attribute. On the other case (*reference-based*) conditions, $|\mathcal{CMB}|$ represents the number of shared distinct values between A_i and A_j. Next, candidates that do not satisfy $shared\_ratio(\psi_{ij}) < \alpha$ are pruned (line 8 of algorithm 3).

Algorithm 4: build_correlated_message_buffer

```
1  build_correlated_message_buffer ( V : list(m.A_x − A_x − m.id ) )
2  begin
3      tmpVal ← null ;
4      tmpBuffer ← ∅ ;
5      while (V.hasNext()) do
6          if tmpVal = V.A_x then
7              tmpBuffer.add(V.id, V.A_x);
8              /** if x = i put id in set_i else put it in set_j **/ ;
9          else
10             if tmpBuffer ≠ ∅ then
11                 CMB.add(tmpBuffer);
12             else
13                 tmpBuffer ← ∅ ;
14                 tmpVal ← V.A_x ;
15                 tmpBuffer.add(V.id, V.A_x);
16 return CMB
```

Computing/Finding Instances: The compute-instances function, depicted in algorithm 5, applies a DFS-like algorithm to explore the $\mathcal{CMB}$. It is called only in the case of *reference-based* conditions. It is in charge of grouping together the messages correlated by a condition ψ_{A_i,A_j} in order to form individual process instances. It takes as input a buffer $\mathcal{CMB}$ associated with a couple of attributes A_i, A_j. We recall that a buffer $\mathcal{CMB}$ produced by the function Build Correlated Message Buffer 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 messages $m1$ and $m2$ that appear in $\mathcal{CMB}$ are correlated by

Algorithm 5: Compute Instances

```
1  Compute_intances ( CMB )
2  begin
3      Stack s; for c ∈ CMB and c is not visited do
4          s.push(c);
5          c.visited ← true ;
6          n ← getNextUnvisitedNeighbor(c);
7          if n ≠ null then
8              n.visited ← true ;
9              s.push(n);
10         else
11             instances.add(s.pull());
12 return instances;
```

the condition ψ_{A_i,A_j} if and only if one of the following conditions is satisfied:

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

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

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

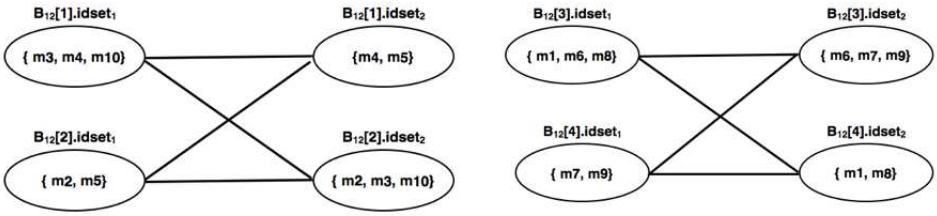

Figure 4.1: Bipartite graph of $\mathcal{CMB}$ with two connected components.

The function compute-instances can be better viewed as a computation of the connected components [15] of an undirected bipartite graph. The vertices of such a graph are the sets appearing in the columns $idset_1$ and $idset_2$ of $\mathcal{CMB}$ and the edges are constructed as follows: let $i, j \in [1, |\mathcal{CMB}|]$, then there is an edge between $\mathcal{CMB}[i].idset_1$ and $\mathcal{CMB}[j].idset_2$ if: $i = j$ (condition (i) above) , or $\mathcal{CMB}[i].idset_1 \cap \mathcal{CMB}[j].idset_2 \neq \emptyset$ (condition (ii) above). The condition (iii) is achieved by the computation of the connected components of this graph. Each connected component corresponds to a discovered process instance.

However, in the case of *key-based* conditions, each set of correlated messages corresponding to a distinct value forms a process instance. In other words, each vertex in the graph forms a connected component.

Example 7 *Figure 4.1 depicts the graph corresponding to the buffer $\mathcal{CMB}$ of Table 4.4. 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* = $\{m_1, m_6, m_7, m_8, m_9\}$

Finally, using user provided threshold β, the non interesting instances are pruned (line 10 of algorithm 3). By computing 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_2 of condition ψ are defined. The PI_ratio is compared with β and the conditions that do not satisfy the criteria are pruned.

Discussion In this section we presented our first algorithm called sorted values centric algorithm devoted to discover candidate atomic correlation conditions. The algorithm relies on one MapReduce job, hence a less overheads involved by the framework to schedule tasks. In addition, It sorts the intermediate data to build the correlated message buffer in one iteration and efficiently computes the process instances. In fact, sorting data requires the use of *value to key* pattern (values are appended to the key to form a composite key) which make the map-outputs size twice larger. Therefore, this fact causes an important overhead while the data are transferred between Map and Reduce nodes and may affect the algorithm's performance. To fix such problem we introduce Hashed values centric algorithm in the following.

4.2.3 Hashed Values Centric Algorithm

In order to avoid generating and transferring intermediate data with duplicated values, we propose the **H**ashed **V**alues **C**entric (HVC) algorithm. HVC, depicted at algorithm 6 and 7, processes each candidate atomic correlation condition independently. Also, it can be implemented in a single MapReduce job. The Map generates the set of all possible candidate atomic conditions from the set of attributes in $\mathcal{L}$. It ensures that each pair will be allocated to the corresponding Reduce by assigning a single key to each pair. The main difference w.r.t SVC lies in the keys used to distinguish the target Reducers. Unlike SVC algorithm, in HVC composite keys are not used. Therefore, the map-output data size is not duplicated and, also, not sorted. Table 4.5 shows an example of the outputs of log $\mathcal{L}$ in 4.3 processed by two mappers executed the map in algorithm 6. Once the *Reduce*, depicted in algorithm 7, receives it corresponding data, it proceeds as follow :

1. Build the correlated message hash buffer (line 3 of algorithm 7).
2. Pruning non-interesting candidates (line 5 of algorithm 7).
3. Computing process instances (line 6 of algorithm 7).

These steps are described below.

Build Correlated Message hash Buffer : The function is depicted in algorithm 8. It aims at grouping together message-ids having same values, and separate those coming from A_i from those coming from A_j in different sets (IdSet1 to A_i and IdSet2 to A_j). Since, the input of the Reduce are not sorted, a hash table is used to store the

Algorithm 6: Hashed Values Centric Map_function.

```
1 Map ( K : unused, V : {A_1, A_2, ..., A_n, id} )
2 begin
3     foreach A_i ∈ V do
4         foreach A_j ∈ V do
5             OutputKey ← A_i, A_j ;
6             OutputValue_i ← {V.A_i − A_i − V.id} ;
7             OutputValue_j ← {V.A_j − A_j − V.id} ;
8             output ({OutputKey }, OutputValue_i);
9             output ({OutputKey }, OutputValue_j);
```

Algorithm 7: Hashed Values Centric Reduce_function

```
1 Reduce ( K : ψ_{A_i,A_j}, V : list(m.A_x − A_x − m.id ) )
2 begin
3     CMB ← build_correlated_message_hash_buffer(V) ;
4     shared_ratio(K) ← |CMB| / |L| ;
5     if shared_ratio(K) < α then
6         PI_ψ ← compute_instances(CMB);
7         if ψ has ImbalancedPI(PI, β) then
8             output(K, PI) ;
```

Algorithm 8: build correlated message hash buffer

```
1 build_correlated_message_hash_buffer ( V : list(m.A_x − A_x − m.id ) )
2 begin
3     hashtable CMB ;
4     while (V.hasNext()) do
5         tmpBuffer ← CMB.lookup(V.A_x);
6         tmpBuffer.add(V.id, V.A_x);
7         CMB.add(tmpBuffer);
8         tmpBuffer ← ∅
9 return CMB
```

correlated messages and, by consequent, several iterations are required to achieve this task. Moreover, an additional step is needed to clean the buffer by deleting entries with empty IdSets. This stage is less performance then that of *SVC* algorithm.

Mapper 1 outputs			
key	val	tag	Id
A_j=A_j	C_3	i	m_1
A_j=A_j	C_1	i	m_3
A_j=A_j	C_4	j	m_1
A_j=A_j	C_2	j	m_2
A_j=A_j	C_2	j	m_3
A_j=A_j	C_2	i	m_2
A_j=A_j	C_1	i	m_4
A_j=A_j	C_2	i	m_5
A_j=A_j	C_1	j	m_5
A_j=A_j	C_1	j	m_4

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

Table 4.5: Example of a log and the outputs, w.r.t. to (A_1, A_2), of two mappers.

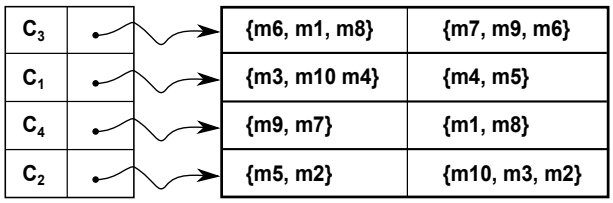

Figure 4.2: Correlated Messages Hash Buffer

Example 8 *Using as input the buffer at Table 4.5, the buffer $\mathcal{CMB}$ produced by the function* Build Correlated Message hash Buffer *is depicted at Figure 4.2. Many iterations are required to fill the buffer.*

Pruning Non-interesting Candidates : Candidate conditions non-satisfy the criterion $shared\_ratio(\psi) < \alpha$ are pruned.

Compute Instances : A DFS-like algorithm is applied to compute the transitive closure of the $\mathcal{CMB}$ and deduces the process instances entailed by each candidate conditions. The same algorithm is used to compute discovered instances as in algorithm 3

with few critical changes. The main modification is applied to getNextUnvisitedNeighbor(). Since data are not sorted, checking for the intersection requires, in worst case, $2 \times |idSet1| \times |idSet2|$ operations. In order to avoid a such high number of operations, a hash function is used as following:

- Read all element in IdSet1 and store them by their hash code in temporary hash set T.
- For each element id in Idset2 computes its hash code and proceed as follow :
  - Stores (id) and return false, If *hash(id)* doest not exist,
  - Return true, otherwise.

Using this property we reduce the number of operation to $2 \times (|idSet1| + |idSet2|)$.

Finally, conditions that partition the log into a few number of long instances or a high number of long instances are pruned using the Imbalanced_PI criterion.

Discussion In this section we present the Hashed Value Centric algorithm used for discovering candidate atomic correlation conditions. The algorithm relies on one MapReduce job. It avoid using *key-to-value* pattern for sorting map-output data. Therefore, it reduces the overheads involved by duplicating values in the SVC algorithm during the transferring data. However, it requires several iterations to build the correlated message buffer and an additional step to clean it.

One problem with Sorted Values Centric or Hashed Values Centric algorithms is due to the existence of non-correlating values. Therefore, not every message from A_i will be correlated with one or many messages from A_j for a given condition ψ_{A_i,A_j}. Such non-correlating values are identified only after building $\mathcal{CMB}$ which involves a wasted time to eliminate such values. To cope with this problem, we propose the per-split correlated messages algorithm which distributes the computation of the $\mathcal{CMB}$ over processing nodes and anticipates such non-correlating values before constructing the process instances.

4.2.4 Per-Split Correlated Messages Algorithm

The per-split correlated message ($PSCM$) algorithm has two phases, each corresponding to a separate MapReduce jobs. This algorithm is introduced to parallelize the computation of the $\mathcal{CMB}$ presented in the two previous algorithms. In case of referenced-based

conditions, not all values are included into the intersection of the distinct values of A_i and A_j. This property is stated more formally as $\{ \exists \mathcal{V}, \mathcal{V} \in distinct(A_i) \cup distinct(A_j)$ and $\mathcal{V} \notin distinct(A_i) \cap distinct(A_j)\}$. In this case, $\mathcal{V}$ should be ignored and all messages having this value should be eliminated. This is done by the phase one of the algorithm, which can be seen as a pre-processing step. We describe below the two phases.

Phase 1: The Map function adds the attribute values to the outputted key (the key refers to the condition name) (line 5 and 6 of algorithm 9). Therefore, it ensures that all messages from the same condition having the same value will be allocated into a single Reduce. The **Reduce** will receive the sets of message-ids having the same value. Each Reduce will produce a single row of the $\mathcal{CMB}$ for each condition. It fills the row buffer by putting message-ids from A_i into IdSet1 (resp. A_j into IdSet2). Rows with empty IdSet1 or empty IdSet2 are ignored. This step is similar to the standard SQL query (self-join of Log $\mathcal{L}$) where the join is computed for each pair of attributes A_i, A_j.

Phase 2: The **Map** function is the function identity, the output-key is the condition name and the output-value is a row of $\mathcal{CMB}$. The **Reduce** function receives all rows of the same condition. First, it groups them in single buffer $\mathcal{CMB}$, then it applies the compute_instance() function as in HVC to compute the set of instances entailed by each condition. Finally, it prunes non-interesting condition based on Imbalanced_PI criterion.

4.3 Handling Reducers Insufficient Memory

As seen previously to evaluate a condition, the reducer in the SVC, HVC and $PSCM$ algorithms receives as input a list of tuples corresponding to a projection of two column (A_i and A_j) on the log L i.e, it receives all values present in two attributes (for reference-based condition). To proceed with computing the correlated message buffer, the entire input data must be loaded in the main memory. It is worth noting that we already eliminate duplicated messages from the correlated messages set using the data structure T_1 and T_2 (see example 5). However, it may happen that either Reducer-input data or the produced ($\mathcal{CMB}$) does not fit entirety in main memory. So if this is the case, then we can use, as alternative, a disk-based solution to cope with this issue. In the following we present an extension to our algorithms for the case where data do not fit in main-memory.

Algorithm 9: Per-Split Correlated Messages Algorithm phase1

Input: $\mathcal{K}$: unused, $\mathcal{V}$: a record from the log file
Output: $\mathcal{K} : A_i = A_j, \mathcal{V} : \{IdSet1\}, \{IdSet2\}$

```
1  Map ( K : unused, V : {A_1, A_2, ..., A_n, id} )
2  begin
3      foreach A_i ∈ V do
4          foreach A_j ∈ V do
5              OutputKey1 ← A_i, A_j+ V .A_i ;
6              OutputKey2 ← A_i, A_j+ V .A_j ;
7              OutputValue_i ← {A_i − V. id} ;
8              OutputValue_j ← {A_j − V. id} ;
9              output ({OutputKey1 }, OutputValue_i);
10             output ({OutputKey2 }, OutputValue_j);
```

Input: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: a set of message-ids having same value
Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: a row of $\mathcal{CMB}$

```
11 Reduce ( K : ψ_{A_i,A_j}, V : list(A_x-m.id ) )
12 begin
13     while (V.hasNext()) do
14         IdSet_x ← V.A_x.id ;
15         B.setIdSets(IdSet_1, IdSet_2) ;
16     output(K, B) ;
```

4.3.1 Disk-Based Extension

A disk-based extension is used to handle insufficient memory at the reduce stage, this may require a large number of I/O operations. A critical change are performed to the $compute\_correlated\_messages\_buffer$ (presented in algorithm 4) function. Instead of storing each row of the buffer into main memory, the function will store it in a separate file into a disk with the index position as a name. Figure 4.3 shows an example of how $\mathcal{CMB}$ rows are stored and processed. In the case of *key-based* condition, if the condition is interesting, all rows are streamed directly to HDFS, since there is no need to compute transitive closure to find instances. Otherwise in case of *reference-based* condition, and at the *compute instances* step, all rows are stored into local disk except, for instance, the first one. Then, the reduce reload rows later to check whether two rows (rows) belong to the same process instance. If the intersection holds between two rows, saying row 1 with

Algorithm 10: Per-Split Correlated Messages Algorithm phase2

Input: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: a row of $\mathcal{CMB}$

Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: a row of $\mathcal{CMB}$

`/* The Map Function is the Map Identity. */`

Reduce_Function

Input: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V} : \mathcal{CMB}$

Output: $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V}$: *PI* set Of discovered instances

`Reduce_Configure`

$|\mathcal{L}| \leftarrow$ count_rows_log();

$\alpha \leftarrow$ getUserThreshold();

$\beta \leftarrow$ getUserThreshold();

`Reduce (` $\mathcal{K} : \psi_{A_i,A_j}$, $\mathcal{V} : \mathcal{CMB}$ `)`

begin

  PI $\leftarrow$ compute_instances($\mathcal{CMB}$);

  PI_Ratio($\mathcal{K}$) $\leftarrow \frac{|PI|}{max|A_i,A_j|}$;

  if $PI\_Ratio(\mathcal{K}) \leq beta$ **then**

    output($\mathcal{K}$, *PI*) ;

  else

    write($\mathcal{K}$ is not interesting condition based on Imbalanced_PI creterion) ;

row 3, then row 3 is marked as *visited* and rows 1 is removed from main memory and stored into disk. The index of rows 1 is pushed into the stack and next row is loaded to check intersection with row 3, and so on. When there is no rows to load or no intersection, the index in the tail of the stack is pulled and the corresponding row is merged with the row present in memory. As presented in the Figure 4.3 (find process instance step), a stack is used to keep track of all connected rows that form a single process instance.

4.3.2 Multi-Pass Process Instances Discovery Algorithm

Various data analysis techniques requires *iterative* computations, like PageRank [78], HyperText-induced topic Search [61], clustering [58], neural network analysis [49], social network analysis [103], recursive relational queries [18, 19, 11, 10], internet traffic analysis [70, 71] and Apriori-based algorithms [65]. These techniques have a common particularity: data are processed iteratively until the computations satisfies a convergence or terminating condition. We propose an iterative algorithm, illustrated in algorithm 11, to deal

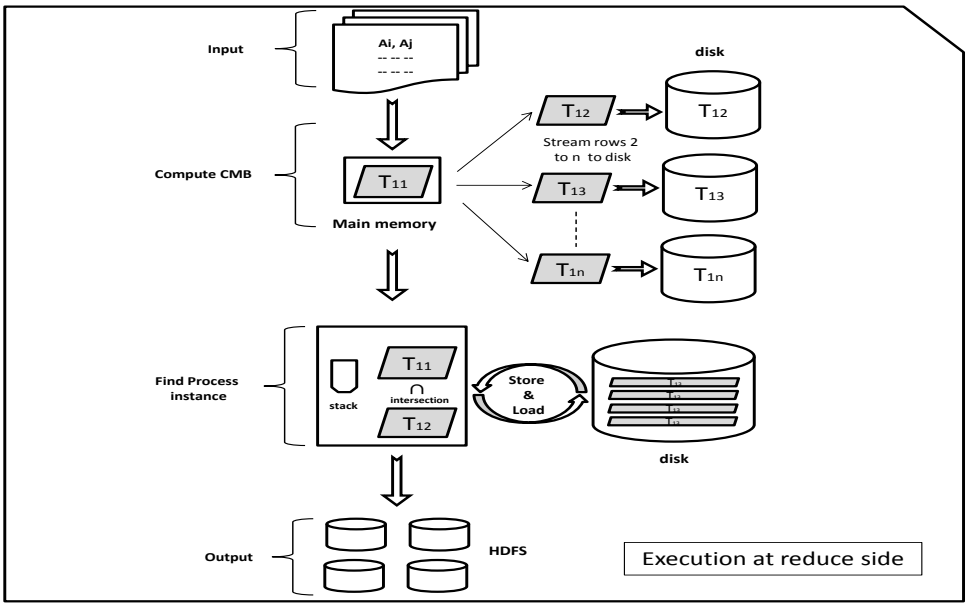

Figure 4.3: Data-flow at the reduce side.

with the problem of computing process instances at the Reduce side in case of limited memory. Indeed, the computation performed by algorithm 11 is similar to the standard calculation of transitive closure on undirected bipartite graph. As mentioned previously, we use a *depth-first traversal* algorithm to discover process instances for *reference-based* conditions.

This algorithm is dedicated to *reference-based* conditions, where the computation of the transitive closure is needed to construct process instances. Furthermore, the algorithm scales well with regard to the size of the log and the number of distinct values that appear on each attribute in the input log files. Moreover, the workload can be, evenly, balanced over the cluster nodes. As its name indicates the algorithm is iterative and consists of, at most, $\mathcal{O}(\log_2(r))$[1] passes and stops when intersection between all rows is empty, meaning that rows can not be merged together. Each pass is a MapReduce job, it merges at least two rows of the $\mathcal{CMB}$. The input of the algorithm is $\mathcal{CMB}$ based on T_2 data structure and we expect that there is intersection between idSets, in other words there is transitivity and the process instances are correlated using *reference-based* condition. The *multi-pass* algorithm, in spirit, is similar to a self-join algorithms performed on $\mathcal{CMB}$. Where, each Map function, outlined in algorithm 12, works on a part of the $\mathcal{CMB}$. First,

[1] r corresponds to the number of rows in $\mathcal{CMB}$ and it refers to the number of shared distinct values between two attributes A_i, A_j.

it loads the entire $\mathcal{CMB}$ into local storage. Then, reads, from the HDFS, a row_x from the corresponding part and probes the $\mathcal{CMB}$ to perform a self-join-like operation. It checks whether for each row_y the condition (ii) in 12 is satisfied (line 4 of algorithm 12). Finally, the Map outputs the row_y as key while the row_x as value if the intersection is not empty, otherwise row_x is discarded. At the Reduce side, associated with each key (row) their will be a set values (rows). The Reduce joins all the rows in the values list with that presented in the key and produces a new row (line 4 of algorithm 13). If none of Reducer does not produce a new row the entire algorithm terminates. Otherwise, the Reduce outputs are used as input for the next iteration and the $\mathcal{CMB}$ is broadcasted to all nodes via *DistributedCache* (line 8 algorithm 11). A **combiner** function can be used in conjunction with the Map to consolidate intermediate values to minimize the cost of transferring data over the network.

Algorithm 11: multi-pass main algorithm.

Input: $\mathcal{K}$: unused, $\mathcal{V}$: a row from $\mathcal{CMB}$
Output: $\mathcal{K}$: $index$, $\mathcal{V}$: row

```
1 while CMB ≠ ∅ do
2     store CMB in distributedCache ;
3     Map() function ;
4     CMB ← Reduce() function ;
```

Algorithm 12: multi-pass map function.

```
1 Map_function
  Input: K : unused, V : a row from CMB
  Output: K : index, V : row_x
2 read CMB from distributedCache ;
3 foreach row_i ∈ CMB do
4     if (V.IdSet1 ∩ row_i.IdSet2) OR (V.IdSet2 ∩ row_i.IdSet1) then
5         output(row_i, V);
```

Algorithm 13: multi-pass reduce function.

```
1 reduce_function
  Input: 𝒦 : a row , {𝒱 } : a list of rows
  Output: 𝒦 : unused, 𝒱 : a new row
2 outputrow.IdSet1 ← key.IdSet1 ;
3 outputrow.IdSet2 ← key.IdSet2 ;
4 foreach v ∈ 𝒱 do
5     outputrow.IdSet1 ← outputrow.IdSet1 ∪ v.IdSet1 ;
6     outputrow.IdSet2 ← outputrow.IdSet2 ∪ v.IdSet2 ;
7 output(null, outputrow);
```

4.4 Evaluation Of The Proposed Algorithms

This section shows the evaluation of the proposed algorithms and estimates their cost using the cost model introduced in section 2.6.2. The theoretical number of all possible atomic correlation condition is $\mathcal{N}=\frac{k(k+1)}{2}$, where k is the number of attributes in $\mathcal{L}$. We assume that each condition is processed by a single node.

4.4.1 Complexity Analysis

Unlike the complexity mentioned in [74] which is $O\ (N.|\mathcal{L}|^2)$, the algorithms presented above are in parallel and each condition is processed independently from the others. Therefore, the time complexity to explore all the space of correlation conditions is $O\ (p)$. Since, the main computation is done by reducers, we omit the map complexity. p is different from one algorithm to another. It consists of the sum of (i) the time complexity of computing correlated messages. (ii) The time complexity of computing instances.

- In the case of sorted data, The worst case time complexity of building correlated message buffer ($\mathcal{CMB}$) is $O\ (|\mathcal{L}|)$. The worst case of computing instances is in $O\ (d \times s \times 2)$ where d is the number of distinct values of $A_i \cap A_j$ and s is size of the largest IdSet.

- In the case of non sorted data, computing correlated messages takes $O\ (\mathcal{L}^2)$, the case of building the hash table, for each value we need to look for it at the hash table. The worst case for computing instances has complexity of $O\ (d \times s^2)$.

4.4.2 Cost-Model-Based Analysis

In this section we evaluate the algorithms based on the cost model introduced in section 2.6.2. MapReduce framework introduces some overheads that should be taken into account to estimate the runtime of algorithms. Almost all of its operations are done in background and transparent to the user. These overheads are mostly affected by the amount of data, read, sorted or transferred. Assume that the number of messages in $\mathcal{L}$ is $|\mathcal{L}|$ and k is the number of attributes in $\mathcal{L}$. Table 4.6 shows the estimated cost of each step of the three algorithms.

Discussion: As shown at Table 4.6, the estimated performances differences of the proposed algorithms differs significantly. Each algorithm has a strength and weakness. Consider *SVC* algorithm, it has the best complexity for computing correlated messages and the process instances, but it is less effective during map_sort, transferring data and reduce_sort phases because of the large size of intermediate data. Unlike *SVC*, *HVC* is less efficient for computing correlated messages, but it outperforms *SVC* during the transferring data phase. The third algorithm uses two steps and, hence, involves more overheads by the framework and affects the algorithm's performance. However, this overheads may be negligible and the *PSCM* outperforms the other algorithms in many cases, as we will see in the experimental evaluations. As a result of this analysis, we can notice the following observation:

- *SVC* is best in computing correlated messages and process instances.
- *HVC* has the lowest overheads during sorting, transferring and merging intermediate data.
- *PSCM* uses more parallelism to compute correlated messages and can use more resources as input data gets larger.

These analysis are validated by the experimental evaluation of the proposed algorithms in the next section.

[2]Map-output-record : the number of record outputted by the Map
[3]Map-output-size : the size of the Map outputted
[4] Passes = MergeSpillsPass(Spills, Factor)

Table 4.6: Algorithms Estimated Costs.

steps	SVC	HVC	PSCM-p1	PSCM-p2
T_{read}	$C_r \times \frac{\lvert\mathcal{L}\rvert}{m}$			$C_r \times \frac{\mathcal{CMB}}{m}$
MOR[2]	$\frac{\lvert\mathcal{L}\rvert}{m} \times k^2$			$\frac{\lvert\mathcal{CMB}\rvert}{m}$
kv_width	$2 \times (val + id + tag) + key$	$val + id + tag + key$	$val + id + tag + key$	row of $\mathcal{CMB}$
MOS[3]	$MOR \times kv\_width$			$\frac{\mathcal{CMB}}{m}$
# Spills	$\frac{mor}{B \times Q \times P \times 2^{16}}$			
SpillSize	$\frac{mos}{\lvert Spills\rvert}$			
T_{map_sort}	$C_l \times spillSize \times 2[\lvert spills\rvert + Passes]$[4]			
T_{tr}	$C_r \times m \times mos$			
T_{sort_reduce}	$C_l \times kv\_width \times \lvert\mathcal{L}\rvert \times 2\lceil(\log_{Factor}(m))\rceil$			
T_{reduce_write}	$C_r \times \mathcal{CMB}_\psi$			

4.5 Experimental Evaluation

In this section, we describe the performance evaluation of the proposed algorithms for candidate atomic correlation conditions. To better understand the performance of parallel algorithms we measured their absolute time as well as their speed up and scale up [36].

4.5.1 Environment

We ran experiments on a cluster of 5 virtual machines. Each machine has one AMD Opteron processor 6234 2.40GHz with four cores, 4 GB of RAM, and 50 GB of Hard disks. Thus the cluster consists of 20 cores and 5 disks. However, one of the five nodes is used to run the master daemon (JobTracker) to manage the Hadoop jobs and the Hadoop Distributed Files System (NameNode) to manage the input and output files. Each node is configured with Ubuntu 12.04 LTS 64-bits operating system, Java 1.7 with a 64-bits server JVM, and Hadoop 0.20.2. Each node run four Maps and four Reduces task. We run our algorithms over the following datasets.

4.5.2 DataSets

We run our algorithms on 3 different datasets:

- **SCM.** This dataset is the interaction log of SCM [4] business service, developed based on the supply chain management scenario provided by WS-I (the Web Service Interoperability organization). There are eight Web services realizing this business service. The interaction log of Web services with clients was collected using a real-world commercial logging system for Web services, HP SOA Manager. The services in SCM scenario are implemented in Java and use Apache Axis as SOAP implementation engine and Apache Tomcat as Web application server [74].

  This dataset has 19 attributes and 4050 messages, each corresponding to an activity invocation. This dataset mainly provides an example of a system, whose its instances are correlated in a chain-based form.

- **Robostrike.** One month collected datasets from a multi-player on-line game service named *Robostrike*[5]. Players exchange XML messages with game server con-

[5] http://www.Robostrike.com/

taining several activities that can be performed during a game session. This dataset has 18 attributes and more than 1.8 million messages.

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

We increase the data by factor of 100, 500 and 1000, we refer to the data set as "SCM" $\times\, n$ where $n \in \{100, 500, 1000\}$ represents the increase factor. Before starting experiments we extract attributes and their values from XML documents using ETL-like preprocessing and represent them as event tuples in csv files. This files are then grouped into buckets with different size (e.g. RS10K contains one millions events of RobotStrike dataset). Next, we upload them into the hadoop distributed file system.

For all the experiments, we tokenize the data by values and we use the attribute names to represent the correlation condition (e.g. $A_1 = A_2$), the lower bound threshold and upper bound threshold used in *non-repeating values* criterion are 0.01 and 0.8 respectively. We define, also, 0.5 as a threshold to prune correlation conditions based on $Imbalanced\_PI$ criterion.

4.5.3 Experiments

As a first experiment, we study the main differences between the 3 algorithms (Sorted Values Centric, Hashed Values and Per-Split Correlated messages algorithms). We run the algorithms for three different size of data SCM$\times$ n (where n $\in$ {100, 500, 1000}), we breakdown the total execution time into two main steps (Map and reduce). Furthermore, the reduce is, also, subdivided into 3 steps (shuffle, sort and CMB+CI). $\mathcal{CMB}$ and CI denote respectively computing *correlated message buffer* and *compute instances*.

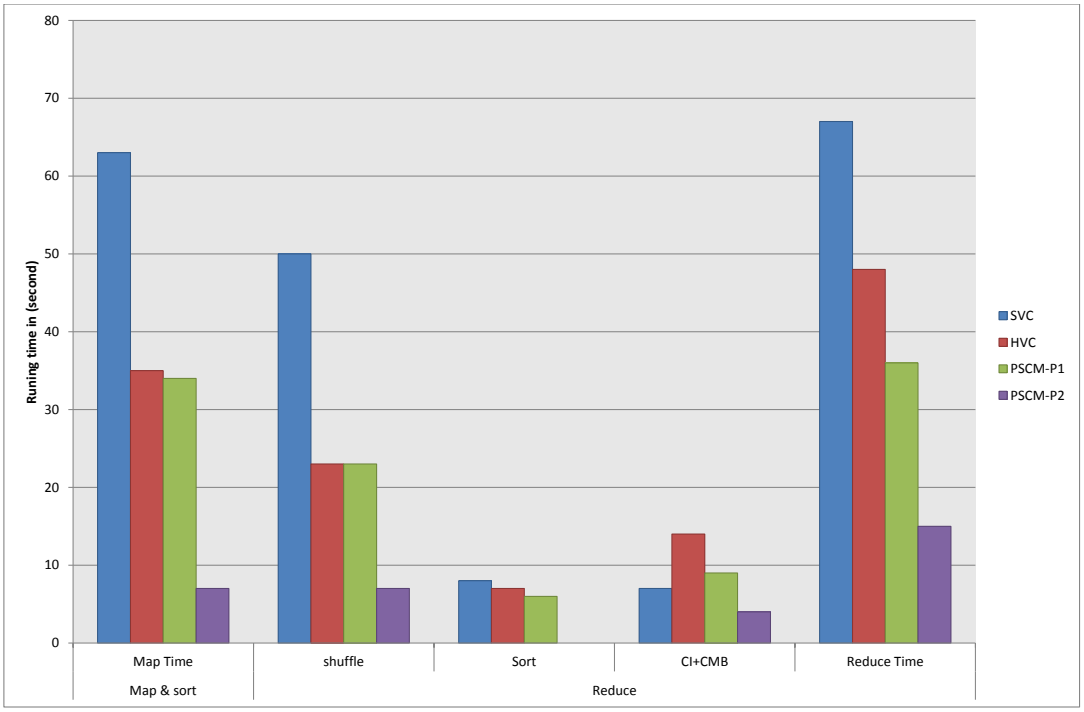

Figure 4.4: Time breakdown SCM×100.

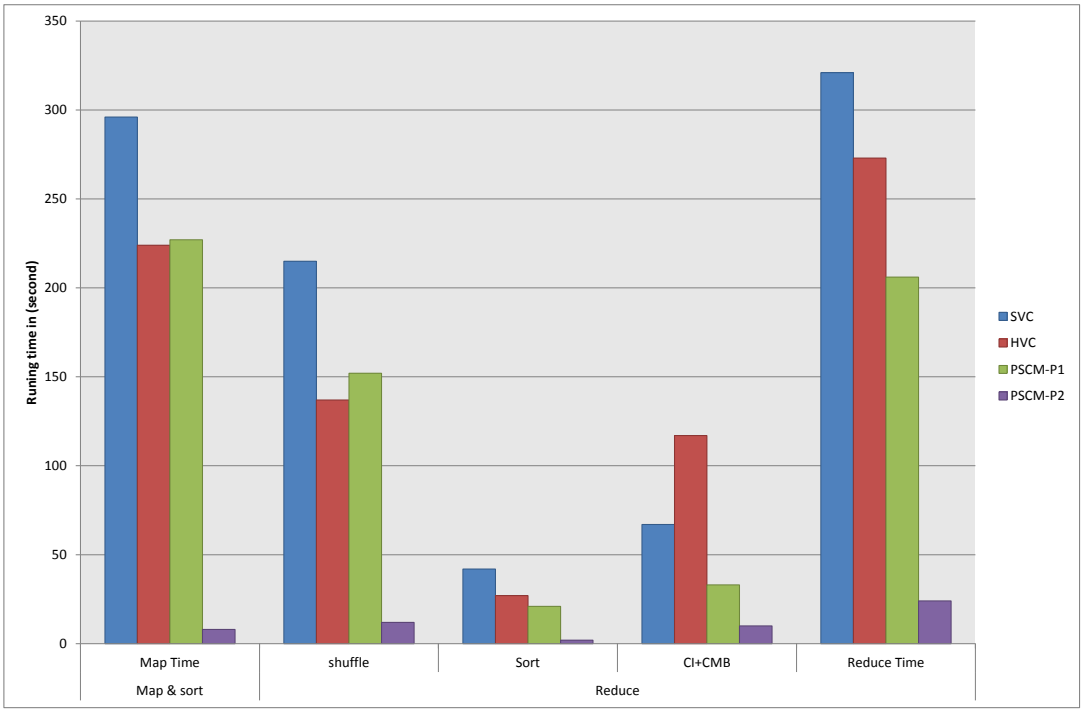

Figure 4.5: Time breakdown SCM×500.

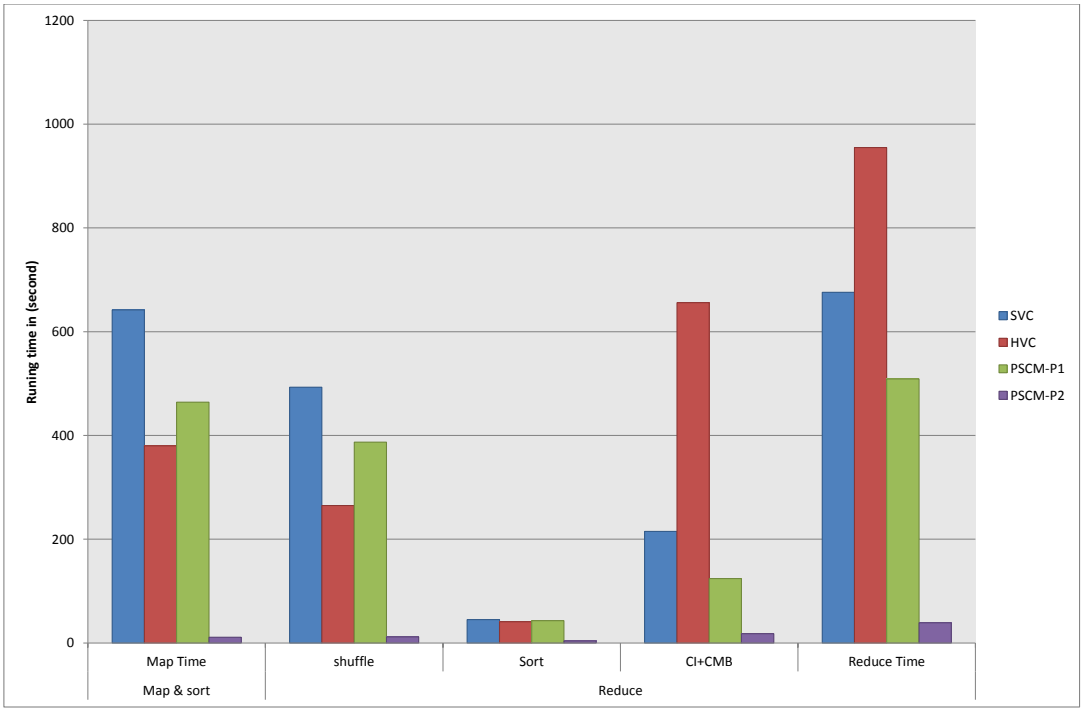

Figure 4.6: Time breakdown SCM×1000.

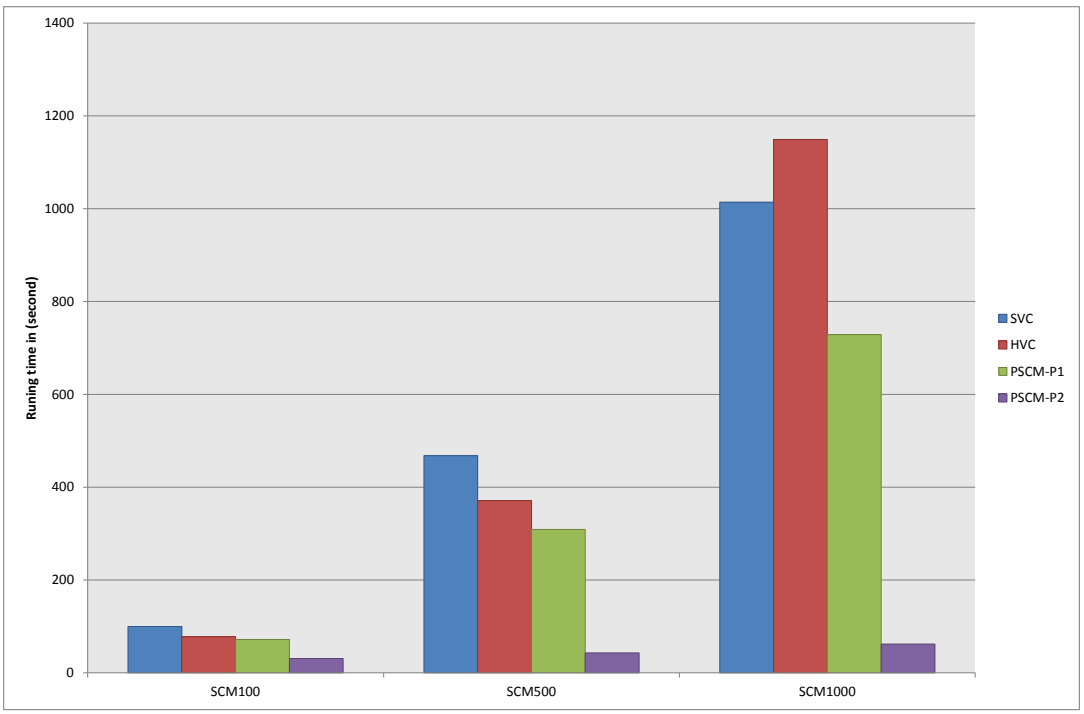

Figure 4.7: Total Run time on SCM×n datasets.

Figures 4.4, 4.5 and 4.6 show the execution time proportion of each step on 5-nodes cluster for different dataset size (represented by the factor n). Per-Split Correlated Messages consists of two MR jobs, the first phase is denoted as *PSCM-p1* and the second as *PSCM-p2*.

Starting with the *map* phase, we observe that *SVC* is always the most expensive algorithm. This is because the *SVC*'s map-outputs size is equivalent to twice as those of *HVC* and *PSCM*. This fact implies moving a large amount of data over network during the shuffle phase. A significant difference between *HVC*'s map and *PSCM-p1*'map can be observed in Figure 4.6. This is caused by the difference in the map-selectivity in each algorithm. In other words, The *PSCM-p1*'s map produces a large number of keys than *HVC'*map. This difference may not be seen in case of small data size. Moving to the *reduce* phase, the *shuffle* and *sort* phases are directly affected by the map-output data. Therefore, we do not observe significant changes in performance (*SVC* is always the worst). On the other hand, and during *CMB+CI* phases *SVC* shows better performance than *HVC* and *PSCM* in Figure 4.4. This is because *SVC*'Reducers handle a sorted data. But in Figures 4.5 and 4.6, *PSCM* was the best because it divides this step on two stages.

Finally, Figure 4.5.3 shows the total execution time of the three algorithms. For the n equals 100 and 500, *HVC* was the best algorithm, this is because *SVC* is less efficient due to the large size of intermediate data, and *PSCM* has an additional overheads due to scheduling another MapReduce step. Whereas, for the largest dataset size *PSCM* was the best. This means that the framework overheads became negligible when the size of the workload increased.

In order to evaluate the scale up and speed up of the algorithms, we performed two different experiments. First we fix the number of nodes and we vary the size of the input data (Robostrike), then we fix the input data (Robostrike and SCM×500) size and we vary the number of processing nodes.

Figure 4.9 presents the running time gathered from executing the 3 algorithms on *RobotStrike* dataset. We start with 200k messages as input log size then we vary by adding 200k for next steps until 1800k, as the data increased the running time of the three algorithms increase linearly. The x-axis shows the size of input data in messages. The y-axis shows the elapsed time in second, and in Figure 4.8 shows the amount of intermediate-data transferred over the network between nodes in the shuffle phase (in

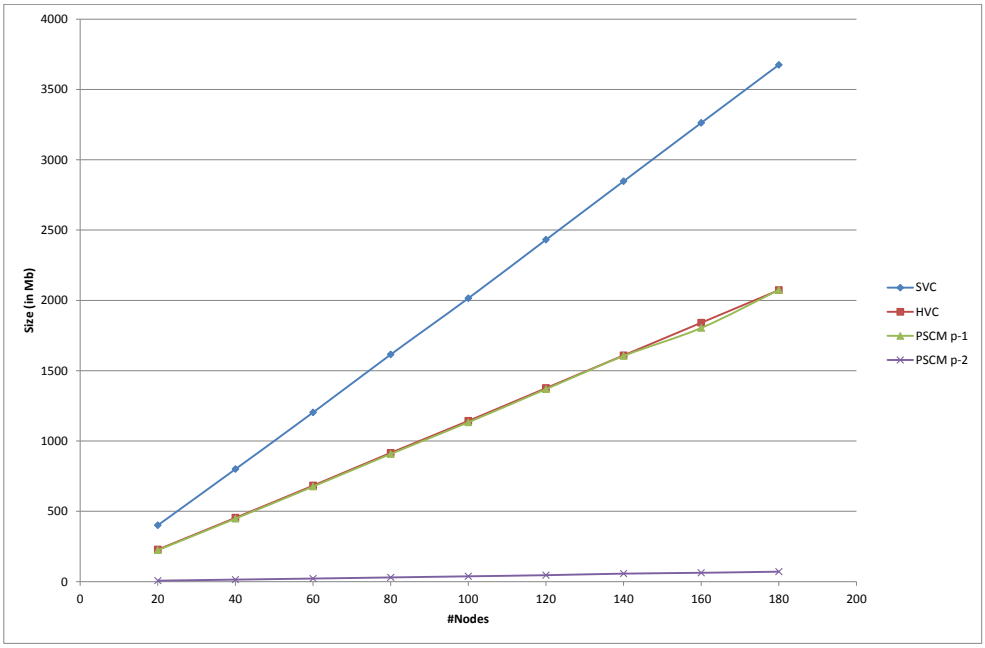

Figure 4.8: The evolution of the amount of data moved over the network.

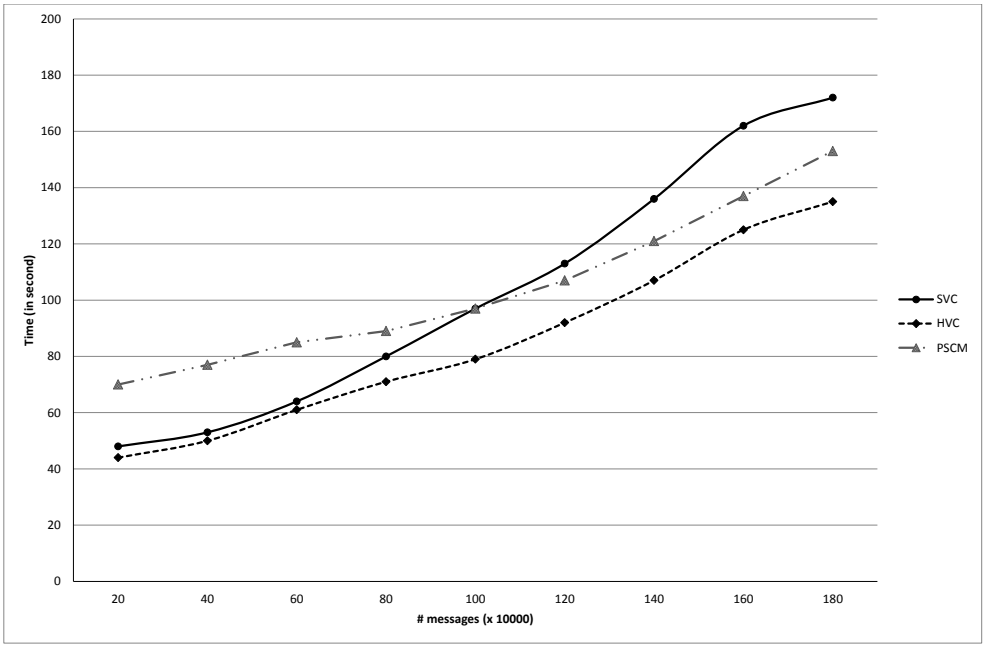

Figure 4.9: Running time of three algorithms on different data size (RobotStrike).

megabytes).

In Figure 4.9. Moving from left to right, we observe that as the data size get bigger the time for the three algorithms increase linearly. The main observations seen in Figure 4.9 are: (1) *Hashed Values Centric* algorithm denoted as *HVC* was the best algorithm for all the size of data. (2) *Sorted Values Centric* algorithm denoted as *SVC* has approximately the same performance as *HVC* until 800 thousands messages where it increases significantly. This is because the cost of sorting and shuffling data to nodes over network started to dominate. *SVC* was the worst algorithm from 1000k to 1800k messages. Finally, (3) *Per-Split Correlated Messages* algorithm was the worst algorithm for small size of data and this is mainly duo to the additional cost of overheads involved by writing and reading into HDFS the result of the first stage and the cost of scheduling new tasks. But, when the data size get bigger ($> 1000\ k\ messages$) *PSCM* outperforms *SVC* because dividing the work into two stages becomes an advantage and avoid processing a large amount of data.

In Figure 4.8, we observe that the 3 algorithms have constant increased curves, where *SVC* has the largest amount of data moved over network. This fact is caused by adding the value to the key-part to sort the map-outputs. *HVC* and *PSCM-p1* have the same size, since they don't replicate the value. *PSCM-p2* has the smallest intermediate data size, this is due to eliminating non used messages in *PSCM-p2*.

Robostrike dataset: In Figure 4.10 we plot the running time of the three algorithms on the same Robostrike data size and we increase the size of the cluster from 1 to 5 nodes. We can see that the *HVC* was the best for all the settings from 1 to 5 nodes (as in previous graphs). The *PSCM-p1* scales faster, this is due to the balancing of the high number of map-output groups (keys) over nodes. Unlike in *SVC* and *HVC*, the routing keys in *PSCM* are numerous and therefore requires the same number of reduce instances. In addition, the reducers in *PSCM-p1* will receive only message ids of a single condition and having same value. On the other hand, *PSCM-p2* has the smallest speed up. The main raison for the poor speed up of *PSCM-p2* was due to: *(1)* the low size of input data (already eliminated by *PSCM-p1*), *(2)* it only groups already computed correlated messages and applies the pruning phase. The three algorithms time decreases as the number of nodes increase.

Figure 4.11 shows the same result as in Figure 4.10 but plotted on *"relative scale"*.

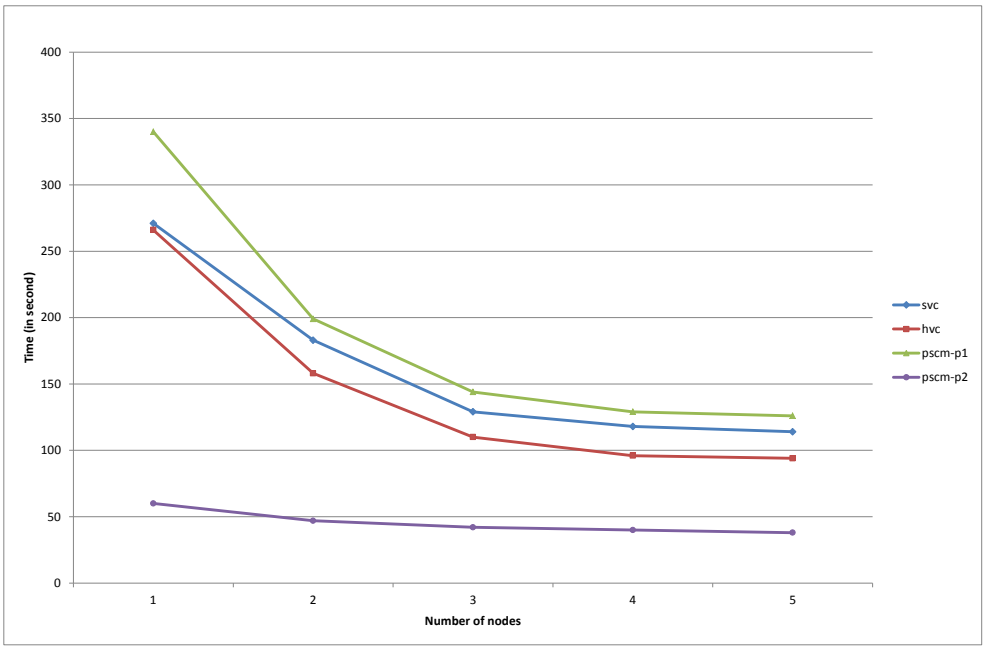

Figure 4.10: Running time of the algorithms for Robostrike data set on different cluster sizes.

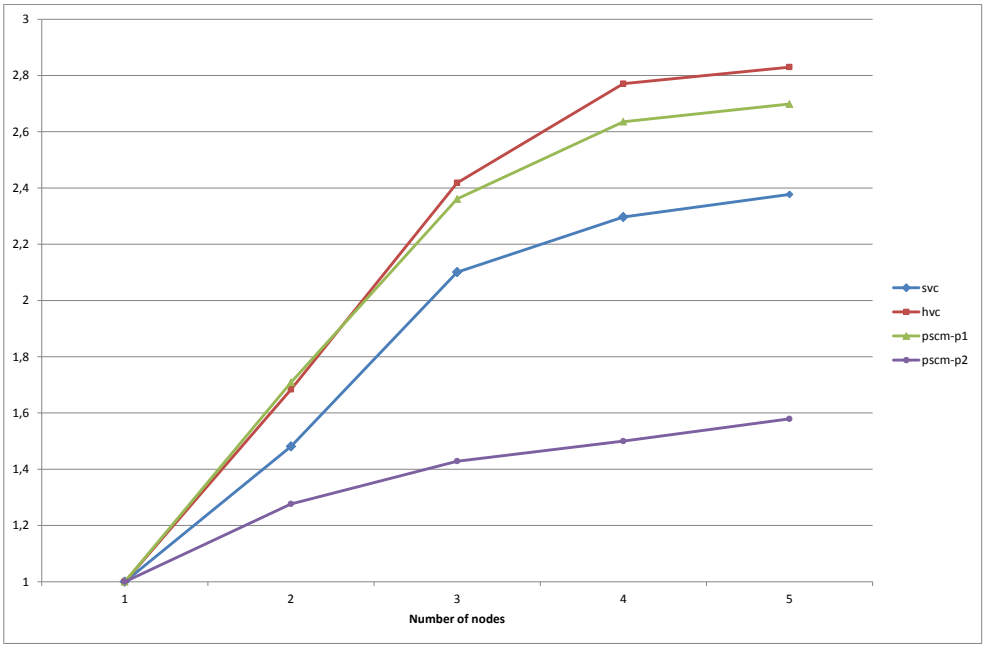

Figure 4.11: Relative running time of the algorithms for Robostrike data set on different cluster sizes.

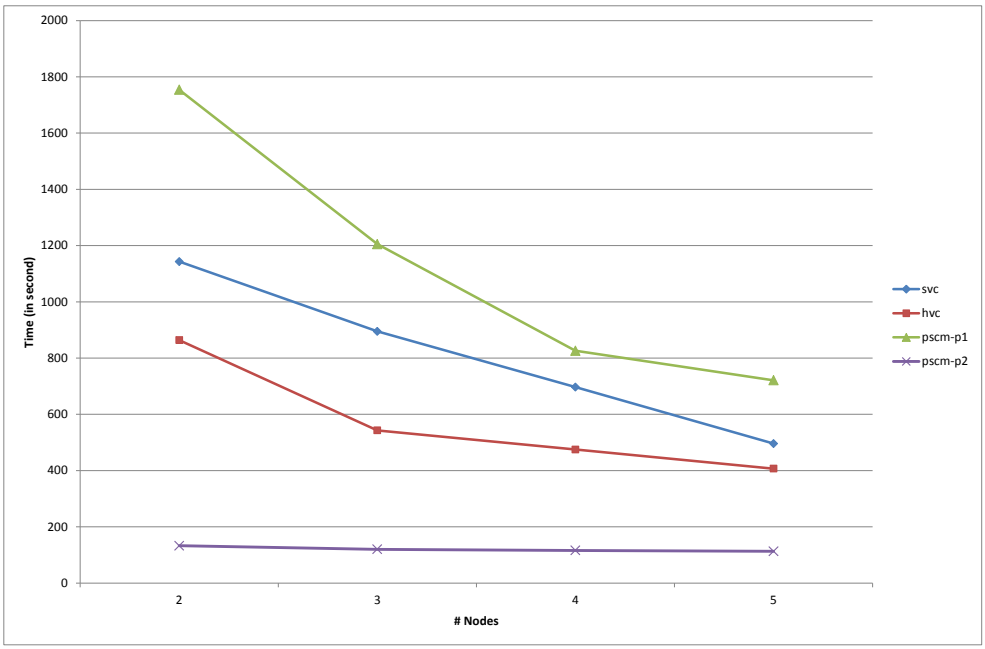

Figure 4.12: Running time of the algorithms for SCM×500 data set on different cluster sizes.

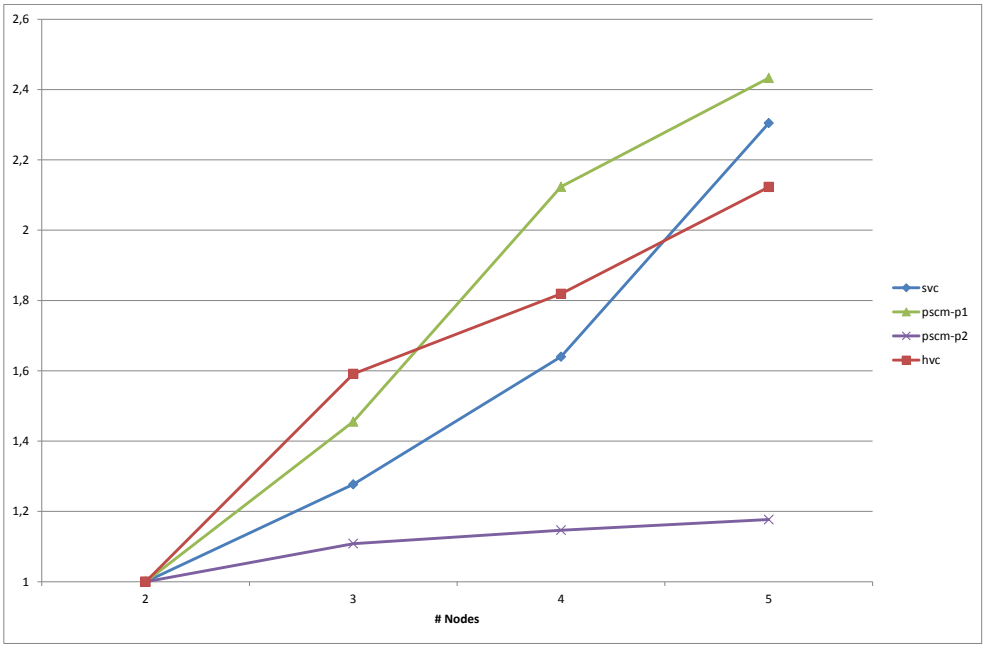

Figure 4.13: Relative running time of the algorithms for SCM×500 data set on different cluster sizes.

In other words, we plot the ratio between the running time of the current cluster size and the smallest size of the cluster. For example, for the 4-node cluster, we plot the ratio between the running time on the 1-node cluster and the running time on the 4-node cluster. We can see that the fastest algorithm was *HVC*. Also we can see that *PSCM-p1* surpassed *HVC* in 2-node cluster then decrease. This fact is due to cost of scheduling tasks and partitioning large number of groups. However, all the three algorithms have approximately the same speed up curves and scales well.

SCM×500 dataset: Similar to Robostrike dataset, we performed the same experiments on SCM×500 dataset. Figure 4.12 shows the running time of the three algorithms on SCM×500 data set on different cluster sizes. The three curves follow the same pattern as in Figure 4.10. *HVC* was always the best. We observe, also, that the *SVC* running time decrease linearly as the cluster size increased. *PSCM-p1* running time was the worst for all cluster sizes. However, its speed up scales faster than *HVC* and *SVC* as shown in Figure 4.13. We can observe in Figure 4.13 that *HVC* was the slowest and outperformed by *SVC* and *PSCM-p1* which scales faster as the cluster size increases.

4.6 Discussion

In this chapter we have studied the problem of discovering candidate atomic conditions in parallel using the MapReduce framework. We proposed one, two and multi-pass approaches and we explored two solutions for the one stage approach (*SVC* and *HVC* algorithms). We showed how to efficiently partition the log across several nodes in the cluster in order to process each candidate independently from others. We also provided an adequate data structure which clearly decrease both the computation time and the size of correlated messages sets. We describe two ways to perform a transitive closure computation using a depth-first-search-like algorithm in order to discover the process instances entailed by interesting *reference-based* candidate conditions while we eliminate such computations for *key-based* candidate conditions. We addressed an extension to deal with some extreme cases when the nodes are overloaded. Given our proposed algorithms, we implemented them in Hadoop and analysed their performance characteristics on real and synthetic datasets. Besides this, and as a final experiment, we run the approach on SCM data set synthetically replicated to 10000 times (more than 40 million messages) and the number of attributes is, also, duplicated. However, the running time to process

$SCM2 \times 10000$, more than 12GB of, data using our approach on 10 nodes was 1 hour and 11 minutes.

Chapter 5

Discovering Composite Conditions

Contents

5.1 Introduction

In this Chapter we introduce two MapReduce-Based algorithms to compute composite conditions (i.e., Conjunctive and Disjunctive). Such correlation conditions are computed from the set of atomic correlation condition discovered in the previous step (c.f., section 4.2), using conjunction operator (respectively disjunction operator). The main contributions of this chapter are as follows:

- We introduce two strategies to partition the space of computation vertically in order to process each (sub)-partition in parallel.
- We introduce the *correlation condition-based partitioning* to partition the computation space in parallel and achieve the processing in a single MapReduce pass. We refer to this concept as the set of *partitioning conditions*.
- We describe technique to avoid unnecessary computation to compute process instances entailed by composite correlation conditions.
- We describe a second approach based on a horizontal partitioning of the computation space. Each level of the lattice is processed by a single MapReduce job.

The rest of the chapter is organized as follows: in section 5.2, we present a *single-pass* approach to achieve the composite correlation condition discovery task. Then, in section 5.3 we present a *multi-pass* approach to discover composite correlation conditions. Next, in section 5.4 we present evaluation experiments of the two approaches. Finally, we summarize the chapter in section 5.5.

5.2 Single-Pass Composite Condition Discovery algorithms

Unlike *atomic correlation condition discovery*, this step is more complex and challenging in terms of space of computations and the high number of candidate composite conditions (as shown in Figure 5.1). Therefore, performance strategies should be adopted in order to deal with this issue. Figure 5.1 shows an example of a lattice (space) of computation of composite conjunctive conditions generated by four *atomic correlation conditions* (ψ_1, ψ_2, ψ_3 and ψ_4). In this section we present a *single-pass* MapReduce-based approach to discover conjunctive and disjunctive correlation conditions.

	CustomerId	OrderId
m_1	C1	P1
m_2	C2	P2
m_3	C2	P1
m_4	C1	P2
m_5	C2	P2
m_6	C1	P2
m_7	C2	P1
m_8	C1	P1

Table 5.1: a snapshot of example log.

5.2.1 Discovering Conjunctive Conditions

Usually messages in logs are not only correlated by a single atomic condition. Indeed, several conditions (i.e., several attributes) can also correlate messages and partition the logs into relevant instances. This case can be viewed as composite keys in relational databases, where multiple attributes are used to identify rows. For instance, messages in Table 5.1 can be correlated using values of attributes **CustomerID** (ψ_1: $m_i.CustomerID = m_j.CustomerID$) and **OrderID** ($\psi_2$: $m_i.OrderID = m_j.OrderID$). In this case, we note $\psi = \psi_{1\wedge 2} = \psi_1 \wedge \psi_2$. In this example, $\psi_{1\wedge 2}(\mathcal{L}) = \{< m_1, m_8 >, < m_2, m_5 >, < m_3, m_7 >, < m_4, m_6 >\}$

Conjunctive Correlation Condition A *Conjunctive correlation condition* consists of conjunction of at least two atomic conditions. It has the following form: $\Phi = \psi_1 \wedge \psi_2 \wedge \ldots \psi_n$. Where, ψ_is, $1 \leq i \leq n$ are atomic conditions.

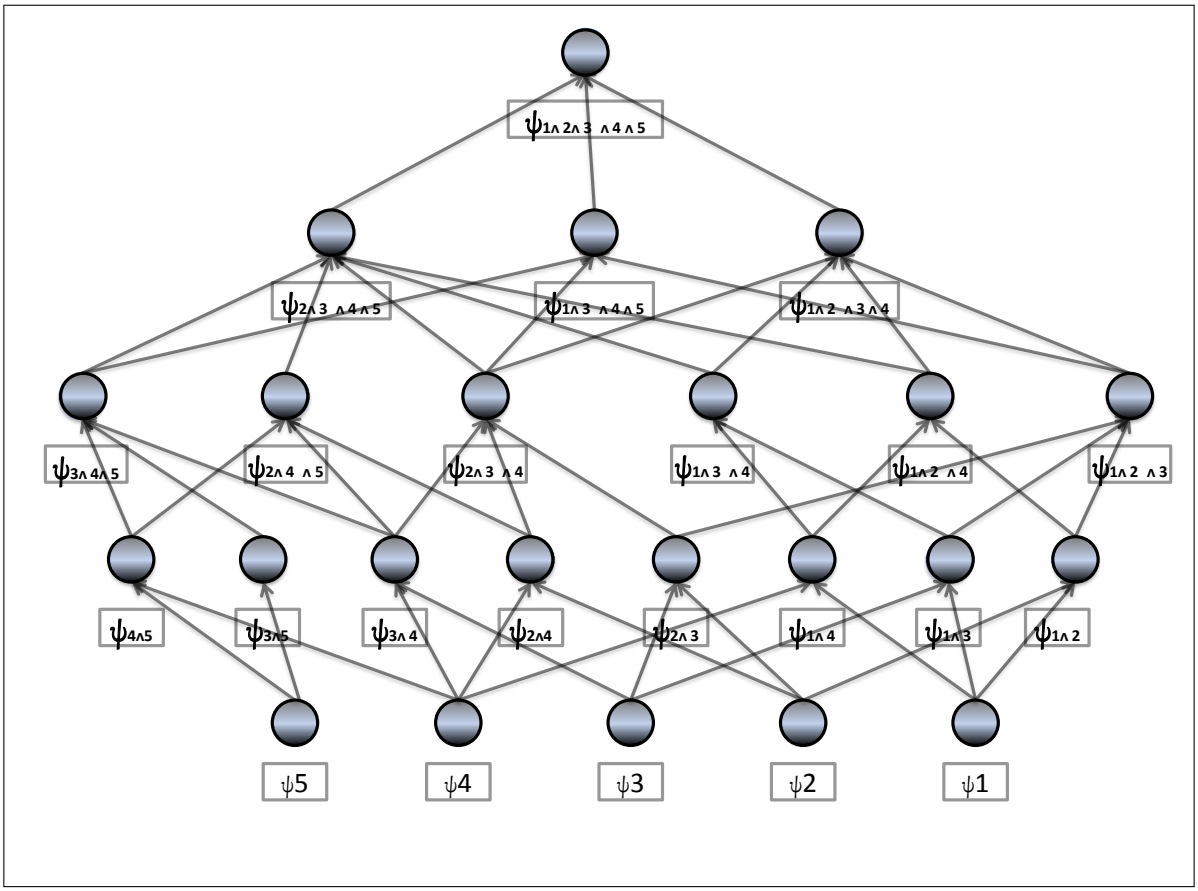

Figure 5.1: Lattice generated by 5 atomic conditions.

Conjunctive conditions are computed using conjunctive operator on atomic conditions: let ψ_1 and ψ_2 be two atomic conditions elicited during the previous step, then the goal is to compute the process instances entailed by the condition $\psi_1 \wedge \psi_2$, noted $\psi_{1\wedge 2}$. More generally, given a set AC of atomic conditions, the goal is to identify the set of *minimal* conjunctive conditions that partition the log into interesting process instances. As explained in [74], such a task can be achieved using a levelwise-like approach [68, 110] where, roughly speaking, each level is determined by the length, in terms of number of conjuncts, of the considered conditions. Starting from atomic conditions (level 1), the discovery process consists in two main parts: (i) generating candidate conditions of level l from candidates of level $l-1$, and (ii) pruning non interesting conditions. At each level,

the process instances associated with each generated candidate condition are computed and used to prune, if any, the considered candidate condition.

Example 9 *Consider as an example a set of atomic condition* $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$. *The candidate conditions at each level are shown at Table 5.2. Figure 5.2 illustrates the space of computations.*

Level 1	$\psi_1, \psi_2, \psi_3, \psi_4, \psi_5$
Level 2	$\psi_{1\wedge2}$, $\psi_{1\wedge3}$, $\psi_{1\wedge4}$, $\psi_{1\wedge5}$, $\psi_{2\wedge3}$, $\psi_{2\wedge4}$, $\psi_{2\wedge5}$, $\psi_{3\wedge4}$, $\psi_{3\wedge5}$, $\psi_{4\wedge5}$
Level 3	$\psi_{1\wedge2\wedge3}$, $\psi_{1\wedge2\wedge4}$, $\psi_{1\wedge2\wedge5}$, $\psi_{1\wedge3\wedge4}$, $\psi_{1\wedge3\wedge5}$, $\psi_{1\wedge4\wedge5}$, $\psi_{2\wedge3\wedge4}$, $\psi_{2\wedge3\wedge5}$, $\psi_{2\wedge4\wedge5}$, $\psi_{3\wedge4\wedge5}$
Level 4	$\psi_{1\wedge2\wedge3\wedge4}$, $\psi_{1\wedge2\wedge3\wedge5}$, $\psi_{1\wedge2\wedge4\wedge5}$, $\psi_{1,3\wedge4\wedge5}$, $\psi_{2\wedge3\wedge4\wedge5}$
Level 6	$\psi_{1\wedge2\wedge3\wedge4\wedge5}$

Table 5.2: Candidates space.

To cast the algorithm Level-wise into a MapReduce framework, the main issue to deal with is how to distribute the candidates among reducers such that the generation and pruning computations are effectively parallelized. We propose to partition the space of candidates (see Figure 5.2) in such a way that an element of the partition can be handled by a unique reducer. This enables to avoid multiple MapReduce steps in order to compute conjunctive conditions. Henceforth, each element of the partition is called a *chunk*.

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

Example 10 *Continuing with the previous example with* $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$ *and assuming that* $PC = \{\psi_1, \psi_2\}$, *we obtain four possible chunks:*

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

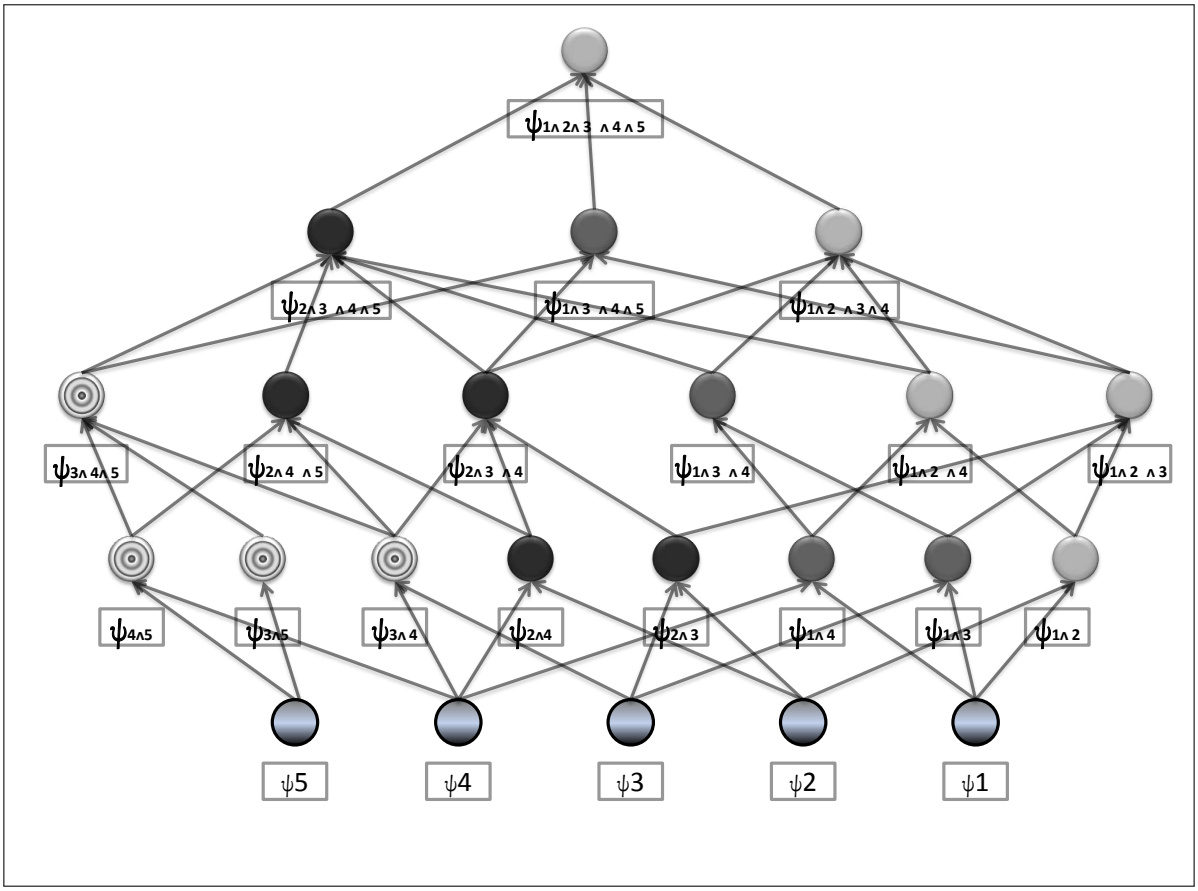

Figure 5.2: The lattice of generated candidate composite condition. Each partition is represented by a single color.

- $(\bar{\psi}_{\mathbf{1}}, \psi_{\mathbf{2}})$*: 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 5.3 and Figure 5.2 represent the obtained partition of the space of candidate conditions presented at Table 5.2 using $PC = \{\psi_1, \psi_2\}$. Each column in Table 5.3 contains a chunk of the space of candidates that can be processed separately by a given reducer.

Note that the obtained chunks are balanced (i.e., they have the same number of candidate conditions[1]) and they form a partition of the initial space of candidates. It is also worth noting that each chunk can be treated separately from the others in order to compute the corresponding interesting conditions.

[1]In Table 5.3, if we also consider that $\emptyset \in (\bar{\psi}_1, \bar{\psi}_2)$, then every chunk will have 8 candidates.

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

Table 5.3: Partitioned candidates space.

Algorithm 14: Conjunctive-MR

Input: AC, $PC = \{\psi_1, \ldots, \psi_l\}$

Output: set of interesting atomic conditions

```
1  begin
2      Map (key: null, c_a: an atomic condition in AC) ;
3          P ← {ψ_1, ψ̄_1} × ... × {ψ_l, ψ̄_l} ;
4      for p ∈ P do
5          if ac ∈ p then
6              output(p, ac) ;
7              else
8                  for r ∈ Reducers do
9                      output(r, ac) ;
       /* (k', V = list(v')) is an intermediate key-list of values pair   */
10     Reduce (k', V) ;
11         PC ← ∅ ;
12         RC ← ∅ ;
       /* P is the set of partitioning conditions, subset of AC.          */
       /* RC =AC \ P                                                      */
13         {PC, RC} ← BuildSets(V);
14     if PC ≠ ∅ then
15         ψ_PC ← Conjunctive_Partitioning_conditions(PC)
           CC ← Level_wise_p(ψ_P, RC) ;
16     else
17         CC ← Level_wise(RC) ;
18         output(CC) ;
```

The high level structure of the algorithm that enables to compute conjunctive conditions, called Conjunctive-MR, is given at algorithm 14. The algorithm takes as input a set of atomic conditions (AC) and iteratively generates candidates of higher level based on candidates in lower level, then it prunes non interesting ones. A classical candidate generation procedure, e.g., see the *Apriori* algorithm [13], computes candidates at level l by a *self-join* on level l - 1. For example, if both $\psi_{1\wedge2}$ and $\psi_{1\wedge3}$ appear at level 2, than the candidate $\psi_{1\wedge2\wedge3}$ will be generated at level 3. The Map function will be in charge of partitioning the space of computations. For a given condition ψ_x, it proceeds as follows:

1. Checks the presence of ψ_x in PC.
2. ψ_x is sent to the corresponding reducers, i.e., sent to each reducer in charge of processing a chunk that contains ψ_x (line 6 of algorithm 14), if $\psi_x \in PC$.
3. ψ_x is sent to all reducers (line 9 of algorithm 14), otherwise.

At the Reduce side, (subset of) the partitioning conditions (PC) are buffered separately from the remainder conditions (RC[2]). Then, each reducer first computes the conjunction of the partitioning conditions presented in its corresponding chunk, i.e., it computes ψ_{PC}, where $\psi_{PC} = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_l$ (line 15 of algorithm 14). After that, a *level-wise* algorithm, depicted in algorithms 16, is applied to explore the space of candidate conjunctive conditions built on ψ_{PC}. For example, a given reducer in charge of a chunk with $PC = \{\psi_1, \psi_2, \psi_3\}$ and $RC = \{\psi_4, \psi_5\}$, then it computes ψ_{123}, after that it applies the level-wise algorithm to compute $\psi_{123\wedge4}$, $\psi_{123\wedge5}$ and $\psi_{123\wedge4\wedge5}$. At each iteration of the level-wise the reducers proceed as follows:

(i) Computing process instances entailed by ψ_{PC} and ψ_x (where $\psi_x \in RC$). The first step computes the messages correlated by $\psi_{PC\wedge x}$. Indeed, this operation (line 6 algorithm 16) relies on the following property: two messages m and m' are correlated by the conjunctive condition ψ_{12} if they are correlated by both ψ_1 and ψ_2 (i.e., $<m, m'> \in \mathcal{CMB}_{\psi_1} \cap \mathcal{CMB}_{\psi_2}$).

(ii) Pruning candidate conjunctive conditions. At this step, non interesting conditions are pruned using *ImbalancedPI* criterion (line 8 algorithm 16). We check if the condition $PI\_ratio(\psi_{PC\wedge x}) < \beta$ is satisfied or not. Using conjunctive operator implies a new criteria that can be applied to prune non interesting conditions. The first criterion is

[2]Atomic conditions that do not belong to partitioning condition set are referenced as RC, (i.e., RC=$AC \setminus PC$).

referred as *notMon(ψ)*, it is used to check the monotony of the number and the length of instances with respect to the conjunctive operator. It says that, conjunctive conditions that do not increase the number of instances w.r.t. the number of instances already discovered by their respective conjuncts and do not decrease the length of the discovered instances is considered as non interesting and therefore pruned. Secondly, if the set of correlated messages of psi_x is included in that of ψ_{PC} (or vice versa), then, the condition $\psi_{PC \wedge x}$ is discarded.

(iii) Generating candidate conditions. Candidate conjunctive condition of $level_l$ are formed using non-pruned from $level_{l-1}$. In algorithm 16 candidate conjunctive conditions in the first level are computed by conjunction of condition $\psi_P C$ with each condition from RC. From the second level to higher, conjunctive conditions are computed by joining conditions from the previous level with those from RC (e.g., $level_1 = \{\psi_{13}, \psi_{14}\}, RC = \{\psi_3, \psi_4\}$ then $level_2 = \{\psi_{134}\}$). Should recall that redundancy is eliminated, since $\psi_{123} = \psi_{213} = \psi_{231}$ only ψ_{123} is computed.

Algorithm 15: *Level_wise*

Input: RC
Output: CC

```
1  begin
2      l ← 0 ;
3      Level_0 ← RC ;
4      repeat
5          l ← l + 1 ;
6          foreach ψ_i ∈ Level_{l−1} do
7              foreach ψ_j ∈ RC do
8                  ψ_cc ← ψ_i ∧ ψ_j ;
9                  CMB_{ψ_cc} ← CMB_{ψ_i} ∩ CMB_{ψ_j} ;
10                 if not inc(CMB_{ψ_cc}) then
11                     PI_{ψ_cc} ← compute_instances(CMB_{ψ_cc}) ;
12                     if is_mon(ψ_cc) and ψ_cc has not ImbalancedPI(PI_{ψ_cc}) then
13                         Level_l ← Level_l ∪ ψ_cc ;
14         CC ← CC ∪ Level_l ;
15     until Level_l = ∅;
16     return CC
```

Algorithm 16: *Level_wise_p*

Input: ψ_P, RC

Output: CC

```
1  begin
2      CC ← ψ_P ;
3      l ← 0 ;
4      foreach condition ψ_i ∈ RC do
5          ψ_{PC∧i} ← ψ_PC ∧ ψ_i ;
6          CMB_{ψ_{PC∧i}} ← CMB_{ψ_PC} ∩ CMB_{ψ_i} ;
7          if not inc(CMB_{ψ_cc}) then
8              PI_{ψPC∧i} ← compute_instances(CMB_{ψ_{PC∧i}}) ;
9              if is_mon(ψ_cc) and ψ_{PC∧i} has not ImbalancedPI(PI_{ψPC∧i}) then
10                 Level_0 ← ψ_{PC∧i} ;
11     CC ← CC ∪ Level_0 ;
12     repeat
13         l ← l + 1 ;
14         foreach ψ_i ∈ Level_{l−1} do
15             foreach ψ_j ∈ RC do
16                 ψ_cc ← ψ_i ∧ ψ_j ;
17                 CMB_{ψ_cc} ← CMB_{ψ_i} ∩ CMB_{ψ_j} ;
18                 if not inc(CMB_{ψ_cc}) then
19                     PI_{ψ_cc} ← compute_instances(CMB_{ψ_cc}) ;
20                     if is_mon(ψ_cc) and ψ_cc has not ImbalancedPI(PI_{ψ_cc}) then
21                         Level_l ← Level_l ∪ ψ_cc ;
22         CC ← CC ∪ Level_l ;
23     until Level_l = ∅;
24     return CC
```

Example 11 *Using the algorithm* Conjunctive-MR *with inputs as* $AC = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$, *enables to generate the whole space of candidates described in Table 5.3.*

An exceptional case needs a different processing occurs when the reducer chunk does not contain any partitioning condition (i.e., $PC = \emptyset$, e.g., the fourth column in Table 5.3). The algorithm devoted to handle this special case is depicted in algorithm 15. The candidate conjunctive conditions in the first level are directly computed from RC and the algorithm follows the same behaviour of algorithm 16 from the step *(ii)*.

5.2.2 Discovering Disjunctive Conditions

Messages	Service	InvId	PayId	ShipId
m_1	invoice	i1		
m_2	invoice	i2		
m_3	Pay	i1	P1	
m_4	Pay	i2	P2	
m_5	Ship		P2	S2
m_6	Ship		P1	S1
m_7	OrderFulfil			S2
m_8	OrderFulfil			S1

Table 5.4: a snapshot of example log.

Business Processes in modern enterprises are, rarely, executed by a single centralized system. Indeed, several systems cooperate together to achieve the enterprise business objective. However, applications or web services interact together by sending and receiving messages. Such interactions are specified in a *process choreography* [104, 81], that allows for multiple concrete implementations, in which traditional information systems cannot support. Therefore, their historical execution informations (log files) are dispersed across several systems and data sources. In such enterprises, the process spans many systems, and each system may have a different correlation method to correlate messages. A disjunction of conditions deals with this issue, where several conditions are used to correlate messages which are correlated differently [74]. For example, in the Table 5.4 a shipment message references a payment message while a payment message references an

invoice message. In this case, 3 conditions are needed to correlate those messages R_{ψ_1} : $m_x.InvID = m_y.InvID = \{\langle m_1, m_3\rangle, \langle m_2, m_4\rangle\}$, $R_{\psi_2} : m_x.PayID = m_y.PayID = \{\langle m_3, m_6\rangle, \langle m_4, m_5\rangle\}$ *and* $R_{\psi_3} : m_x.ShipID = m_y.ShipID = \{\langle m_5, m_7\rangle, \langle m_6, m_8\rangle\}$. We note:

$$\psi_{1\vee2\vee3} = \psi_1 \vee \psi_2\psi_3 = \{\langle m_1, m_3, m_6, m_8\rangle, \langle m_2, m_4, m_5, m_7\rangle\}.$$

Disjunctive Correlation Condition A *Disjunctive correlation condition* consists of a disjunction of at least two atomic or conjunctive conditions. It has the following form: $\Phi = \psi_1 \vee \psi_2 \vee \ldots \psi_n$. Where, ψ_is, $1 \leq i \leq n$ are atomic conditions and/or conjunctive conditions.

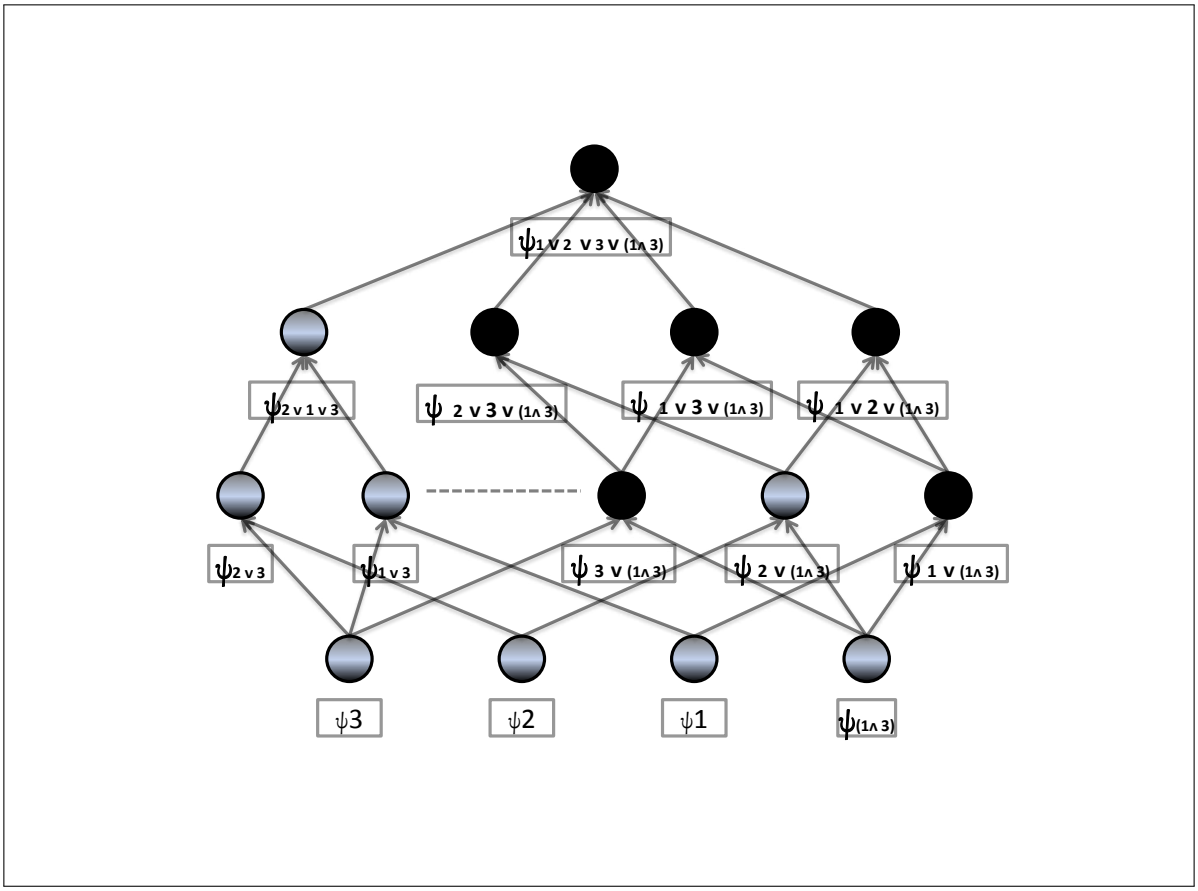

Figure 5.3: Lattice generated by 3 atomic conditions and one conjunctive condition.

Given an atomic and a conjunctive conditions ψ_i and $\psi_{cc} \in \{AC\} \cup \{CC\}$, the goal is to find interesting minimal disjunctive conditions of the form $\psi_{dc} = \psi_i \vee \psi_{cc}$. Discovering candidate disjunctive conditions is also performed by an iterative level-wise approach

[68, 110]. The correlation discovery process consists in three main parts: (i) generating candidate conditions of level l from candidates of level $l-1$, (ii) computing process instances of the new candidates and, finally, (iii) pruning non interesting conditions.

Example 12 *Consider as an example a set of atomic and conjunctive conditions* $AC\_CC = \{\psi_1, \psi_2, \psi_3, \psi_{1\wedge3}\}$. *The candidate conditions at each level are shown in the Table 5.5 and Figure 5.3 illustrates the space of computations. Note that as explained in the sequel, conditions in black are discarded.*

Level 1	$\psi_1, \psi_2, \psi_3, \psi_{1\wedge3}$
Level 2	$\psi_{1\vee2},\ \psi_{1\vee3},\ \psi_{1\vee(1\wedge3)},\ \psi_{2,3},\ \psi_{2\vee(1\wedge3)},\ \psi_{3\vee(1\wedge3)}$
Level 3	$\psi_{1\vee2\vee3},\ \psi_{1\vee2\vee(1\wedge3)},\ \psi_{1\vee3\vee(1\wedge3)},\ \psi_{2\vee3\vee(1\wedge3)}$
Level 4	$\psi_{1\vee2\vee3\vee(1\wedge3)}$

Table 5.5: Candidates space.

In order to explore the computation space of candidate disjunctive conditions, we adopt the same strategy for computing candidate conjunctive conditions (see section 5.2.1).

Let AC set of atomic conditions and CC set of Conjunctive conditions. We define the partitioning conditions $PC = \{\psi_1, \ldots, \psi_l\} \subset AC \cup CC$ a subset of $AC \cup CC$. Then, we form $\mathcal{P}$, the partitions of the space of candidates, 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 $\phi_1 \in \{\psi_1, \bar{\psi_1}\}$, for $i \in [1, l]$, represents a chunk of the space of candidate conditions.

Example 13 *Continuing with the previous example and assuming that* $PC = \{\psi_1, \psi_3\}$, *we obtain four possible chunks:*

- $(\psi_{\mathbf{1}}, \psi_{\mathbf{3}})$*: contains the candidates with subscripts* 1 *and* 3,
- $(\psi_{\mathbf{1}}, \bar{\psi}_{\mathbf{3}})$*: contains the candidates with subscript* 1 *but without* 3,
- $(\bar{\psi}_{\mathbf{1}}, \psi_{\mathbf{3}})$*: contains the candidates with subscript* 3 *but without* 1,
- $(\bar{\psi}_{\mathbf{1}}, \bar{\psi}_{\mathbf{3}})$*: contains the candidates without the subscripts* 1 *and* 3,

Table 5.6 shows the obtained chunks of the space of candidate conditions of Table 5.2 partitioned using $PC = \{\psi_1, \psi_2\}$. *Each column of the table contains a chunk of the space of candidates that can be processed separately by a given reducer.*

Algorithm 17: Disjunctive-MR

Input: $AC, CC, P = \{\psi_1, \ldots, \psi_l\}$

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 ac ∈ p then
6              output(p, ac) ;
7              else
8                  for r ∈ Reducers do
9                      output(r, ac) ;
       /* (k', V = list(v')) is an intermediate key-list of values pair   */
10     Reduce (k', V) ;
11         PC ← ∅ ;
12         RC ← ∅ ;
       /* PC is the set of partitioning conditions, subset of AC ∪ CC.    */
       /* RC = {AC ∪ CC} \ P                                              */
13         {PC, RC} ← BuildSets(V);
14     if P ≠ ∅ then
15         ψ_PC ← Conjunctive_Partitioning_conditions(PC)
           DC ← Level_wise_p(ψ_P C, RC) ;
16     else
17         DC ← Level_wise(RC) ;
18         output(DC) ;
```

The proposed algorithm devoted to compute disjunctive conditions, called Disjunctive-MR, is given at algorithm 17. The algorithm has the atomic conditions (AC) and conjunctive conditions (CC) as inputs and iteratively builds candidate of higher levels based on candidate in lower levels. Recall, that candidate in level l are computed by a *self join* on level l -1. The Map, in Disjunctive-MR will be in charge of partitioning the space of computations.

At the **reduce** side we apply the level-wise, with slight difference, to explore the space of candidate conditions. Each iteration consists of the following steps:

	$(\psi_{\mathbf{1}}, \psi_{\mathbf{3}})$	$(\psi_{\mathbf{1}}, \bar{\psi}_{\mathbf{3}})$	$(\bar{\psi}_{\mathbf{1}}, \psi_{\mathbf{3}})$	$(\bar{\psi}_{\mathbf{1}}, \bar{\psi}_{\mathbf{3}})$
Level 1		ψ_1	ψ_3	$\psi_2, \psi_{(1\wedge 3)}$
Level 2	$\psi_{1\vee 3}$	$\psi_{1\vee 2}, \psi_{1\vee(1\wedge 3)}$	$\psi_{3\vee 2}$, $\psi_{3\vee(1\wedge 3)}$	$\psi_{2\vee(1\wedge 3)}$
Level 3	$\psi_{1\vee 3\vee 2}$, $\psi_{1\vee 3\vee(1\wedge 3)}$	$\psi_{1\vee 2\vee(1\wedge 3)}$	$\psi_{3\vee 2\vee(1\wedge 3)}$	
Level 4	$\psi_{1\vee 2\vee 3\vee(1\wedge 3)}$			

Table 5.6: Partitioned candidates space.

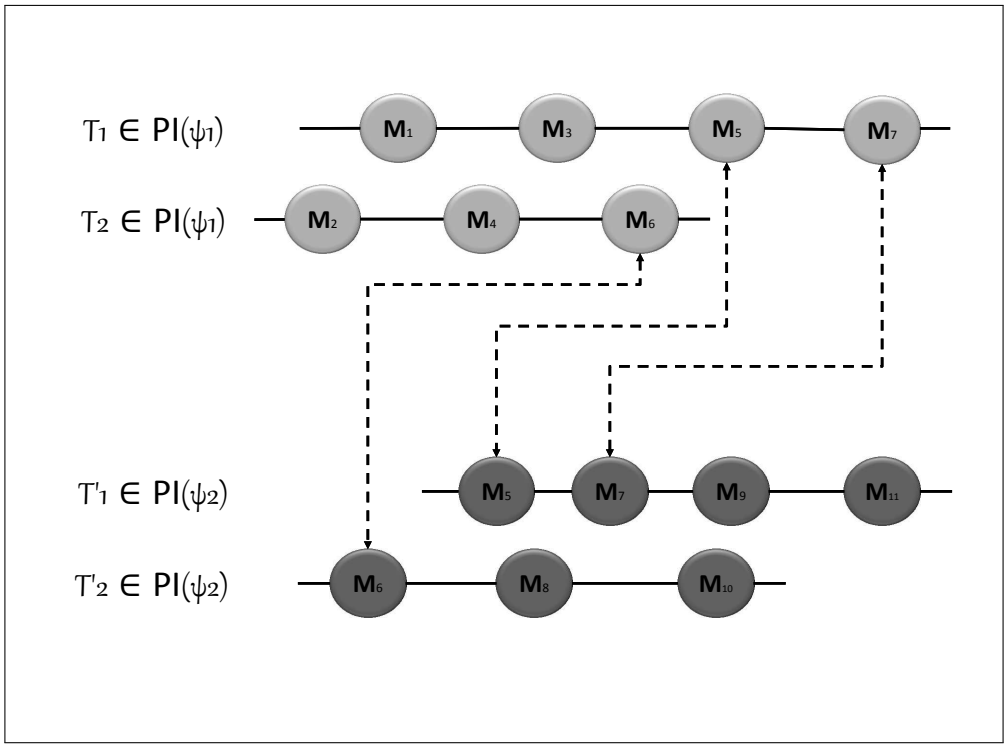

Figure 5.4: Connected instances.

(i) Computing process instances. This step is challenging and require an intensive computation. To find process instances of a disjunctive conditions $\psi_{1\vee 2}$ in [74] they, first, compute the set of correlated message pairs $R_{\psi_{1\vee 2}}$. $R_{\psi_{1\vee 2}}$ is the union of correlated message pairs of ψ_1 and those of ψ_2. After that, process instances are computed based on finding connected components from the set of correlated messages $R_{\psi_{1\vee 2}}$. In order to minimize computations, process instances entailed by disjunctive conditions can be computed directly from already discovered instances in previous steps (atomic and con-

junctive) and avoid additional step of processing. Therefore, we need only to find a link between a process instance from ψ_1 with a process instance from ψ_2 to form a new process instance. Figure 5.4 illustrates an example of connected instances. In order to formalize this property we introduce the operator $\otimes$ as following:

For each instance σ in $PI_{\psi_{1\vee2}}$, it exists at least two instances σ_1 and σ_2 from PI_{ψ_1} and PI_{ψ_2} respectively, where intersection of σ_1 and σ_2 is not empty and $\sigma = \sigma_1 \otimes \sigma_2$. $\otimes$ is the joining of two instances and it is computed if and only if $\sigma_1 \cap \sigma_2 \neq \emptyset$. More formally: $\forall \sigma \in PI_{\psi_{1\vee2}}$ *then* $\exists(\sigma_1, \sigma_2) \in PI_{\psi_1} \times PI_{\psi_2}$ *where* $\sigma_1 \cap \sigma_2 \neq \emptyset$ *and* $\sigma_1 \otimes \sigma_2 \subseteq \sigma$

Example 14 *As shown in Figure 5.4, we have:*

- $\sigma \in PI_{\psi_{1\vee2}}, \exists(\sigma_1, \sigma_1') = \{\langle m_1, m_3, m_5, m_7\rangle, \langle m_5, m_7, m_9, m_{11}\rangle\} \in PI_{\psi_1} \times PI_{\psi_2}$, *where* $\sigma_1 \cap \sigma_2 = \{m_5, m_7\}$ *and* $\sigma = \sigma_1 \otimes \sigma_1' = \{\langle m_1, m_3, m_5, m_7, m_9, m_{11}\rangle\}$,
- $\sigma' \in PI_{\psi_{1\vee2}}, \exists(\sigma_2, \sigma_2') = \{\langle m_2, m_4, m_6\rangle, \langle m_6, m_8, m_{10}\rangle\} \in PI_{\psi_1} \times PI_{\psi_2}$, *where* $\sigma_2 \cap \sigma_2' = \{m_6\}$ *and* $\sigma' = \sigma_2 \otimes \sigma_2' = \{\langle m_2, m_4, m_6, m_8, m_{10}\rangle\}$.

Finally, we find $PI_{\psi_{1\vee2}} = \{\sigma,\ \sigma'\} = \{\langle m_1, m_3, m_5, m_7, m_9, m_{11}\rangle,\ \langle m_2, m_4, m_6, m_8, m_{10}\rangle\}$

(ii)Pruning Candidate disjunctive conditions. At this step, non-interesting conditions are pruned using the following criterion:

- *ImblancedPI* criterion is applied to check if $PI\_ratio > \alpha$ since instances formed based on disjunction of ψ_1 and ψ_2 are less numerous than those of ψ_1 and ψ_2.
- *monotonic property*, for a given disjunctive condition, if the number of instances does not decrease or the length of the instances does not increase, then such disjunctive condition is pruned.
- *associativity and inclusion properties.* Conditions that combine the disjunction and conjunction of the same atomic condition are not needed to be computed, this criterion is referred as *not assoc.* If PI_{ψ_1} is included in PI_{ψ_2}, then we have $PI_{\psi_{1\vee2}} = PI_{\psi_2}$. Hence, the condition is discarded, this criterion is referred as *not inc.*

(iii) Generating candidate conditions. Candidate conjunctive condition of $level_l$ are formed using non-pruned candidates from $level_{l-1}$.

Algorithm 18: *Level_wise_p*

Input: ψ_P, RC
Output: DC

```
1  begin
2      DC ← ψ_P ;
3      l ← 0 ;
4      foreach condition ψ_i ∈ RC do
5          ψ_{PC∨i} ← ψ_PC ∨ ψ_i ;
6          if not assoc(ψ_PC) or not inc(ψ_PC, ψ_i) then
7              PI_{ψ_{PC∨i}} ← compute_instances(CMB_{ψ_{PC∨i}}) ;
8              if is_mon(ψ_dc) and ψ_{PC∨i} has not ImbalancedPI(PI_{ψ_{PC∨i}}) then
9                  Level_0 ← ψ_{PC∨i} ;
10     DC ← DC ∪ Level_0 ;
11     repeat
12         l ← l + 1 ;
13         foreach ψ_i ∈ Level_{l−1} do
14             foreach ψ_j ∈ RC do
15                 ψ_dc ← ψ_i ∨ ψ_j ;
16                 if not assoc(ψ_dc) or not inc(ψ_P, ψ_i) then
17                     PI_{ψ_dc} ← compute_instances(CMB_{ψ_cc}) ;
18                     if is_mon(ψ_dc) and ψ_dc has not ImbalancedPI(PI_{ψ_dc}) then
19                         Level_l ← Level_l ∪ ψ_dc ;
20         DC ← DC ∪ Level_l ;
21     until Level_l = ∅;
22     return DC
```

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

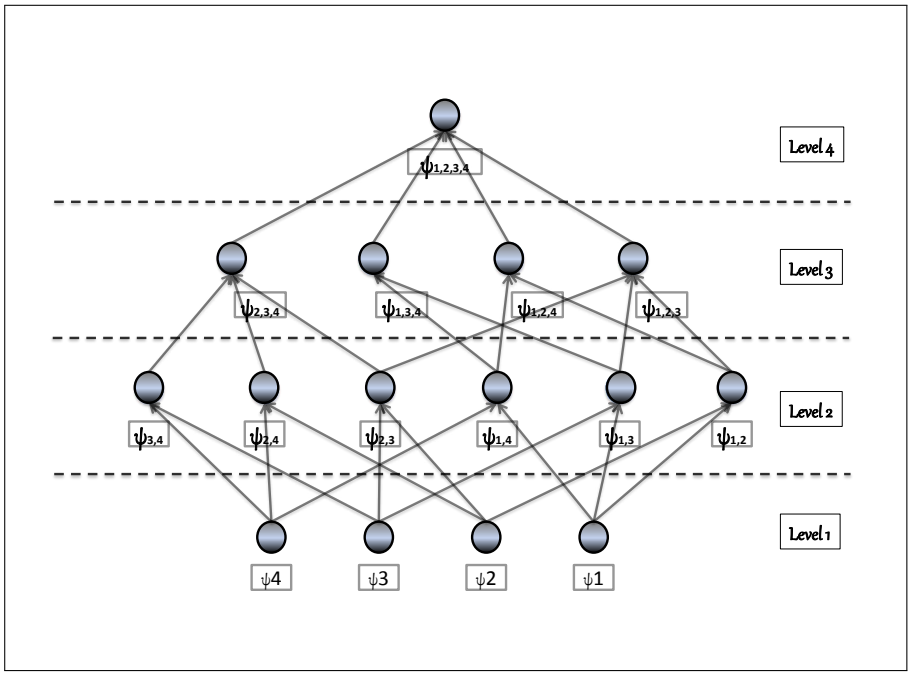

Figure 5.5: Lattice generated by 4 atomic conditions. Each level of the lattice is processed by separate MapReduce job.

5.3 Muti-Pass composite Conditions Discovering algorithms

To enhance the performance of the algorithm presented previously and minimize the workload of each node, we introduce a multi-pass algorithm to discover candidate composite

conditions (i.e., conjunctive or disjunctive). Besides generating and processing candidates, the aim of the algorithm is to minimize the workload allocated to each node by adopting a new strategy of partitioning and splitting the space of candidate exploration.

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

Algorithm 19: multi-pass main algorithm.

Input: $\mathcal{K}$: unused, $\mathcal{V}$: AC
Output: $\mathcal{K}$: unused, $\mathcal{V}$: CC

```
1  begin
2      k ← 0 ;
3      level_k ← AC ;
4      while level_k ≠ ∅ do
5          η ← ∅ ;
6          forall the ψ ∈ level_k do
7              η ← η ∪ ψ ;
8          store η in distributedCache ;
9          k ← k + 1 ;
10         Map() function ;
11         level_k ← Reduce() function ;
```

As illustrated in Figure 22, the *multi-pass composite conditions discovery* algorithm partitions the lattice horizontally, i.e by levels. It discovers relevant candidate composite conditions presented in $level_k$ in $iteration_k$. Each level is distinguished by the number of atomic conditions merged together (e.g, $\psi_{1,2,3}$ is in $level_3$). Also, it is handled by a single MapReduce job. The first iteration (job) of the algorithm combines the set of *candidate atomic conditions*, discovered in the previous stage, to generate conditions of $level_2$ then select interesting candidates, based on criteria, to be fed to the next iteration. Afterwards, every $iteration_k$ generates the candidate of $level_k$ from the selected

candidates, those that are not pruned, of $iteration_{k-1}$.

Algorithm 20: Multi-pass map function.

```
1  Map_Configure
2  begin
3      load η from distributedCache ;
4      foreach ψ_i, ψ_j ∈ η do
5          ψ_{i⊙j} ← ψ_i ⊙ ψ_j ;
           /* ⊙ ∈ {∧, ∨}                                          */
6          if notAssoc(ψ_{i⊙j}) then
7              CC ← ψ_{i⊙j} ;
8  Map_function
   Input: K : unused, V : ψ_i ∈ level_k
   Output: K : ψ, V : CMB_ψ
9  begin
10     n ← extract the condition name from ψ_i;
11     foreach ψ ∈ CC do
12         if ψ contains n then
13             output(ψ, ψ_i) ;
```

Algorithm 21: multi-pass reduce function.

```
1  Reduce_function
   Input: K : ψ, V : CMB_{ψ⟩}, CMB_{ψ|}
   Output: K : ψ, V : ψ_c c
2  ψ_cc ← ψ_i ∧ ψ_j ;
3  if not inc(ψ_P, ψ_i) then
4      PI_{ψ_cc} ← compute_instances(CMB_{ψ_cc}) ;
5      if is_mon(ψ_dc) and ψ_cc has not ImbalancedPI(PI_{ψ_cc}) then
6          output(ψ, ψ_c c);
```

For a given iteration k and before the Map functions, outlined in the algorithm 20, start their execution, an initialization function is called to load from the *DistributedCache* the

non-pruned candidate conditions[3] from iteration $k-1$(line 3 of algorithm 20), excepting the first iteration which loads the *candidates atomic conditions*. Then, it combines these candidate to generates a set of new candidates (line 5 of algorithm 20). Thereafter, it applies the *associativity* criterion to clean the list from non-interesting candidate (line 6 of algorithm 20). The Map function then retrieves the *correlated message buffers*[4] ($\mathcal{CMB}_\psi$) from HDFS. Next, from each $\mathcal{CMB}_\psi$, it extracts the condition name and uses it to probes the list of keys built in the initialization function for testing whether any key contains the condition name. Hereafter, the Map function produces the key-value pair (*key*, $\mathcal{CMB}_\psi$) for all keys that has the condition name as part (lines from 10 to 13).

In the Reduce function, each reducer will receive a single candidate ψ as key, which corresponds to the candidate that will be processed by this reducer. Associated with that key the set of values are two conditions of $level_{k-1}$ such as the combination of their name produces the key. Before, computing instances at line 4, the reducer checks whether the conditions satisfies the *non Inclusion* and *Trivial Union* criteria. If so, a *DFS*-like algorithm is applied to discover the process instances involved by the composite conditions. After that, the reducers verifies whether new candidate condition induces a new interesting process instances by carrying out the *monotonicity* and *imbalanced_PI* criteria. If the condition survives the criteria, then the reducer outputs the key-value pairs (ψ, $\mathcal{CMB}_\psi$).

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

Example 15 *Figure 5.6 illustrates an example of the algorithm execution. The algorithm has 4 candidate atomic conditions as input. Each iteration combines previous result to build current level candidates. The last iterations produces the candidate containing all the candidates atomic conditions present in the input.*

[3]We store only the condition names which is composed by the attribute names (line 8 of algorithm 19). For convenience, we use only ψ or ψ_i to represent the condition name.

[4]In case of *candidate disjunctive conditions* and for optimization reasons we load/store the process instances instead of $\mathcal{CMB}$.

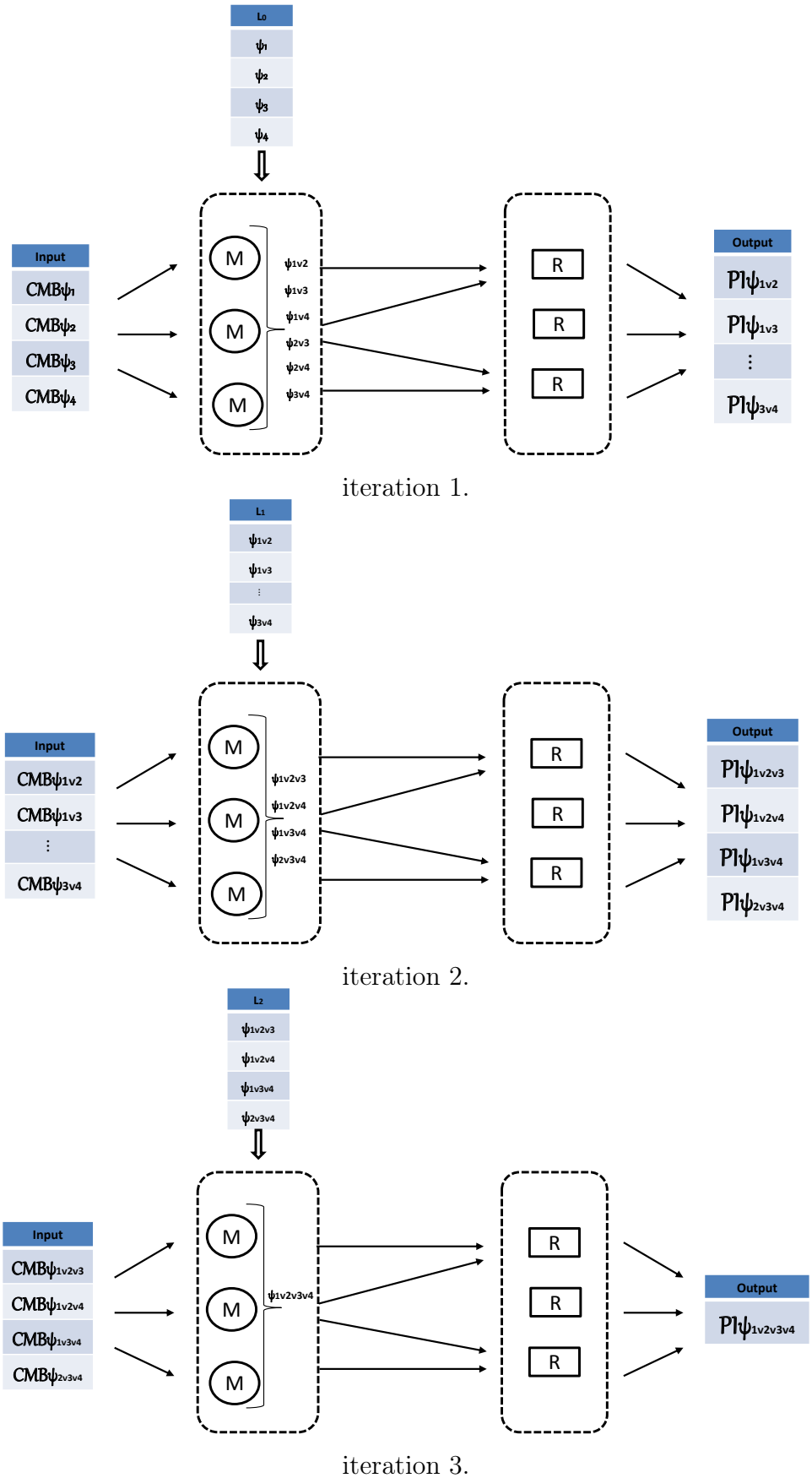

Figure 5.6: An execution example of the algorithm with 3 iterations.

5.4 Experimental Evaluation

In this section, we present the performance evaluation of the proposed algorithms for candidate composite correlation conditions. The absolute time as well as speed up and scale up are measured and discussed.

Environment And Datasets. We ran all the experiments reported in this chapter on the same environment configuration described in Chapter 4. We used the same dataset namely *SCM*×x and *RobotStrike*.

5.4.1 Experiments.

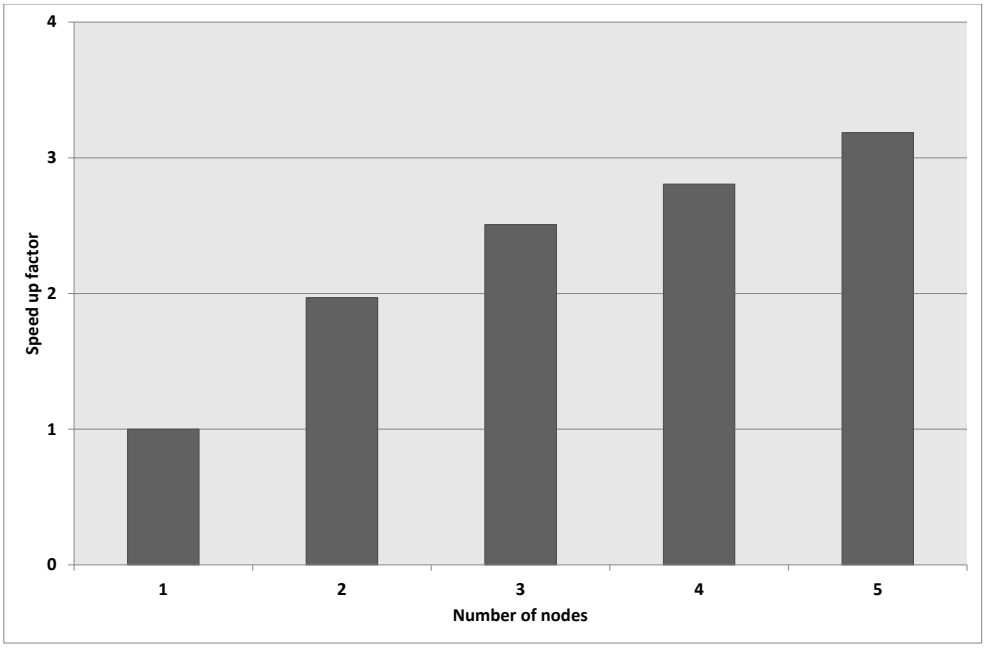

Figure 5.7: Relative running time of the single-pass conjunctive conditions algorithm for RobotStrike dataset set on different cluster sizes.

Single-pass conjunctive conditions algorithm. We ran the *single-pass conjunctive conditions* algorithm on a fixed size of *RobotStrike* dataset and we vary the number of nodes from 1 to 5. We fix the number of *partitioning conditions* to 3 and by consequent the number of Reduce tasks required to achieve the computations is 2^3. Figure 5.7 shows

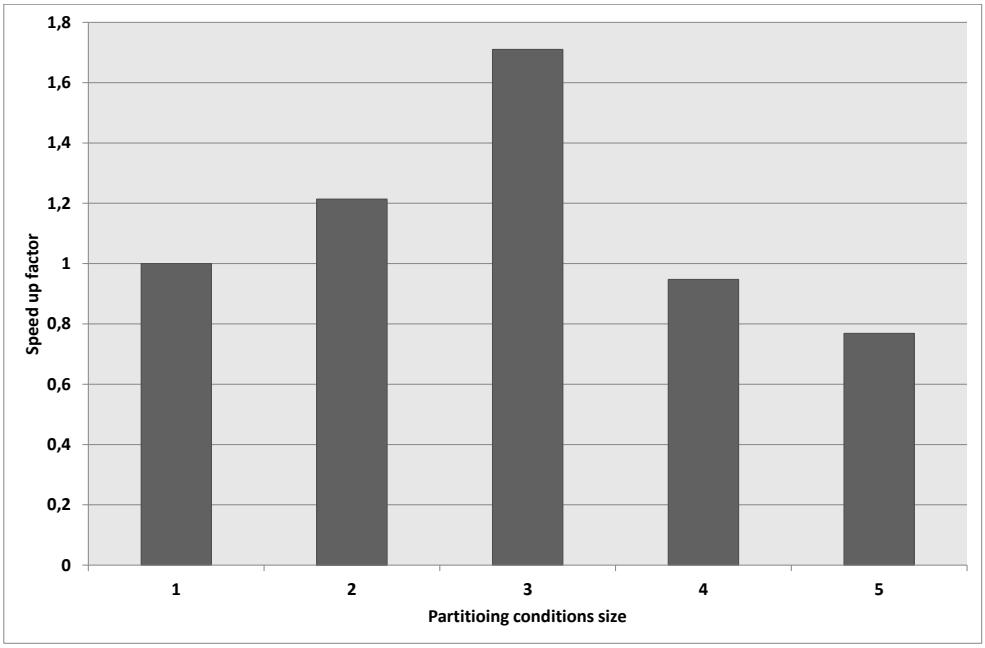

Figure 5.8: Relative running time of the single-pass conjunctive conditions algorithm for RobotStrike dataset set on 5-nodes cluster with different partitioning conditions sizes.

the evolution of the relative running time w.r.t the number of nodes. The running time decreases as the number of nodes is increased. The breaking point can be observed moving from configuration with 1-node to configuration with 2-nodes, where the running time decreases, approximately, by half. This is because the two nodes receive the same workload. Adding more nodes decreases the running time with factor of 0.4. This is due two reasons: (i) nodes do not have the same workload (e.g., with 3-nodes configuration, two nodes receive three tasks and one receive two tasks), (ii)) some tasks take more time in execution than the others depending on the amount of candidates that are pruned before computing their process instances.

In the second experiment, we fixed the number of nodes to 5 than we varied the number of partitioning conditions from 1 to 5. Figure 5.8 shows the relative execution time of the different configuration. We start with *partitioning conditions* size (p) equal to 1 this implies 2^1 Reduce tasks. Therefore, only 2 nodes were working where the other 3 nodes are idle. We observe that the execution time increases than it decreases comparing to the first configuration.

- Configuration 2 and 3, the execution time increases because each node processes at

most two Reduce tasks. Also, all nodes receive a piece of workload (no idle nodes).

- configuration 4 and 5, increasing p spawns more Reduce tasks. However, it involves larger intermediate data size and scheduling a huge number of tasks. These overheads affect the performance of the algorithm and decrease the running time.

Based on this evaluation, we conclude that a good value of p w.r.t the number of nodes is in factor of two (e.g., if we have 10 nodes cluster size p should equal to 4).

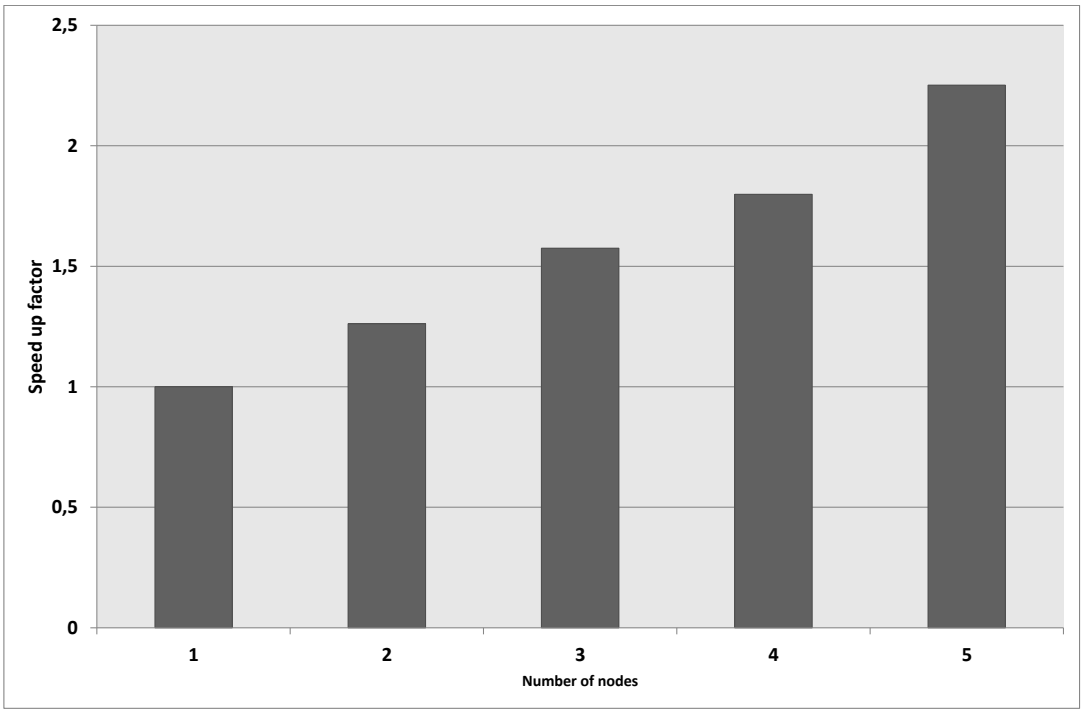

Figure 5.9: Relative running time of the single-pass disjunctive conditions algorithm for SCM×x dataset set on different cluster sizes.

Single-pass disjunctive conditions algorithm. To evaluate the speed up of the *single-pass disjunctive condition* algorithm, we ran the algorithm on a fixed size of SCM×x (x=30) and we fixed the number of *partitioning condition* to 3 (2^3 Reduce tasks). Figure 5.9 shows the evolution of the execution time. We observe that each time we add a node the running time speeds up by a factor of 0.5. Therefore, using 5-node cluster speeds up the running time by approximately a factor of 2.3. Even if the lattice is equally partitioned and the nodes receive the same workload (number of candidates) the experiments show a poor speed up of the algorithm. This fact is due to the computation of

candidates in some nodes and the earlier pruning of candidates in the other nodes. For example, 2 nodes receive 2 candidates, the first prunes one candidate and computes the process instances for the second candidates, the second node computes both candidates. In this case, it is obvious that the second node will spend more time than the first one.

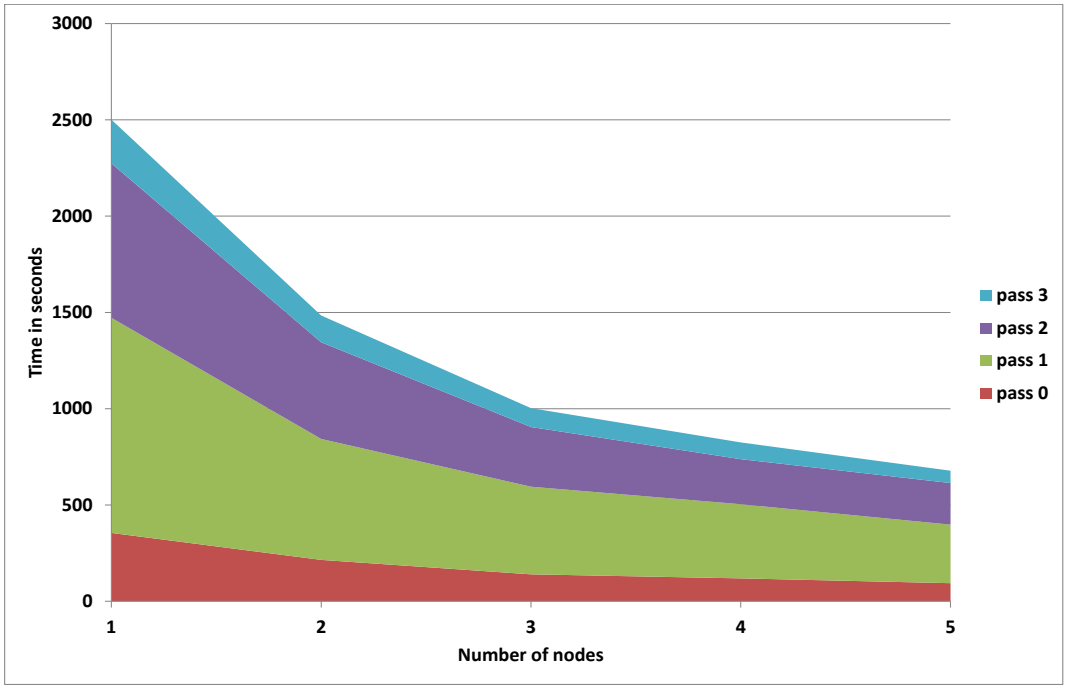

Figure 5.10: Running time of the multi-pass disjunctive conditions algorithm for SCM×x data set on different cluster sizes.

Multi-pass disjunctive conditions algorithm. In order to evaluate the *multi-pass composite conditions* algorithm, we fixed the dataset size at ×30 with a duplicated number of attributes and we varied the cluster size from 1 to 5. The number of the input atomic conditions is 14 therefore the number of candidate to be explored is equal to 2^{14}. Figure 5.10 shows the details of running time of each pass of the algorithm as we varied the cluster size. For each configuration, on average, 44% of the time is spent on computing candidates in *pass 1*, 33% on processing *pass 2*, 14% on processing *pass 0* and about 9% on processing the last pass. The reason that *pass 1* and *pass 2* have the important share in the execution time is due to the large number of candidates generated at this passes. Indeed, a theoretical number of candidates in *pass 1* starting with 14 atomic conditions is 4095 candidate conditions. In other hand, the last pass spent less time because the

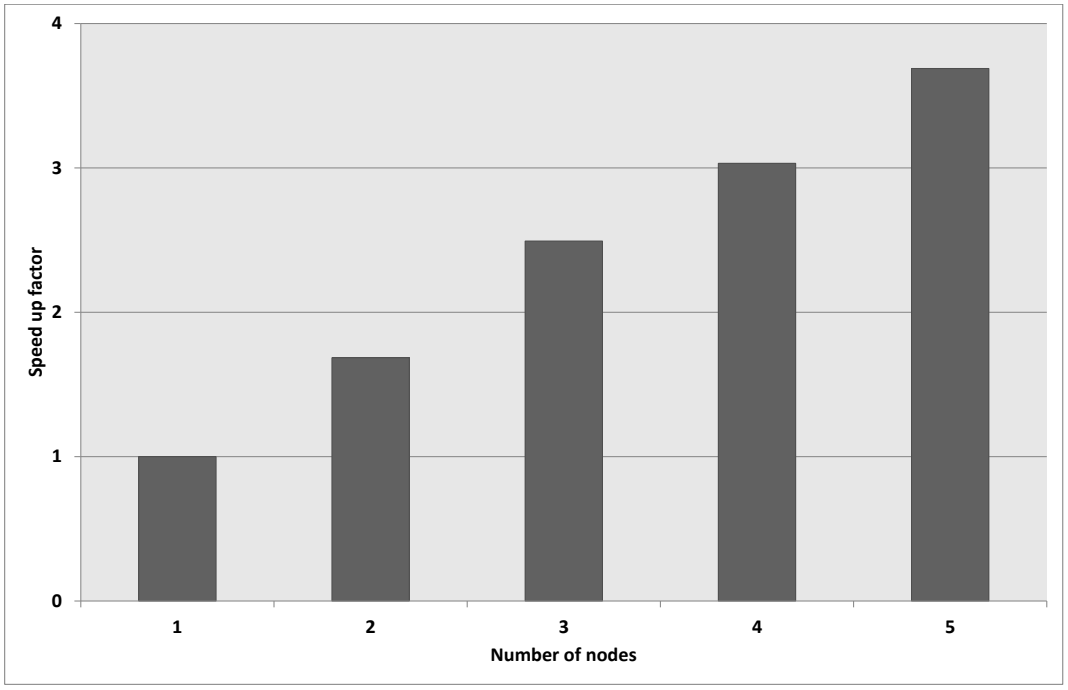

Figure 5.11: Relative running time of the multi-pass disjunctive conditions algorithm for SCM×x dataset set on different cluster sizes.

number of candidate conditions decreases (in the top of the lattice). Also, we observe that increasing the number of nodes decreases the execution time. To better understand the speed up characteristics of the *multi-pass composite conditions* algorithm,in Figure 5.11 we plotted the same results on relative scale. That is, the y-axis shows how much faster the running time becomes as we increase the cluster size. From this figure, we can observe that the execution time decreased by a factor of 3.7 which is better than the *single-pass* approach (by a factor of 2.4). The reason for this result is that, *(i)* in single-pass approach, nodes receive large workloads and may be overloaded or compute a large number of candidates where (ii) in *multi-pass* approach candidates conditions are redistributed over nodes for each pass. Moreover, nodes receive small workloads at each iteration.

5.5 Discussion

In this chapter we have studied the problem of discovering candidate composite conditions in parallel using the MapReduce framework. We proposed single and multi-pass

approaches and we provide two algorithms for the single stage approach to discover conjunctive and disjunctive candidate correlation conditions. We showed how to efficiently partition the space of computation across several nodes in the cluster in order to process each (sub)-partition independently from others. We also provided a useful property to eliminate unnecessary computation during computing process instance phase. We implement our approach on hadoop and we perform a set of experiments to evaluate both scalability and speed up of the algorithms. The experiments showed that the worst speed up was the *single-pass disjunctive conditions* algorithm with factor of 2.3 where the other algorithms exceeded the factor 3.2. In order to improve the speed up and scale up of these approach, a preprocessing step can be used to detect only interesting candidates by using pre pruning heuristics and measures. This idea is a subject of future works.

Chapter 6

Conclusions and Future Work

A challenging issue for business process monitoring/processing is event correlation which refers to grouping together event logs to identify end-to-end process instances. Correlating message logs based on their content is a requirement in various application domains. However, this task represents a real challenge because of, (i) modern enterprise systems become increasingly federated, loosely coupled and continually growing in size and complexity, (ii) simultaneously they generate a large size of event-data representing business activities stored in log files. Therefore, in today's event-driven systems, it is necessary to perform such task on multiple processing nodes in order to handle a large recorded data sets. In this thesis we described and analysed how event correlation discovery task can be supported efficiently on data-intensive parallel platform namely MapReduce.

We presented algorithms devoted to atomic correlation conditions discovery. We showed how to efficiently deal with problems such as partitioning, replication, and multiple inputs by manipulating the keys used to route the data between nodes of a MapReduce cluster. We also provided an adequate data structure used to store correlated messages in order to decrease memory usage. We proposed several alternatives consisting of one, two or many MapReduce jobs for discovering atomic correlation conditions. We also proposed a disk-based alternative to handle insufficient memory at Reduce side. We closed this chapter with an experimental evaluation of the proposed algorithms.

Also, we described algorithms devoted to discover composite candidate correlation conditions. We proposed two alternatives consisting on one and multi-pass MapReduce jobs. For the one-pass algorithm, we introduced the concept of partitioning conditions, a subset of the input conditions, in order to partition the space of computation vertically and perform the correlation discovery in a single MapReduce job. We also showed how to connect process instances to form new instances, in the case of disjunctive candidates, in order to reduce the computations. We proposed a multi-pass strategy based on a

horizontal partitioning of the lattice and process each level in a single MapReduce job. Finally, we performed experimental evaluation of both strategies on 5-nodes cluster.

Finally, the experimental results showed that the proposed algorithms can process large data sets within a reasonable time and they scale well w.r.t to dataset sizes and cluster sizes.

Future Work

In this thesis we focused on the problem of event correlation discovery using *MapReduce*. One possible future direction to improve our approach is to explore alternative data structures, for example, distributed non-relational database as Hbase [5] or HadoopDB [9] to integrate and store event related data. Such kind of data structure provides advantages in distributed cloud storage systems as tables are always sorted by their key and thus can be easily distributed horizontally over several machines. Besides this, it is interesting to suggest new statistic calculations, based on attributes values, to determine correlation among events. Using such storage gives the opportunity to provide a near to real-time incremental approach where events are processed as soon as they are received.

Also, it will be interesting to collaborate with data mining researchers to investigate well known data mining algorithms such as *Apriori* using MapReduce framework to improve the strategy of data partitioning.

This thesis addressed the problem of event correlation discovery from business event logs in parallel shared nothing framework, which is only the first step to achieve business process mining techniques. Another future research direction, is to extend the approach to construct (in parallel) the logical description of the business process. In this case, atomic conditions are used to discover sub models where disjunctive conditions are used to build the whole process model. This thesis is the starting step toward using MapReduce in business process management domain. Therefore, there are many issues to be studied. We expect that this area to grow up as there are many application domains, including monitoring business process and querying historical business events, that require event correlation discovery.

Bibliography

[1] *Workflow management: models, methods, and systems.* MIT Press, Cambridge, MA, USA, 2002.

[2] 2011. http://aws.amazon.com/.

[3] 2011. http://aws.amazon.com/ec2/instance-types/.

[4] 2011. http://managementsoftware.hp.com/products/soa.

[5] 2013. http://hbase.apache.org/.

[6] Ibm websphere business process management software., 2013. http://www-01.ibm.com/software/websphere/.

[7] Wil M. Aalst. Transactions on petri nets and other models of concurrency ii. chapter Process-Aware Information Systems: Lessons to Be Learned from Process Mining, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[8] Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and Eric Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Technol.*, 8(3):13:1–13:30, May 2008.

[9] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, August 2009.

[10] Foto N. Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 1–8, New York, NY, USA, 2011. ACM.

[11] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 132–143, New York, NY, USA, 2012. ACM.

[12] Rakesh Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 255–266, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[13] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Chile*, pages 487–499, 1994.

[14] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[15] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and algorithms*. Addison Wesley, 983.

[16] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.

[18] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 16–52, New York, NY, USA, 1986. ACM.

[19] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. *SIGMOD Rec.*, 15(2):16–52, June 1986.

[20] Roger S. Barga and Hillary Caituiro-Monge. Event correlation and pattern detection in cedr. In *Proceedings of the 2006 international conference on Current Trends in Database Technology*, EDBT'06, pages 919–930, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] 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.

[22] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[23] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distrib. Parallel Databases*, 29(3):185–216, June 2011.

[24] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[25] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[26] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Paul G. Brown and Peter J. Hass. Bhunt: automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 668–679. VLDB Endowment, 2003.

[28] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[29] Fabio Casati, Malú Castellanos, Umeshwar Dayal, and Norman Salazar. A generic solution for warehousing business process data. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande,

Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB*, pages 1128–1137. ACM, 2007.

[30] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[31] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, July 1998.

[32] Brian F. Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J. Kistler, P. P. S. Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, and Raymie Stata. Building a cloud for yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[33] 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.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[35] J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2004.

[36] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[37] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

[38] Xin Dong and Alon Y. Halevy. A platform for personal information management and integration. In *CIDR*, pages 119–130, 2005.

[39] Schahram Dustdar and Robert Gombotz. Discovering web service workflows using web services interaction mining. *International Journal of Business Process Integration and Management*, 1:256–266(11), 27 February 2007.

[40] Iman Elghandour and Ashraf Aboulnaga. Restore: reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, February 2012.

[41] Shimon Even. *Graph Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.

[42] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, December 2005.

[43] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[44] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, April 1995.

[45] Claude Godart and Olivier Perrin. *Les processus métiers: Concepts, modèles et systèmes*. Traité Informatique et Systèmes d'Information, IC2. Hermes, May 2009.

[46] Daniela Grigori, Fabio Casati, Malu Castellanos, Umeshwar Dayal, Mehmet Sayal, and Ming-Chien Shan. Business process intelligence. *Comput. Ind.*, 53(3):321–343, April 2004.

[47] Daniela Grigori, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. Improving business process quality through exception understanding, prediction, and prevention. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 159–168, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[48] Apache Hadoop, 2013. http://hadoop.apache.org/.

[49] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural network design*. PWS Publishing Co., Boston, MA, USA, 1996.

[50] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[51] Herodotos Herodotou and Shivnath Babu. A what-if engine for cost-based mapreduce optimization. *IEEE Data Eng. Bull.*, 36(1):5–14, 2013.

[52] Apache hive, 2013. http://hive.apache.org/.

[53] D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, January 1995.

[54] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 647–658, New York, NY, USA, 2004. ACM.

[55] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul G. Brown, and Ashraf Aboulnaga. Cords: automatic generation of correlation statistics in db2. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1341–1344. VLDB Endowment, 2004.

[56] Yannis Ioannidis, Raghu Ramakrishnan, and Linda Winger. Transitive closure algorithms based on graph traversal. *ACM Trans. Database Syst.*, 18(3):512–576, September 1993.

[57] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[58] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[59] Jaql, 2013. http://www.jaql.org/.

[60] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: an in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, September 2010.

[61] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999.

[62] Frank Leymann and Dieter Roller. *Production workflow: concepts and techniques*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[63] 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 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM.

[64] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, pages 1–2, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[65] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 76:1–76:8, New York, NY, USA, 2012. ACM.

[66] Guojun Liu, Ming Zhang, and Fei Yan. Large-scale social network analysis based on mapreduce. In *CASoN*, pages 487–490. IEEE Computer Society, 2010.

[67] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 191–202. VLDB Endowment, 2002.

[68] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledgediscovery. *Data Min. Knowl. Discov.*, 1(3):241–258, January 1997.

[69] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[70] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the 2005 ACM SIGMETRICS international*

conference on Measurement and modeling of computer systems, SIGMETRICS '05, pages 50–60, New York, NY, USA, 2005. ACM.

[71] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. *SIGMETRICS Perform. Eval. Rev.*, 33(1):50–60, June 2005.

[72] Hamid Motahari, Regis Saint-Paul, Boualem Benatallah, and Fabio Casati. Protocol discovery from web service interaction logs. In *IEEE ICDE 07*, April 2007.

[73] Hamid R. Motahari Nezhad. *Discovery and Adaptation of Process Views*. PhD thesis, THE UNIVERSITY OF NEW SOUTH WALES, 2008.

[74] 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.

[75] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, September 2010.

[76] Business Process Model Object Management Group and Notation, 2013. http://www.bpmn.org//.

[77] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 267–273, Berkeley, CA, USA, 2008. USENIX Association.

[78] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[79] Wim De Pauw, Robert Hoch, and Yi Huang. Discovering conversations in web services using semantic correlation analysis. In *ICWS*, pages 639–646. IEEE Computer Society, 2007.

[80] 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.

[81] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003.

[82] Erhard Rahm and Hong H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[83] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, Cambridge, 2012.

[84] 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.

[85] Hicham Reguieg, Farouk Toumani, Hamid Reza Motahari-Nezhad, and Boualem Benatallah. Using mapreduce to scale events correlation discovery for business processes mining. In *Proceedings of the 10th international conference on Business Process Management*, BPM'12, pages 279–284, Berlin, Heidelberg, 2012. Springer-Verlag.

[86] Szabolcs Rozsnyai, Aleksander Slominski, and Geetika T. Lakshmanan. Discovering event correlation rules for semi-structured business processes. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 75–86, New York, NY, USA, 2011. ACM.

[87] Szabolcs Rozsnyai, Roland Vecera, Josef Schiefer, and Alexander Schatten. Event cloud - searching for correlated business events. In *CEC/EEE*, pages 409–420. IEEE Computer Society, 2007.

[88] Sigal Sahar. Interestingness via what is not interesting. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '99, pages 332–336, New York, NY, USA, 1999. ACM.

[89] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[90] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 743–754, New York, NY, USA, 2004. ACM.

[91] Josef Schiefer, Heinz Roth, Hannes Obweger, and Szabolcs Rozsnyai. Event data warehousing for complex event processing. In Pericles Loucopoulos and Jean-Louis Cavarero, editors, *RCIS*, pages 203–212. IEEE, 2010.

[92] Weiyi Shang, Zhen Ming Jiang, Bram Adams, and Ahmed E. Hassan. Mapreduce as a general framework to support research in mining software repositories (msr). In Michael W. Godfrey and Jim Whitehead, editors, *MSR*, pages 21–30. IEEE, 2009.

[93] Mirko Steinle, Karl Aberer, Sarunas Girdzijauskas, and Christian Lovis. Mapping moving landscapes by mining mountains of logs: novel techniques for dependency model generation. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1093–1102. VLDB Endowment, 2006.

[94] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[95] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[96] I.H. Toroslu, G.Z. Qadah, and L. Henschen. An efficient database transitive closure algorithm. *Applied Intelligence*, 1994.

[97] 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.

[98] W M P Van Der Aalst. Process mining: Discovery, conformance and enhancement of business processes. *Media*, 136(2):352, 2011.

[99] 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.

[100] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, September 2004.

[101] Wil M. P. Van Der Aalst, Arthur H. M. Ter Hofstede, and Mathias Weske. Business process management: a survey. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pages 1–12, Berlin, Heidelberg, 2003. Springer-Verlag.

[102] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[103] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[104] Mathias Weske. Business process management: Concepts, languages, architectures. *Media*, 15(2):368, 2012.

[105] WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.

[106] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[107] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[108] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[109] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[110] Mohammed Javeed Zaki. Parallel sequence mining on shared-memory machines. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale*

Parallel KDD Systems, SIGKDD, pages 161–189, London, UK, UK, 2000. Springer-Verlag.

[111] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 30(2):425–436, May 2001.

[112] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 425–436, New York, NY, USA, 2001. ACM.

[113] Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071, 2010.

List of Figures

List of Tables

Using MapReduce To Scale Events Correlation Discovery For Process Mining

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 dissertation, 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.

Key words: MapReduce; Business Process; Process Mining; Process Discovery; Event Correlation.