Scalable Process Discovery using Map-Reduce

Joerg Evermann
Memorial University of Newfoundland, Canada

Abstract—Process discovery is an approach to extract process models from event logs. Given the distributed nature of modern information systems, event logs are likely to be distributed across different physical machines. Map-Reduce is a scalable approach for efficient computations on distributed data. In this paper we present Map-Reduce implementations of two well-known process mining algorithms to take advantage of the scalability of the Map-Reduce approach. We present the design of a series of mappers and reducers to compute the log-based ordering relations from distributed event logs. These can then be used to discover a process model. We provide experimental results that show the performance and scalability of our implementations.

Index Terms—Map-Reduce, Process Mining, Process Discovery, Alpha Algorithm, Flexible Heuristic Miner, Workflow Management

1 Introduction

Process mining is the analysis of event logs. Many information systems produce event logs that capture the actions of their users. Examples of event logs are page requests of web-servers and business-object method calls in ERP systems. Process discovery is that area of process mining that deals with the identification of the processes followed by system users, for example the process of ordering a product on an e-commerce web-site, or the process of scheduling a manufacturing order in an ERP system. Process discovery is important when the underlying systems are not process-aware [1], [2]. Another area of process mining is compliance assurance, determining whether the actually executed process conforms to a prescribed one [3]. Recently, process mining has been applied to web-service discovery and improvement [4].

Modern information systems, such as web-crawlers, web-servers, ERP systems, databases, etc., are increasingly distributed, with replicated instances deployed on multiple physical machines, for example as part of a load-balancing architecture or for geographic proximity to users. Given the distributed nature and the increasing size of event logs, it is natural to look for a distributed way to mine these for processes. The Map-Reduce approach [5] is a scalable means of analyzing distributed data and performing distributed computations on such data. In this paper, we describe how two well-known process mining algorithms, the Alpha algorithm [6] and the Flexible Heuristics Miner (FHM) [7], [8], [9], can be implemented using Map-Reduce. Such implementations take advantage of the natural fit between distributed event logs and distributed computations, to make the algorithms scalable to large data sets.

The remainder of the paper is structrured as follows. We first briefly present the general idea of process mining and the general Map-Reduce approach to distributed computation. We then describe our implementation of the Alpha algorithm using Map-Reduce, with a detailed description of the required mappers, reducers, and data types. Following this, we describe our implementation of the FHM algorithm using Map-Reduce, again with a detailed description of the mappers, reducers, and data types. Experimental results are presented for both algorithms to demonstrate performance and scalability. We then discuss our results and the relationship to prior work, before concluding with an outlook to future work.

2 Process Mining from Event Logs and Running Example

In this section, we illustrate the general concepts of process mining, using an example from [6]. We use this as a running example throughout the remainder of the paper.

Event logs minimally contain information about events referring to a case (process instance), the event type, and a timestamp. Events indicate the execution of instances of activities or tasks (e.g. the completion of a work item in a workflow-management system). Hence, event types refer to activities and we use the terms event type, task, and activity interchangeably in this paper. Consider the event log of 19 events for 5 cases presented in Table 1. Each case begins with execution of A and finishes with execution of D. If activity B is executed, then activity C is also executed. However, for some cases, activity C is executed before activity B. The time-ordered sequence of activities for each case is called a trace.

TABLE 1
An example event log

Case ID	Activity ID	Time Stamp
Case 1	Activity A	1
Case 2	Activity A	2
Case 3	Activity A	3
Case 3	Activity B	4
Case 1	Activity B	5
Case 1	Activity C	6
Case 2	Activity C	7
Case 4	Activity A	8
Case 2	Activity B	9
Case 2	Activity D	10
Case 5	Activity A	11
Case 4	Activity C	12
Case 1	Activity D	13
Case 3	Activity C	14
Case 3	Activity D	15
Case 4	Activity B	16
Case 5	Activity E	17
Case 5	Activity D	18
Case 4	Activity D	19

Definition 1. *(Trace) A trace $\sigma = t_1 \dots t_n$ is a temporally ordered sequence of events for one case (process instance) in an event log.* □

Our example log in Table 1 contains five traces, of which only three are unique:

$$\sigma_1 = \sigma_3 = ABCD, \sigma_2 = \sigma_4 = ACBD, \sigma_5 = AED$$

The workflow net [10] in Figure 1 shows one of an infinite number of process models that can give rise to the event log in Table 1.

Process mining approaches can be distinguished to what extent they are able to accommodate ″noisy″ data, where noise is defined not as errors in the log, but as ″outliers″ [1], i.e. highly unusual traces. For example, the Alpha algorithm [6] does not take ″noisy″ data into account, resulting in overly complex process models as the algorithm attempts to account also for unusual traces. In contrast, the Flexible Heuristics Miner (FHM) [7], [8], [9] was specifically designed to work on noisy data and in low-structured domains, i.e. domains where very different traces may be found.

Generally, process mining first defines and computes log-based ordering relations between the elements of a trace. The specific log-based ordering relations differ between process mining approaches, but are generally based on the concept of which activity follows (or does not follow) which other activity. The log-based ordering relations are subsequently aggregated. For example, it may be important to know whether or how often two events follow each other, but it is not important to retain each instance or associated trace. Because of this, once the log-based ordering relations are computed, the process mining algorithms operate on a comparatively small data set to derive the process model. However, the FHM algorithm does require individual traces for later phases of the algorithm (Section 5)

3 The Map-Reduce Framework

Map-Reduce is a programming approach for large scale data processing in a distributed computing environment [5]. Conceptually, it uses a two-phase approach.

In the first phase, the `map()` function accepts as input a series of `(InKey, InValue)` pairs from an input reader and provides as output a series of `(OutKey, OutValue)` pairs. Figure 2 illustrates this using a simple example from [11]. The input text files are provided to `map()` as a series of `(InKey, InValue)` pairs where the key represents the file offset, and the values represent distinct lines in the file. The output of `map()` is a series of `(OutKey, OutValue)` pairs where the key represents a year, and the value represents an observed temperature.

In the intermediate phase, this output of `map()` is presented to `shuffle()` which sorts the input by key and collects the values for different keys. It provides output as a series of tuples of the form `(OutKey, OutValue1, OutValue2, ...)`. In the example in Figure 2, this creates a list of temperatures for each year. This output set is partitioned and each partition sent to a reducer in the second phase.

In the second phase, the `reduce()` function takes as input a series of tuples of the form `(OutKey, OutValue1, OutValue2, ...)` and provides as output a series of `(Out2Key, Out2Value)` pairs. In the example in Figure 2, the reducer computes the maximum of the list of temperatures for each year.

The data types for `InKey, InValue, OutKey, OutValue, Out2Key, Out2Value` may be different from each other, and may be arbitrarily complex data types, provided that `OutKey` is comparable (i.e. a `compareTo()` function is provided). Hence, the important aspect in implementing a Map-Reduce based algorithm is the combination of suitable sequences of `map()` and `reduce()` function with appropriate data types for the keys and values.

The following sections define such a sequence of mappers, reducers, and data types for the Alpha and the FHM algorithm. Both algorithms require two map-reduces phases (″jobs″) to compute the log-based dependency relations that form the basis for the algorithms. The general idea for both implementations is to design the key and value data types so that the first reducer sees and operates on complete traces, and the second reducer sees and operates on complete information about the log-based ordering relations of an event pair (e.g. (A, B)) and its inverse (e.g. (B, A)). The FHM algorithm requires further map-reduce jobs to mine the split and join types.

4 Implementing the Alpha Algorithm using Map-Reduce

This section describes our implementation of the Alpha algorithm using the Map-Reduce framework. We

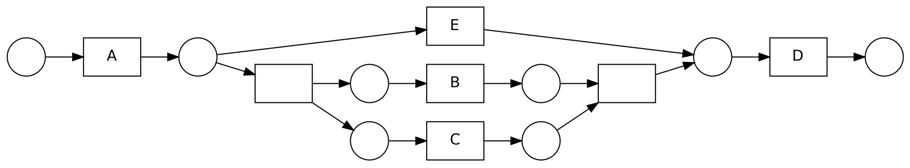


Fig. 1. A workflow net corresponding to the workflow log in Table 1

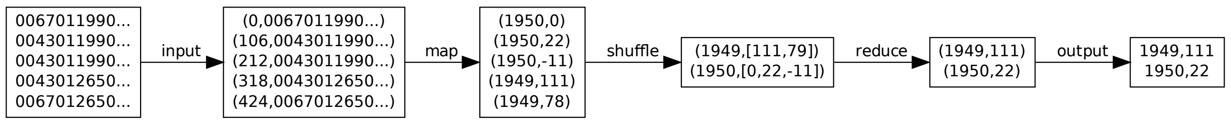


Fig. 2. Map-Reduce example, from [11]

begin by describing the algorithm with a focus on the log-based ordering relations.

4.1 The Alpha Algorithm

The Alpha process mining algorithm discovers one possible workflow net [10] from the observed event logs under the assumption that there is no noise in the event log data. The Alpha algorithm first identifies log-based ordering relationships between activities. These are termed "causal" relationships in [6]. For example, when an activity always follows another activity but not vice-versa, there may be a causal relationship between the activities. The identification of possible causal relationships is based on four types of ordering relations.

Definition 2. *(Log-based ordering relations for the Alpha algorithm) Let T be a set of activities and W be an event log over T. Let a, $b \in T$:*

- *$a >_w b$ iff there is a trace $\sigma = t_1t_2t_3 \dots t_{n-1}$ in W such that $\sigma \in W$ and $t_i = a$ and $t_{i+1} = b$ for $i \in \{1, \dots, n-2\}$*
- *$a \rightarrow_w b$ iff $a >_w b$ and $b \not>_w a$*
- *$a \|_w b$ iff $a >_w b$ and $b >_w a$*
- *$a \#_w b$ iff $a \not>_w b$ and $b \not>_w a$* □

The first relation ($>_w$) is the basic temporal order in the event log from which the other relations are computed. The second relation ($\rightarrow_w$) represents a possible causal order between activities. The third relation ($\|_w$) represents potential parallelism. The last relation ($\#_w$) represents those activities that never follow each other directly.

From these log-based ordering relations, the Alpha algorithm generates a workflow net (P_w, T_w, F_w). Specifically, T_w consists of event types which occur in at least one trace in W, $T_i \subseteq T_w$ consists of event types at the begining of traces in W and $T_o \subseteq T_w$ consists of event types at the end of traces in W. In our example:

$$T_w = \{A, B, C, D, E\}, \quad T_i = \{A\}, \quad T_o = \{D\}$$

Next, the set X_w is determined from the log-based ordering relations as follows. Let Q, R be sets of activities. Then $(Q, R) \in X_w$ iff there is a causal relation from each element of Q to each element of R (i.e. all pairwise combinations of elements of Q and R are in $\rightarrow_w$) and the members of Q and R are not in $\|_w$. In our example:

$$\begin{aligned} X_w = \{&(\{A\},\{B\}),(\{A\},\{C\}),(\{A\},\{E\}),(\{B\},\{D\}),\\ &(\{C\},\{D\}),(\{E\},\{D\}),(\{A\},\{B,E\}),\\ &(\{A\},\{C,E\}),(\{B,E\},\{D\}),(\{C,E\},\{D\})\} \end{aligned}$$

The set Y_w are the maximally contained elements of X_w. In our example:

$$\begin{aligned} Y_w = \{&(\{A\},\{B,E\}),(\{A\},\{C,E\}),(\{B,E\},\{D\}),\\ &(\{C,E\},\{D\})\} \end{aligned}$$

The set of places P_w contains an input place i_w, an output place o_w and one place $p_{(Q,R)}$ for every member of Y_w. In our example:

$$\begin{aligned} P_w = \{&i_w, o_w, p_{(\{A\},\{B,E\})}, p_{(\{A\},\{C,E\})}, p_{(\{B,E\},\{D\})},\\ &p_{(\{C,E\},\{D\})}\} \end{aligned}$$

Finally, the flow relation F_w is determined as the union of the following sets:

$$\begin{aligned} F_w = &\{(a, p_{(Q,R)})|(Q,R) \in Y_w \wedge a \in Q\} \cup\\ &\{(p_{(Q,R)}, b)|(Q,R) \in Y_w \wedge b \in R\} \cup\\ &\{(i_w, t)|t \in T_I\} \cup \{(t, o_w)|t \in T_O\} \end{aligned}$$

In our example:

$$F_w = \{(i_w, A), (A, P_{(\{A\},\{B,E\})}), (P_{(\{A\},\{B,E\})}, B), \ldots, (D, o_w)\}$$

It is important to note that the computation of the log-based ordering relations occurs on the event logs. The subsequent computations of P_w, T_w and F_w are performed on considerably smaller data sets, whose upper bound is a function of the number of distinct event types in the event logs, rather than the size of the event logs themselves. Hence, it is primarily the computation of the log-based ordering relations that can benefit from the distributed computation model provided by Map-Reduce. To compute these relations, we define two sets of mappers and reducers, following each other.

4.2 Map-Reduce for $>_w$ and $\not>_w$

In general, information for each case may not be entirely contained on a single physical compute node. For example, a single user session may span multiple physical compute nodes because of load-balancing reasons; a single user session may involve the logs of multiple systems on different nodes; or automatic log rotation may have split a single user session into different event log files. Hence partial case information may be present on different compute nodes and a mapper cannot assume complete case information. Thus, the first step is to extract the events and timestamps for each case from the log on each node so that complete case information is presented to the subsequent reducer.

The input to `map()` is a log file in the form of Table 1. The standard `TextInputFormat` presents a line of the log file to the `map()` function, which is parsed and emitted as a series of $(CaseID, (Event, TimeStamp))$ tuples. The output key is $CaseID$ so that the subsequent reducer sees the complete trace information. The timestamp information is retained because the subsequent `shuffle()` makes no guarantees about the order of the values in the list presented to `reduce()`. Formally:

$$\textbf{map1} : (\text{Int}, \text{Text}) \rightarrow set(\text{CaseID}, (\text{Event}, \text{TimeStamp}))$$

This set of tuples is the input to `shuffle()`, which collects the set of activities for each case ID, i.e. a trace as defined in Definition 1, as input to `reduce()`. Formally:

$$\textbf{shuffle1} : set(\text{CaseID}, (\text{Event}, \text{TimeStamp})) \rightarrow (\text{CaseID}, set(\text{Event}, \text{TimeStamp}))$$

The first reducer computes the log relation $>_w$. Each trace itself is sufficiently small, relative to the size of the event log. Hence, the reducer can operate on it as a whole in memory to sort events by time. As Definition 2 shows, further computations also require information about pairs of activities that are not in $>_w$, i.e. are in the log relation $\not>_w$. However, the absence of an activity pair in $>_w$ for a single trace does not imply that the activity pair is in $\not>_w$ as it may be in $>_w$ for a different case. Thus, we explicitly keep track of activity pairs in $\not>_w$ for each trace.

For example, consider the trace for case 1 in our example as emitted by `shuffle()`: $(1, ((A,1)(B,5)(C,6)(D,13)))$. The trace consists of four activities and hence sixteen possible event pairs. Because the computations of the log-based ordering relations in the subsequent reducer require information about (A,B) as well as (B,A), information about both event pairs must be presented to the same reducer instance, and thus require the same key for the subsequent `shuffle()`. Hence, we write each pair of event IDs in a canonical form, e.g. lowest event ID first, and additionally maintain information about the direction of the collected information, i.e. whether it represents (A,B) or (B,A). Thus, for our example, we need to consider only 10 distinct event pairs: (A,A), (A,B), (A,C), (A,D), (B,B), (B,C), (B,D), (C,C), (C,D), (D,D). For each pair, we must determine whether it (e.g. (A,B)) is in $>_w$ or $\not>_w$ for this trace and whether its "inverse" (e.g. (B,A)) is in $>_w$ or $\not>_w$ for this trace.

Consider the first event pair (A,B). According to the trace, A is immediately followed by B, while the opposite is incorrect, so that we write $[AB, (F^+, NF^-)]$. In our notation $'F'$ stands for "follows" and $'NF'$ for "not follows", expressing the relations $>_w$ and $\not>_w$, respectively. We use the superscripts $+$ and $-$ to indicate the "directionality", i.e. order of the pairs in the relation. Again, rather than associating information with (A,B) and (B,A) separately, we associate this information only with the pair (A,B). The superscripted $+$ shows whether the second event follows (not follows) the first one, while the superscripted $-$ means that the first event follows (not follows) the second one (i.e. signifies the "inverse" of the event pair). Thus, $(AB, (F^+, NF^-))$ means that (A,B) is in $>_w$ and (B,A) is in $\not>_w$.

The `reduce()` function computes the log-based ordering relations in this form for each case (trace) and emits them as key-value pairs, where the composite key is the event pair, while the value is a log relation, either F^+, F^-, NF^+ or NF^-, as illustrated in Figure 3. Formally,

$$\textbf{reduce1} : (\text{CaseID}, set(\text{Event}, \text{TimeStamp})) \rightarrow set((\text{Event}, \text{Event}), (\text{Boolean}, \text{Boolean}))$$

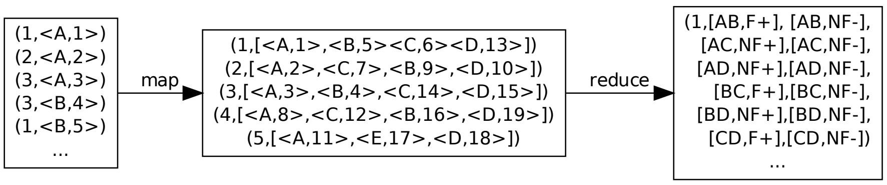


Fig. 3. The first Map-Reduce logical data flow for the Alpha algorithm

4.3 Map-Reduce for $\|_w$, $\#_w$ and $\rightarrow_w$

The possibly multiple reduce nodes in the first Map-Reduce phase may leave the information about each pair of event IDs scattered across different nodes. In the second phase, we use an identity `map()` function to read this information and use a combiner (`combine()`) to aggregate information for each pair of event IDs by removing redundant information. Formally:

$$\begin{aligned}\textbf{map2}: &((\text{Event}, \text{Event}), (\text{Boolean}, \text{Boolean}))\\ &\rightarrow ((\text{Event}, \text{Event}), (\text{Boolean}, \text{Boolean}))\\ \textbf{combine2}: &((\text{Event}, \text{Event}), (\text{Boolean}, \text{Boolean}))\\ &\rightarrow ((\text{Event}, \text{Event}), (\text{Boolean}, \text{Boolean}))\end{aligned}$$

Consider the following information that is passed unchanged by `map()` to `combine()`.

(AB, F^+)
(AB, NF^-)
(AB, NF^+)
(AB, NF^-)
(AB, F^+)
...

Removing the duplicate log relation values using `combine()`, the intermediate shuffle phase collects information for the same key (i.e. event pair) and provides the following input to `reduce()`:

$(AB, (F^+, NF^-, NF^+))$
...

Note that it is not possible to compute the log-based ordering relations in the `combine()` function, as the combiners are not guaranteed complete information about a pair of event IDs.

Each reducer is then presented with the complete information about each event pair (A, B) and its inverse (B, A). The reduce function can now compute and emit the log-based ordering relations $\rightarrow_w$, $\#_w$ and $\|_w$ according to Definition 2. Formally:

$$\begin{aligned}\textbf{reduce2}: &((\text{Event}, \text{Event}), set(\text{Boolean}, \text{Boolean}))\\ &\rightarrow set((\text{Event}, \text{Event}), \text{LogRelation}))\end{aligned}$$

The distinction between F and NF is formally a boolean, and the distinction between $+$ and is formally a boolean. Hence, $\{F+, NF-, NF+\}$ is formally set of 3 pairs of booleans.

In the example we have:

$(AB, \|_w)$
...

The second data flow is illustrated in Figure 4. The output of this final `reduce()` function provides the input of the later phases of the Alpha algorithm which constructs the workflow net.

The multiple reducers of this phase may leave outputs scattered in different files. As each key is processed only once, there are no duplicates to filter and the separate output files need only be combined. This can be done as a subsequent map-reduce job or after retrieving the results from the Hadoop cluster.

4.4 Performance

We conducted an experimental study to evaluate the scalability and effectiveness of our Alpha implementation. The PLG process log generator [12] can randomly create process models based on parameters such as the proportion of XOR splits, AND splits, etc. We used this to create a process model containing 47 activities. We then used PLG to create an event log with 10,000 randomly created traces from this process model. We replicated this event log 500 times, for a total of 5,000,000 event traces. The total file size was approximately 80GB, a size where the use of the Map-Reduce approach becomes viable. We processed these logs using the Amazon Elastic Map-Reduce (EMR) service that provides the Hadoop system[1], a widely adopted open-source implementation of Map-Reduce. Input, intermediate results and outputs were stored on the Amazon S3 service.

To provide a baseline for demonstrating the benefits of using the scalable map-reduce framework, we first configured a Map-Reduce cluster with a single medium size compute node[2], and specified a single map and single reduce task. Specifying the number of map tasks does not impose a hard constraint but is only a suggestion to the Hadoop framework. As a result, EMR ran two concurrent map tasks on the single compute node. This represents a single-server

1. http://hadoop.apache.org
2. AWS EC2 m1.medium machine type, single thread, 3.75GB

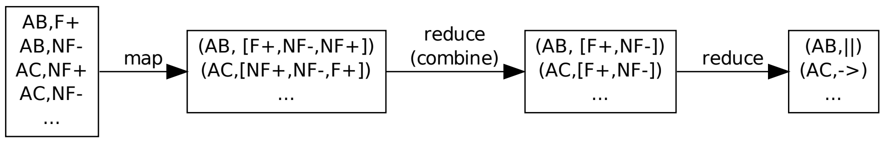


Fig. 4. The second Map-Reduce logical data flow for the Alpha algorithm

TABLE 2
Performance results for a single medium size Map-Reduce task node, total job execution time 1 day 1 hour

Phase	**CPU time (h:m:s.millis)**	**Number of output tuples**	**Number of tasks**	**Number of slots**
Phase 1, map	0:34:17.930	184,464,000	500	2
Phase 1, reduce	2:29:37.070	4,919,394,000	1	1
Phase 2, map	18:4:5.800	4,919,394,000	1113	2
Phase 2, combine		29,963,958		
Phase 2, reduce	0:5:29.750	1,012	1	1

TABLE 3
Performance results for a cluster of 10 medium size Map-Reduce task nodes, total job execution time 1 hour 24 minutes

Phase	**CPU time (h:m:s.millis)**	**Number of output tuples**	**Number of tasks**	**Number of slots**
Phase 1, map	0:32:47.060	184,464,000	500	20
Phase 1, reduce	2:34:53.150	4,919,394,000	21	10
Phase 2, map	11:40:7.200	4,919,394,000	1113	20
Phase 2, combine		29,964,565		
Phase 2, reduce	0:5:55.650	1,012	21	10

TABLE 4
Performance results for a cluster of 10 high performance Map-Reduce task nodes, total job execution time 8 minutes

Phase	**CPU time (h:m:s.millis)**	**Number of output tuples**	**Number of tasks**	**Number of slots**
Phase 1, map	0:54:58.430	184,464,000	500	480
Phase 1, reduce	2:2:59.620	4,919,394,000	108	120
Phase 2, map	8:43:31.160	4,919,394,000	1188	480
Phase 2, combine		30,203,398		
Phase 2, reduce	0:9:46.610	1,012	108	120

setup and serves as a useful baseline. The performance results and number of processed tuples at each phase are shown in Table 2. The total job execution time was 1 day and 1 hour.

We chose not to use existing implementations of the Alpha algorithm as a baseline for comparison, because these are in-memory implementations and do not scale beyond a certain data set size. We also wanted to focus on the scalability that can be achieved by a parallel map-reduce implementation and rule out differences in the use of internal data structures and implementation details of the algorithm.

Next, for a more realistic case, we provisioned a cluster with 10 medium instances as Hadoop task nodes in addition to master and controller nodes. The performance results and number of processed tuples at each phase are shown in Table 3. The total job execution time was 1 hour and 24 minutes. To further demonstrate the scalability, rather than increasing the number of nodes in the cluster, we increased the performance of each node. Using a cluster of 10 high-performance compute nodes[3] reduced the total job execution time to just 8 minutes (Table 4).

5 Implementing the Flexible Heuristics Miner using Map-Reduce

This section describes our implementation of the Flexible Heuristics Miner (FHM) algorithm using the Map-Reduce framework. This more recent algorithm was developed to address the realities of noisy event logs and low-structured domains [7], [8], [9].

3. AWS EC2 cc2.8xlarge machine type, 32 threads, 60.5GB

5.1 The Flexible Heuristic Miner Algorithm

The Flexible Heuristic Miner (FHM) algorithm is designed to be used with noisy event log data in that it allows the exclusion of rare or unusual traces that should be considered "outliers" for the discovery of the process model. The FHM algorithm defines three log-based ordering relations:

Definition 3. *(Log-based ordering relations for the FHM algorithm) Let T be a set of activities and W be an event log over T. Let a, $b \in T$:*

- *$a >_w b$ iff there is a trace $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ in W such that $\sigma \in W$ and $t_i = a$ and $t_{i+1} = b$ for $i \in \{1, \ldots, n-2\}$*
- *$a >>_w b$ iff there is a trace $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ in W such that $\sigma \in W$ and $t_i = t_{i+2} = a$ and $t_{i+1} = b$ for $i \in \{1, \ldots, n-3\}$*
- *$a >>>_w b$ iff there is a trace $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ in W such that $\sigma \in W$ and $t_i = a$ and $t_j = b$ and $i < j$ for $i, j \in \{1, \ldots, n-1\}$* □

The first relation ($>_w$) is the basic temporal order in the event logs (sequence of activities) that is also used by the Alpha algorithm and represents direct successors of activities in the log. The second relation ($>>_w$) represents loops of length two between activities. The third relation ($>>>_w$) represents the general successor relationship, either direct or indirect, irrespective of the distance of the two activities in the log.

From these log-based ordering relations, the FHM algorithm constructs "dependency measures" that express the relative frequency of the occurrence of elements in each log-based ordering relation. These dependency measures indicate the "certainty" that there is a true dependency relation between two events A and B, or that there is truly a loop of length two.

Definition 4. *(Dependency measures for the FHM algorithm) Let T be a set of activities and W be an event log over T. Let $a, b \in T$, $|a|$ the number of times a occurs in W, $|a >_w b|$ the number of times $a >_w b$ occurs in W, $|a >>_w b|$ the number of times $a >>_w b$ occurs in W, $|a >>>_w b|$ the number of times $a >>>_w b$ occurs in W. Then:*

$$a \Rightarrow_w b = \left(\frac{|a >_w b| - |b >_w a|}{|a >_w b| + |b >_w a| + 1} \right) \quad \textit{if } (a \neq b)$$

$$a \Rightarrow_w a = \left(\frac{|a >_w a|}{|a >_w a| + 1} \right)$$

$$a \Rightarrow_w^2 b = \left(\frac{|a >>_w b| - |b >>_w a|}{|a >>_w b| + |b >>_w a| + 1} \right)$$

$$a \Rightarrow_w^l b = 2 \left(\frac{|a >>>_w b| - \text{abs}(|a| - |b|)}{|a| + |b| + 1} \right)$$

□

Consider the following example traces, which have been slightly modified from the running example for the Alpha algorithm to include loops of length two:

$$\sigma_1 = \sigma_3 = ABACD, \sigma_2 = \sigma_4 = ACBCD, \sigma_5 = AED$$

TABLE 5
Subset of log-based relation counts for FHM for the example log. Counts for event types not listed are zero.

$\lvert A >_w B\rvert = 2$	$\lvert B >_w A\rvert = 2$
$\lvert A >_w C\rvert = 4$	$\lvert C >_w A\rvert = 0$
$\lvert B >_w C\rvert = 2$	$\lvert C >_w B\rvert = 2$
$\lvert C >_w D\rvert = 4$	$\lvert D >_w C\rvert = 0$
$\lvert A >_w E\rvert = 1$	$\lvert E >_w A\rvert = 0$
$\lvert E >_w D\rvert = 1$	$\lvert D >_w E\rvert = 0$
$\lvert A >>_w B\rvert = 2$	$\lvert C >>_w B\rvert = 2$
$\lvert A >>>_w A\rvert = 2$	$\lvert A >>>_w B\rvert = 2$
$\lvert A >>>_w C\rvert = 4$	$\lvert A >>>_w D\rvert = 7$
$\lvert B >>>_w D\rvert = 4$	$\lvert B >>>_w C\rvert = 2$
$\lvert C >>>_w C\rvert = 2$	$\lvert C >>>_w D\rvert = 2$

TABLE 6
Subset of dependency measures for FHM for the example log. Dependencies for event types not listed are zero.

$A \Rightarrow_w B = 0$	$B \Rightarrow_w A = 0$
$A \Rightarrow_w C = 4/5$	$C \Rightarrow_w A = 0$
$B \Rightarrow_w C = 0$	$C \Rightarrow_w B = 0$
$C \Rightarrow_w D = 4/5$	$D \Rightarrow_w C = 0$
$A \Rightarrow_w E = 1/2$	$E \Rightarrow_w D = 1/2$
$A \Rightarrow_w^2 B = 2/3$	$C \Rightarrow_w B = 2/3$
$A \Rightarrow_w^l A = 4/15$	$A \Rightarrow_w^l B = -1/6$
$A \Rightarrow_w^l C = 3/7$	$A \Rightarrow_w^l D = 10/13$
$B \Rightarrow_w^l D = 3/5$	$B \Rightarrow_w^l C = 0$
$C \Rightarrow_w^l C = 4/13$	$C \Rightarrow_w^l D = 1/6$

Table 5 shows the frequencies for the log-based ordering relations for the FHM algorithm. Table 6 shows the resulting dependency measures for the different types of dependencies.

The FHM algorithm uses these frequency-based dependency measures to define a dependency graph. The algorithm to construct the dependency graph can be found in Definition 6 in [9]. It is at this phase that the algorithm can take noise into account by requiring certain threshold frequencies to be met in order for the dependency to be considered "real" or "true". Five such thresholds are defined:

- δ_a: The absolute dependency threshold
- δ_{L1L}: The length-one-loops threshold
- δ_{L2L}: The length-two-loops threshold
- δ_l: The long-distance dependency threshold
- δ_r: The relative-to-best threshold

By default, $\delta_a = \delta_{L1L} = \delta_{L2L} = \delta_l = 0.9, \delta_r = 0.05$. In our example, all dependencies are below the default thresholds of 0.9 because of the small log size. This reflects the fact that such a small sample size does not permit much confidence in the support of the relation, even when relations are frequent in the

TABLE 7
Dependency Graph for Running Example (with $\delta_a = \delta_{L1L} = \delta_{L2L} = 0.5$)

Pre X	Activity X	Post X
$\{B\}$	A	$\{B, C, D, E\}$
$\{A, C\}$	B	$\{A, C, D\}$
$\{A, B\}$	C	$\{B, D\}$
$\{B, C, E\}$	D	$\{\}$
$\{A\}$	E	$\{D\}$

TABLE 8
Augmented CNet for Running Example (with $\delta_a = \delta_{L1L} = \delta_{L2L} = 0.5$)

Pre X	Activity X	Post X
$\{\{B\}^2\}$	A	$\{\{B\}^2, \{C\}^4, \{E\}^1\}$
$\{\{A\}^2, \{C\}^2\}$	B	$\{\{A\}^2, \{C\}^2\}$
$\{\{A\}^4, \{B\}^2\}$	C	$\{\{B\}^2, \{D\}^4\}$
$\{\{C\}^4, \{E\}^1\}$	D	$\{\}$
$\{\{A\}^1\}$	E	$\{\{D\}^1\}$

log. For our example, the dependency graph with $\delta_a = \delta_{L1L} = \delta_{L2L} = 0.5$ is given in Table 7.

As the name implies, the dependency graph indicates only which event types depend on other event types, but does not indicate whether a particular event type is followed by an AND, an XOR, or an OR split (or, conversely, whether a particular event type is preceded by an AND, an XOR, or an OR join). To identify the splits in the process model, each trace is run "forwards" against the dependency graph. For example, if the dependency graph, as in our example, states that tasks B, C, D, and E depend on task A (i.e. they can possibly be activated by the occurrence of task A), we wish to find out whether A in some traces activates B and in other traces C, or whether in all traces B and C are activated, or some combination. The first case would represent an XOR split, the second case would represent an AND split, and the third case an OR split. To identify joins in the process model, the same process is used but the traces are run "backwards" against the dependency graph. The result of this step is an augmenented causal net (CNet) which contains information about the frequencies with which one event activates another event in the workflow log. The reader is referred to Section IV B in [9] for details. The augmented CNet for our example is shown in Table 8.

As with the Alpha algorithm, we focus on the design of the appropriate mappers and reducers for the computation of the dependency measures, the dependency graph, and the augmented causal net. We define five sets of mappers and reducers.

5.2 Map-Reduce for the Log-based Ordering Relations

The first Map-Reduce phase computes the log-based ordering relations $>_w$, $>>_w$, $>>>_w$ as well as the element counts $|a|$ that are required for computing the long-distance dependency $\Rightarrow^l_w$ in the next phase.

The mapper of the first phase is identical to the first-phase mapper for the Alpha algorithm. Again, the input to `map()` is a log file in the form of Table 1, which is parsed and emitted as a series of $(CaseID, (Event, TimeStamp))$ tuples. The output key is $CaseID$ so that the subsequent reducer sees the complete trace for a case. The time stamp is retained because `shuffle()` makes no guarantees about the order of the values in the list presented to `reduce()`. Formally:

$$\begin{aligned}\textbf{map1} :& (\text{Int}, \text{Text}) \\ & \rightarrow set(\text{CaseID}, (\text{Event}, \text{TimeStamp})) \\ \textbf{shuffle1} :& set(\text{CaseID}, (\text{Event}, \text{TimeStamp})) \\ & \rightarrow (\text{CaseID}, set(\text{Event}, \text{TimeStamp}))\end{aligned}$$

As Definition 4 shows, further computations require information about pairs of activities as well as about the "inverse" pair. Thus, the design of the first-phase reducer is similar to that for the Alpha algorithm, in that we again require information about the pair (A, B) and the pair (B, A) to be seen by the same subsequent reducer to compute the dependency measures. For example, the computation of $a \Rightarrow_w b$ requires information about $a >_w b$ as well as $b >_w a$. Hence, as for the first-phase reducer for the Alpha algorithm, we associate information about both "directions" with the same event pair and maintain an indicator of the "directionality".

For example, consider the trace for case one in our example as emitted by `shuffle()`: $(1, ((A, 1)(B, 5)(A, 6)(C, 7)(D, 13)))$. As with the first-phase reducer of the Alpha algorithm, we write each pair of event IDs in a canonical form, e.g. lowest first. Thus, we need to consider only 10 distinct event pairs: (A, A), (A, B), (A, C), (A, D), (B, B), (B, C), (B, D), (C, C), (C, D), (D, D). For each pair, we must determine whether it (e.g. (A, B)) is in $>_w$, $>>_w$ or $>>>_w$ and whether its "inverse" (e.g. (B, A)) is in $>_w$, $>>_w$, or $>>>_w$.

Consider the first event pair (A, B). According to the trace, A is immediately followed by B, while the opposite is not the case, so that we write $[AB, F^+]$. In our notation, $'F'$ stands for "directly follows" and expresses the relation $>_w$. Similar to the output of the first-phase reducer for the Alpha algorithm, the superscripted $+$ and $-$ signs indicate "directionality": A superscripted $+$ indicates that the second event directly follows the first one, while $-$ means that the first event directly follows the second one (i.e. signifies the "inverse" of the event pair). We compute information about the two other log-based ordering relations ($>>_w$ and $>>>_w$) in a similar way and write $[AB, L2^+]$ or $[AB, L2^-]$ to signify that either $(A, B) \in >>_w$ or $(B, A) \in >>_w$ ("$L2$" signifies length-two-loops). We write $[AB, S^+]$ or $[AB, S^-]$ to signify that either $(A, B) \in >>>_w$ or $(B, A) \in >>>_w$ ("S" signifies the general (direct or indirect) successor relation). Additionally, we also compute the frequency of each element and emit each occurrence of an event A in the same format as $[AA, C^+]$ ("C" signifies the count of each element). This last relation is directionless and by default associated with a $+$ superscript.

The `reduce()` function computes the log-based ordering relations in this form for each case (trace) and emits them as key-value pairs. The composite key is the event pair, while the value is a tuple comprising an integer expressing the type of log-based ordering relation ("F", "$L2$", "S", or "C"), a boolean value indicating the directionality, and an integer expressing the number of occurrences of this relationship in this trace. Formally:

$$\begin{aligned}\textbf{reduce1} :&(\text{CaseID}, set(\text{Event}, \text{TimeStamp}))\\ &\rightarrow set((\text{Event}, \text{Event}),\\ &\qquad(\text{Integer}, \text{Boolean}, \text{Integer}))\end{aligned}$$

Including the occurrence counts for each case, rather than emitting each occurrence as a separate tuple, significantly reduces the data volume written to the file system after this phase, which must be read in the subsequent phase. Figure 5 illustrates the data flow in the first phase.

5.3 Map-Reduce for the Dependency Measures

The possibly multiple reduce nodes in the first Map-Reduce phase may leave the information about each pair of event IDs scattered across different nodes. The map function for this phase is the identity map that simply reads the information and allows the shuffle phase to provide complete information for each pair of event IDs to a subsequent reducer. As with the implementation of the Alpha algorithm, we are able to use a combiner for optimization. Because the dependency measures (Definition 4) are commutative, but not associative, as required for a combiner function, the combiner does not compute dependency measures, but instead it aggregates occurrence count information for each relation for each event pair as a key. Formally:

$$\begin{aligned}&\textbf{combine2} :\\ &\quad set((\text{Event}, \text{Event}), (\text{Integer}, \text{Boolean}, \text{Integer}))\\ &\quad \rightarrow set((\text{Event}, \text{Event}), (\text{Integer}, \text{Boolean}, \text{Integer}))\end{aligned}$$

Consider for example the following output by the mapper (subscripts indicate occurrence counts, duplicates occur because these tuple were produced by different reducers of the previous phase):

(AB, F_1^+)
$(AB, L2_1^-)$
$(AB, L2_1^+)$
$(AB, L2_1^-)$
(AB, F_1^+)
(AB, F_1^-)
(AB, F_1^+)
...

The combiner counts, for each event pair, and for each direction (indicated by the superscripted $+$ or $-$), the number of occurrences of the log-based ordering relation in its input (subscripted). For this example, it emits the following output tuples:

(AB, F_3^+)
$(AB, L2_2^-)$
$(AB, L2_1^+)$
(AB, F_1^-)
...

The intermediate shuffle phase collects the complete information for each event pair and provides it to the `reduce()` function. In our example, assuming that no other mapper processes information about the event pair (A, B), the `shuffle()` function presents the following as input to `reduce()`:

$(AB, (F_3^+, L2_2^-, L2_1^+, F_1^-))$
...

The reduce function computes and emits the dependency measures according to Definition 4. First, the reducer aggregates the occurrence counts of the log-based ordering relation that it receives for each event pair and for each direction (indicated by the superscripted $+$ or $-$) in the same way as the combiner did. Thus, in our example, it computes $|A >_w B|$ and $|B >_w A|$, $|A >>_w B|$ and $|B >>_w A|$, and $|A >>>_w B|$ and $|B >>>_w A|$. For our example, the reducer determines the following counts:

$|A >_w B| = 3$
$|B >_w A| = 1$
$|A >>_w B| = 1$
$|B >>_w A| = 2$
...

Next, the reducer uses these occurrence counts to compute the dependency measures (Definition 4). For our example, the reducer emits the following dependency measures:

$A \Rightarrow_w B = 0.4$	(emitted as $(AB, F, 0.4)$)
$B \Rightarrow_w A = -0.4$	(emitted as $(BA, F, -0.4)$)
$A \Rightarrow_w^2 B = 0.75$	(emitted as $(AB, L2, 0.75)$)
$B \Rightarrow_w^2 A = 0.75$	(emitted as $(BA, L2, 0.75)$)
...	...

The reducer output, in contrast to the mapper output, does not associate different directionalities with the same event pair: $A \Rightarrow_w B$ is emitted seperately from $B \Rightarrow_w A$. To reduce the data volume, only dependency measures > 0 are emitted.

The long-distance dependency measure $\Rightarrow_w^l$ requires not only information about the count of the successor (direct or indirect) relation, but also the occurrence counts for each of the events. Because

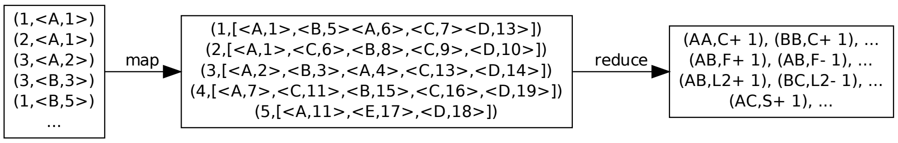

(1,<A,1>)
(2,<A,1>)
(3,<A,2>)
(3,<B,3>)
(1,<B,5>)
...
map
(1,[<A,1>,<B,5><A,6>,<C,7><D,13>])
(2,[<A,1>,<C,6>,<B,8>,<C,9>,<D,10>])
(3,[<A,2>,<B,3>,<A,4>,<C,13>,<D,14>])
(4,[<A,7>,<C,11>,<B,15>,<C,16>,<D,19>])
(5,[<A,11>,<E,17>,<D,18>])
reduce
(AA,C+ 1), (BB,C+ 1), ...
(AB,F+ 1), (AB,F- 1), ...
(AB,L2+ 1), (BC,L2- 1), ...
(AC,S+ 1), ...

Fig. 5. The first Map-Reduce logical data flow for the FHM algorithm

it is not possible to join this information in this Map-Reduce phase as it is associated with different keys, the reducer emits the raw occurrence counts $|A >>>_w B|, |B >>>_w A|, |A|$, and $|B|$ instead of the dependency measure.

Formally, a dependency measure is a tuple comprising an event pair, an integer value expressing the type of dependency measure (directly follows "F", length-two-loops "$L2$", successor "S") and a floating point value containing the measure itself. The key that is created by the reducer is a constant integer value c for all tuples, as the following phase (construction of the dependency graph) requires all tuples to be provided to the same reducer.

$$\begin{aligned}\textbf{reduce2}: &\\ &((\text{Event}, \text{Event}), set(\text{Integer}, \text{Boolean}, \text{Integer}))\\ &\rightarrow set(c, (\text{Event}, \text{Event}, \text{Integer}, \text{Float}))\end{aligned}$$

The second data flow is illustrated in Figure 6.

5.4 Map-Reduce for the Dependency Graph

The third Map-Reduce phase constructs the dependency graph from the information produced by the previous phase. The map function for this phase is an identity map that simply collects the information produced by the previous phase reducers (recall that each tuple had the same constant key). This information, and user-supplied information about the thresholds $\delta_a, \delta_{L1L}, \delta_{L2L}, \delta_l, \delta_r$, is provided to a single reducer. The reducer uses the algorithm in Definition 6 of [9] to construct the dependency graph. In contrast to the description in [9] the long-distance relationships are added to the dependency graph at this point, rather than after the computation of the split/join information in the next step, so that a recomputation of that information is not required. Formally,

$$\begin{aligned}\textbf{reduce3}: &\\ &set(c, (\text{Event}, \text{Event}), set(\text{Integer}, \text{Float}))\\ &\rightarrow set(c, (\text{Event}, \text{Event}))\end{aligned}$$

The dependency graph is simply a set of event pairs that indicates which event depends on which other event(s). For example, the dependency graph in Table 7 is encoded as follows:

$$\begin{aligned}&(1, (A, B))\\ &(1, (A, C))\\ &\dots\\ &(1, (C, D))\\ &(1, (E, D))\end{aligned}$$

While the data volume flowing into the reducer is very small compared to the size of the original log information, this phase represents a bottle-neck in the amount of possible parallelization of the FHM algorithm, as the dependency graph is a pre-requisite for the following phase of computing the split/join information.

5.5 Map-Reduce for the Augmented Causal Nets (Split/Join Information)

Phase 4 of the Map-Reduce implementation of the FHM algorithm uses the dependency graph and the event logs to compute the split and join frequencies for each event type in the dependency graph. The dependency graph that is produced in the previous phase is "side-loaded" through Hadoop's distributed cache mechanism to each compute node in this phase and is read by each reducer instance on initialization.

The map function for this phase is identical to that in the first phase, and in the first phase of the Alpha algorithm. It reads the log files and emits a series of $(CaseID, (Event, TimeStamp))$ tuples. The subsequent shuffle phase then provides complete traces to each reducer. Formally, a trace is a tuple of a case ID and a set of event-timestamp pairs.

$$\begin{aligned}\textbf{map4}: &(\text{Int}, \text{Text})\\ &\rightarrow set(\text{CaseID}, (\text{Event}, \text{TimeStamp}))\\ \textbf{shuffle4}: &set(\text{CaseID}, (\text{Event}, \text{TimeStamp}))\\ &\rightarrow (\text{CaseID}, set(\text{Event}, \text{TimeStamp}))\end{aligned}$$

The reducer function for this phase of the FHM implementation accepts a trace and, with the aid of the dependency graph, computes the split and join information as described in [8], [9]. The output of the reducer is an augmented causal net of the form in Table 8 as a set of tuples comprising the activating or activated event type (i.e. that event for which split/join information is provided), a boolean value indicating directionality (i.e. whether this tuple expresses split or join information), a set of event

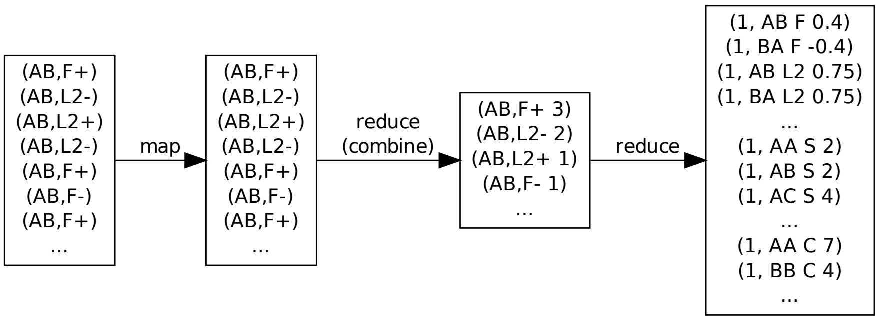


Fig. 6. The second Map-Reduce logical data flow for the FHM algorithm

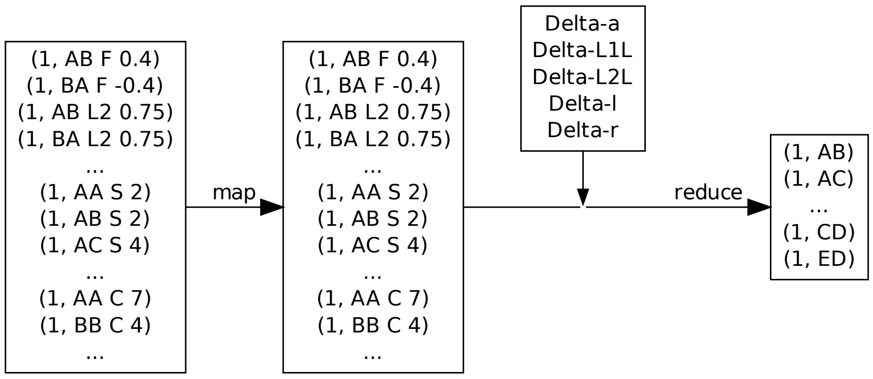


Fig. 7. The third Map-Reduce logical data flow for the FHM algorithm

types (i.e. the set that is activated by or activates the focal event), and the frequency count. For example, the augmented CNet in Table 8 would be output as key-value pairs as follows:

$((A, \{B\}, -), 2)$
$((B, \{A\}, -), 2)$
$((B, \{C\}, -), 2)$
$((C, \{A\}, -), 4)$
$((C, \{B\}, -), 2)$
$((D, \{C\}, -), 4)$
$((D, \{E\}, -), 1)$
$((E, \{A\}, -), 1)$
$((A, \{B\}, +), 2)$
$((A, \{C\}, +), 4)$
$((A, \{E\}, +), 1)$
$((B, \{A\}, +), 2)$
$((B, \{C\}, +), 2)$
$((C, \{B\}, +), 2)$
$((C, \{D\}, +), 4)$
$((E, \{D\}, +), 1)$

We use the information about the event types as keys and the frequency count as value so that the reducer of the subsequent phase can aggregate the count information for the same keys. The data flow in this phase of the FHM algorithm is shown in Figure 8. Formally, the reduce function for this phase is defined as:

$$\begin{aligned}\textbf{reduce4}\ :&(\text{CaseID}, set(\text{Event}, \text{TimeStamp}))\\ &\rightarrow set((\text{Event}, set(\text{Event}), boolean), \text{Integer})\end{aligned}$$

Finally, to collect the information from different reducers in this phase and to aggregate frequency counts, a final map-reduce phase is added. The map in this phase is an identity map function. The reducer in this phase receives the specific event types as key and a set of frequency counts as values. The output is the sum of frequency counts for that key. The reduce function is formally defined as:

$$\begin{aligned}\textbf{reduce5}\ :&((\text{Event}, set(\text{Event}), boolean), set(\text{Integer}))\\ &\rightarrow ((\text{Event}, set(\text{Event}), boolean), \text{Integer})\end{aligned}$$

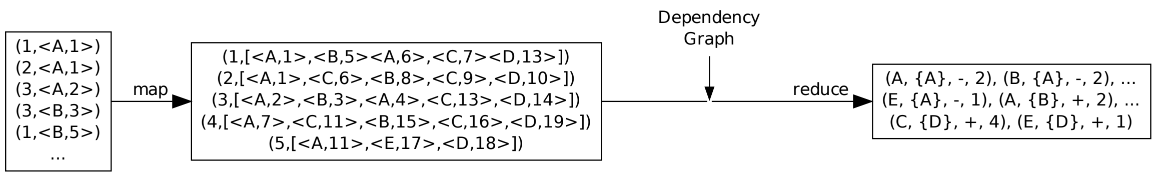

(1,<A,1>)
(2,<A,1>)
(3,<A,2>)
(3,<B,3>)
(1,<B,5>)
...
map
(1,[<A,1>,<B,5><A,6>,<C,7><D,13>])
(2,[<A,1>,<C,6>,<B,8>,<C,9>,<D,10>])
(3,[<A,2>,<B,3>,<A,4>,<C,13>,<D,14>])
(4,[<A,7>,<C,11>,<B,15>,<C,16>,<D,19>])
(5,[<A,11>,<E,17>,<D,18>])
Dependency Graph
reduce
(A, {A}, -, 2), (B, {A}, -, 2), ...
(E, {A}, -, 1), (A, {B}, +, 2), ...
(C, {D}, +, 4), (E, {D}, +, 1)

Fig. 8. The fourth Map-Reduce logical data flow for the FHM algorithm

5.6 Performance

Again, we conducted an experimental study to evaluate the scalability and effectiveness of our FHM implementation. We used the same inputs as for the evaluation of our Alpha implementation. We also used the same configuration of the Amazon Elastic Map Reduce service. The performance results and number of processed tuples at each phase are shown in Tables 9 to 11. On our base-line single compute node, the total job execution time was 22 hours and 21 minutes. On a cluster of 10 medium-sized compute nodes, the total job execution time was 2 hours and 1 minute. To further demonstrate the scalability, we again increased the performance of each node. Using a cluster of 10 high-performance compute nodes reduced the total job execution time to 17 minutes (Table 11). As with the performance evaluation of the Alpha algorithm, we conducted this experiment as a proof-of-concept to demonstrate that our approach results in scalable computation of the log-based ordering relations using Map-Reduce.

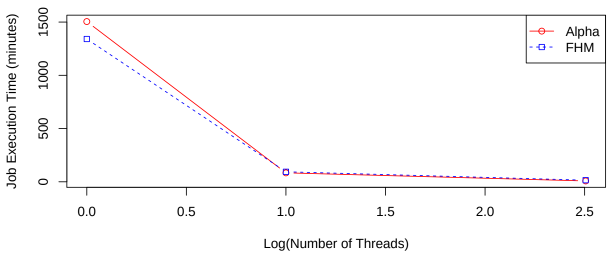


Fig. 9. Total job execution time as a function of available compute threads

6 Discussion

We have seen that running both the Alpha and the FHM algorithm on only a single machine takes approximately a full day on a data volume of only 5 million event traces (80GB). Our experiments have shown that both algorithms can be efficiently parallelized, as the computation of log-based relations for different pairs of event types are independent. Further, running individual traces against the dependency graph in the later stage of the FHM can also be done in parallel as the individual traces are independent. Hence, we have achieved total job execution times as low as as 8 minutes for the Alpha algorithm, and as low as 15 minutes for the FHM algorithm. Figure 9 also shows the total job execution times as a function of the number of total compute threads available on the cluster, showing graphically how the algorithm implementations scale with increasing computational resources.

A look at Tables 4 and 11 shows that our high-performance experimental condition is essentially the limit of the parallelization for this input size, with almost as many map slots as map tasks available in most phases and fewer reduce tasks than available slots in most phases. Further parallelization can possibly be achieved by splitting the input log files into smaller partitions, although the configurations used in our experiment are default Hadoop configurations that represent a good balance between parallelization and compute overhead. Both algorithms appear to be of approximately the same complexity (taking the same time), and the improvements due to paralellization are approximately the same.

The performance results can still be improved upon by carefully tuning the Map-Reduce cluster configuration and by improving the individual compute node performance, e.g. by including solid state drives (SSD) for storage. However, we conducted this experiment as a proof-of-concept in order to demonstrate that our approach results in scalable, efficient computation of the log-based order relations using Map-Reduce, rather than as a demonstration of the many existing Map-Reduce tuning techniques [13]. While 80GB log files do not necessarily require a Map-Reduce approach and could possibly even be performed in-memory on a single node, it is easy to imagine log file sizes of 80TB or even 80PB, which do require a distributed, secondary-storage based approach like the one used here.

We note three features of the Map-Reduce implementations of these algorithms that may present challenges. First, both implementations require multiple reducers that are sometimes preceded by identity map functions. However, while the Map-Reduce framework provides options to have multiple mappers

TABLE 9
Performance results for a single medium size Map-Reduce task node, total job execution time 22 hours 21 minutes

Phase	CPU time (h:m:s.millis)	Number of output tuples	Number of tasks	Number of slots
Phase 1, map	0:34:7.910	184,464,000	500	2
Phase 1, reduce	3:57:26.640	2,729,277,000	1	1
Phase 2, map	11:23:30.150	2,729,277,000	1112	2
Phase 2, combine		47,236,112		
Phase 2, reduce	0:3:50.100	1,849	1	1
Phase 3, map	0:0:2.430	1,849	4	2
Phase 3, reduce	0:0:1.660	269	1	1
Phase 4, map	0:34:17.870	184,464,000	500	2
Phase 4, reduce	1:34:53.890	261,699,500	1	1
Phase 5, map	0:47:5.680	261,699,500	103	2
Phase 5, combine		238,793		
Phase 5, reduce	0:0:18.350	390	1	1

TABLE 10
Performance results for a cluster of 10 medium size Map-Reduce task nodes, total job execution time 1 hour 34 minutes

Phase	CPU time (h:m:s.millis)	Number of output tuples	Number of tasks	Number of slots
Phase 1, map	0:31:21.910	184,464,000	500	20
Phase 1, reduce	4:06:30.560	2,729,277,000	21	10
Phase 2, map	7:45:19.820	2,729,277,000	1113	20
Phase 2, combine		47,314,036		
Phase 2, reduce	0:5:3.490	1,849	21	10
Phase 3, map	0:0:19.860	1,849	36	20
Phase 3, reduce	0:0:1.770	269	1	10
Phase 4, map	0:32:48.590	184,464,000	500	20
Phase 4, reduce	1:35:7.310	261,699,500	21	10
Phase 5, map	0:49:06.440	261,699,500	105	20
Phase 5, combine		232,663		
Phase 5, reduce	0:0:4.330	390	1	10

and a single reducer in a Map-Reduce job, there is currently no option to provide a single mapper and multiple reducers. The required identity mapper for the second phase of the implemented algorithms means that a large data volume has to be written to and then read again from the file system, limiting performance. While newer versions of Hadoop introduce the `ChainReducer` class, this only allows mappers to be added after the reducer, not to add further reducers. Similarly, while Cascading[4] allows the simple definition of multi-phase Map-Reduce applications, it is ultimately built on Map-Reduce and thus inserts identity mappers between different phases. Other computational approaches, like Pig[5] and Hive[6] are abstractions that are also built on top of Map-Reduce and therefore, similar to Cascading, possibly provide an easier definition of the computations, but do not address this challenge.

Second, as the design of the mapper and the numbers reported in Tables 2–4 and Tables 9–11 show, the volume of data throughout the process does not monotonically decrease. Instead, the output of the first-phase reducer for both algorithms is much larger than the input (both in terms of key-value pairs as well as in terms of bytes written to the file system). This characteristic of the algorithms compounds the Map-Reduce limitation that we just discussed. Future research in this area may focus on specifically designing a process discovery algorithm with awareness of the Map-Reduce limitations or identifying an alternative scalable, distributed computational approach without the limitations of the Map-Reduce framework.

Third, for the FHM algorithm in particular, the computation of a single dependency graph as the middle phase of the algorithm presents a "bottleneck" in the degree of parallelization that can be achieved. During this phase, a single reducer is required, leaving significant compute resources idle (albeit for a short time, as the construction of the graph is relatively quick).

7 Related Work

While process mining and Map-Reduce are both active research areas, we are only aware of one prior study that has combined the two. Reguieg et al. [14] were the first who applied Map-Reduce to process

4. www.cascading.org
5. pig.apache.org
6. hive.apache.org

TABLE 11
Performance results for a cluster of 10 high-performance Map-Reduce task nodes, total job execution time 15 minutes

Phase	CPU time (h:m:s.millis)	Number of output tuples	Number of tasks	Number of slots
Phase 1, map	0:54:39.110	184,464,000	500	480
Phase 1, reduce	3:0:33.690	2,729,277,000	108	120
Phase 2, map	6:26:1.400	2,729,277,000	1188	480
Phase 2, combine		47,441,548		
Phase 2, reduce	0:9:50.180	1,849	108	120
Phase 3, map	0:0:36.970	1,849	108	480
Phase 3, reduce	0:0:2.330	269	1	120
Phase 4, map	0:56:21.420	184,464,000	500	480
Phase 4, reduce	1:9:2.620	261,699,500	108	120
Phase 5, map	1:22:16.260	261,699,500	540	480
Phase 5, combine		293,006		
Phase 5, reduce	0:0:7.260	390	1	120

discovery. However, their approach aims at discovering ″event correlations″ in systems where events are not explicitly associated with cases through a case ID. Using Map-Reduce, event logs are mined to identify events that satisfy ″correlation conditions″. Once the correlation between events are identified, the process model for each set of the recognized process instances can be discovered. However, the actual process discovery from correlated event logs is outside the scope of their Map-Reduce based approach. Our study differs from [14] in that the input event logs in our study contain an explicit case identifier so that event correlation is not needed. Their approach can be viewed as a complement to ours to perform event correlation if required, followed by our own approach to mine workflow models, both using the scalable Map-Reduce approach.

While the Map-Reduce based approaches presented here are suitable for the discovery of a process model at a single point in time, a related challenge is to update the discovered process model based on additional events in order to avoid having to repeatedly perform computations over the entire log. An adaptation of the heuristics miner [7], [9] for ″Streaming Process Discovery″ (SPD) has been proposed in [15]. While that approach does not update the discovered process model for each new event but instead updates a finite queue of events that is used for the mining algorithm to periodically re-discover the process model, the streaming heuristics miner [15] could extend our proposal in a complementary manner: Once the initial process model has been discovered from the existing large event logs, periodic updates could be performed using the proposal in [15].

Strongly related to our approach is the work on log decomposition to tackle the problem of big logs using a divide and conquer strategy by van der Aalst [16], [17], [18], [19]. The notion of passages is proposed in [16] as a way to decompose logs. Passages are special pairs of sets of event types whose log projections may be analyzed separately and the resulting process models be merged with correctness guarantees. A drawback to passages is that prior knowledge about the process model is required to identify passages. This makes passages more suitable to distributed conformance checking (another branch of process mining) rather than process discovery. Later work by van der Aalst [17], [18] generalized the approach to provide correctness guarantees when projecting the logs on arbitrary but overlapping sets of event types. In that work, the discovered partial models are merged using a model merging approach. This was also the approach that our own prior work [20] used intuitively when first implementing the Alpha algorithm on Map-Reduce. However, we based our process on the specifics of the Alpha algorithm (and the FHM algorithm in this work) whereas [17], [18] generalize their correctness guarantees to any process discovery algorithm. One can view this work as an instantiation or implementation of van der Aalst's divide and conquer approach, as we distribute the processing for different event pairs to different reduce nodes and combining results later. In our case, we do not require process model merging techniques as the results for the Alpha and FHM algorithm can simply be combined as the set union of individual result tuples (after phase 2 of the Alpha algorithm, in phases 3 and 5 of the FHM algorithm).

More generally situated within the application of scalable ″Big Data″ techniques to process mining, instead of specifically focused on process discovery, are two other studies. Munoz-Gama et al. [21] recognize the challenge posed by very large event logs. However, they investigate the question of how best to perform conformance checking of large event logs against a given process model. Similarly, Ekanayake et al. [22] also recognize the challenge of large event logs. However, they tackle the problem that large logs tend to produce ″spaghetti-like″ process models. Their approach improves on the existing trace clustering by splitting process models by model variants as well as hierarchically in order to generate models of reduced complexity. Their approach is also complementary to ours: Given event logs of sufficient size and with a large number of activity types, the result of the Alpha algorithm will also be "spagehtti-like" and can be transformed as proposed in [22].

8 Conclusion

Workflow mining collects runtime information to discover a process model from the recorded event log of an information system. In this paper, we addressed the process discovery problem for large event logs by using the Map-Reduce framework. Map-Reduce offers a scalable model for distributed computation across multiple cluster nodes and is a natural fit with the distributed nature of modern information systems and their event logs. We presented a sequence of Map-Reduce operations to derive the log-based ordering relations that are the input for the Alpha algorithm and a second sequence of Map-Reduce operations to derive the dependency measures for the Flexible Heuristics Miner (FHM). We presented the results of experimental studies to demonstrate the performance and scalability of our implementation.

While the challenge that large data sets pose to process mining and process management in general has been recognized, this is the first work that applies the widely-used Map-Reduce framework to the problem of process discovery. Given the increasing availability of data and the importance of "Big Data" management to organizations, our future work will investigate how our Map-Reduce based implementation can be usefully combined with other techniques to reduce the complexity that stems from large event logs, such as log partitioning or trace clustering [1] and how other mining algorithms can be paralellized on Map-Reduce. For example, genetic algorithms are inherently parallel in nature, and this might lend itself to implementing genetic process mining [23], [24] efficiently on Map-Reduce.

References

[1] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Heidelberg, Germany: Springer Verlag, 2011.

[2] ——, "Process mining: Overview and opportunities," *ACM Trans. Manage. Inf. Syst.*, vol. 3, no. 2, pp. 7:1–7:17, Jul. 2012.

[3] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance checking of service behavior," *ACM Trans. Internet Technol.*, vol. 8, no. 3, pp. 13:1–13:30, May 2008.

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

[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04, Berkeley, CA, USA, 2004, pp. 10–10.

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

[7] A. Weijters and W. van der Aalst, "Rediscovering workflow models from event-based data using little thumb," *Integrated Computer-Aided Engineering*, vol. 10, no. 2, pp. 151–162, 2004.

[8] A. Wijters and J. Ribeiro, "Flexible heuristics miner (FHM)," Eindhoven University of Technology, BETA Working Paper Series WP334, 2010.

[9] A. Weijters and J. Ribeiro, "Flexible heuristics miner (FHM)," in *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining CIDM 2011, Paris, France*, 2011.

[10] W. M. P. van der Aalst, "The application of petri nets to workflow management," *The Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[11] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Sebastopol, 2009.

[12] A. Burattin and A. Sperduti, "PLG: A framework for the generation of business process models and their execution logs," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, M. zur Muehlen and J. Su, Eds. Springer Berlin Heidelberg, 2011, vol. 66, pp. 214–219.

[13] S. B. Joshi, "Apache hadoop performance-tuning methodologies and best practices," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 241–242.

[14] H. Reguieg, F. Toumani, H. Motahari-Nezhad, and B. Benatallah, "Using mapreduce to scale events correlation discovery for business processes mining," in *Business Process Management*, ser. Lecture Notes in Computer Science, A. Barros, A. Gal, and E. Kindler, Eds. Springer Berlin Heidelberg, 2012, vol. 7481, pp. 279–284.

[15] A. Burattin, A. Sperduti, and W. M. P. van der Aalst, "Heuristics miners for streaming event data," *CoRR*, vol. abs/1212.6383, 2012.

[16] W. M. van der Aalst, "Decomposing process mining problems using passages," in *Proceedings of Petri Nets 2012*, S. Haddad and L. Pomello, Eds., 2012, pp. 72–91.

[17] ——, "Decomposing petri nets for process mining: A generic approach," *Distributed and Parallel Databases*, vol. 31, pp. 471–507, 2013.

[18] ——, "Process mining in the large: A tutorial," in *Proceedings of eBISS 2013*, 2014, pp. 33–76.

[19] ——, "A general divide and conquer approach for process mining," in *Proceedings of the 2013 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2013, pp. 1–10.

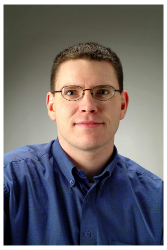
[20] J. Evermann and G. Assadipour, "Big data meets process mining: Implementing the alpha algorithm with map-reduce," in *Proceedings of the ACM Symposium on Applied Computing*, 2014.

[21] J. Munoz-Gama, J. Carmona, and W. M. van der Aalst, "Conformance checking in the large: Partitioning and topology," in *Business Process Management*, ser. Lecture Notes in Computer Science, F. Daniel, J. Wang, and B. Weber, Eds. Springer Berlin Heidelberg, 2013, vol. 8094, pp. 130–145.

[22] C. C. Ekanayake, M. Dumas, L. Garćia-Bañuelos, and M. Rosa, "Slice, mine and dice: Complexity-aware automated discovery of business process models," in *Business Process Management*, ser. Lecture Notes in Computer Science, F. Daniel, J. Wang, and B. Weber, Eds. Springer Berlin Heidelberg, 2013, vol. 8094, pp. 49–64.

[23] W. M. van Der Aalst, A. A. de Medeiros, and A. Weijters, "Genetic process mining," in *Applications and Theory of Petri Nets 2005*. Springer, 2005, pp. 48–69.

[24] A. Medeiros, A. Weijters, and W. Aalst, "Genetic process mining: an experimental evaluation," *Data Mining and Knowledge Discovery*, vol. 14, no. 2, pp. 245–304, 2007.



Joerg Evermann received his PhD in Information Systems from the University of British Columbia. Prior to being a faculty member at Memorial University, Dr. Evermann was a lecturer in Information Systems with the School of Information Management at the University of Wellington, New Zeland. Dr. Evermann's interests are in business process management, statistical research methods, and information integration. Dr. Evermann has published his research in more than 60 peer-reviewed publications. His work has appeared in high-quality journals, such as IEEE Transactions on Software Engineering, IEEE Transactions on Knowledge and Data Engineering, Organizational Research Methods, Structural Equation Modeling, Journal of the AIS, Information systems, and Information Systems Journal.