Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A decentralized approach for mining event correlations in distributed system monitoring

Gang Wu[a], Huxing Zhang[a], Meikang Qiu[b], Zhong Ming[c,*], Jiayin Li[b], Xiao Qin[d]

[a] School of Software, Shanghai Jiao Tong University, Shanghai, 200240, China
[b] Department of Electrical and Computer Engineering, University of Kentucky, Lexington, KY 40506, USA
[c] College of Computer Science and Software, Shenzhen University, Shenzhen 518060, China
[d] Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA

ARTICLE INFO

Article history:
Received 30 March 2011
Received in revised form
28 August 2012
Accepted 14 September 2012
Available online 23 September 2012

Keywords:
Distributed system
Data mining
Decentralized
Event correlations

ABSTRACT

Nowadays, there is an increasing demand to monitor, analyze, and control large scale distributed systems. Events detected during monitoring are temporally correlated, which is helpful to resource allocation, job scheduling, and failure prediction. To discover the correlations among detected events, many existing approaches concentrate detected events into an event database and perform data mining on it. We argue that these approaches are not scalable to large scale distributed systems as monitored events grow so fast that event correlation discovering can hardly be done with the power of a single computer. In this paper, we present a decentralized approach to efficiently detect events, filter irrelative events, and discover their temporal correlations. We propose a MapReduce-based algorithm, *MapReduce-Apriori*, to data mining event association rules, which utilizes the computational resource of multiple dedicated nodes of the system. Experimental results show that our decentralized event correlation mining algorithm achieves nearly ideal speedup compared to centralized mining approaches.

1. Introduction

With the growth of large scale distributed systems such as cluster systems and cloud computing systems [24], the key to building an efficient and reliable distributed environment is to monitor and control nodes, services, and applications. Monitoring such a large system requires continuous collecting of performance attribute values (i.e., CPU-usage, memory-usage) with a fixed frequency. As the system scales, the overhead of monitoring can become prohibitive [29].

Events detected during monitoring can help administrators to quickly pinpoint and troubleshoot bottlenecks, failures and other problems. For example, Fisichella et al. in [15] proposed a three stage process to detect unsupervised public health events. By the term event, we mean that a performance attribute whose observed value exceeds a given threshold, and failure events can be regarded as a special case. However, as the system complexity continues to grow, failures become norms instead of exceptions [44]. Conventional approaches such as checkpointing often prove counter-effective [16]. Thus, research on failure management has shifted onto failure prediction and related proactive management technologies [25,30].

It has been long recognized that events are not independent but correlated. Past studies [40] on failure analysis revealed important patterns in failure distribution. In particular, the events are temporally correlated in long time spans [37]. Previous research efforts have shown event correlation is helpful for resource allocation, job scheduling, and failure prediction [43,16,17].

Event correlation patterns, which describe the co-occurrence among different events, can be discovered through data mining approaches. However, classic data mining algorithms such as *Apriori* [33] suffer from exponential exploring space to mine frequent patterns. As data mining is both computationally intensive and data intensive, when data sets are large, scaling up the performance of data mining is a crucial challenge. Besides, Apriori based approaches are incapable of discovering temporal relationship among detected events. Previous works [5,36] mine temporal event correlations in sensor networks. However, in these approaches, all data is stored in a centralized database. As the scale of the system increases, aggregating events into one centralized database will be less efficient.

Centralized data mining approaches are not applicable in several cases. First, in some scientific fields where monitoring may last over years, the detected events would be too huge to be

* Corresponding author.
E-mail addresses: wugang@cs.sjtu.edu.cn (G. Wu), ralf0131@sjtu.edu.cn (H. Zhang), qiumeikang@yahoo.com, mqiu@engr.uky.edu (M. Qiu), mingz@szu.edu.cn (Z. Ming), jli6@engr.uky.edu (J. Li), xqin@auburn.edu (X. Qin).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.09.007

Table 1
Notations for monitoring and mining framework.

Symbol	Description
λ	The monitoring time interval
σ	The duration for monitoring
ξ_{low}	The lower threshold of a given attribute
ξ_{up}	The upper threshold of a given attribute
TID	The unique ID for a transaction
lifetime	The temporal duration of a given item
sup	The temporal support of a given item
conf	The confidence of a given association rule
min_sup	The minimal temporal support threshold
min_conf	The minimal confidence threshold

centralized. Second, in some time critical fields, event correlation mining can hardly be done with the power of a single computer in acceptable time. In contrast, decentralized data mining is particularly suitable for applications that typically deal with very large amounts of data (e.g., transaction data, scientific simulation, and telecom data) that cannot be analyzed through a traditional paradigm within an acceptable time. We also argue that events do not exist throughout the monitoring period. Therefore, the temporality of event correlations must be addressed.

In this paper, we focus on discovering temporal event correlations efficiently. To speed up the process of mining event correlations, we suggest that events should be aggregated to a set of databases rather than a centralized database, where the event correlation mining can be done in parallel. Moreover, to reduce the aggregating overhead, events are supposed to be locally filtered in each node before being passed. Finally, a Map-Reduce based event correlation mining is performed on the set of databases to discover event correlations in parallel. Event correlations are regarded as event association rules, which indicate the temporal relations among events based on common intervals of activities. The event association rules facilitate to predict the failure occurrence of the system as well as to improve the system's quality-of-service. For example, we are expected to receive an event from a certain node within a certain time interval, but it does not appear actually. Then we may infer that a failure is likely to occur on that node.

The contributions of this paper are summarized as follows:

1. We improve the event-correlation-mining approach by taking into account the temporality of detected events.
2. We propose an approach to efficiently filtering irrelative events locally to reduce the number of events to be aggregated.
3. We present a MapReduce-based algorithm to mine event correlations among a set of dedicated nodes in parallel.

The remainder of this paper is organized as follows: Section 2 describes the model and concepts. Section 3 gives an example to illustrate our approach. Section 4 introduces the detailed algorithms. Experiments are given in Section 5. Section 6 reviews the related work. Section 7 concludes the paper as well as discussing the future work.

2. Models and concepts

In this section, we first introduce the monitoring and event detection framework, and then give the model for mining monitoring association rules. Table 1 summarizes the notation we used.

2.1. Monitoring framework

The structure of the distributed system is considered as a hierarchical monitoring tree, where the nodes are divided into *Super Node* (SN), *Admin Node* (AN) and *Working Node* (WN). SN is the root node of the entire monitoring tree, which has a global control of the system. AN is a middle (non-leaf) node, which is in charge of a set of WNs. Each WN is a leaf node in the monitoring tree. The task of detecting events is distributed among dedicated monitoring agents on each WN. An agent is an application-level monitoring program that runs independently of other applications in the system and communicates with the outside world via message-passing [4]. For each WN, it has a local storage to keep the detected events during monitoring. All the nodes share a *Global Distributed File System* (GDFS), which is built upon their local storage. Each node has the ability to download files from GDFS to its local storage, as well as upload files to GDFS.

A monitoring request contains long running activities used for observation, analysis, and control of large scale distributed systems and applications they host. Each agent periodically collects values of certain attributes and compares them with given thresholds. Formally, we define a monitoring request as follows:

Definition 2.1. A monitoring request req $= (\lambda, \sigma, \{(a, \xi^a_{\mathrm{low}}, \xi^a_{\mathrm{up}}) \mid a \in A, \xi^a_{\mathrm{low}}, \xi^a_{\mathrm{up}} \in \Phi\})$, where λ is the monitoring time interval and σ is the duration for monitoring. It also contains a set of tuples, each consists of a monitoring attribute **a** in attribute set A and its correspondent lower threshold ξ^a_{low} and upper threshold ξ^a_{up} in threshold set Φ.

The monitoring request is initiated by SN, and then it propagates to all WNs through ANs. Finally, it activates the agent to start monitoring. The goal of monitoring is to check whether a set of system performance attributes, i.e., CPU utilization, memory utilization, network bandwidth, and any kind of custom attributes at application level, have exceeded a correspondent threshold. If the threshold is exceeded at a certain time slot in a certain place, an event is detected. Note that we are only interested in whether an event is detected, rather than the corresponding attribute value. Formally, we give the definition of event as follows:

Definition 2.2. Let A be a set of attributes we observe, define event $e = (|a| > \xi^a_{\mathrm{up}}$ or $|a| < \xi^a_{\mathrm{low}})$, where $a \in A$ is the attribute whose value is larger than the upper threshold ξ^a_{up} or is smaller than the lower threshold ξ^a_{low}.

2.2. Event correlation mining framework

Based on the association rule definition in a traditional database, we define *item*, *transaction* and *transaction database* in our domain.

Let $N = \{n_1, n_2, \ldots, n_m\}$ be a set of nodes in a distributed system. Assume that time is divided into equal-sized slots $\{t_1, t_2, \ldots, t_n\}$ such that $t_{i+1} - t_i = \lambda$ for all $1 \leq i \leq n$, where λ is the size of each slot.

Definition 2.3. An item $i = (e, n)$ is a couple of event and node, where $n \in N$ and e is an event detected on node n.

Definition 2.4. A transaction $T_t = \{i_1, i_2, \ldots, i_n\}$ is a set of items collected at time slot t.

Definition 2.5. A transaction database DB $= \{T_1, T_2, \ldots, T_n\}$ consists of a set of transactions, each of which is associated with a unique identifier, called its TID. We say that a transaction T contains X, a set of certain items, if $X \subseteq T$.

For example, let a be the CPU utilization and $\xi^a_{\mathrm{low}} = 10\%$, $\xi^a_{\mathrm{up}} = 50\%$. Assume that at time slot t_0, it is observed that CPU utilization is 60% on node n_1 and is 55% on node n_2. Thus an event $e = (|a| > 50\%)$ is detected both on nodes n_1 and n_2. Hence, a transaction at time slot t_0 contains two items, $i_1 = (e, n_1)$ and $i_2 = (e, n_2)$, which is denoted by $T_{t_0} = \{i_1, i_2\}$.

Each item has its life cycle in DB, which explicitly represents the temporal duration of it. Below we define *lifetime* of an item.

Definition 2.6. Let i be an item, define the lifetime of i, $[t_i, t_j]$, as a time interval within which the associated event has been detected, where $1 \leq i \leq j \leq n$. It is denoted by: $lifetime(i) = l(i) = [t_i, t_j]$.

For instance, if event e is detected at time slot 1, 2, and 3 on node n, then $lifetime((e, n)) = [1, 3]$.

Definition 2.7. Let D be a set of items. If the cardinality of D is k, D is called k-itemset, denoted by D_k. The lifetime of a k-itemset ($k > 1$) is defined as follows:

$$lifetime(D_k) = \bigcap_{i=1}^{n} lifetime(D_i), \quad D_i \in D_k. \tag{1}$$

As an example, let $k = 2$, item $i_1 = (e_1, n_1)$, item $i_2 = (e_2, n_2)$, $lifetime(i_1) = [1, 3]$, $lifetime(i_2) = [2, 4]$, thus $lifetime(\{i_1, i_2\}) = lifetime(i_1) \cap lifetime(i_2) = [2, 3]$.

Subsequently, we extend the existing association rule model to give the definition of *temporal support* and *confidence*.

Definition 2.8. Let D be a set of items, the temporal support of D, denoted by $sup(D, lifetime(D))$ or $sup(D, l(D))$, is the number of transaction that contains D over its lifetime. Support is also referred to as frequency.

A k-itemset D is called *frequent* (or *large*) k-th item set if its frequency is greater than or equal to a given minimum temporal support threshold. Note that frequency and support are used interchangeably here. The add operation of temporal support is defined as follows:

$$sup(i_1, l(i_1)) + sup(i_2, l(i_2)) = sup(i_1 \cup i_2, l(i_1) \cap l(i_2)). \tag{2}$$

Definition 2.9. The event association rule is defined in the form of $(D_1 \Rightarrow D_2, lifetime(D_1 \cup D_2), conf)$, where D_1, D_2 are both a set of items and $D_1 \cap D_2 = \varnothing$. The rule $(D_1 \Rightarrow D_2, l)$ holds with confidence $conf$ if $conf$% of transactions in DB that contain D_1 also contain D_2 within $lifetime(D_1 \cup D_2)$. It is the conditional probability, which can be denoted as follows:

$$conf\,(D_1 \Rightarrow D_2, l(D_1 \cup D_2)) = \frac{sup(D_1 \cup D_2, l(D_1 \cup D_2))}{sup(D_1, l(D_1 \cup D_2))}. \tag{3}$$

We now formally define the event correlation mining problem in distributed system monitoring as follows:

Problem Statement 1: Given an set of events E, a set of nodes N, a set of items $I = \{(e, n) | e \in E, n \in N\}$, a minimal temporal support min_sup, and a minimal confidence min_conf, find all frequent itemsets $\cup_k F_k$, such that $\forall D \in \cup_k F_k$, $sup(D, l(D)) \geq min\_sup$. Find all event association rules R, such that $\forall r = (D_1 \Rightarrow D_2, l(D_1 \cup D_2), conf) \in R$, $conf \geq min\_conf$, where $D_1, D_2 \in \cup_k F_k$, $D_1 \cap D_2 = \varnothing$, and $D_1 \cup D_2 \in \cup_k F_k$.

3. Motivational examples

In this section, we use an illustrative example to show how we are able to mine frequent event correlations in our scenario. Let $N = \{n_1, n_2, n_3\}$ be the nodes in a particular distributed system. Let $a_1 = cpu\_utilization$, $a_2 = memory\_utilization$, $a_3 = network\_utilization$, $\xi_{up}^{a_1} = 50\%$, $\xi_{up}^{a_2} = 60\%$, $\xi_{up}^{a_3} = 55\%$, $\xi_{low}^{a_1} = \xi_{low}^{a_2} = \xi_{low}^{a_3} = 0\%$, which implies event $e_1 = (|a_1| > \xi_{up}^{a_1})$, $e_2 = (|a_2| > \xi_{up}^{a_2})$, $e_3 = (|a_3| > \xi_{up}^{a_3})$. Assume that the time slot size is equal to 5 min and the monitoring process is initiated at 00:00 and lasts 30 min.

Table 2 shows the detected events within 30 min. For example, at the end of 00:00, node n_1 detects event e_1, node n_2 detects both events e_2 and e_3, both nodes n_3 and n_5 detect event e_3, and nothing is detected in node n_4. The monitoring process is repeated periodically at the end of each time slot until the end of the monitoring period.

To apply our algorithm, we first transform the collected events into a form of transactions. Table 3 shows the monitoring data in the form of transactions. Note that the transactions may be distributed in different nodes.

Our objective is to find the frequently occurred correlations between different items. In general, an Apriori-like iterative process is conducted. Each pass finds a frequent itemset in parallel on multiple nodes. Table 4 shows the temporal support for each item and finds frequent 1-itemset with given minimal temporal support $min\_sup = 3$. Only 3 items are identified as frequent 1-itemset after this pass.

Table 5 shows candidate 2-itemset based on frequent 1-itemset as well as frequent 2-itemset. As a result, only 1 frequent 2-itemset is identified. Our process terminates here because no further candidate itemset is found.

The result reveals useful information that item (n_1, e_1) and (n_2, e_3) are frequently co-occurred, which can be concluded as an association rule:

$((n_1, e_1) \Rightarrow (n_2, e_3), 10 \text{ min}, 100\%)$.

Table 2
Events detected on different nodes.

Time slot	n_1	n_2	n_3	n_4	n_5
00:00	e_1	e_2, e_3	e_3	/	e_3
00:05	e_1	e_2, e_3	/	/	/
00:10	e_1	e_3	e_2	e_1	/
00:15	/	e_3	/	e_1	e_3
00:20	e_2	/	e_1	e_1, e_2	e_2
00:25	e_2	/	e_1	e_1, e_2	e_1
00:30	/	e_1	/	/	e_1

Table 3
Monitoring data in the form of transactions.

TID	Time slot	Transaction
001	00:00	$(n_1, e_1), (n_2, e_2), (n_2, e_3), (n_3, e_3), (n_5, e_3)$
002	00:05	$(n_1, e_1), (n_2, e_2), (n_2, e_3)$
003	00:10	$(n_1, e_1), (n_2, e_3), (n_3, e_2), (n_4, e_1)$
004	00:15	$(n_2, e_3), (n_4, e_1), (n_5, e_3)$
005	00:20	$(n_1, e_2), (n_3, e_1), (n_4, e_1), (n_4, e_2), (n_5, e_2)$
006	00:25	$(n_1, e_2), (n_3, e_1), (n_4, e_1), (n_4, e_2), (n_5, e_1)$
007	00:30	$(n_2, e_1), (n_5, e_1)$

Table 4
Candidate 1-itemset and frequent 1-itemset ($min\_sup = 3$).

Candidate 1-itemset	Support	Lifetime	Frequent 1-itemset
(n_1, e_1)	3	[00:00, 00:10]	√
(n_1, e_2)	2	[00:20, 00:25]	
(n_2, e_3)	4	[00:00, 00:15]	√
(n_2, e_2)	2	[00:00, 00:05]	
(n_2, e_1)	1	[00:30, 00:30]	
(n_3, e_3)	1	[00:00, 00:00]	
(n_3, e_2)	1	[00:10, 00:10]	
(n_3, e_1)	2	[00:20, 00:25]	
(n_4, e_1)	4	[00:10, 00:25]	√
(n_4, e_2)	2	[00:20, 00:25]	
(n_5, e_3)	2	[00:00, 00:15]	
(n_5, e_2)	1	[00:20, 00:20]	
(n_5, e_1)	2	[00:25, 00:30]	

Table 5
Candidate 2-itemset and frequent 2-itemset ($min\_sup = 3$).

Candidate 2-itemset	Support	Lifetime	Frequent 2-itemset
$\{(n_1, e_1), (n_2, e_3)\}$	3	[00:00, 00:10]	√
$\{(n_1, e_1), (n_4, e_1)\}$	1	[00:10, 00:10]	
$\{(n_2, e_3), (n_4, e_1)\}$	2	[00:10, 00:15]	

If event e_1 is detected on node n_1, then there is a 100% chance of detecting event e_3 on node n_2 within 10 min.

Recall that given a database of transactions, together with a minimum support *min_sup* and a minimum confidence *min_conf*, the problem of data mining is to generate all the association rules with their confidence greater than or equal to the given threshold. It is widely recognized that the association rule mining process can be decomposed into two phases:

1. Find frequent patterns whose support counts are greater than or equal to *min_sup*.
2. Generate association rules whose confidence are greater than or equal to *min_conf*.

As the first phase performs in an iterative way to generate frequent patterns, this phase dominates the overall cost of mining association rules. Therefore, the key to success is how to efficiently reduce the cost brought by the first sub-problem.

In the next section, we will propose a decentralized approach to significantly reduce the cost and speed up the mining process.

Algorithm 1: Local Event Detection

```
   Input: Node $n$, total time $T$, time interval $\lambda$
   Output: $R$: A set of records containing detected events
 1 $R \leftarrow \varnothing$, $start\_time \leftarrow get\_current\_time()$ ;
 2 $t \leftarrow get\_current\_time()$ ;
 3 while $(t + \lambda) \leq (start\_time + T)$ do
 4     foreach Attribute $a$ do
 5         $value \leftarrow get\_attribute\_value(a, t)$;
 6         if value $> \xi^a_{up}$ then $e \leftarrow (|a| > \xi^a_{up})$ ;
 7         else if value $< \xi^a_{low}$ then $e \leftarrow (|a| < \xi^a_{low})$ ;
 8         else continue ;
 9         $R \leftarrow R \cup \{(t, n, e)\}$ ;
10     end
11     $t \leftarrow get\_current\_time()$ ;
12 end
13 $output(R)$ ;
```

4. The algorithms

In this section, we first introduce the local event detection algorithm. Then we propose *MapReduce-Apriori*, a MapReduce based association rule mining algorithm, on the detected events to efficiently filter certain events before being passed and then detect their correlations in parallel.

4.1. Local event detection

At the end of each time slot, each agent in its WN checks if any threshold of attribute is exceeded. If there is, the agent will store a record $r = (t, (n, e))$ into its local storage, indicating that an event e is detected on node n at time slot t. If no attribute exceeds its threshold, then nothing is recorded. Algorithm 1 shows a formal description of local event detection.

4.2. MapReduce-Apriori

MapReduce [13] is a programming model and an associated implementation for processing massive data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. As component failures become norms instead of exceptions, MapReduce offers ease of programming and fault tolerance.

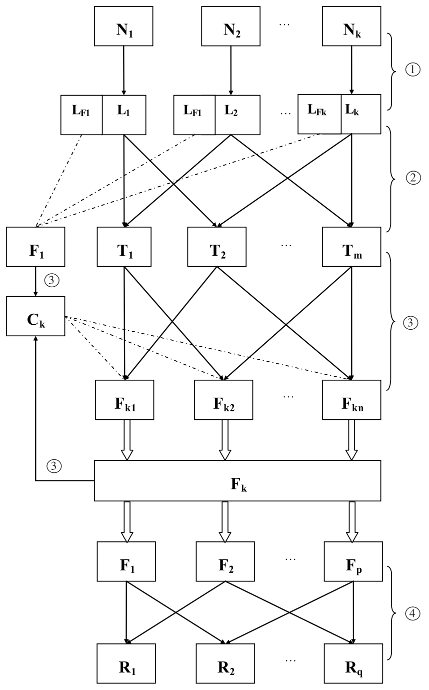


Fig. 1. Overall framework of MapReduce-Apriori.

The *MapReduce-Apriori* approach consists of four phases: local event filtering, merging events into transaction, global frequent set mining, and global monitoring association rule generation.

Fig. 1 depicts the four phases of *MapReduce-Apriori*:

1. *Local event filtering*: Filter items whose temporal support is less than *min_sup* as well as the records that contain these items.
2. *Merge events into transaction*: A MapReduce approach to combine events occurs within same time slot into a transaction.
3. *Global frequent itemset mining*: A multi-pass MapReduce approach to discover frequent itemsets.
4. *Global association rule generation*: A MapReduce approach to generate event association rules.

MapReduce-Apriori starts with several input parameters, including a minimal support *min_sup* and a minimal confidence *min_conf*. At this time, each node has stored the detected events in its local storage and nothing is currently shared in GDFS.

4.2.1. Local filtering

In this phase, we aim at reducing the communication cost when uploading records into GDFS. The idea behind is to filter out the items whose temporal support are less than *min_sup*. We do this because if an item is not locally frequent, it cannot be globally frequent. Once the monitoring phase is over, a scan on each WN is performed, which traverses all the records in the local storage to calculate the lifetime and temporal support for each item. If the temporal support of a certain time is greater than or equal to *min_sup*, all the records that contain the item will be retained. Once the filtering phase is over, all the retained records are uploaded to GDFS. Note that the records may be stored in different nodes. Algorithm 2 depicts the details for local filtering.

For example, let us reconsider the events presented in Table 2. Given $min\_sup = 3$, only item (n_2, e_3) will be retained in node n_2, as $lifetime((n_2, e_3)) = [00 : 00, 00 : 15]$ and $sup((n_2, e_3), [00 : 00, 00 : 15]) = 4 > 3$. Therefore, after local event filtering, only 4 records out of 7 will be retained, which significantly reduces the number of records being passed. Note that no record will be passed in node n_3 and n_5 because no item is supported by at least 3 records.

Algorithm 2: Local Filtering

```
Input: A item set $I$, A record set $R$, $min\_sup$
Output: $I' \subseteq I, \forall i \in I', sup(i, l(i)) \geq min\_sup, R' \subseteq R, \forall r \in R'$
        contains at least one item $i \in I'$
$I' \leftarrow \varnothing; R' \leftarrow \varnothing$ ;
foreach item $i \in I$ do
    $support\_count \leftarrow 0$ ;
    foreach record $r \in$ R do
        if $r$ contains $i$ then
            $support\_count \leftarrow support\_count + 1$ ;
        end
    end
    if $support\_count \geq min\_sup$ then
        foreach record $r \in R$ do
            $lifetime(i) \leftarrow [0, 0]$ ;
            if $r$ contains $i$ then
                $R' \leftarrow R' \cup \{r\}$ ;
                $lifetime(i) \leftarrow lifetime(i) \cap r.timeslot$ ;
            end
        end
        $I' \leftarrow I' \cup \{(i, support\_count, lifetime(i))\}$ ;
    end
end
$output(I')$ ; $output(R')$ ;
```

4.2.2. Merging records into transaction

The events in GDFS need to be grouped by time slot before discovering their correlations. This process can be conducted in parallel. Here a MapReduce pass is employed to merge records into transactions. Each mapper takes in an input pair in the form of (key $=$ *recordID*, value $=$ *record*), where *record* $= (t, (n, e))$ is a record that generated previously. It splits *record* into a time slot t and an item (n, e), and outputs a key-value pair (key $= t$, value $= (n, e)$). After all mapper instances have completed, for each distinct key t, the MapReduce infrastructure collects its corresponding values as *items*, and feeds reducers by key-value pair (key $= t$, value $=$ *items*), where *items* is a set of items with the same time slot. The reducer receives the key-value pair, merges all items with the same time slot into a transaction T, and associates its *TID* with t. Finally, it outputs the key-value pair (key $=$ *TID*, value $= T$). Algorithm 3 presents the pseudo code of this step.

Algorithm 3: Merging records into transaction

```
Input: A set of records in GDFS
Output: A set of transactions in GDFS
Procedure Mapper(key=recordID, value=record): ;
begin
    $ts \leftarrow record.t$ ;
    $item \leftarrow record.item$ ;
    output(key=ts, value=item)
end
Procedure Reducer(key=ts, value=items): ;
begin
    Transaction $T.TID \leftarrow ts$ ;
    foreach item $i \in items$ do
        $T \leftarrow T \cup \{i\}$ ;
    end
    output(key=TID, value=T)
end
```

4.2.3. Global frequent itemset mining

As transactions are distributed in GDFS, the MapReduce model is well fitted for our scenario to parallel the global frequent itemset mining phase. Here several iterative MapReduce passes are introduced. During the k-th MapReduce pass, the frequent k-itemsets are generated. It loops until no larger frequent itemset is found.

Recall that the frequent itemset mining has two sub-steps:

1. Generate candidate k-itemset C_k according to the frequent $(k-1)$-itemset F_{k-1}.
2. Generate frequent k-itemset F_k from C_k by eliminating itemsets whose support is less than *min_sup*.

The first sub-step, known as *Candidate Generation*, takes two steps to produce the result. First, in the join step, $\forall p, q \in F_{k-1}, p \neq q$, it joins p with q to get C_k, where p and q has the same $k-2$ items. Second, in the prune step, it deletes all itemset $c \in C_k$ such that some $(k-1)$-itemset of c is not in F_{k-1} [33]. In the join step, we identify a key property in our scenario to efficiently speed up the C_k generation, which is based on the following property:

Lemma 4.1. *$\forall$ frequent k-itemset i_k and frequent j-itemset i_j, given minimum support min_sup and time interval $\lambda \neq 0$, let $lifetime(i_k \cup i_j) = [t_1, t_2]$, if $\frac{|t_2-t_1|}{\lambda} < min\_sup$, then $i_k \cup i_j$ is not a frequent item set.*

Proof. Assume that $i_k \cup i_j$ is a frequent item set, with $sup(i_k \cup i_j, l(i_k \cup i_j)) = sup' \geq min\_sup$, where $l(i_k \cup i_j) = [t_1, t_2]$. Since $\frac{|l(i_k \cup i_j)|}{\lambda} \geq sup'$, we can conclude $\frac{|l(i_k \cup i_j)|}{\lambda} \geq sup' \geq min\_sup$, which contradicts the given condition $\frac{|t_2-t_1|}{\lambda} < min\_sup$. $\square$

For example, consider the data in Table 4, given $min\_sup = 3$, when generating candidate 2-itemset from frequent 1-itemset, itemset $i_1 = \{(n_1, e_1), (n_4, e_1)\}$ and $i_2 = \{(n_2, e_3), (n_4, e_1)\}$ can be pruned from candidate 2-itemsets, as $\frac{|lifetime(i_1)|}{\lambda} = \frac{|00:10-00:10|}{5} = 0 < 3$ and $\frac{|lifetime(i_2)|}{\lambda} = \frac{|00:15-00:10|}{5} = 1 < 3$. Algorithm 4 shows the extended candidate generation algorithm.

Algorithm 4: Extended Candidate Generation

```
Input: $F_{k-1}$: Frequent (k-1)-itemsets
Output: $C_k$: Candidate k-itemsets
$C_k \leftarrow \varnothing$ ;
foreach $p, q \in F_{k-1}$ do
    if $\frac{|lifetime(p \cup q)|}{\lambda} \geq min\_sup$ then
        $C_k \leftarrow C_k \cup$ join$(p, q)$ ;
    end
end
foreach $c \in C_k$ do
    foreach (k-1)-subsets $s \in c$ do
        if $s \notin F_{k-1}$ then
            delete $c$ from $C_k$ ;
        end
    end
end
$output(C_k)$ ;
```

In the second sub-step, multiple MapReduce passes are introduced, each would generate all frequent k-itemsets from candidate k-itemsets by counting the support for each candidate itemset C_k. Each mapper loads all C_k generated in the first sub-step before it receives a key-value pair (key $=$ *TID*, value $= T$). Then, for each candidate k-itemset $C_k^i \in C_k$, the map function decides whether T contains C_k^i, if so, it outputs a key-value pair (key $= C_k^i$, value $= 1$). The reducers aggregate the values with the same key, re-calculate $lifetime(C_k^i)$, and judge if the total support count is no less than *min_sup*. It outputs a key-value pair (key $= C_k^i$, value $= sup(i, lifetime(i))$), where $C_k^i \in F_k$ is a frequent k-itemset. Algorithm 5 gives the pseudo code for the frequent itemset mining.

4.2.4. Global association rule generation

To generate association rules from frequent itemsets, it is necessary to find all the subsets for every frequent itemset. Given a frequent itemset D, we must find, for each proper subset $D_1 \subsetneq D$, the event association rule $(D_1 \Rightarrow (D - D_1), [t_1, t_2], conf)$, such that $conf \geq min\_conf$. A problem here proposed by Ale and Rossi [3] is that given a frequent itemset i, an exponential re-calculation over all its subset is needed in order to get the temporal support for each subset. Theoretically, we have

$$conf(D_1 \Rightarrow (D - D_1), lifetime(D)) = \frac{sup(D, lifetime(D))}{sup(D_1, lifetime(D))}. \quad (4)$$

However, in the frequent item set mining phase, we only calculate $sup(D_1, lifetime(D_1))$ rather than $sup(D_1, lifetime(D))$. As $lifetime(D) \subseteq lifetime(D_1)$, if we replace $lifetime(D)$ with $lifetime(D_1)$, the actual value would be no larger than the theoretical value.

To avoid it, Ale and Rossi [3] assume that an itemset uniformly occurrs over its lifetime, thus the chance of an appearance in any subset will be the same. We argue that this assumption is not valid because in our situation, the events do not uniformly occur during its lifetime.

Algorithm 5: MapReduce Frequent Itemset Mining

```
   Input: $F_1$: Frequent 1-itemset, DB, min_sup
   Output: $\cup_k F_k$: all frequent itemsets
 1 Procedure Mapper(key=TID, value=T): ;
 2 begin
 3     $C_k \leftarrow load\_candidate\_itemset()$ ;
 4     foreach candidate itemset $C_k^i \in C_k$ do
 5         if T contains $C_k^i$ then
 6             $l_T \leftarrow [T.TID, T.TID]$ ;
 7             $sup(C_k^i, l_T) \leftarrow 1$;
 8             output(key=$C_k^i$, value=$sup(C_k^i, l_T)$) ;
 9         end
10     end
11 end
12 Procedure Reducer(key=$C_k^i$, value=supports): ;
13 begin
14     $sup(C_k^i, l(C_k^i)) \leftarrow 0$ ;
15     foreach $sup(C_k^i, l_T)$ in supports do
16         $sup(C_k^i, l(C_k^i)) \leftarrow sup(C_k^i, l(C_k^i)) + sup(C_k^i, l_T)$ ;
17     end
18     if $sup(C_k^i, l(C_k^i)) \geq min\_sup$ then
19         output(key=$C_k^i$, value=$sup(C_k^i, l(C_k^i))$) ;
20     end
21 end
22 Procedure FrequentItemsetMining(): ;
23 begin
24     for $k \leftarrow 1; F_k \neq \varnothing; k \leftarrow k + 1$ do
25         $F_k \leftarrow \varnothing$ ;
26         $C_k \leftarrow ExtendedCandidateGeneration(F_{k-1})$ ;
27         $F_k \leftarrow F_k \cup$ MapReduce($C_k$, Ts) ;
28     end
29     output($\cup_k F_k$) ;
30 end
```

To overcome this problem, we propose an enhanced lifetime format for an itemset, which records all the time slots that the itemset occurs instead of recording only the minimal and maximal time slot. Based on the enhanced format, we can directly count $sup(D_1, lifetime(D))$ without rescanning the database. For example, the lifetime of 2-itemset $\{(n_1, e_1), (n_2, e_3)\}$ in Table 5 can be recorded as [00:00, 00:05, 00:10] rather than [00:00, 00:10].

To parallelize this phase into a MapReduce pass, each mapper reads from a frequent itemset F as well as its temporal support $sup(F, l(F))$, and outputs key-value pairs (key $= F$, value $= F_1$), where $F_1 \subsetneq F$. Each reducer calculates confidence $conf(F_1 \Rightarrow (F - F_1), l(F))$. Algorithm 6 describes the MapReduce-based association rule generation. Note that the *get_temporal_support* function will calculate $sup(F_1, l(F))$ based on the enhanced lifetime format. The association rules mined are ranked by their confidence, which makes it very straightforward to pick up the top-K strong rules.

Algorithm 6: MapReduce Association Rule Generation

```
   Input: $\bigcup_{i=1}^{k} F_k$: frequent (1~k)-itemsets, min_conf
   Output: R: A set of association rules, whose confidence is no
           less than min_conf.
 1 Procedure Mapper(key=F, value=$sup(F, l(F))$): ;
 2 foreach subset $F_i \subsetneq F$ do
 3     $sup(F_i, l(F)) \leftarrow get\_temporal\_support(F_i, l(F))$ ;
 4     $V_{F_i} \leftarrow (F_i, sup(F_i, l(F)))$ ;
 5     output(key=$(F, sup(F, l(F)))$, value=$V_{F_i}$) ;
 6 end
 7 Procedure Reducer(key=$(F, sup(F, l(F)))$, value=values): ;
 8 foreach $V_{F_i} = (F_i, sup(F_i, l(F))) \in values$ do
 9     $conf \leftarrow sup(F, l(F)) / sup(F_i, l(F))$ ;
10     if $conf \geq min\_conf$ then
11         $rule \leftarrow (F_i \Rightarrow (F - F_i), l(F), conf)$ ;
12         output(key=conf, value=rule) ;
13     end
14 end
```

In summary, the benefits of our proposed algorithm are as follows: (1) There is no need to centralize all data into one database. Instead, all the events are collected into multiple nodes. (2) The time of mining is dramatically reduced, especially in some time-critical areas. (3) The events are locally filtered, thus the number of events being passed is reduced. (4) MapReduce is the most suitable approach to deal with today's large scale distributed systems and large amount of data. (5) We design the Extended Candidate Generation algorithm to efficiently reduce the number of candidates being generated.

5. Experimental results

We implemented the proposed algorithms on Apache Hadoop,[1] an open source implementation of Google's MapReduce framework. Apache Hadoop is a Java software framework that supports data-intensive distributed applications under a free license. Hadoop enables applications to work with thousands of nodes and petabytes of data. Besides, the *Hadoop Distributed File System* (HDFS) is a suitable implementation of GDFS for our scenario.

We first built a synthetic distributed system and mimicked the monitoring environment to generate synthetic events. Then we applied MapReduce-Apriori on the synthetic events to discover event

[1] http://hadoop.apache.org

Table 6
Parameters for synthetic event generation.

Parameter	Description
T	The total number of monitoring time slots
N	The number of nodes
A	The number of attributes that need to monitor
$\xi_{\min}$	The minimal attribute upper threshold
$\xi_{\max}$	The maximum attribute upper threshold
min_sup_rate	The minimal temporal support rate for mining

Table 7
Synthetic datasets ($0.7 \le \xi \le 1.0$).

Dataset	# of nodes	# of time slots	# of attributes	Total items
N50A10T5k	50	5 000	10	500
N50A10T10k	50	10,000	10	500
N50A10T20k	50	20,000	10	500
N50A10T40k	50	40,000	10	500
N50A10T70k	50	70,000	10	500
N50A10T100k	50	100,000	10	500
N100A10T10k	100	10,000	10	1000
N150A10T10k	150	10,000	10	1500
N200A10T10k	200	10,000	10	2000
N50A15T10k	50	10,000	15	750
N50A20T10k	50	10,000	20	1000
N50A25T10k	50	10,000	25	1250

correlations. To demonstrate the effectiveness of MapReduce-Apriori, we also implemented the classic Apriori algorithm and applied it on a centralized database. We compared the execution time of mining event correlations for both algorithms.

5.1. Synthetic event generation

The simulator generates events with several input parameters, which are described in Table 6. Upon receiving such parameters, the simulator first generates $|A|$ attribute upper thresholds between $\xi_{\min}$ and $\xi_{\max}$ randomly (lower threshold set to 0 by default). At each time slot, the simulator will generate $|A|$ distinct attribute values on each node, and judge whether a value exceeds its corresponding upper threshold. Note that each dataset generation is independent, and the generation of attribute value satisfies a uniform distribution $U(0, 1)$ over the possible number of time slots. The synthetic event generation program is run on a desktop PC with Pentium Dual-Core CPU E5300 at 2.6 GHz and 2 GB memory.

Our simulation is done on the following scenarios:

- Given $N = 50, A = 10, \xi_{\min} = 0.7, \xi_{\max} = 1.0$, with the increase of T, collect detected events that are distributed in different nodes within T.
- Given $T = 10\,000, A = 10, \xi_{\min} = 0.7, \xi_{\max} = 1.0$, with the increase of N, collect detected events that are distributed in different nodes within T.
- Given $T = 10\,000, N = 50, \xi_{\min} = 0.7, \xi_{\max} = 1.0$, with the increase of A, collect detected events that are distributed in different nodes within T.

Table 7 shows the generated datasets. The first column lists the name of a dataset. The number of nodes in the second column indicates the scale of a synthetic distributed system. The third column lists the total number of time slots we need to monitor. The fourth column lists the number of attributes in a distributed system. The last column multiplies the number of nodes and the number of attributes to get the number of distinct items. For example, dataset N50A10T5k simulates a monitoring task on a 50-node synthetic distributed system, which checks 10 different performance attributes within 5000 time slots.

Once the event generation is over, the simulator immediately performs local event filtering on the generated events with given

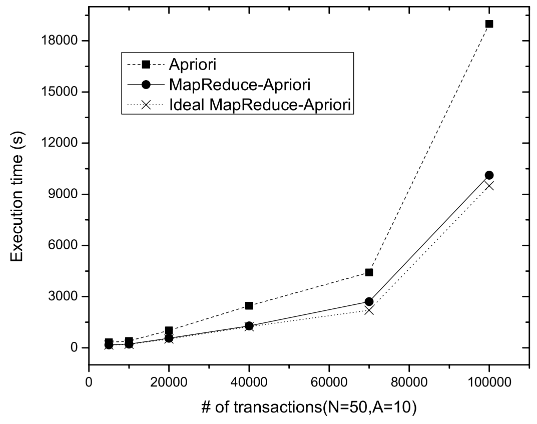


Fig. 2. Increasing monitoring time slots.

min_sup_rate. We manually choose *min_sup_rate* $= 0.07$. Table 8 describes the filtering result. For each dataset, the second column lists the total number of events detected. The third column lists the number of events filtered during the local event filtering phase. The fourth column lists the percentage of filtered events. The fifth column lists the number of distinct items filtered. The last column lists the percentage of filtered items. Clearly, our local filtering approach effectively prunes some infrequent events and items. The average percentage of filtered events is 8.03% and the average percentage of filtered items is 26.41%. In the best case, half of the total items have been filtered, bringing 16.32% decrease of the number of events to be passed.

5.2. Relative performance

In this section, we applied two algorithms on the events after filtering to mine their correlations. Apriori runs on 1 single PC with a Pentium Dual-Core CPU E5300 at 2.6 GHz and 2 GB of memory. MapReduce-Apriori runs on a 2-node Hadoop cluster consisting of two physically different PCs, each with a Pentium (R) Dual-Core CPU E5300 @ 2.6 GHz and 2 GB of memory. We measured the execution time of two algorithms under different datasets, including the frequent itemset mining time and the association rule generation time.

Figs. 2, 4, 6 and 7 are drawn in order to compare the performance of two algorithms in various scenarios. In all the figures, the dash line represents the performance of Apriori, while the solid line represents the performance of MapReduce-Apriori. Note that the ideal performance of MapReduce-Apriori (*Ideal MapReduce-Apriori*) is represented by the dot line. In our experiment, the ideal performance of MapReduce-Apriori would reduce the execution time of Apriori by half, gaining double speedup.

5.2.1. Varying monitoring time slots

We applied the two algorithms on 6 datasets, N50A10T5k, N50A10T10k, N50A10T20k, N50A10T40k, N50A10T70k, and N50A10T100k, to study their performance when varying monitoring time slots. With $N = 50$ and $A = 10$, Fig. 2 shows how the execution time of two algorithms reacts to the increasing number of monitoring time slots. Note that the increase of monitoring time slots would result in the increase of total number of transactions, as each TID is associated with a unique time slot number. Hence, we directly replace monitoring time with the number of transactions in the figure. When the number of transactions is relatively small, i.e., $T = 5$ k, 10 k, and 20 k, the difference between two algorithms is not very obvious. As the number of transactions grows, MapReduce-Apriori begins to show its advantage. In addition, the

Table 8
Local filtering result (*min_sup_rate* = 0.07).

Dataset	# of detected events	# of filtered events	% of filtered events (%)	# of filtered items	% of filtered items (%)
N50A10T5k	330,003	38,608	11.70	128	25.6
N50A10T10k	561,391	91,608	16.32	250	50.0
N50A10T20k	1,367,163	102,147	7.47	106	21.2
N50A10T40k	3,045,858	182,591	5.99	100	20.0
N50A10T70k	5,715,207	249,107	4.36	100	20.0
N50A10T100k	8,315,240	276,305	3.32	100	20.0
N100A10T10k	1,262,790	105,981	8.39	400	40.0
N150A10T10k	2,165,142	301,121	13.91	600	40.0
N200A10T10k	2,723,228	200,049	7.35	400	20.0
N50A15T10k	1,182,979	20,263	1.71	100	13.3
N50A20T10k	1,355,369	134,527	9.93	281	28.1
N50A25T10k	2,046,988	121,130	5.92	234	18.7

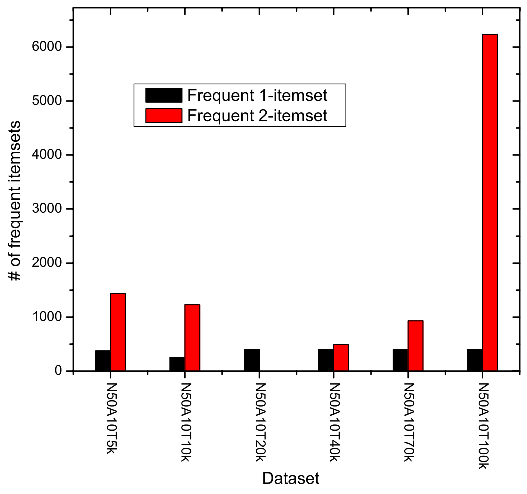


Fig. 3. # of frequent itemsets mined by increasing monitoring time slots.

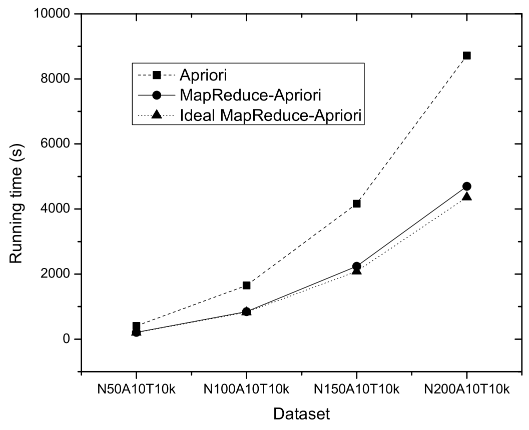


Fig. 4. Increasing number of nodes in a synthetic system.

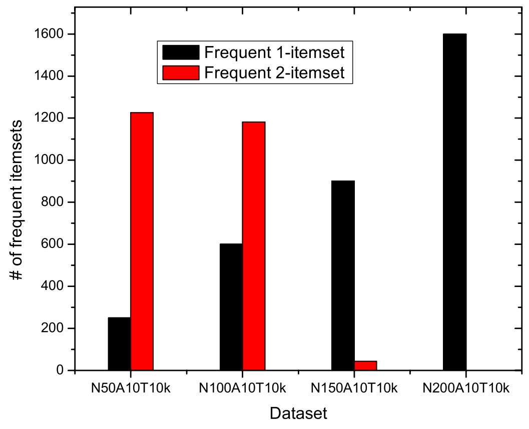


Fig. 5. # of frequent itemsets mined by increasing # of nodes.

performance of MapReduce-Apriori is very close to that of Ideal MapReduce-Apriori.

Next, we studied the frequent itemsets mined and found that for each dataset, only frequent 1-itemsets and frequent 2-itemsets are mined. As the *min_sup_rate* is chosen to be 0.07, there are no ternary event correlations that are supported so many times. Hence, no frequent 3-itemset is discovered. Fig. 3 depicts the number of frequent itemsets mined in different datasets. The black and red column represents the frequent 1-itemsets and frequent 2-itemsets, respectively. From the figure we can see no frequent 2-itemset is mined in dataset N50A10T20k, while over 6000 frequent 2-itemsets are mined in dataset N50A10T100k. This may possibly explain why the execution time of N50A10T20k grows so fast in Fig. 2.

5.2.2. Varying the number of nodes in a synthetic system

We applied the two algorithms on 4 datasets, N50A10T10k, N100A10T10k, N150A10T10k, and N200A10T10k, to study their performance when varying the number of nodes in a synthetic distributed system. The increase of the number of nodes would result in the increase of the number of different items to mine. Note that dataset N50A10T10k has already been experimented with previously. Therefore we adopt the results directly. Fig. 4 compares the execution time of two algorithms applied to different datasets. From the figure we can see that MapReduce-Apriori outperforms Apriori and achieves nearly ideal speedup. The number of frequent itemsets mined is shown in Fig. 5. In dataset N50A10T10k and N100A10T10k, the number of frequent 2-itemsets is larger than that of frequent 1-itemsets. However, in dataset N150A10T10k, the number of frequent 2-itemsets is much smaller. Unfortunately, there is no frequent 2-itemset mined in dataset N200A10T10k while 1600 frequent 1-itemsets are mined. In this case, the *min_sup_rate* might be too high to mine binary event correlation.

5.2.3. Varying attributes in the system

We applied the two algorithms on 4 datasets, N50A10T10k, N50A15T10k, N50A20T10k and N50A25T10k, to study the relative performance when varying the number of attributes in synthetic distributed system. The increase of the number of attributes would result in the increase of the number of different items to mine. Note that dataset N50A10T10k has been experimented previously. Fig. 6 compares the execution time of two algorithms. As Fig. 6 illustrates, the running time of two algorithms both grow with the increase of attribute numbers. As expected, MapReduce-Apriori outperforms Apriori, reducing the execution time nearly by half.

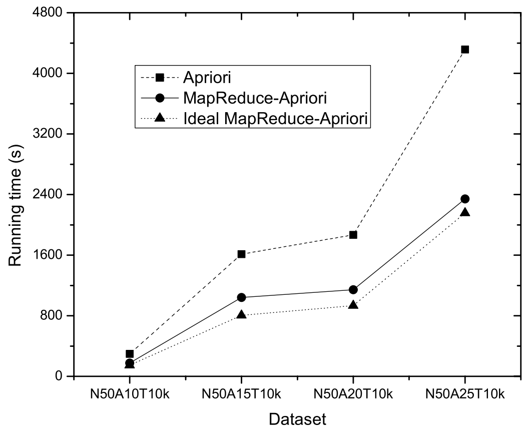


Fig. 6. Increasing number of attributes in synthetic system.

Table 9
N50A10T10k-new local filtering result.

min_sup_rate	# of FE	% of FE (%)	# of FI	% of FI (%)
0.06	6,497	0.76	11	2.2
0.07	30,915	3.62	50	10.0
0.08	48,879	5.72	73	14.6
0.09	71,122	8.33	100	20.0

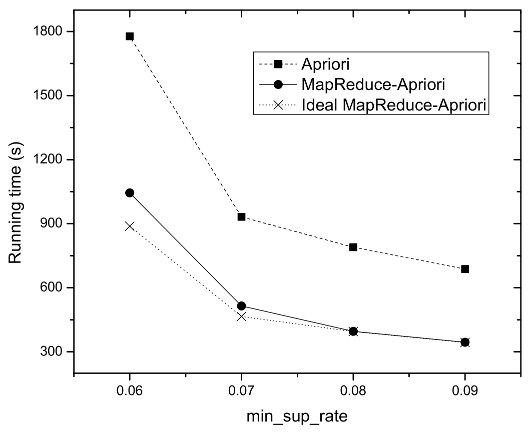


Fig. 7. Increasing *min_sup_rate*.

5.2.4. Varying min_sup_rate in a specific dataset

To evaluate the performance of two algorithms in different *min_sup_rate*, we have generated a brand new dataset, *N50A10T10k-new*, and conducted the following experiments.

First we increased *min_sup_rate* from 0.06 with interval 0.01 and measured the number of events and items our local filtering algorithm filtered. Next, we applied both MapReduce-Apriori and Apriori on the filtered events and recorded execution time of them.

Table 9 describes the filtering result. For each distinct *min_sup_rate*, the second column lists the number of events filtered. The third column lists the percentage of events filtered. The fourth column lists the number of items filtered. The last column lists the percentage of items filtered. Note that the total number of events detected is 854,157, and the total number of items to mine is 500. From Table 9 we can see that filtering efficiency grows with the increase of *min_sup_rate*.

Fig. 7 compares the execution time of two algorithms in different *min_sup_rate*. As we can see, the execution time decreases with the increase of *min_sup_rate*. Besides, MapReduce-Apriori completely outperforms Apriori and gradually converges to the ideal case.

6. Related work

Distributed system monitoring.

A large body of research addresses the design of distributed monitoring systems, and demonstrates that hierarchical aggregation is an effective approach to achieve good scalability [28,41,14,10,20,29,21,1,38]. Abadi et al. [1] present a system called REED for robust and efficient event detection in sensor networks. Al-Shaer et al. [4] propose a monitoring architecture that employs a hierarchical event filtering approach to distribute monitoring load and limit event propagation. Cormode et al. [10] present a distributed-tracking scheme for maintaining accurate quantile estimates with provable approximation guarantees. Deligiannakis et al. [14] extend prior work on in-network data aggregation to support the approximate evaluation of queries to reduce the number of exchanged messages among the nodes.

Khanna et al. [21] propose an autonomous and hierarchical monitor system, which is used to provide fast detection to distributed system failures. Madden et al. present the *Tiny AGgregation* (TAG) service for aggregation in low-power, distributed wireless environments, distributing and executing simple queries efficiently in networks of low-power wireless sensors [28]. Yalagandula and Dahlin proposed a *Scalable Distributed Information Management System* (SDIMS) [41]. The SDIMS serves as a basic building block for large-scale distributed applications, providing detailed nearby information and summary views of global information. In [14], the in-network data aggregation has been extended by providing the support of the approximate evaluation of queries to reduce the number of exchanged messages among the nodes. Algorithmic solutions were provided for the problem of continuously tracking complex holistic aggregates in a distributed streaming network [10].

REMO [29] is a resource aware application state monitoring system that considers multi-task optimization and node level resource constraints. Silberstein et al. [39] proposed a frame monitoring problem in a wireless sensor network as one of monitoring node and edge constraints to detect network failure. Sharfman et al. proposed a geometric approach by splitting the global monitoring task into a set of constraints applied in local nodes [38]. Cuzzocrea [12] used CAMS (Cube-based Acquisition model for Multidimensional Streams) to tame the multidimensionality of real-life data streams efficiently through combining OLAP (OnLine Analytical Processing) flattening and OLAP aggregation processes.

The nature of our algorithm is a MapReduce based count distribution parallel Apriori, so it is quite different between our approach and other Apriori algorithms. In the monitoring phase, the main difference is that our approach stores events into multiple nodes rather than a single node. Also, our approach uses a local filter to reduce the event size being passed.

Temporal data mining. With the rapid development of the distributed monitoring system, mining temporal event correlations in a distributed environment became realistic. Roddick and Spiliopoulou [35] survey the issues and solutions in temporal data mining. Ale and Rossi [3] expand the notion of association rules incorporating time to the frequent itemset discovered. Chen and Petrounias [7] identify the valid period and periodicity of patterns with more specifical association rules. Li et al. [22] extend the Apriori algorithm and develop two optimization techniques to take advantage of the special properties of the calendar-based patterns. Römer [36] proposes an in-network data mining technique to discover frequent patterns of events with certain spatial and temporal properties.

Distributed and parallel data mining. There is a large amount of related work addressing the distributed and parallel data mining problems. The *Parallel Data Mining* (PDM) for association rules implemented the serial *Direct Hashing and Pruning* (DHP) in a parallel manner [31]. It is based on the use of a parallel hash table. The communicating of the counts of each location in the hash table makes it inefficient. There are a number of other distributed algorithm such as Parallel Eclat [42], Intelligent Data Distribution [19], and Hybrid Distribution [19], all of which are based on the task distribution paradigm, allowing each processor to handle a subset of the candidate itemset. Park and Kargupta [32] present a brief overview of the distributed data mining algorithms, systems, and applications.

Agrawal and Shafer [2] consider the problem of mining association rules on a shared-nothing multiprocessor, and present several approaches to parallel the association rule mining algorithm. The Data Distribution algorithm in [2] provided a solution for the memory problem of the count distribution algorithm, splitting the generation process among the processors in a round robin manner. However, our approach has made some improvements. First, we design the Extended Candidate Generation Algorithm (Algorithm 4) to dramatically reduce the number of candidate in each iteration. Second, we use the MapReduce framework to implement the algorithm, which is different from [2]. Third, in [2], the count distribution assumes data are centralized into one database. In our approach, the data are events which are collected into multiple nodes and there is no centralized database.

Li et al. [23] propose a parallel FP-growth algorithm to mine frequent items for query recommendation. Chang et al. [6] summarize several parallel algorithms for mining large-scale rich-media data. Cheung et al. [8] propose a distributed association rule mining algorithm to generate a small number of candidate sets and reduce the number of messages to be passed at mining association rules. The strength of our approach is that we require less synchronization than [8], which is very critical in a large distributed system. the weakness is that our approach will generate more candidates in each iteration than [8].

Boukerche and Samarah [5] mine sensor behavioral patterns and sensor association rules to improve the quality of service in wireless ad-hoc sensor networks. Loo et al. [27] propose lossy counting based online mining algorithm for discovering inter-stream associations from large sensor networks. Ren and Guo [34] present a distributed frequent items mining algorithm to mine frequent items from sensory data in wireless sensor networks. Congiusta et al. [9] discuss how grid computing can be used to support distributed data mining.

MapReduce framework. The low cost, data intensive MapReduce framework [13] has recently became a popular tool for the distributed data mining. Cryans et al. adapted the Apriori algorithm to MapRedcue in the search for relation between entities [11]. Yang et al. presented an improved Apriori algorithm based on the MapReduce mode to handle massive datasets with a large number of nodes on Hadoop platform [26]. Hammoud proposed a MapReduce based association rule miner for extracting strong rules from large datasets [18], which is used to develop a large scale classifier.

7. Conclusion and future work

In this paper, we proposed an extended model of mining event association rules in a distributed system, which considered the temporality of detected events. In addition, we proposed a MapReduce based algorithm to efficiently filter irrelative events and discover their temporal correlations. Our experimental results show that our algorithm outperforms the Apriori-based centralized mining approach and achieves nearly ideal speedup.

Our work will be extended in two dimensions. First, in order to improve the throughput of the mining approach, we are considering extending our framework to enable pipelining of a frequent itemset mining phase and association rule generation phase. Second, we plan to deploy our framework in a real distributed environment to further validate the effectiveness of the proposed algorithms.

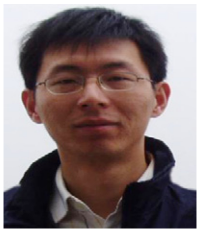
Acknowledgments

This work was supported in part by the National Basic Research Program of China 2011CB302601; NSF CNS-1249223 and NSFC 61071061; NSFC 61170077, SZ-HK Innovation Circle Proj. ZYB200907060012A, NSF GD:10351806001000000, S & T Proj. of SZ JC200903120046A; the NSF CNS-0915762 (CSR), CCF-08452578 (CAREER), CNS-0917137 (CSR).

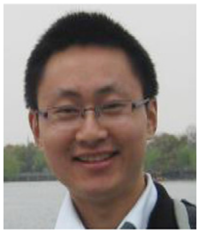
References

[1] D.J. Abadi, S. Madden, W. Lindner, Reed: robust, efficient filtering and event detection in sensor networks, in: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 769–780.
[2] R. Agrawal, J.C. Shafer, Parallel mining of association rules, The IEEE Transactions on Knowledge and Data Engineering 8 (1996) 962–969.
[3] J.M. Ale, G.H. Rossi, An approach to discovering temporal association rules, in: Proceedings of the 15th ACM Symposium on Applied Computing, pp. 294–300.
[4] E. Al-Shaer, H. Abdel-Wahab, K. Maly, Hifi: a new monitoring architecture for distributed systems management, in: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, pp. 171–178.
[5] A. Boukerche, S. Samarah, A novel algorithm for mining association rules in wireless ad hoc sensor networks, IEEE Transactions on Parallel and Distributed Systems 19 (2008) 865–877.
[6] E.Y. Chang, H. Bai, K. Zhu, Parallel algorithms for mining large-scale rich-media data, in: Proceedings of the 17th ACM International Conference on Multimedia, pp. 917–918.
[7] X. Chen, I. Petrounias, Mining temporal features in association rules, in: Proceedings of the 3rd European Conference on Principles of Data Mining and Knowledge Discovery, pp. 295–300.
[8] D.W. Cheung, J. Han, V.T. Ng, A.W. Fu, Y. Fu, A fast distributed algorithm for mining association rules, in: Proceedings of the 4th International Conference on Parallel and Distributed Information Systems.
[9] A. Congiusta, D. Talia, P. Trunfio, Service-oriented middleware for distributed data mining on the grid, Journal of Parallel and Distributed Computing 68 (2008) 3–15.
[10] G. Cormode, M. Garofalakis, S. Muthukrishnan, R. Rastogi, Holistic aggregates in a networked world: distributed tracking of approximate quantiles, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 25–36.
[11] J.D. Cryans, S. Ratte, R. Champagne, Adaptation of apriori to mapreduce to build a warehouse of relations between named entities across the web, in: International Conference on Advances in Databases, Knowledge, and Data Applications, pp. 185–189.
[12] A. Cuzzocrea, Cams: olaping multidimensional data streams efficiently, in: Proc. of DaWaK, pp. 48–62.
[13] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 137–150.
[14] A. Deligiannakis, Y. Kotidis, N. Roussopoulos, Hierarchical in-network data aggregation with quality guarantees, in: Proceedings of the 9th International Conference on Extending DataBase Technology.
[15] M. Fisichella, A. Stewart, A. Cuzzocrea, K. Denecke, Detecting health events on the social web to enable epidemic intelligence, in: Proc. of SPIRE, pp. 87–103.
[16] S. Fu, C. Xu, Exploring event correlation for failure prediction in coalitions of clusters, in: Proceedings of the 19th ACM/IEEE Conference on Supercomputing, pp. 41:1–41:12.
[17] S. Fu, C. Xu, Quantifying event correlations for proactive failure management in networked computing systems, Journal of Parallel and Distributed Computing 70 (2010) 1100–1109.
[18] S. Hammoud, MapReduce network enabled algorithms for classification based on association rules, Ph.D. Thesis, Brunel University, 2011.
[19] E.H. Han, G. Karypis, V. Kumar, Scalable parallel data mining for association rules, in: ACM SIGMOD International Conference on Management of Data, pp. 277–288.
[20] R. Huebsch, B. Chun, J.M. Hellerstein, B.T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, A.R. Yumerefendi, The architecture of pier: an Internet-scale query processor, in: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research, pp. 28–43.
[21] G. Khanna, P. Varadharajan, S. Bagchi, Automated online monitoring of distributed applications through external monitors, IEEE Transactions on Dependable and Secure Computing 3 (2006) 115–129.
[22] Y. Li, P. Ning, X.S. Wang, S. Jajodia, Discovering calendar-based temporal association rules, Data & Knowledge Engineering 44 (2003) 193–218.

[23] H. Li, Y. Wang, D. Zhang, M. Zhang, E.Y. Chang, PFP: parallel FP-growth for query recommendation, in: Proceedings of the 2008 ACM Conference on Recommender Systems.
[24] J. Li, M. Qiu, Z. Ming, G. Quan, X. Qin, Z. Gu, Online optimization for scheduling preemptable tasks on IaaS cloud systems, Journal of Parallel and Distributed Computing (JPDC) 72 (5) (2012) 666–677. (Most Download Paper of JPDC in 2012).
[25] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, R. Sahoo, Bluegene/l failure analysis and prediction models, in: Proceedings of the 2006 International Conference on Dependable Systems and Networks, pp. 425–434.
[26] X.Y. Yang, Z. Liu, Y. Fu, MapReduce as a programming model for association rules algorithm on Hadoop, in: International Conference on Information Sciences and Interaction Sciences, 2002, pp. 99–102.
[27] K.K. Loo, I. Tong, B. Kao, D. Cheung, Online algorithms for mining inter-stream associations from large sensor networks, in: Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 143–149.
[28] S. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, Tag: a tiny aggregation service for ad-hoc sensor networks, ACM SIGOPS Operating Systems Review 36 (2002) 131–146.
[29] S. Meng, S.R. Kashyap, C. Venkatramani, L. Liu, REMO: Resource-aware application state monitoring for large-scale distributed systems, in: Proceedings of the 29th IEEE International Conference on Distributed Computing Systems, pp. 248–255.
[30] J.W. Mickens, B.D. Noble, Exploiting availability prediction in distributed systems, in: Proceedings of the 3rd Conference on Networked Systems Design & Implementation—Volume 3, NSDI'06.
[31] J.S. Park, M. Chen, P. Yu, Efficient parallel data mining for association rules, in: International Conference on Information and Knowledge Management, pp. 31–36.
[32] B.H. Park, H. Kargupta, Distributed data mining: algorithms, systems, and applications, in: Data Mining Handbook, 2002, pp. 341–358.
[33] A. Rakesh, S. Ramakrishnan, Fast algorithms for mining association rules in large databases, in: Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487–499.
[34] M. Ren, L. Guo, Mining recent approximate frequent items in wireless sensor networks, in: Proceedings of the 6th International Conference on Fuzzy Systems and Knowledge Discovery, pp. 463–467.
[35] J.F. Roddick, M. Spiliopoulou, A survey of temporal knowledge discovery paradigms and methods, The IEEE Transactions on Knowledge and Data Engineering 14 (2002) 750–767.
[36] K. Römer, Discovery of frequent distributed event patterns in sensor networks, in: Proceedings of the 5th European Conference on Wireless Sensor Networks, pp. 106–124.
[37] B. Schroeder, G.A. Gibson, A large-scale study of failures in high-performance computing systems, in: Proceedings of the 2006 International Conference on Dependable Systems and Networks, pp. 249–258.
[38] I. Sharfman, A. Schuster, D. Keren, A geometric approach to monitoring threshold functions over distributed data streams, in: ACM SIGMOD International Conference on Management of Data, pp. 301–312.
[39] A. Silberstein, R. Braynard, J. Yang, Constraint chaining: on energy-efficient continuous monitoring in sensor networks, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 157–168.
[40] D. Tang, R.K. Iyer, Analysis and modeling of correlated failures in multicomputer systems, IEEE Transactions on Computers 41 (1992) 567–577.
[41] P. Yalagandula, M. Dahlin, A scalable distributed information management system, in: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer communications, pp. 379–390.
[42] M.J. Zaki, Scalable algorithms for association mining, The IEEE Transactions on Knowledge and Data Engineering 12 (2000) 372–390.
[43] W. Zhou, J. Zhan, D. Meng, D. Xu, Z. Zhang, Logmaster: mining event correlations in logs of large scale cluster systems, The Computing Research Repository abs/1003.0951 (March, 2010).
[44] X. Zhu, X. Qin, M. Qiu, QoS-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters, IEEE Transactions on Computers (TOC) 60 (6) (2011) 800–812.



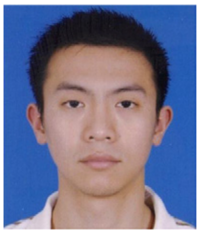
Gang Wu received the Ph.D. degree in Computer Science from the National University of Defense Technology in China in 2000. He is now an associated professor in the School of Software, Shanghai Jiao Tong University. Before entering Shanghai Jiao Tong University, he had worked at the National University of Defense Technology as a lecturer from 2000 to 2002 and as an associated professor from 2002 to 2005. He is the director of the Adaptive Distributed Computing lab of Shanghai Jiao Tong University. His research interests include distributed system, pervasive computing, cloud computing, etc.



Huxing Zhang is currently pursuing a Masters Degree in the School of Software at Shanghai Jiao Tong University in China. He received his B.S. degree in Software Engineering at Shanghai Jiao Tong University in 2009. His research interests include distributed, and cloud systems, particularly data mining in distributed systems.

Meikang Qiu received the B.E. and M.E. degrees from Shanghai Jiao Tong University, China. He received the M.S. and Ph.D. degrees in Computer Science from the University of Texas at Dallas in 2003 and 2007, respectively. He had worked at the Chinese Helicopter R&D Institute and IBM. Currently, he is an assistant professor of ECE at the University of Kentucky. He is an IEEE Senior member and has published 140 papers. He is the recipient of the ACM Transactions on Design Automation of Electronic Systems (TODAES) 2011 Best Paper Award. He also received four other best paper awards (IEEE EUC'09, IEEE/ACM GreenCom'10, IEEE CSE'10, IEEE ICESS'12) and one best paper nomination (IEEE EmbeddedCom'09). He also holds 2 patents and has published 3 books. His research is supported by NSF, ONR, and the Air Force. He has been awarded a Navy summer faculty in 2012 and an Air Force summer faculty in 2009. He has been on various chairs and TPC members for many international conferences. He served as the Program Chair of IEEE EM-Com'09. His research interests include embedded systems, computer security, and wireless sensor networks.

Zhong Ming is a professor at the College of Computer and Software Engineering of Shenzhen University. He is a member of a council and senior member of the Chinese Computer Federation. His major research interests are in software engineering and embedded systems. He led two projects of the National Natural Science Foundation, and two projects of the Natural Science Foundation of Guangdong province, China.



Jiayin Li received the B.E. and M.E. degrees from Huazhong University of Science and Technology (HUST), China, in 2006 and 2008, respectively. He obtained a Ph.D. degree from the Department of Electrical and Computer Engineering (ECE), University of Kentucky in 2012. His research interests include software/hardware co-design for embedded systems and high performance computing.

Xiao Qin received the B.S. and M.S. degrees in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1996 and 1999, respectively, and the Ph.D. degree in computer science from the University of Nebraska-Lincoln in 2004. He is currently an associate professor of Computer Science at Auburn University. Prior to joining Auburn University in 2007, he had been an assistant professor with the New Mexico Institute of Mining and Technology (New Mexico Tech) for three years. He won an NSF CAREER award in 2009. His research interests include parallel and distributed systems, real-time computing, storage systems, fault tolerance, and performance evaluation. His research is supported by the US National Science Foundation, Auburn University, and Intel Corporation. He is a senior member of the IEEE and the IEEE Computer Society.