An efficient parallel algorithm for mining weighted clickstream patterns

An efficient parallel algorithm for mining weighted clickstream patterns

Huy M. Huynh[a], Loan T.T. Nguyen[b,c], Bay Vo[d,*], Zuzana Komínková Oplatková[a], Philippe Fournier-Viger[e], Unil Yun[f]

[a]*Faculty of Applied Informatics, Tomas Bata University in Zlín, Nám. T.G. Masaryka 5555, Zlín 76001, Czech Republic*

[b]*School of Computer Science and Engineering, International University, Ho Chi Minh City 700000, Viet Nam*

[c]*Vietnam National University, Ho Chi Minh City 700000, Viet Nam*

[d]*Faculty of Information Technology, Ho Chi Minh City University of Technology (HUTECH), Ho Chi Minh City 700000, Vietnam*

[e]*School of Humanities and Social Sciences, Harbin Institute of Technology, Shenzhen 518055, China*

[f]*Department of Computer Engineering, Sejong University, Seoul 05006, Republic of Korea*

[*]*Corresponding author. E-mail addresses: huynh@utb.cz (H.M. Huynh), nttloan@hcmiu.edu.vn (L.T.T. Nguyen), vd.bay@hutech.edu.vn (B. Vo), oplatkova@utb.cz (Z.K. Oplatková), philfv@hit.edu.cn (P. Fournier-Viger), yunei@sejong.ac.kr (U. Yun).*

ABSTRACT

In the Internet age, analyzing the behavior of online users can help webstore owners understand customers' interests. Insights from such analysis can be used to improve both user experience and website design. A prominent task for online behavior analysis is clickstream mining, which consists of identifying customer browsing patterns that reveal how users interact with websites. Recently, this task was extended to consider weights to find more impactful patterns. However, most algorithms for mining weighted clickstream patterns are serial algorithms, which are sequentially executed from the start to the end on one running thread. In real life, data is often very large, and serial algorithms can have long runtimes as they do not fully take advantage of the parallelism capabilities of modern multi-core CPUs. To address this limitation, this paper presents two parallel algorithms named DPCompact-SPADE (Depth load balancing Parallel Compact-SPADE) and APCompact-SPADE (Adaptive Parallel Compact-SPADE) for weighted clickstream pattern mining. Experiments on various datasets show that the proposed parallel algorithm is efficient, and outperforms state-of-the-art serial algorithms in terms of runtime, memory consumption, and scalability.

Keywords: Frequent pattern mining, weighted clickstream patterns, parallelism

1. Introduction

In recent decades, the amount of data stored in databases has dramatically increased, providing opportunities to analyze it to find useful information. However, analyzing data by hand is time-consuming. Moreover, because data often has a complex structure and is large, it is difficult to identify interesting relationships between data elements. To address these issues, the field of Knowledge Discovery and Database has emerged, also known as data mining. A fundamental task in data mining

is pattern mining. The goal of pattern mining is to find frequently occurring patterns such as frequent sequences of purchases made by customers. Those patterns can then be analyzed to obtain insights into the habits of customers and using that knowledge, stores can be adapted to their customers' needs. There are many pattern mining tasks, but the most fundamental ones are frequent itemset mining (FIM), association rule mining (ARM), and sequential pattern mining (SPM). FIM and ARM were proposed by Agrawal et al. [**1**]. Though useful, FIM and ARM do not consider the order between items, and thus SPM [**3**] was proposed to address this issue. It consists of finding subsequences of purchased items that appear frequently in a set of sequences.

Clickstream pattern mining (CPM) is a specialized problem derived from SPM that has attracted the attention of many researchers recently because there is a need for analyzing user interactions on websites. It aims at finding patterns representing a series of events such as sequences of user clicks and accessed URLs. For instance, a person that is browsing an online webstore generates a clickstream in the form of a user log (user clickstream) containing a sequence of URLs. Other types of data such as user actions on computers (e.g., deleting files or opening folders) and DNA sequences can also be modeled as clickstreams.

Initially, studies of CPM considered that all items are of equal importance (i.e. non-weighted). But for several applications, items do not have the same importance. Therefore, it was proposed to associate weights with items to find more useful patterns. However, integrating weights can make the task more complex, since measures used to select patterns may not respect the anti-monotonicity property used to reduce the search space. Additionally, traditional CPM algorithms are often serial. They can only utilize the resources of a single thread or core. This makes their performance unable to scale with multiple core computer architecture, which consists of multiple cores and can execute several threads in parallel.

To address the above limitations of traditional CPM algorithms, this paper proposes an effective parallel method for mining weighted clickstream patterns on a single node (a computer with shared memory and multiple cores). The main contributions are as follows:

1. A depth dynamic load balancing strategy is proposed to dynamically distribute work and avoid idling.
2. A heuristic sampling strategy is proposed to avoid performance downgrading on databases with short patterns.
3. Based on the proposed strategies and our previous Compact-SPADE algorithm in [**22**], two parallel algorithms called DPCompact-SPADE, and APCompact-SPADE are designed for mining frequent weighted clickstream patterns.
4. Experiments were conducted to compare the performance of the designed algorithm on various datasets. It was found that APCompact-SPADE and DPCompact-SPADE outperformed the state-of-the-art algorithms. Additionally, APCompact-SPADE can adapt its strategy to not be severely affected by databases with short patterns.

The rest of the paper is organized as follows. **Section 2** reviews related work. **Section 3** introduces important concepts and the Weighted Clickstream Pattern Mining (WCPM) problem. **Section 4** introduces a base algorithm for WCPM. Then, Section 5 describes how to parallelize the base algorithm of **Section 4**. Thereafter, Section 6 presents the results and a discussion. Finally, the last section concludes and discusses future work.

2. Related work

Agrawal and Srikant proposed the problem of SPM by adding the concept of sequential ordering to FIM [**2**]. They designed AprioriAll, which is inspired by the Apriori algorithm for FIM. As SPM started to gain more attention from researchers, many SPM algorithms were proposed, and they can be categorized as horizontal or vertical algorithms.

Algorithms of the horizontal family use a horizontal database format where each row contains a sequence id and a list of itemsets. An example of such a database is shown in **Table 1** and will be described in more detail in the next section. Popular horizontal algorithms are AprioriAll [**2**], and PrefixSpan [**32**]. AprioriAll is the first sequential pattern mining algorithm that used a search space pruning property called the Apriori property (or anti-monotonicity). The property states that all the subpatterns of a frequent sequential pattern must be frequent. Pei et al. [**32**] proposed PrefixSpan, an algorithm that recursively creates projected databases of smaller size. Each time the database is reduced, frequent patterns are larger, and scanning the database is faster due to the reduced size. Many authors have extended PrefixSpan to deal with various kinds of pattern mining problems. For example, PrefixSpan was adapted to mine patterns in large uncertain databases [**48**].

Algorithms in the vertical family use a vertical database format, in which sequence ids containing a pattern and the location of that pattern in the sequence are stored in memory. Some popular algorithms are SPADE [**47**], PRISM [19], and more recently CM-SPADE and CM-SPAM [**13**]. SPADE [**47**] was reported to be one of the most efficient algorithms [**13**]. It uses a concept of equivalence class and sublattice decomposition to divide the whole lattice (search space) into multiple fragments. Each fragment can fit into computer memory and be processed independently. PRISM [**19**] uses a special type of vertical database based on prime block encoding. Recently, Fournier-viger et al. [**13**] proposed the CMAP (i.e., co-occurrence map) structure, which stores co-occurrence information for a given database. CMAP is used to filter infrequent candidates early to speed up the mining process. It is integrated into SPAM [**6**] and SPADE [**47**] to create CM-SPADE and CM-SPAM. CM-SPADE and CM-SPAM were reported to be considerably faster than previous state-of-the-art methods.

Handling the weight constraint in SPM was first proposed by Yun et al. [**45**]. However, the work had to alter the original item weights to maintain the anti-monotonicity property for search space pruning. Alternatively, the maximum sequence weight was used as a restriction to preserve the anti-monotonicity property [**4**]. Extending this idea, Mukesh [**31**] modified the weight formula in [**45**] and combined it with the concept of time intervals to focus on events that occur in short time intervals.

Many FIM algorithms also have been proposed to handle weights [**43,44**]. Yun et al. [**43**] obtained an anti-monotonicity property by using the maximum and minimum weight ranges for the problem of weighted interesting pattern mining. Yun et al. [**44**] proposed the WMFP-tree and WMFP-array structures and combined them with weights to mine maximal frequent patterns in data streams. This paper, however, utilizes a weighted formula that preserves the anti-monotonicity property and is similar to the studies of Vo et al. and Lee et al. [**27,39**]. The weight of a pattern is the average weight of all its elements. However, while these studies are for itemset mining, the algorithm proposed in this paper applies a similar formula for the clickstream mining problem, which is more complex as the sequential ordering of items must be considered. Some researchers used a different weight formula, such as Wu et al. and Gan et al. [**18,30,40**]. Besides using weights, other researchers suggested using multiple constraints [**16,25,29,33,36,37**].

Table 1 A horizontal clickstream database [**21**].

CID	User clickstream
1	*a,c,c,d,f*
2	*d,c,b,a*
3	*f,b,f,c,b*
4	*e,a,c,b,c,b,f*
5	*a,b,c,e*

Clickstream pattern mining has numerous applications [**8,14,23**] (e.g. web log analysis and intrusion detection). However, most studies applied SPM algorithms to mine clickstream patterns rather than using or developing specialized CPM algorithms. For example, Cooley et al. and Demiriz [**9,10**] used SPM algorithms to discover interesting clickstream patterns of user browsing behaviors, while Ting et al. [**35**] analyzed unexpected clickstream patterns of users to support and improve a website's design. In the security domain, Lee and Stolfo [**28**] used association rules generated from clickstream patterns of system calls to build an intrusion detection classifier.

Parallel pattern mining algorithms. Most algorithms mentioned above are serial algorithms, designed to run on a single thread. However, as the data grows bigger and the problems get more complex, the runtime to solve those problems gets longer. It is easy to encounter big datasets in real applications where serial algorithms have very long runtimes. Additionally, multiprocessor computers are now easily available. As the current trend is to increase the number of cores or clusters instead of single-core speed, serial algorithms cannot benefit from those improvements because of their single-threaded nature. Thus, parallelism is one of the key factors to improve runtime performance. Parallel data mining algorithms, in general, have provided a large performance improvement over their serial counterparts. For example, the PARMA algorithm [**34**] was proposed to mine approximate association rules using a computer cluster. PARMA runs on the MapReduce framework and was reported to have a good speedup. A general-purpose framework was also proposed for mining frequent itemsets based on MapReduce [**7**], which groups similar transactions and processes each group on different cluster nodes. Parallel versions of PrefixSpan were also designed [**26,42**] to improve its performance by utilizing the Spark and MapReduce frameworks. MGPUCPM [**5**] is a parallel co-location mining algorithm that combines an effective update strategy and multiple GPUs. Since GPUs have less memory than computer RAM, the algorithm splits the data into multiple parts and transfers them to the GPU's memory when needed, with a speedup of up to 18 times being reported. Djenouri et al. [**12**] went even further by parallelizing on multiple cluster nodes, each equipped with a GPU to achieve much better performance compared with single thread mining algorithms (a speedup of up to 350 times). Vanahalli and Patil [**38**] proposed an algorithm called FCCI to mine frequent colossal closed itemsets. The algorithm combines two main strategies, an effective improved parallel processing algorithm to prune irrelevant features and an effective row cardinality table to check the closeness of rowsets. Their algorithm is parallelized by using the Open Multi-Processing application programming interface and manages to achieve a good increase in speed.

As sequential pattern mining gains more attraction from researchers, many algorithms have been proposed to keep up with the challenges of processing big datasets. However, parallelizing SPM algorithms can be challenging and complex [**17**]. Many things must be considered to achieve a performance gain, such as synchronization or workload balancing. Researchers have proposed several parallel SPM algorithms for different tasks, designed either for shared memory computers or distributed memory computers with dynamic or static load balancing.

For distributed memory machines, static load balancing is generally used because individual nodes (computers) can quickly access fragments of a database stored in their memory. Work is typically divided into several parts that are then assigned to each node such that they do not have to interact with each other. However, work distribution becomes crucial, and a poor distribution can lead to a very poor speedup. Several methods [**24,41**] have been proposed to estimate the relative effort for completing subtasks beforehand so that work can be distributed evenly. Some studies even suggested using GPUs in combination with CPUs in large clusters to achieve a greater speedup [**11,12**].

For shared-memory machines, dynamic load balancing is more efficient. SPADE is an efficient serial SPM algorithm that has been extended with parallelism in two main studies [**10,46**]. In the earlier study pSPADE was proposed for SPM using a hardware distributed shared memory machine named SGI Origin 2000, in which each processor is located on a different board but the memory is shared across processors. The latter, webSPADE, was developed for multiple processor Wintel machines, in which parallelism is achieved by multiple threads that are managed by the operating system. While the first study focused on SPM, the latter one was mainly about web clickstreams. Both algorithms use task parallelism but the latter also relies on data parallelism. Task parallelism was achieved by assuming that each branch of the lattice (representing the search space) can be processed independently as a task on a separate thread or core. Both algorithms combine parallelism with a breadth-first search.

In this paper, parallelism is considered for a shared memory machine. The parallelism approach that is employed is similar to webSPADE but used with a depth-first search while also considering weights, to develop an efficient parallel algorithm for WCPM.

3. Problem statement

This section introduces important concepts and presents a formal definition of the problem of WCPM. Some notations and symbols are summarized in **Table 2** for quick reference.

Let there be a set of symbols $A = \{a_1, a_2, a_3, \cdots, a_n\}$ representing different actions (such as mouse clicks to open folders), and a set of positive real value $W = \{w_1, w_2, w_3, \cdots, w_n\}$, in which each value w_i is a weight indicating the importance of the action $a_i \in A$ (e.g. see **Table 3**).

A **clickstream** $X = (x_1, x_2, \cdots, x_m)$ is a series of actions that happen one after another. For each $x_i \in A$ for $i \in [1, m]$, i is called the position of action x_i in X. Furthermore, if an action x_z follows another action x_w in$X$, x_w is said to happen before x_z in X, which is denoted by $x_w <_T x_z$. A clickstream X is called a k-clickstream if it contains k actions $(k = |X|)$.

For example, the clickstream $X = (a, b, e, b, f, c)$ is a 6-clickstream. The action b appears twice (at positions 2 and 4, if the first position is 1).

Assume that we have two clickstreams $X = (x_1, x_2, \cdots, x_m)$ and $Y = (y_1, y_2, \cdots, y_n)$ $(m \leq n)$. If there exist integers $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ such that $x_1 = y_{i1}$, $x_2 = y_{i2}$ $x_k = y_{ik}$, we call Y a **super clickstream** of X, and X is a **sub clickstream** of Y, which is denoted by $X \subseteq_S Y$ (Y contains X).

For example, (a, d), (a, c, c, f) and (a, c) are sub clickstreams of (a, c, c, d, f) in **Table 1**. However, (b,f) is not a sub clickstream of (a, c, c, d, f).

Table 2 Notation/Symbol table for a quick reference.

Notation/Symbol	Meaning
$A = \{a_1, a_2, a_3, \cdots, a_n\}$	A set of symbols representing different actions (such as mouse clicks)
$W = \{w_1, w_2, w_3, \cdots, w_n\}$	A set of positive real value. Each value w_i is a weight indicating the importance of the action $a_i \in A$
X, Y, Z	Clickstream patterns
x_i, y_i, z_i	Actions in the corresponding patterns (X, Y, Z respectively)
P_i	A pattern candidate
$cw(Y)$	Weight of the pattern Y
CDB_w	Clickstream database weight
$ws(Y)$	Weighted support of the pattern Y
$last_{Pi}$	Last action in the pattern P_i
$[P]$	An equivalence class $[P]$, or a set of patterns which have P as their prefix. P is also a pattern. For example, <a, b> is a member of $[a]$ because <a> is a prefix of <a, b>.
ω	Minimum support threshold
$\mathscr{W}$	Weighted co-occurrence map
k-class	Level k equivalence class. For example, a 1-class is level 1 equivalence class, which contains frequent patterns of length two as its members. A 2-class contains patterns of length three as its members and so on.
unprocessed k-class	A k-class in which its children (which are $k + 1$-patterns) are not yet discovered or visited.

Table 3 Action weights for the database of **Table 1** [**21**].

Action	Weight
a	0.5
b	0.9
c	0.2
d	0.4
e	0.7
f	0.6

a	
CID	Order
1	1
2	4
3	Ø
4	2
5	1

b	
CID	Order
1	Ø
2	3
3	2,5
4	4,6
5	2

c	
CID	Order
1	2,3
2	2
3	4
4	3,5
5	3

d	
CID	Order
1	4
2	Ø
3	Ø
4	Ø
5	Ø

e	
CID	Order
1	Ø
2	Ø
3	Ø
4	1
5	4

f	
CID	Order
1	5
2	Ø
3	1,3
4	7
5	Ø

Fig. 1. A vertical clickstream database [**21**].

A **user clickstream** is a series of actions that are generated by a user while the user performs various actions like navigating folders on a computer or surfing the Internet. A **clickstream database** CDB is a collection of pairs {cid, user clickstream} such that each pair contains a user clickstream and a unique clickstream id cid. A clickstream database is usually in a horizontal format (as in **Table 1**) and can be converted to a vertical format (as in **Fig. 1**). If a clickstream is contained in at least one user clickstream of an input database then the clickstream is called a (**clickstream**) **pattern** (i.e. P is a clickstream pattern if $\exists X \in CDB : P \subseteq sX$).

Assuming that Y is a user clickstream, the **weight** of Y (e.g. **Table 4**) is defined as follows:

$$cw(Y) = \frac{\sum_{c_i \in Y} w_i}{|Y|}$$

And the weight of a clickstream database CDB, denoted by CDB_w, is calculated as follows:

$$CDB_w = \sum_{Y \in CDB} cw(Y)$$

Table 4 Weights of user clickstreams for the example database of **Table 1** [**21**],

CID	Weight
1	0.38
2	0.5
3	0.64
4	0.57
5	0.58
CDB_w	2.67

Table 5 Some frequent weighted patterns when the minimum weighted support ω = 0.4 [**21**].

Pattern	Weighted Support
a	0.76
b	0.86
c	1
e	0.43
f	0.6
b,c	0.67
b,c,b	0.45
...	

The weighted support of a clickstream pattern Y is the sum of weights of user clickstreams, where Y appears, divided by the database weight. Formally, it is defined as:

$$ws(Y) = \frac{\sum_{X \in CDB \wedge Y \subset_s X} cw(X)}{CDB_w}$$

For a given clickstream database CDB, the **problem of weighted clickstream pattern mining** is to find all frequent weighted clickstream patterns. A clickstream *Y* is called a **frequent weighted clickstream pattern** if *Y* appears in at least one user clickstream of the database and its weighted support is greater than or equal to a minimum frequent weighted threshold ω, set by the user. Some frequent weighted patterns are shown in **Table 5**.

4. Mining frequent weighted clickstream patterns

This section briefly introduces the Compact-SPADE algorithm [**22**] and its components, which is extended to propose our parallelized algorithms in **Section 5**. First, Subsection 4.1 presents the WICList (Weighted ID-Compact Value List) data structure that Compact-SPADE uses to store information about each pattern. Then, Subsection 4.2 explains the candidate generation method used by Compact-SPADE. Finally, Subsection 4.3 describes WCMAP for eliminating unpromising patterns early reduces the amount of work.

The main process of Compact-SPADE is expanding a lattice of prefix-based classes (**Fig. 2**). It first starts by finding visible 1-patterns (i.e. 1-clickstream patterns) and computing their weighted support. Compact-SPADE then discards the infrequent patterns, and the remaining frequent ones are combined to form the next longer visible clickstream patterns. The process is then repeated following a depth-first search manner to find all remaining frequent weighted patterns.

4.1. WICList

The WICList is a data structure used to store both ids of clickstreams and their positions together in an integer to reduce memory usage. Let sid,pos, and dlen be respectively an id of a user clickstream, a position in a clickstream (of which id is equal to sid), and the minimum number of decimal digits required to hold the number of positions of the longest clickstream in the given database CDB.

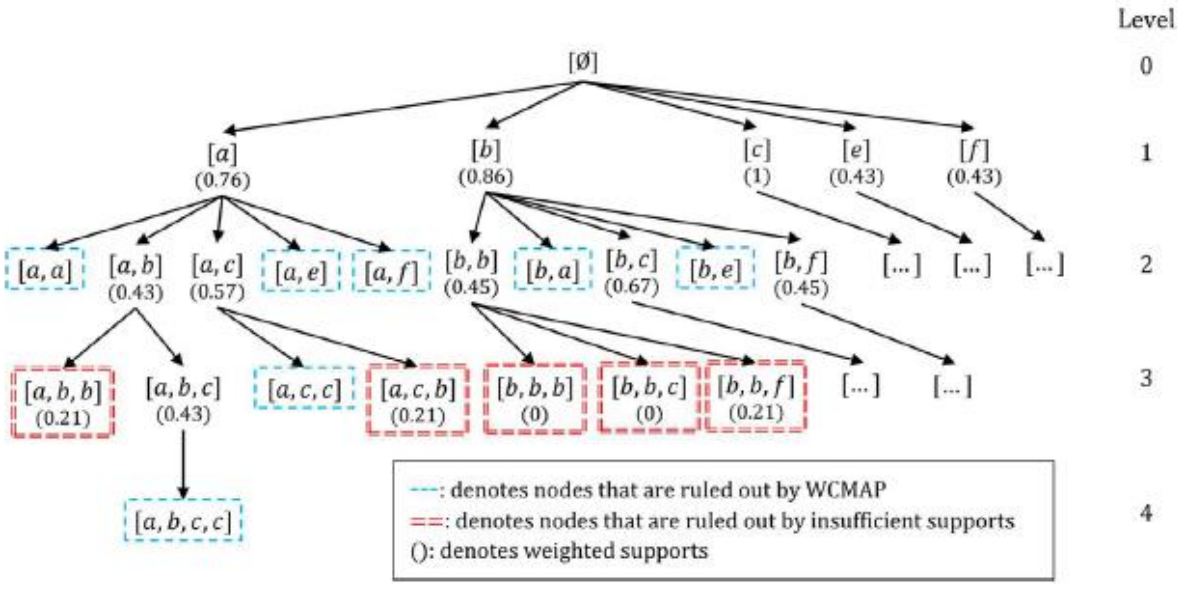

Fig. 2. A part of the traversed lattice for the example database.

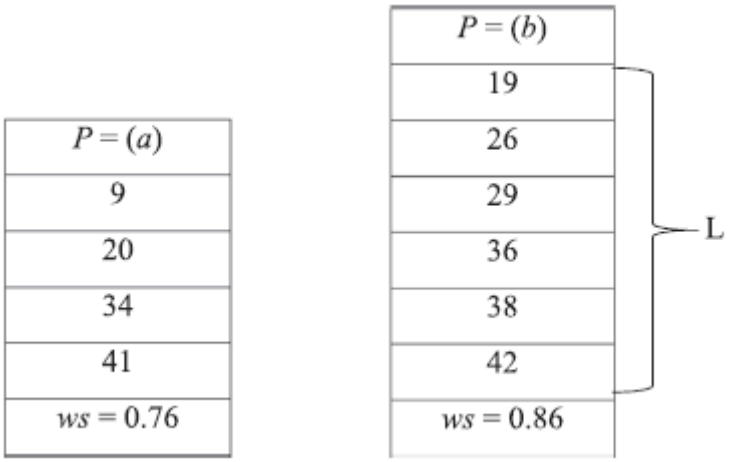

Fig. 3. WICLists of 1-patterns (a) and (b) with $dlen_{bit}$ = 3 (because the length of the longest user clickstream does not exceed 7).

A **compact value** *v* is defined as an integer number that encodes both a clickstream id sid and a position pos provided that $dlen_{bit}$ is known. Let there be an integer bit_X representing the bitmap of an integer value *X*, & denote the AND operator of two-bit arrays (or integer numbers), ≪ *N* denotes the logical left-shift operator and dlenbit be the minimum number of binary digits of the longest user clickstream. A compact value is created as:

$$cval(cid, pos, dlen_{bit}) = \{\alpha \in \mathbb{Z} | (bit_{cid} \ll dlen_{bit}) \& bit_{pos}\}$$

For example, assume that we have *cid* = 11,*pos* = 14, and dlen = 5. Then, bit_{cid} = 1011 and bit_{pos} = 1110. Because the result bit_v = cval(11,14,5) = 101101110, the compact value is *v* = 366.

Let *v* be a compact value, ≫ *N* be the logical right-shift operator, and A be the bitwise XOR operator. The cid and pos values can be retrieved from a compact value *v* by applying the following two functions, respectively:

$$ccid(v, dlen_{bit}) = \{\alpha \in \mathbb{Z} | bit_v \gg dlen_{bit}\}$$

$$cpos(v, dlen_{bit}) = \{\alpha \in \mathbb{Z} | bit_{ccid(v,dlen_{bit})} \ll dlen^b_{bit} it_v\}$$

Fig. 3 shows the WICLists of patterns for the database of **Table 1**. The second element of the WICList of (*a*) is the compact value *v* = 20, from which we can obtain *cid* = 2 and *pos* = 4 by using the above retrieval functions with $dlen_{bit}$ = 3.

4.2. Candidate generation

The proposed Compact-SPADE algorithm explores the search space by combining pairs of patterns to generate larger patterns. The process is based on the concept of lattice decomposition and prefix-

based classes which is introduced in [**47**]. Let there bean integer $k \geq 1$,and P_1 and P_2 be two frequent weighted (k + 1)-patterns, *X* be a common k-prefixof P_1 and P_2, $last_{P1}$ be the last action of P_1 and $last_{P2}$ be the last action of P_2. Then, P_1 = (X, $last_{P1}$) and P_2 = (X, $last_{P2}$) are said to belong to the same equivalence class $[X]_{\theta k}$ (because they share the same k-prefix, which is X). The equivalence class $[X]_{\theta k}$ is also called the k-class [X] in this paper.

If $P_1 \neq P_2$, two (k + 2)-candidates can be generated from these patterns, which are {P_3 = (P_1, $last_{P2}$)andP_4 = (P_2, $last_{P1}$)}. The pattern P_3 belongs to [P_1] and P_4 belongs to the [P_2]. If $P_1 = P_2$, then only (P_1, $last_{P1}$) is generated.

4.3. WCMAP (Weighted co-occurrence map)

To avoid unnecessary candidate generation, Fournier-Viger et al. [**13**] proposed the CMAP structure and a corresponding search space pruning strategy. It consists of pre-calculating the support of 2-patterns (formed from frequent 1-patterns). However, in the original CMAP, weights are not considered. Recently, the WCMAP [**21**] structure was proposed by adding average weights to the CMAP for weighted clickstream pattern mining. **Table 6** depicts part of a WCMAP for a minimum weighted support of ω = 0.4.

Example of using the WCMAP structure. Consider that ω = 0.4 and two frequent weighted patterns P_1 *= (b, c)* and P_2 *= (b,f)* with ws(P_1) *= 0.67* and ws(P_2) = *0.45*. Those patterns can be used to generate two candidates P_3 *= (b, c,f)* and P_4 *= (b,f,c)*. To determine if the candidates P_3 and P_4 are frequent, we can create WL_{P3} and WLP4 and compare WL_{P3}.ws and WL_{P4}.ws with ω. But by looking in the WCMAP, P_3 and P_4 can be identified as infrequent because the 2-patterns formed by the last actions of P_1 and P_2 (i.e., *(c,f)* and *(f, c))* are infrequent in the WCMAP (i.e. $\mathscr{W}(c,f)$ = 0.34 < ω and $\mathscr{W}(f,c)$ = 0.24 < ω). Thus, we can skip building WLP_3.ws and WLP_4.

Table 6 A part of a WCMAP for ω = 0.4 [**21**].

First action	Second action	Weighted support
...	...	...
c	*a*	0.19
	b	0.64
	c	0.34
	e	0.22
	f	0.34
...	...	...
f	*b*	0.24
	c	0.24
	f	0.24
...	...	...

5. The adaptive parallel Compact-SPADE algorithm

This section presents DPCompact-SPADE and APCompact-SPADE, two parallelized algorithms that extend Compact-SPADE. DPCompact-SPADE (**Section 5.3**) and APCompact-SPADE (**Section 5.4**) consist of two main components: 1) the core algorithm, Compact-SPADE, and 2) parallelism based on a depth load balancing (**Section 5.3**) or an adaptive dynamic load balancing (**Section 5.4**). APCompact-SPADE adapts to two types of datasets, short pattern datasets or long pattern datasets. If a dataset contains several short patterns, which are with the length of two and three, APCompact-SPADE uses horizontal dynamic load balancing (**Section 5.2**). Otherwise, it switches to depth dynamic load balancing (**Section**

5.3). The adaptative process is based on the proposed heuristic sampling method in **Section 5.4**. We also briefly talk about static load balancing in **Section 5.1** to be used as another baseline for performance comparison.

5.1. Static load balancing

Static load balancing is a common method for parallelism due to its ease of implementation. It divides tasks into several partitions that can correspond to the number of assigned threads (or cores, or processors). Each partition is processed by a thread and none of the threads interferes with other tasks. Briefly, the static load balancing is splitting all the elements in a 0-class [Ø] (i.e. the 1-classes) into *n* partitions, starting from the left-most to the right-most unprocessed 1-class. For example, assuming that we have 0-class *[Ø] = {[a], [b], [c], [e], f]}* and the number of threads *n* = 3, then the first thread expands *{[a], [b]}*, the second processes *{[c], [e]}* and the third works on *{f]}*. The parallel Compact-SPADE algorithm that uses this static load balancing is called StaticPCompact-SPADE.

One problem of static load balancing is load imbalance, in which some workloads require longer runtimes than the others. The runtime of the algorithm is usually equal to the longest workload's runtime. Hence, if the tasks are not properly distributed among all the threads, it can reduce the effectiveness of the parallelism.

5.2. Horizontal dynamic load balancing

Another common load balancing method is the horizontal (dynamic) load balancing, used in [**10,46**]. Unlike static load balancing, horizontal load balancing does not split the tasks into several workloads. It instead schedules unprocessed 1-classes and assigns each of them to an idle thread. If there is no idle thread, it waits until a thread is available again to assign the next unprocessed 1-class. We call our algorithm with horizontal load balancing HPCompact-SPADE (Horizontal Parallel Compact-SPADE).

For example, we have 0-class *[Ø] = {[a], [b], [c], [e], f]}* and the number of threads *n* = 3. The first thread processes *{[a]}*, the second processes *{[b]}* and the third works on *{[c]}*. Two unprocessed 1-classes are *{[e], f]}*, and since all threads are taken, the load balancer goes into a waiting state. If the second thread finished its work first, the load balancer will assign *{[e]}* to the second thread. After that, when the first thread becomes available, it will be assigned with *{f]}*. Even when the third thread finishes, the load balancer will put the thread into an idle state since there is no work left. The algorithm finishes when all threads are finished. Algorithm 1 depicts the pseudo-code for HPCompact-SPADE.

Though improving on StaticPCompact-SPADE, HPCompact-SPADE still encounters the same problems, since not every thread is fully utilized as the work nears its end. In the example, the third thread has no work and must wait for the other two threads.

Algorithm 1. HPCompact-SPADE

Algorithm 1. HPCompact-SPADE

HPCompact-SPADE ()
Input: minimum weighted support threshold ω and the clickstream database *CDB*
Output: the set $\mathcal{F}$ containing all frequent weighted clickstream patterns in *CDB*
1 $\mathcal{F} \leftarrow \varnothing$, a global variable (i.e., shared across methods)
2 SCAN *CDB* to determine the class $[\varnothing]$ containing all frequent weighted 1-patterns and their respective WICLists
3 $\mathcal{F} \leftarrow \mathcal{F} \cup [\varnothing]$
4 POPULATE $\mathcal{W}$, a global WCMAP, from $[\varnothing]$
5 $\mathcal{T} \leftarrow$ a global set of threads for parallelism
6 EXECUTE HLoadBalancer-Run() with $[\varnothing]$

HPNode-Expand()
Input: a parent pattern Y and a class $[X]$
7 $[Y] \leftarrow \varnothing$
8 FOR each pattern P_j in class $[X]$ DO
9 $\alpha \leftarrow$ the pattern generated from P_i and P_j with P_i being its prefix //The pattern having prefix P_i belong to class $[P_i]$
10 USE WCMAP $\mathcal{W}$ and WICList of α to determine if α is frequent or not
11 IF α is frequent THEN
12 $[P_i] \leftarrow [P_i] \cup \alpha$//Add the frequent pattern α to its corresponding class
13 $\mathcal{F} \leftarrow \mathcal{F} \cup \alpha$//Register the frequent pattern α to $\mathcal{F}$
14 FOR each pattern y_i in $[Y]$ DO
15 EXECUTE HPNode-Expand() with pattern y_i and class $[Y]$ // Recursively traverse class $[y_i]$

HLoadBalancer-Run()
16 WHILE there are still unprocessed classes in $[\varnothing]$ DO
17 IF $\mathcal{T}$ has idle threads THEN
18 $[Y] \leftarrow$ an unprocessed class in $[\varnothing]$
19 $t \leftarrow$ An idle thread in $\mathcal{T}$
20 EXECUTE HPNode-Expand() with pattern Y and class $[\varnothing]$ using thread t // $[Y]$ is an unprocessed class, and Y is a prefix pattern
21 ELSE
22 WAIT until there is a free idle thread

5.3. Recursive depth dynamic load balancing

The (recursive) depth (dynamic) load balancing is proposed based on the depth-first-search in Compact-SPADE to solve the workload imbalance issue. We call the parallel Compact-SPADE that uses this load balancing DPCompact-SPADE. Depth load balancing has two characteristics that help with generating and distributing workloads. Firstly, it uses recursion to generate parallelism. Secondly, the tasks that are assigned to a thread can be a class at any level. Any thread can transfer its partial workload, which is any k-class, to other idle threads so all threads remain active until the end of the work. Algorithm 2 illustrates the DPCompact-SPADE process.

The algorithm first starts sequentially using a single main thread and scans the whole database to calculate various weight information (i.e. the preparing data stage in **Fig. 4**). After which, the algorithm finds all the frequent weighted 1-patterns, the WICLists, and put those patterns to the 0-class *[Ø]* (line 1 to 4). The WCMAP $\mathcal{W}$ is then created and both a global working queue $\mathcal{Q}$ and thread pool $\mathcal{T}$ are initialized (lines 5 to 6). The algorithm enters load balancing phase after putting [Ø] into the workload queue $\mathcal{Q}$ (line 7) and it assigns [Ø] to the first thread (line 23). The first thread then enters DPNode-Expand () and expand all 1-classes in [Ø]. After filling all the elements of the left-most 1-class, the first thread transfers that 1-class to another idle thread. From here, two threads are running simultaneously and both repeat the same process of filling elements to a k-class and recursively transfer them to the other idle threads. If all threads are taken and $\mathcal{Q}$ is full, no task is transferred and each thread continues to further explore the lattice on its own (line 20). The transferring process starts again when $\mathcal{Q}$ is again not full. As a consequence, the number of threads is prioritized for the classes on the left of the lattice.

Algorithm 2. DPCompact-SPADE

Algorithm 2. DPCompact-SPADE

DPCompact-SPADE ()
Input: minimum weighted support threshold ω and the clickstream database *CDB*
Output: the set $\mathscr{F}$ containing all frequent weighted clickstream patterns in *CDB*
1 $\mathscr{F} \leftarrow \varnothing$, a global variable (i.e., shared across methods)
2 SCAN *CDB* to determine the equivalence class $[\varnothing]$ containing all frequent weighted 1-patterns satisfying ω and their respective WICLists
3 $\mathscr{F} \leftarrow \mathscr{F} \cup [\varnothing]$
4 POPULATE $\mathscr{W}$, a global WCMAP, from $[\varnothing]$
5 $\mathscr{T} \leftarrow$ a global set of threads for parallelism
6 $\mathscr{Q} \leftarrow \varnothing$, a global task queue with a max size $|\mathscr{Q}|_{max} = 3 * |\mathscr{T}|$
7 SUBMIT $[\varnothing]$ as a task to $\mathscr{Q}$
8 EXECUTE DLoadBalancer-Run()

DPNode-Expand()
Input: an equivalence class $[X]$
9 FOR each pattern P_i in class $[X]$ DO
10 $[P_i] \leftarrow \varnothing$
11 FOR each pattern P_j in class $[X]$ DO
12 $\alpha \leftarrow$ the pattern generated from P_i and P_j with P_i being its prefix //The pattern having prefix P_i belong to class $[P_i]$
13 USE WCMAP $\mathscr{W}$ and WICList of α to determine if α is frequent or not
14 IF α is frequent THEN
15 $[P_i] \leftarrow [P_i] \cup \alpha$//Add the frequent pattern α to its corresponding class
16 $\mathscr{F} \leftarrow \mathscr{F} \cup \alpha$//Register the frequent pattern α to $\mathscr{F}$
17 IF $\mathscr{T}$ has idle threads or $|\mathscr{Q}| < |\mathscr{Q}|_{max}$ THEN //$\mathscr{Q}$ is not full
18 IF $[P_i]$ is not empty SUBMIT $[P_i]$ as a task to $\mathscr{Q}$ // Transfer a part of the work to another idle thread
19 ELSE
20 EXECUTE DPNode-Expand() with class $[P_i]$ // Recursively expand $[P_i]$ further using the current thread

DLoadBalancer-Run()
21 WHILE $\mathscr{Q}$ still has tasks to distribute OR any thread in $\mathscr{T}$ is still busy DO
22 IF $\mathscr{T}$ has idle threads AND $|\mathscr{Q}| > 0$ DO // $\mathscr{Q}$ is not empty
23 $[q] \leftarrow$ the top class in $\mathscr{Q}$
24 REMOVE $[q]$ from $\mathscr{Q}$
25 $t \leftarrow$ An idle thread in $\mathscr{T}$
26 EXECUTE DPNode-Expand() with class $[q]$ using thread t
27 ELSE
28 WAIT until a thread finishes its work or new work is pushed into $\mathscr{Q}$

Example of DPCompact-SPADE. Considering three threads t_1, t_2 and t_3 are used, the size of the work queue $\mathscr{Q}$ is two, the input database is described in **Table 1**, and the minimum weighted support is ω = 0.4. DPCompact-SPADE first finds the 0-class *[Ø] = {a, b, c, d, e, f}*, create their respective WICLists and build the WCMAP $\mathscr{W}$.

DPCompact-SPADE then puts *[Ø]* into $\mathscr{Q}$ and assigns t_1 to work on *[Ø]*. After filling in the content of first 1-class *[a] = {(a, b), (a, c)}*, t_1 submit *[a]* to $\mathscr{Q}$. Because threads t_2 and t_3 are free, the load balancer transfers *[a]* to t_2 and t_1 continue to form the content of 1-class *[b]*.

Assuming that t_2 can create the content of 2-class *[a, b]* before t_1 creates *[b]*, then t_2 push *[a, b]* to Ø and t_3 is then assigned with *[a,b]*. After which *[b]* is pushed into Ø because there is no free thread and Ø is not full.

Assuming that *[c]* is created by t_1, yet t_2, t_3 are still busy, so *[c]* is pushed into $\mathscr{Q}$. At this point $\mathscr{Q}$ = *{[b], [c]}* is full, thus when t_1 forms new classes (e.g., *[e], [e, e], [e, a]*), t_1 traverses them by itself until either there is a free thread or $\mathscr{Q}$ is not full. Assuming that t_2 finishes *[a,b]*, then it takes *[b]* from $\mathscr{Q}$. The working queue $\mathscr{Q}$ then only contains *{[c]}*.

DPCompact-SPADE repeatedly applies the aforementioned depth-first search approach with parallelism until the lattice has been fully traversed, and all frequent weighted clickstream patterns have been enumerated.

5.4. Adaptive dynamic load balancing

DPCompact-SPADE cannot perform well on short pattern datasets where most patterns are shifted towards a length of one, two, or three. First, the depth load balancing does not immediately initialize multiple threads right off the bat.

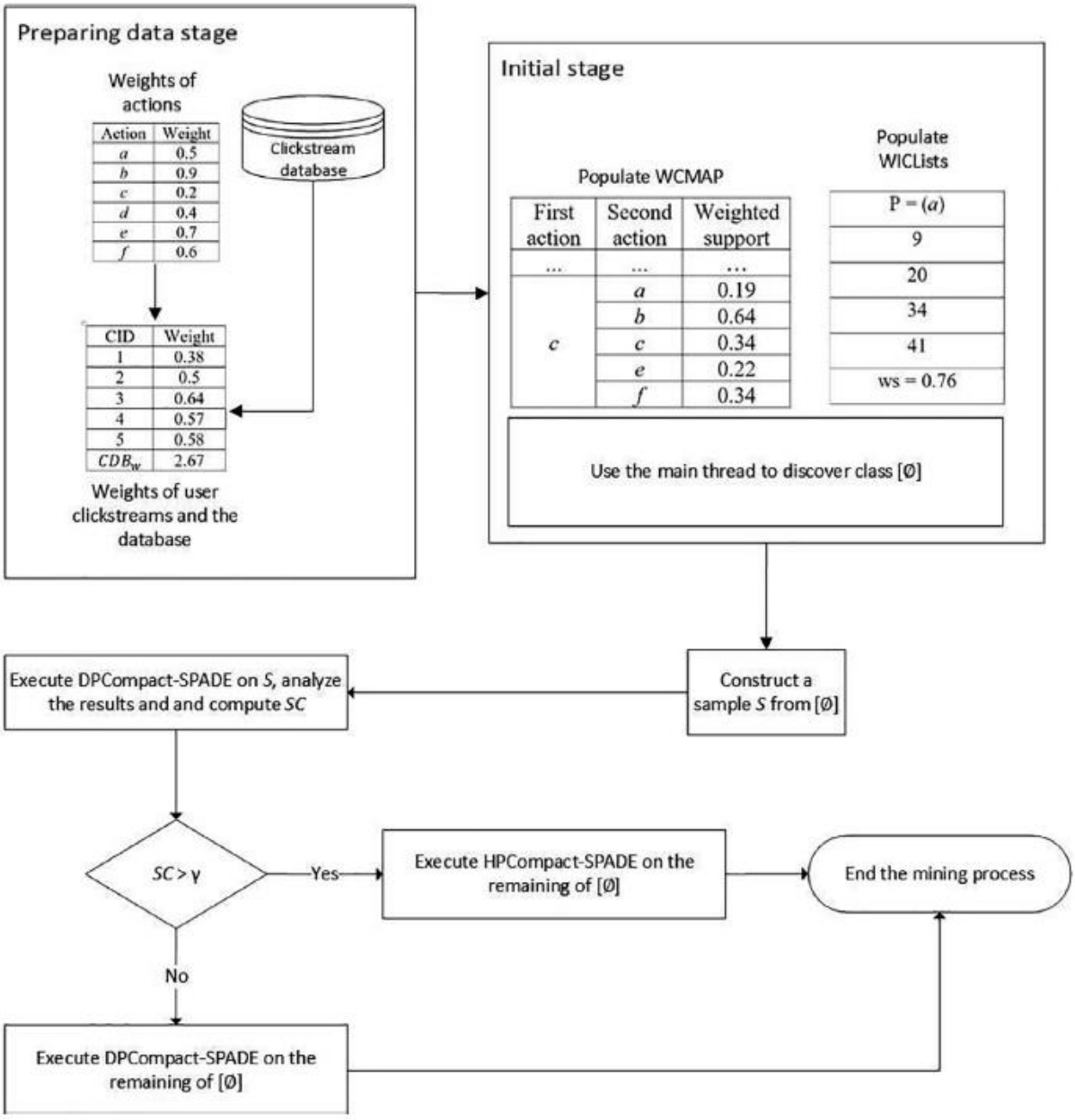

Fig. 4. The flowchart of APCompact-SPADE.

It depends on recursively transferring work via depth-first search. It starts with one thread, then multiplies the occupied threads as the search goes deeper into the lattice. If the lattice is shallow, then

the workload is very short, and tasks are not generated fast enough for all idle threads. In this case, static or horizontal load balancing performs better (which is shown and discussed more in the experimental section). Adaptive (dynamic) load balancing is proposed to tackle this issue. It is based on our proposed heuristic sampling called join estimation via sampling.

Join estimation via sampling. This idea aims to switch the balancing method to the most efficient one, depending on the distribution of patterns and joins in a database.

Let $join_k$ be the number of joins between k-patterns, $join_S$ be the number of all joins, SC = ($\sum^{3}_{k=1}$ $join_k$)/$join_S$, and y be the proportional ratio that is given by the user. If SC > γ, the load balancing switches from depth to horizontal. In our experiments, we use y = 70%. That is, if the joins of 1,2,3-patterns take more than 70% of all joins, horizontal load balancing is preferred over depth load balancing.

In real situations, we do not always know the distribution in advance. Therefore, we propose a heuristic sampling to estimate the distribution and the SC value. The sample *S* is decided as follows:

- 1% of 1-classes in [Ø] are picked to make S. Additionally, the size of S must be at least two and no more than 50.
- The sample S must include 1-class [a] where a is the 1-pattern with the highest weighted support among all 1-patterns in *[Ø]*.
- The remaining 1-classes are selected from left-most to right-most in *[Ø]*.

Adaptive load balancing. The adaptive load balancing is described in Algorithm 3, APCompact-SPADE is DPCompact-SPADE with ALoadBalancer-Run instead of DLoadBalancer-Run. Fig. 4 illustrates the workflow of APCompact-SPADE.

Algorithm 3. ALoadBalancer-Run

Algorithm 3. ALoadBalancer-Run

```
ALoadBalancer-Run()
1   S ← ∅ // The sample set, which is initialized to an empty set
2   [∅] ← the first element in
3   SELECT the 1-class with the highest weighted support from [∅] and put it into S
4   SELECT the remaining 1-classes from [∅] and put them into S
5   EXECUTE DPCompact-SPADE on S and compute SC
6   IF SC > γ THEN
7     EXECUTE HPCompact-SPADE with remaining unprocessed 1-classes in [∅]
8   ELSE
9     EXECUTE DPCompact-SPADE with remaining unprocessed 1-classes in [∅]
```

6. Experimental results

To evaluate the proposed algorithm, experiments were carried out on a computer running Windows 8.1 64 bits, with Java 8, 16 GB of RAM, an Intel Core i7-8750H 2.20 GHz processor with six physical cores, and hyper-threading technology. The algorithms were implemented in Java by extending the SPMF open-source package[1] [**15**]. The Java virtual machine was configured to use up to 10 GB for the memory heap. The turbo boost technology was turned off to stabilize the algorithms' runtime.

Experiments were done using six benchmark databases.[2] Their characteristics are presented in Table 7. FIFA, BIBLE, and SIGN are small-to-medium clickstream databases, while Korasak, Chainstore, and D9000S4 are big databases. Additionally, Chainstore and D9000S4 are short pattern databases, while the others are long pattern databases (**Fig. 10**). Chainstore was originally in a transactional format and converted to the clickstream format. Initially, all databases did not have weights. As such, a weight between 1 and 100 was assigned randomly to each action.

6.1. Evaluation of DPCompact-SPADE and APCompact-SPADE's runtime and memory usage

This section presents experiments on runtime and maximum memory usage of the proposed parallel algorithms. To evaluate the effectiveness and efficiency of DPCompact-SPADE and APCompact-SPADE, we compared them with CM-SPADE [**13**], Parallel DBV [20], StaticPCompact-SPADE (**Section 5.1**), and HPCompact-SPADE (**Section 5.2**). Two algorithms, CM-SPADE [**13**], and Parallel DBV [**20**] are the state-of-the-art algorithms for SPM. While CM-SPADE is serial, Parallel DBV is a parallel algorithm. Those two algorithms are designed for non-weighted sequential pattern mining. Therefore, they need to be integrated with the weighted support calculation. These modified versions are called CM-WSPADE and Parallel WDBV respectively. Additionally, the (Parallel) DBV algorithm was originally implemented in the C# language, so we re-implemented it in Java. Moreover, to evaluate the benefits of parallelism, Compact-SPADE and WDBV were also tested. The compared algorithms were run with different minimum threshold values on different databases (**Figs. 5 and 6**). Additionally, for DPCompact-SPADE and APCompact-SPADE, we set workload queue size $|Q|_{max} = 3*$ number of threads. We did not notice any performance gain but larger memory usage above this value.

It is observed in **Fig. 5** that HPCompact-SPADE, APCompact-SPADE, and DPCompact-SPADE achieve better overall runtime than StaticPCompact-SPADE, CM-WSPADE, WDBV, and Parallel WDBV. However, their runtimes can fluctuate depending on database types. The fastest algorithm on short databases can be the slowest on normal databases. On the four normal databases (FIFA, BIBLE, SIGN, and Korasak), the general order based on runtimes is DPCompact-SPADE > APCompact-SPADE > H PCompact-SPADE > StaticCompact-SPADE $\geq$ Compact-SPADE > Parallel WDBV > CM-WSPADE > WDBV. For example, on BIBLE and at m = 11%, the runtime of DPCompact-SPADE, APCompact-SPADE, HPCompact-SPADE, StaticCompact-SPADE, Compact-SPADE, Parallel WDBV, CM-WSPADE, and WDBV is 5.5, 6.6, 6.4, 10.4, 23.9, 29, 45.6, and 89.6 s, respectively. The gaps get bigger as m decreases. At m = 7%, they need 26.1, 27.2, 41.9, 63.6, 125.9, 162.4, 278.8, and 431.1 s. DPCompact-SPADE runs 1.04,1.6, 2.43, 4.82, 6.22, 10.68, and 16.5 times faster than APCompact-SPADE, HPCompact-SPADE, StaticCompact-SPADE, Compact-SPADE, Parallel WDBV, CM-WSPADE, and WDBV, respectively.

[1] ***http://www.philippe-fournier-viger.com/spmf/.***

[2] *The BIBLE, FIFA, SIGN, Chainstore datasets can be obtained from the SPMF website (**http://www.philippe-fournier-viger.com/spmf/index.php? link=datasets.php**) and Kosarak can be obtained at **http://fimi.ua.ac.be/data/**. D9000S4 is a synthetic database that are generated by our modified version of IBM Quest Data Generator at: **https://github.com/halfvim/quest**.*

Table 7 Characteristics of the test databases.

Database	Clickstream count	Action count	Average clickstream length
BIBLE	36,369	13,905	21.6
FIFA	20,450	2990	34.74
SIGN	730	310	51.99
Korasak	990,002	41,270	8.1
Chainstore	1,112,949	46,086	7.2
D9000S4	9,000,000	5000	3.72

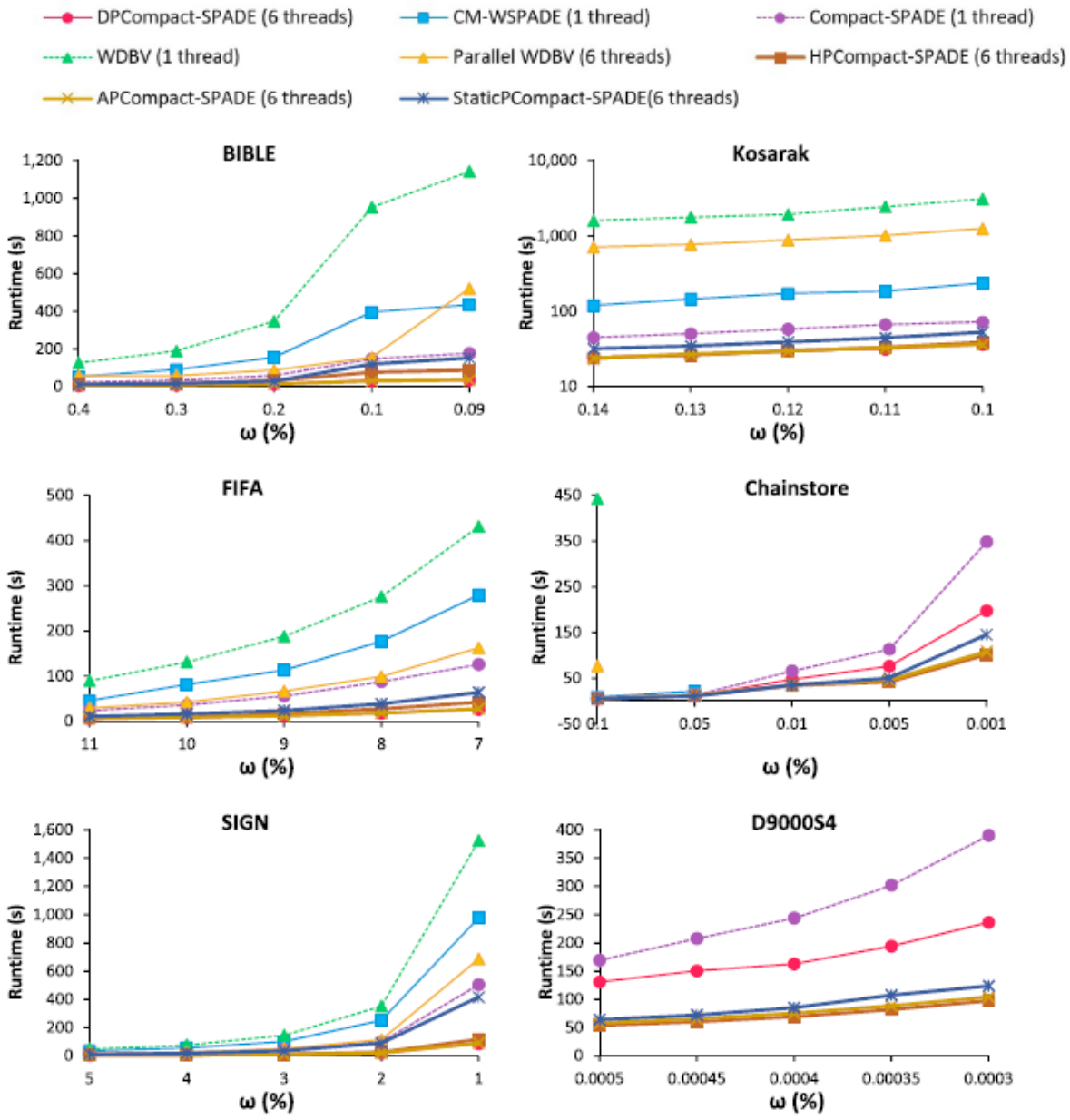

Fig. 5. Runtimes for various minimum weighted support values.

However, on the two short pattern databases, Chain-store and D9000S4, their performance switches around as HPCompact-SPADE becomes the fastest, followed by APCompact-SPADE, StaticCompact-SPADE, DPCompact-SPADE, Compact-SPADE, CM-WSPADE, Parallel WDBV, and finally WDBV. For

example, on Chainstore at m = 0.001%, the runtimes for HPCompact-SPADE, APCompact-SPADE, StaticCompact-SPADE, DPCompact-SPADE, and Compact-SPADE are 101.9, 108.1, 145.3, 197.6, and 348.3 s, respectively. The three algorithms, CM-WSPADE, Parallel WDBV, and WDBV could not run at that threshold.

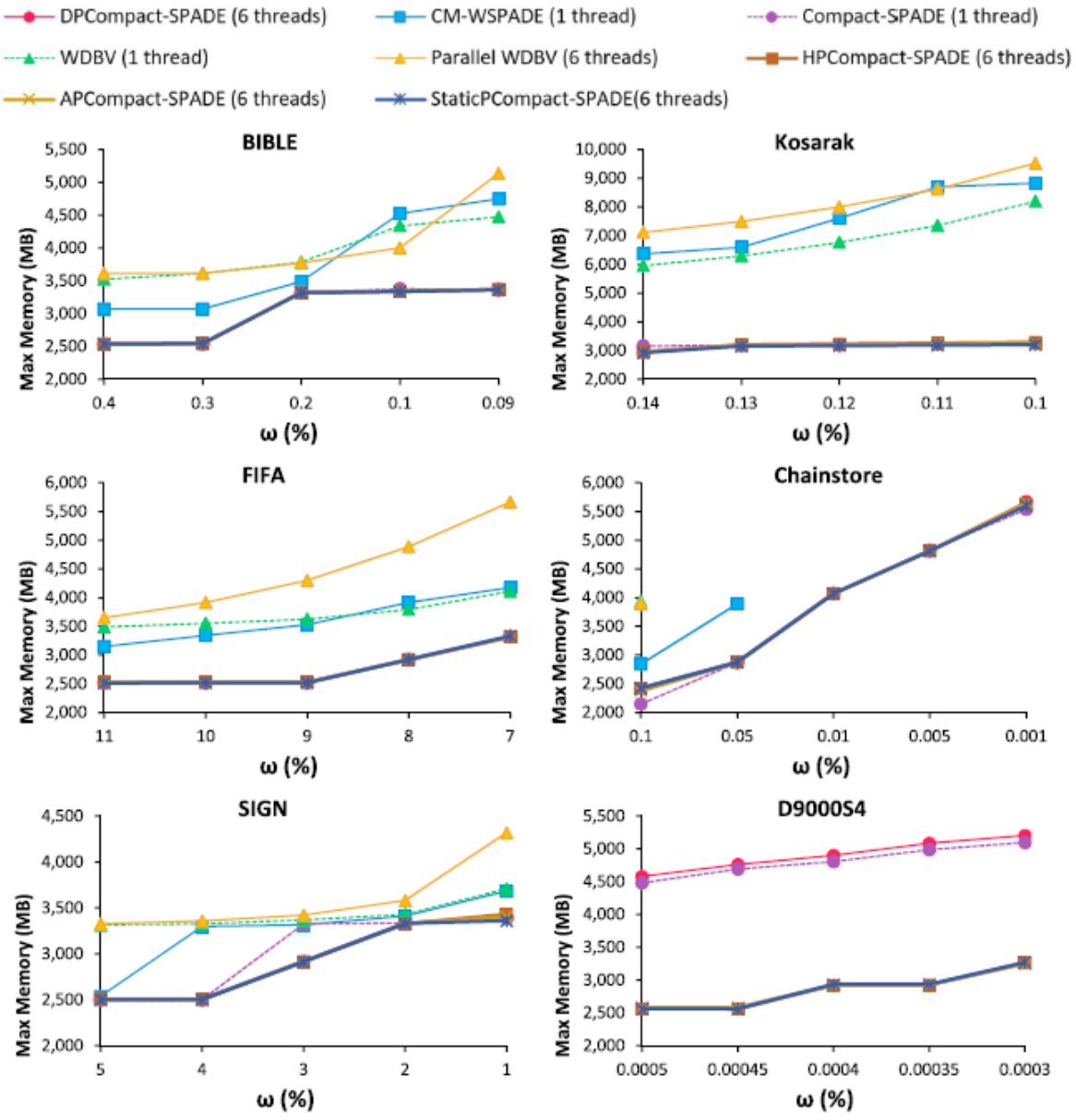

Fig. 6. Maximum memory consumption for various minimum weighted support values.

APCompact-SPADE is neither faster than DPCompact-SPADE on long pattern databases or HPCompact-SPADE on short pattern databases. This is because its sampling process takes small runtimes to estimate the distributions of joins. On every database, it always comes as the second fastest algorithm, while DPCompact-SPADE takes the lead on four normal databases and HPCompact-SPADE is fastest on the two short pattern databases. The time required for sampling is around 0% to 6.5%. The longest sampling time is 6.5% on D9000S4.

Regarding memory consumption, **Fig. 6** shows that the five algorithms with Compact-SPADE as a core (DPCompact-SPADE, APCompact-SPADE, HPCompact-SPADE, StaticCompact-SPADE, and Compact-SPADE) used less memory than CM-WSPADE, WDBV, and Parallel WDBV. Parallel WDBV used the most memory in most cases. CM-WSPADE used more memory than WDBV on Kosarak and was roughly equal to WDBV on SIGN. On FIFA and BIBLE, CM-WSPADE used less memory than WDBV only on some certain thresholds (i.e. $\omega \geq 0.1\%$ on BIBLE and $\omega \geq 8\%$ on FIFA). On Chainstore, CM-WSPADE used less memory than both WDBV and Parallel WDBV. DPCompact-SPADE, APCompact-SPADE, HPCompact-SPADE, and StaticCompact-SPADE consumed as much memory as Compact-SPADE on most datasets. For example, on FIFA, they all use roughly 2400 MB at ω = 11% to 3300 MB at ω = 7%, and the difference in memory consumption is less than 1% at every value of ω. D9000S4 is the only database for which DPCompact-SPADE had a significantly higher memory usage than Compact-SPADE. This illustrates the efficiency of the depth-first parallel search and load balancing methods.

6.2. Multi-thread evaluation

This section evaluates DPCompact-SPADE and APCompact-SPADE in multithreaded environments. All parallel algorithms (DPCompact-SPADE, APCompact-SPADE, HPCompact-SPADE, StaticPCompact-SPADE, Parallel WDBV) were executed with fixed ra values while the maximum numbers of threads vary from 1 to 10 in increments of 2. For each dataset, ra was set to the lowest value in the previous experiment. The runtime speedups and maximum memory usage are presented in **Figs. 7-9**.

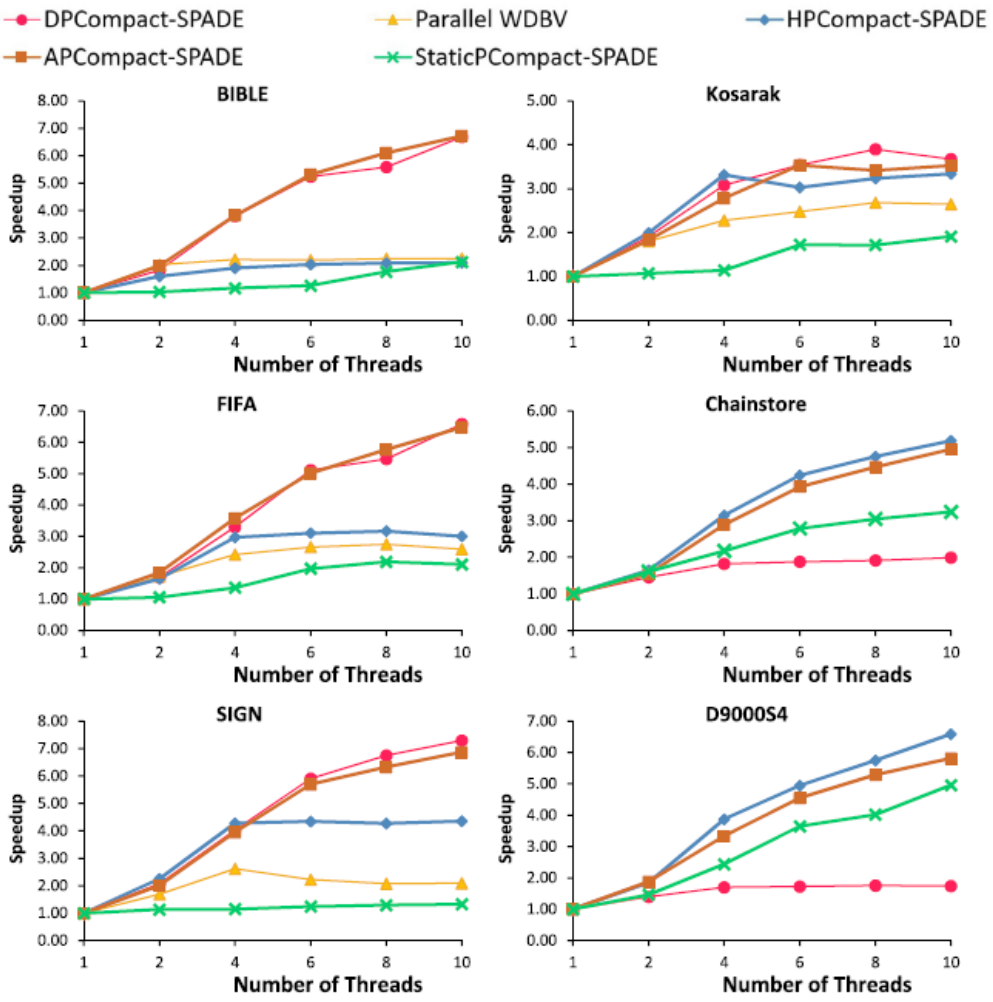

Fig. 7. Speedup of the tested algorithms including the WCMAP runtime for various numbers of threads on the six test databases.

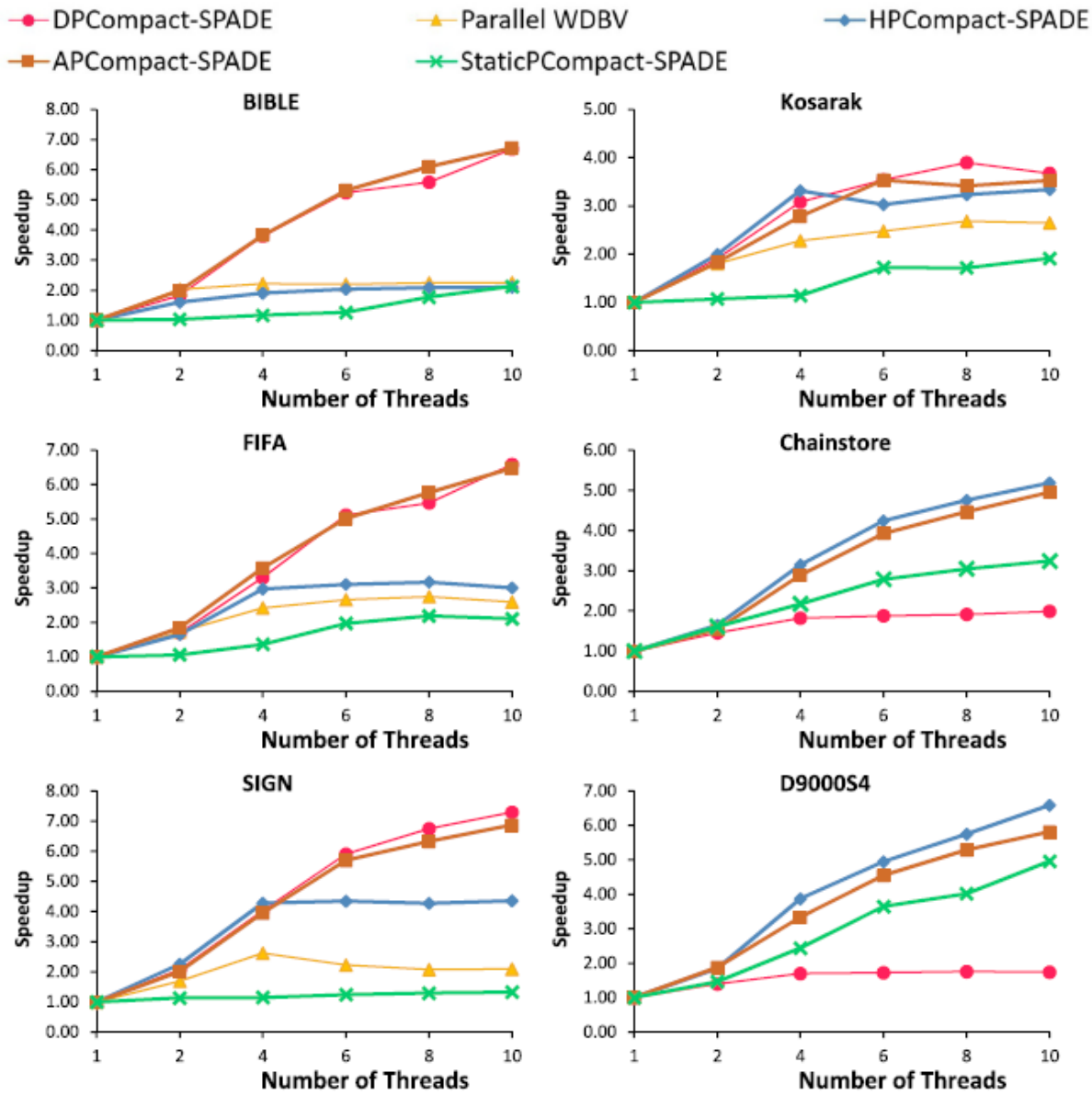

Fig. 8. Speedup of the tested algorithms without the WCMAP runtime for various numbers of threads on the six test databases.

As can be seen in **Fig. 8**, all parallel algorithms scale with the number of threads to a certain extent. However, the scaling is not linear. They all benefit greatly from adding a few threads, but as the number of threads increases further, the performance does not continue to improve, and can even worsen.

The experiment was run with six physical cores and hyper-threading. For this hardware setup, the optimal number of threads was found to be between six to ten. The greatest speedups for DPCompact-SPADE and APCompact-SPADE were achieved using ten threads for SIGN, where the runtimes were reduced by 7.27 and 6.85 times, respectively. The highest speedups for HPCompact-SPADE and StaticPCompact-SPADE were 4.95 and 4.05 times on D9000S4 with ten threads. For Parallel WDBV, the greatest speedups were using eight threads on Kosarak, where the runtime was decreased by 2.68 times. Our proposed parallel algorithms (DPCompact-SPADE and APCompact-SPADE) appear to potentially scale better with the number of cores than the others. The speedups of the four algorithms DPCompact-SPADE, APCompact-SPADE, HPCompact-SPADE, and StaticPCompact-SPADE are quite

small on Kosarak (less than two). Additionally, DPCompact-SPADE also has bad scalability on D9000S4 and Chainstore. The reasons for this behavior are investigated in the next paragraphs.

To have better insights, the construction time of WCMAP was recorded and compared with the total runtime of four parallel Compact-SPADE-based algorithms on the lowest m values (**Table 8**). On Korasak with ω = 0.1%, WCMAP's construction time is 21.9 s, which takes about two-thirds of the four algorithms' total runtimes. Similarly for Chainstore, WCMAP takes up 13.1% to 25.4% of the four algorithms' total runtimes. On D9000S4, it takes from 10% to 23.8%. More time is spent on building WCMAP for those three datasets because there are many big frequent 1-patterns at such low threshold values.

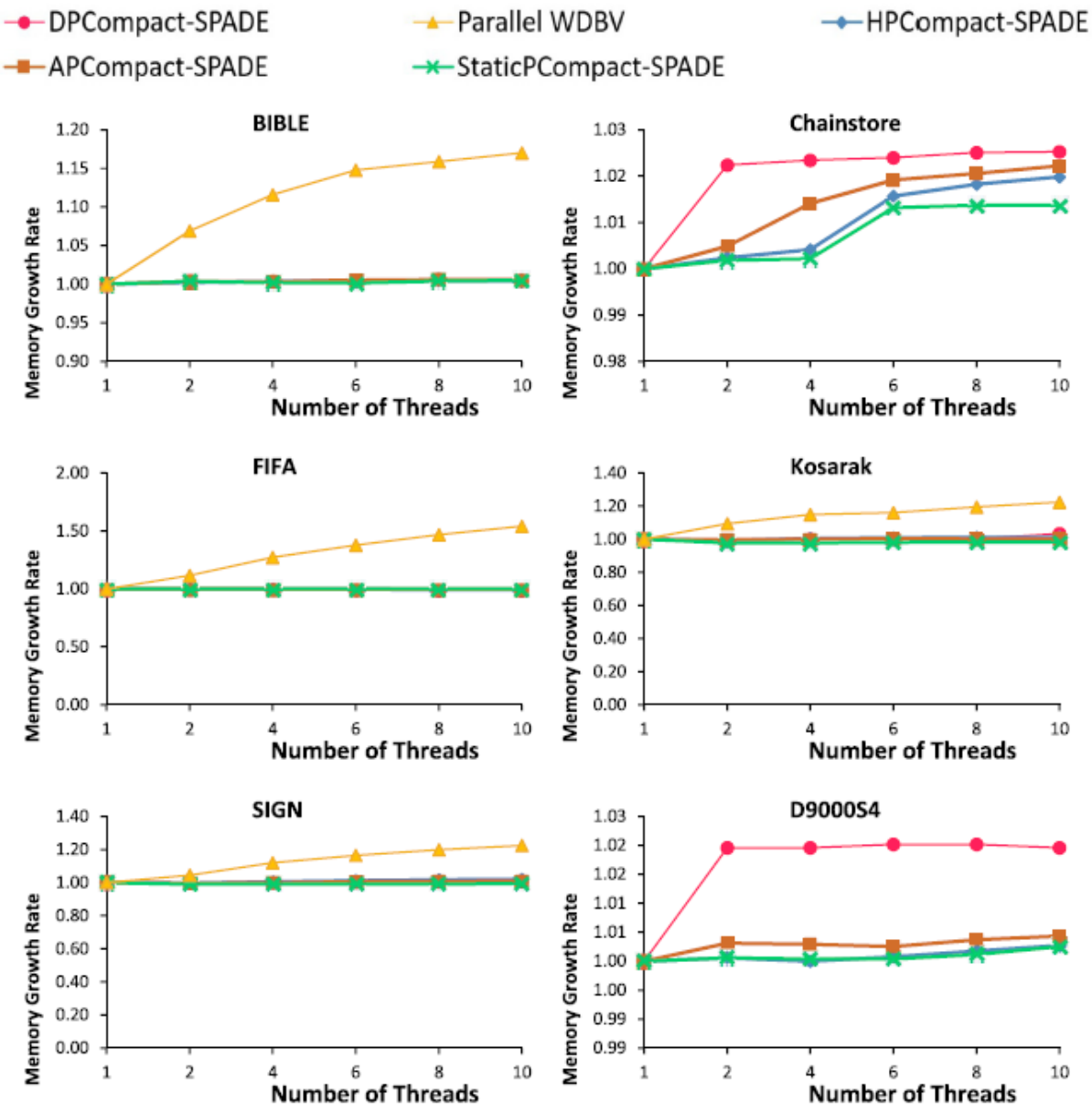

Fig. 9. The memory growth rate of the tested algorithms for various numbers of threads on six test databases.

Table 8 WCMAP's construction time for each database.

Database	ω(%)	WCMAP's construction time (s)	DPCompact-SPADE (s)	HPCompact-SPADE (s)	APCompact-SPADE (s)	Static PCompact-SPADE
BIBLE	0.09	2.5	35.9	88.4	34.5	153.1
FIFA	7	1.9	26.1	41.9	27.2	63.6
SIGN	1	0.3	85.6	116.2	88.7	416.2
Kosarak	0.1	21.9	36.2	38.7	36.3	52.8
Chainstore	0.001	25.9	197.6	101.9	108.1	145.3
D900054	0.0003	23.2	236.1	97.4	103.9	123.9

On other remaining datasets, the construction time took less than 10% of the total runtime. Furthermore, the WCMAP construction process is still serial and thus does not benefit from the increased number of threads. Putting the WCMAP's construction time aside, the highest speedup is pushed from 2.08 to 3.9, 2 to 3.53, 1.96 to 3.34, 1.5 to 1.91 times on Kosarak for DPCompact-SPADE, APCompact-SPADE, HPCompact-SPADE, and StaticPCompact-SPADE, respectively (**Fig. 8**)

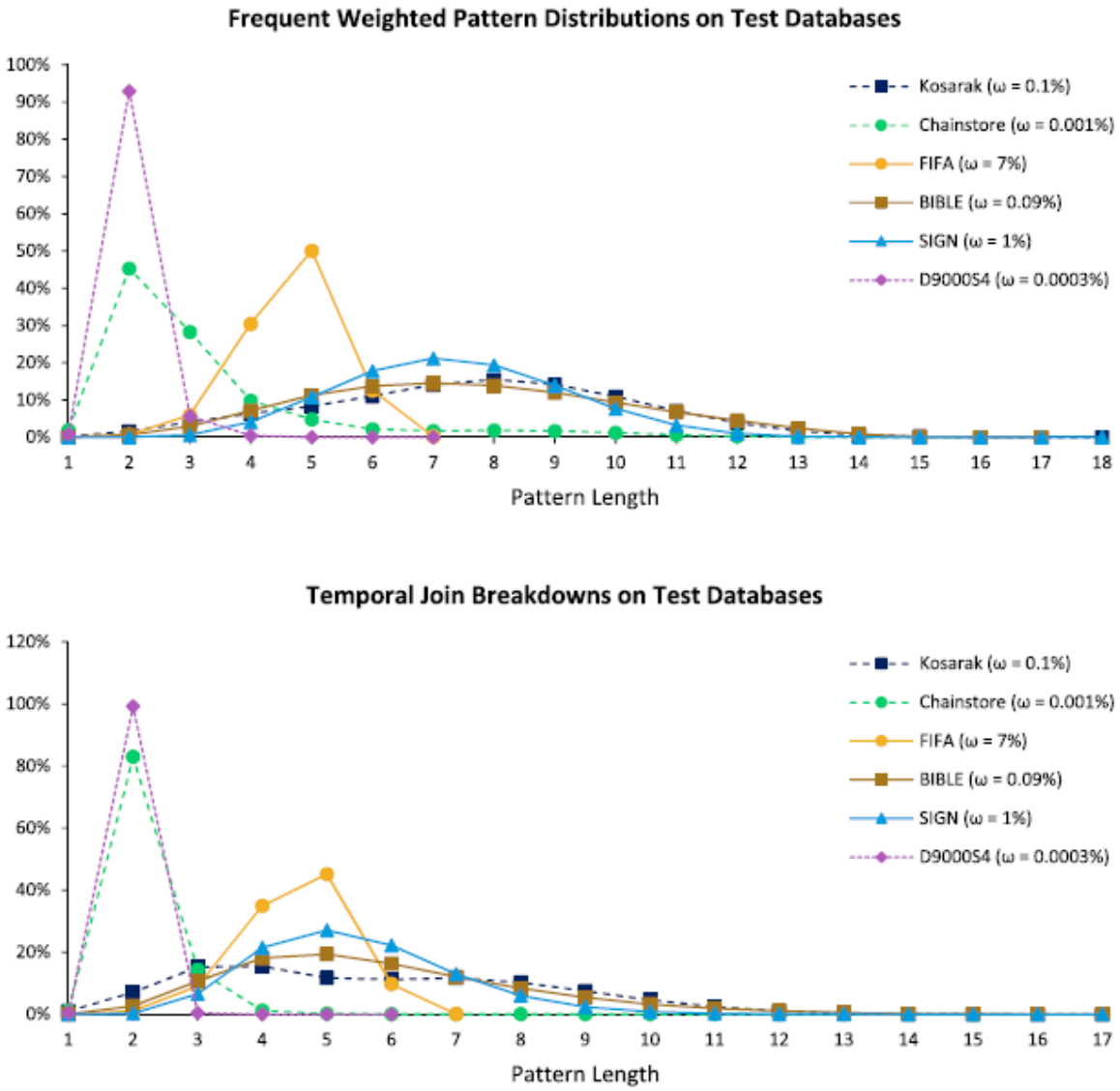

Fig. 10. Frequent weighted pattern distribution and temporal join breakdown for each test database and a fixed m value.

This indicates that on some large databases with many different actions, parallelizing the WCMAP construction could give a performance boost.

Besides the WCMAP construction time issue, there is a second reason why DPCompact-SPADE did not scale well on Chain-store and D9000S4. **Fig. 10** presents distributions of frequent patterns and temporal joins for each of the test databases. We can see that unlike the other databases, Chainstore and D9000S4's pattern and join distributions are heavily shifted towards short patterns, with 2-patterns representing 45% and 93% of all patterns, and joins of 2-patterns representing 82% and 99% of the total, respectively. Those two databases contain very small portions of high-level classes. DPCompact-SPADE handles parallelism by delegating a thread's higher-level classes to other idle threads. With every transfer to a new thread, the classes are smaller. With those databases, the classes do not have a high enough level (>=3), resulting in fewer possibilities for the algorithm to delegate work to other threads. Another limitation is that, if the classes are small and short, individual thread executions of DPNode-Expand() become shorter, comparatively increasing the overhead cost of managing the thread pool. This suggests that DPCompact-SPADE does not scale up well for short pattern databases. The other parallel algorithms do not have this problem because each thread works on a whole 1-class or a group of 1-classes.

Regarding memory consumption (**Fig. 9**), despite our expectations of an increased memory footprint, we observed a roughly 0.01% difference in the maximum memory consumption between all Compact-SPADE-based algorithms. Additionally, Compact-SPADE-based algorithms used less memory than Parallel WDBV and the reason is explained as follows.

To explain this behavior, we first need to look at the data structures. Both Compact-SPADE-based algorithms and Parallel WDBV use data structures that are based on the concept of vertical data format. However, while Compact-SPADE-based algorithms utilize a vertical format, Parallel WDBV uses a semi-vertical one with a dynamic bit vector (DBV). The advantages of DBV were demonstrated [**20**], but its drawback is bit-sparsity in the early stage of the mining process. The DBV may waste memory if it only contains few true bits. For example, a DBV a size of 10,000 bits may contain only 10 true bits, and that causes memory waste.

Secondly, two parameters, the database size, and m, may also affect the effectiveness of DBV. The larger the database size, the larger the DBV likely is and the lower the m value, the fewer true bits it contains. Additionally, the algorithm needs to iterate over the whole bit arrays. If the DBVs are large and sparse, the runtime increases. For example, iterating over DBV of size 10,000 but with only 10 true bits is significantly slower than iterating over a DBV of size 100 containing 10 true bits. Compact-SPADE, on the other hand, does not use bit arrays to represent user clickstream ids. Each element in a WICList represents both a user clickstream and a position index of the pattern. This results in less memory consumption as is observed in **Figs. 6 and 9** for Compact-SPADE-based algorithms.

How to choose the right number of threads? Picking the right number of threads for Compact-SPADE-based algorithms can be tricky, because if the number is too high or too low, optimal performance may not be achieved. Based on the experimental results, we suggest setting the number of threads to about the number of physical cores on the CPU should give a considerable runtime improvement without risking a big increase in memory consumption.

7. Conclusions and future work

WCPM is a data mining task that has many applications because data from various fields can be encoded as a clickstream database. However, current algorithms do not take advantage of parallelism and/or do not consider weights indicating the importance of actions. This paper addressed these issues by presenting DPCompact-SPADE and APCompact-SPADE. Comprehensive experiments have shown that DPCompact-SPADE and APCompact-SPADE outperformed (P) WDBV and CM-WSPADE on all test databases and APCompact-SPADE can adapt to avoid performance degradation on short pattern databases. Moreover, it was found that parallelism improved Compact-SPADE's performance considerably in terms of runtime and scaled well with the number of threads. However, nothing is perfect. The DPCompact-SPADE operates less well when the lattice representing the search space is imbalanced and WCMAP's construction is still performed with a single thread. Moreover, APCompact-SPADE's runtime can lag a bit behind because its sampling may require some runtime to analyze the database's pattern and join distributions.

In future works, we plan to develop methods to further improve the performance by parallelizing the WCMAP construction process and developing alternative parallel approaches that could deal better with imbalanced lattices and small-size WICLists. We also plan to adapt the proposed algorithm for mining patterns in quantitative databases.

References

[1] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, in: Proc. ACM SIGMOD Int. Conf. Manag. Data, 1993, pp. 207-216.

[2] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proc. Int. Conf. Data Eng., 1995, pp. 3-14.

[3] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proc. Elev. Int. Conf. Data Eng., IEEE Comput. Soc. Press, 1996, pp. 3-14.

[4] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, A novel approach for mining high-utility sequential patterns in sequence databases, ETRIJ. 32 (2010) 676-686.

[5] W. Andrzejewski, P. Boinski, Efficient spatial co-location pattern mining on multiple GPUs, Expert Syst. Appl. 93 (2018) 465-483.

[6] J. Ayres, J. Flannick, J. Gehrke, T. Yiu, Sequential pattern mining using a bitmap representation, in: Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min., ACM Press, New York, USA, 2002, pp. 429-435.

[7] A. Belhadi, Y. Djenouri, J.-W. Lin, A. Cano, A general-purpose distributed pattern mining system, Appl. Intell. 50 (9) (2020) 2647-2662.

[8] L. Bermingham, I. Lee, Mining distinct and contiguous sequential patterns from large vehicle trajectories, Knowledge-Based Syst. 189 (2020) 105076.

[9] R. Cooley, B. Mobasher,J. Srivastava, Web mining: information and pattern discovery on the World Wide Web,, in: Proc. IEEE Int. Conf. Tools with Artif. Intell., 1997, pp. 558-567.

[10] A. Demiriz, webSPADE: a parallel sequence mining algorithm to analyze web log data, in: Proc. Int. Conf. Data Min., 2002, pp. 755-758.

[11] Y. Djenouri, A. Belhadi, P. Fournier-Viger, H. Fujita, Mining diversified association rules in big datasets: a cluster/GPU/genetic approach, Inf. Sci. (Ny) 459 (2018) 117-134.

[12] Y. Djenouri, D. Djenouri, A. Belhadi, A. Cano, Exploiting GPU and cluster parallelism in single scan frequent itemset mining, Inf. Sci. (Ny) 496 (2019) 363-377.

[13] P. Fournier-Viger, A. Gomariz, M. Campos, R. Thomas, Fast vertical mining of sequential patterns using co-occurrence information, in: Proc. Pacific-Asia Conf. Knowl. Discov. Data Min, 2014, pp. 40-52.

[14] P. Fournier-Viger, J. Li, J.-C.-W. Lin, T.T. Chi, R. Uday Kiran, Mining cost-effective patterns in event logs, Knowledge-Based Syst. 191 (2020) 105241.

[15] P. Fournier-Viger,J.C.W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, H.T. Lam, The SPMF open-source data mining libraryversion 2, in: Proc.Jt. Eur. Conf. Mach. Learn. Knowl. Discov. Databases, 2016, pp. 36-40.

[16] J. Fowkes, C. Sutton, A subsequence interleaving model for sequential pattern mining, in: Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min., 2016, pp. 835-844.

[17] W. Gan,J.-W. Lin, P. Fournier-Viger, H.-C. Chao, P.S. Yu, A survey of parallel sequential pattern mining, ACM Trans. Knowl. Discov. Data 13(3) (2019) 134.

[18] W. Gan, J.-C.-W. Lin, P. Fournier-Viger, H.-C. Chao, P.S. Yu, HUOPM: high-utility occupancy pattern mining, IEEE Trans. Cybern. 50 (2020) 1195-1208.

[19] K. Gouda, M. Hassaan, M.J. Zaki, Prism: an effective approach for frequent sequence mining via prime-block encoding, J. Comput. Syst. Sci. 76 (2010) 88-102.

[20] B. Huynh, B. Vo, V. Snasel, An efficient method for mining frequent sequential patterns using multi-Core processors, Appl. Intell. 46 (3) (2017) 703-716.

[21] H.M. Huynh, L.T.T. Nguyen, B. Vo, A. Nguyen, V.S. Tseng, Efficient methods for mining weighted clickstream patterns, Expert Syst. Appl. 142 (2019) 112993.

[22] H.M. Huynh, L.T.T. Nguyen, B. Vo, A. Nguyen, V.S. Tseng, Efficient methods for mining weighted clickstream patterns, Expert Syst. Appl. 142 (2020) 112993, https://doi.org/10.1016Zj.eswa.2019.112993.

[23] H.M. Huynh, L.T.T. Nguyen, B. Vo, U. Yun, Z.K.. Oplatková, T.-P. Hong, Efficient algorithms for mining clickstream patterns using pseudo-IDLists, Futur. Gener. Comput. Syst. 107 (2020) 18-30.

[24] R. Kessl, Probabilistic static load-balancing of parallel mining of frequent sequences, IEEE Trans. Knowl. Data Eng. 28 (5) (2016) 1299-1311.

[25] T. Kieu, B. Vo, T. Le, Z.-H. Deng, B. Le, Mining top-k co-occurrence items with sequential pattern, Expert Syst. Appl. 85 (2017) 123-133.

[26] B. Kim, G. Yi, Location-based parallel sequential pattern mining algorithm, IEEE Access 7 (2019) 128651-128658.

[27] G. Lee, U. Yun, K.H. Ryu, Mining frequent weighted itemsets without storing transaction IDs and generating candidates, Int. J. Uncertainty, Fuzziness Knowledge-Based Syst. 25 (01) (2017) 111-144.

[28] W. Lee, S.J. Stolfo, Data mining approaches for intrusion detection data mining approaches for intrusion detection, Proc. Conf. USENIX Secur., 1998.

[29] J.-W. Lin, Y. Djenouri, G. Srivastava, Efficient closed high-utility pattern fusion model in large-scale databases, Inf. Fusion 76 (2021) 122-132.

[30] J.-C.-W. Lin, Y. Djenouri, G. Srivastava, U. Yun, P. Fournier-Viger, A predictive GA-based model for closed high-utility itemset mining, Appl. Soft Comput. 108 (2021) 107422.

[31] M. Patel, N. Modi, K. Passi, An effective approach for mining weighted sequential patterns, in: Proc. Int. Conf. Smart Trends Inf. Technol. Comput. Commun., 2016, pp. 904-915.

[32] J. Pei, J. Han, Q. Chen, M.-C. Hsu, B. Mortazavi-Asl, H. Pinto, U. Dayal, PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth, in: Proc. Int. Conf. Data Eng., 2001, pp. 215-224.

[33] F. Petitjean, T. Li, N. Tatti, G.I. Webb, Skopus: Mining top-k sequential patterns under leverage, Data Min. Knowl. Discov. 30 (5) (2016) 1086-1111.

[34] M. Riondato, J.A. DeBrabant, R. Fonseca, E. Upfal, PARMA: a parallel randomized algorithm for approximate association rules mining in map reduce, in: Proc. 21st ACM Int. Conf. Inf. Knowl. Manag., ACM Press, New York, USA, 2012, p. 85.

[35] I.H. Ting, C. Kimble, D. Kudenko, UBB mining: Finding unexpected browsing behaviour in clickstream data to improve a web site's design, in: Proc. ACM Int. Conf. Web Intell., 2005, pp. 179-185.

[36] T. Van, B. Vo, B. Le, Mining sequential patterns with itemset constraints, Knowl. Inf. Syst. 57 (2) (2018) 311-330.

[37] T. Van, A. Yoshitaka, B. Le, Mining web access patterns with super-pattern constraint, Appl. Intell. 48 (11) (2018) 3902-3914.

[38] M.K. Vanahalli, N. Patil, An efficient parallel row enumerated algorithm for mining frequent colossal closed itemsets from high dimensional datasets, Inf. Sci. (Ny) 496 (2019) 343-362.

[39] B. Vo, F. Coenen, B. Le, A new method for mining frequent weighted itemsets based on WIT-trees, Expert Syst. Appl. 40 (4) (2013) 1256-1264.

[40] J.-M.-T. Wu, J.-C.-W. Lin, A. Tamrakar, High-utility itemset mining with effective pruning strategies, ACM Trans. Knowl. Discov. Data. 13 (2019) 1-22.

[41] K.-M. Yu, J. Zhou, Parallel TID-based frequent pattern mining algorithm on a PC Cluster and grid computing system, Expert Syst. Appl. 37 (3) (2010) 2486-2494.

[42] X. Yu, Q. Li, J. Liu, Scalable and parallel sequential pattern mining using spark, World Wide Web. 22 (1) (2019) 295-324.

[43] U. Yun, Efficient mining of weighted interesting patterns with a strong weight and/or support affinity, Inf. Sci. (Ny) 177 (17) (2007) 3477-3499.

[44] U. Yun, G. Lee, K.H. Ryu, Mining maximal frequent patterns by considering weight conditions over data streams, Knowledge-Based Syst. 55 (2014) 4965.

[45] U. Yun, J.J. Leggett, WSpan: Weighted sequential pattern mining in large sequence databases, in: Proc. Int. IEEE Conf. Intell. Syst., 2006, pp. 512-517.

[46] M.J. Zaki, Parallel sequence mining on shared-memory machines, J. Parallel Distrib. Comput. 61 (3) (2001) 401-426.

[47] M.J. Zaki, SPADE: an efficient algorithm for mining frequent sequences, Mach. Learn. 42 (2001) 31-60.

[48] Z. Zhao, D. Yan, W. Ng, Mining probabilistically frequent sequential patterns in large uncertain databases, IEEE Trans. Knowl. Data Eng. 26 (2014) 1171-1184.