DATA-PERFORMANCE CHARACTERIZATION OF FREQUENT PATTERN MINING ALGORITHMS

Sayaka Akioka

School of Interdisciplinary Mathematical Sciences,
Meiji University, Tokyo, Japan

ABSTRACT

Big data quickly comes under the spotlight in recent years. As big data is supposed to handle extremely huge amount of data, it is quite natural that the demand for the computational environment to accelerates, and scales out big data applications increases. The important thing is, however, the behavior of big data applications is not clearly defined yet. Among big data applications, this paper specifically focuses on stream mining applications. The behavior of stream mining applications varies according to the characteristics of the input data. The parameters for data characterization are, however, not clearly defined yet, and there is no study investigating explicit relationships between the input data, and stream mining applications, either. Therefore, this paper picks up frequent pattern mining as one of the representative stream mining applications, and interprets the relationships between the characteristics of the input data, and behaviors of signature algorithms for frequent pattern mining.

KEYWORDS

Stream Mining, Frequent Mining, Characterization, Modeling, Task Graph

1. INTRODUCTION

Big data quickly comes under the spotlight in recent years. Big data is expected to collect gigantic amount of data from various data sources, and analyze those data across conventional problem domains in order to uncover new findings, or people's needs. As big data is supposed to handle extremely huge amount of data compared to the conventional applications, it is quite natural that the demand for the computational environment, which accelerates, and scales out big data applications, increases. The important thing here is, however, the behavior or characteristics of big data applications are not clearly defined yet.

Big data applications can be classified into several categories depending on the characteristics of the applications, such as behaviors, or requirements. Among those big data applications, this paper specifically focuses on stream mining applications. A stream mining application is such an application that analyzes data, which arrive one after another in chronological order, on the fly. Algorithms specialized for stream mining applications are intensively studied [1-30], and Gaber et al. published a good review paper on these algorithms [31].

High performance computing community has been investigating data intensive applications, which analyze huge amount of data as well. Raicu et al. pointed out that data intensive applications, and stream mining applications are fundamentally different from the viewpoint of data access patterns, and therefore the strategies for speed-up of data intensive applications, and stream mining applications have to be radically different [32]. Many data intensive applications often reuse input data, and the primary strategy of the speed-up is locating the data close to the target CPUs. Stream mining applications, however, rarely reuse input data, so this strategy for data intensive applications does not work in many cases. Modern computational environment has been and is evolving mainly for speed-up of benchmarks such as Linpack [33], or SPEC [34]. These benchmarks are relatively scalable according to the number of CPUs. Stream mining applications are not scalable to the contrary, and the current computational environment is not necessarily ideal for stream mining applications. The simplest approach is to use this template and insert headings and text into it as appropriate. Additionally, many researchers from machine learning domain, or data mining domain point out that the behavior, execution time more specifically, of stream mining applications varies according to the characteristics, or features of the input data. The problem is, however, the parameters, or the methodology for data characterization is not clearly defined yet, and there is no study investigating explicit relationships between the characteristics of the input data, and the behavior of stream mining applications, either.

Therefore, this paper picks up frequent pattern mining as one of the representative stream mining applications, and interprets the relationships between the characteristics of the input data, and behaviors of signature algorithms for frequent pattern mining. The rest of this paper is organized as follows. Section 2 describes a model of stream mining algorithms in order to share the awareness of the problem, which this paper focuses on. Then, the section also briefly introduces related work. Section 3 overviews the application that this paper picks up, and illustrates the algorithms those are typical solutions for the application. Section 4 explains the methodology of the experiments, shows the results, and gives discussions over those results. Section 5 concludes this paper.

2. RELATED WORK

2.1. A Model of Stream Mining Algorithms

A stream mining algorithm is an algorithm specialized for a data analysis over data streams on the fly. There are many variations of stream mining algorithms, however, general stream mining algorithms share a fundamental structure, and a data access pattern as shown in Figure 1 [35]. A stream mining algorithm consists of two parts, stream processing part, and query processing part. First, the stream processing module in stream processing part picks the target data unit, which is a chunk of data arrived in a limited time frame, and executes a quick analysis over the data unit. The quick analysis can be a preconditioning process such as a morphological analysis, or a word counting. Second, the stream processing module in stream processing part updates the data, which are cached in one or more sketches, with the latest results through the quick analysis. That is, the sketches keep the intermediate analysis, and the stream processing module updates the analysis incrementally as more data units are processed. Third, the analysis module in stream processing part reads the intermediate analysis from the sketches, and extracts the essence of the data in order to complete the quick analysis in the stream processing part. Finally, the query

processing part receives this essence for the further analysis, and the whole process for the target data unit is completed.

Based on the model shown in Figure 1, we can conclude that the major responsibility of the stream processing part is to preprocess each data unit for the further analysis, and that the stream processing part has the huge impact over the latency of the whole process. The stream processing part also needs to finish the preconditioning of the current data unit before the next data unit arrives. Otherwise, the next data unit will be lost as there is no storage for buffering the incoming data in a stream mining algorithm. On the contrary, the query processing part takes care of the detailed analysis such as a frequent pattern analysis, or a hot topic extraction based on the intermediate data passed by the stream processing part. The output by the query processing part is usually pushed into a database system, and there is no such an urgent demand for an instantaneous response. Therefore, only the stream processing part needs to run on a real-time basis, and the successful analysis over all the incoming data simply relies on the speed of the stream processing part.

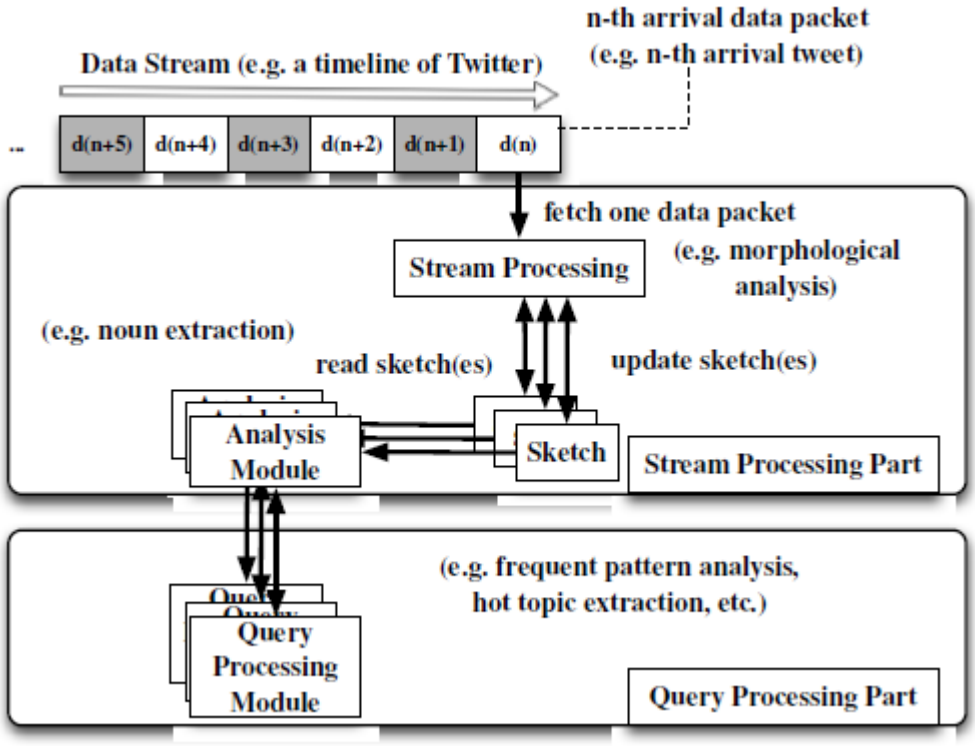

Figure 1. A model of stream mining algorithms.

The model in Figure 1 also indicates that the data access pattern of the stream mining algorithms is totally different from the data access pattern of so-called data intensive applications, which is intensively investigated in high performance computing community. The data access pattern in the data intensive applications is a write-once-read-many [32]. That is, the application refers to the necessary data repeatedly during the computation. Therefore, the key for the speedup of the application is to place the necessary data close to the computational nodes for the faster data accesses throughout the execution of the target application. On the contrary, in a stream mining

algorithm, a process refers to its data unit only once, which is a read-once-write-once style. Therefore, a scheduling algorithm for the data intensive applications is not simply applicable for the purpose of the speedup of a stream mining algorithm.

2.2. A Task Graph

A task graph is a kind of pattern diagrams, which represents data dependencies, control flows, and computational costs regarding a target implementation. A task graph is quite popular for scheduling algorithm researchers. In the process of the development of scheduling algorithms, the task graph of the target implementation works as if a benchmark, and provides the way to develop a scheduling algorithm in a reproducible fashion. The actual execution of the target implementation in the actual computational environment provides realistic measurement, however, the measurement varies according to the conditions such as computational load, or timings at the moment. This fact makes difficult to compare different scheduling algorithms in order to determine which scheduling algorithm is the best for the target implementation with the target computational environment. A task graph solves this problem, and enables fair comparison of scheduling algorithms through simulations.

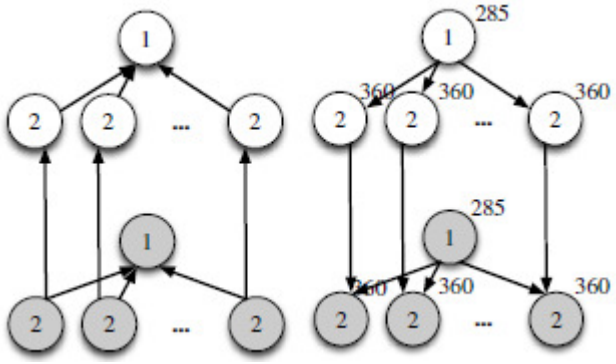

Figure 2. An example of a task graph

for all training data **do**
(1) fetch one data unit v
for all attributes for v **do**
(2-1) update the weight sum for this attribute
(2-2) update the mean value of this attribute
end for
end for

Figure 3. A pseudo code for the training stage of Naïve Bayes algorithm.

As shown in the previous section, the model of a stream mining algorithm has data dependencies both across the processes, and inside one process. Therefore, a task graph for a stream mining algorithm should consist of a data dependency graph, and a control flow graph [36]. Figure 2 is an example of a task graph of the training stage of Naïve Bayes classifier [37]. In Figure 2, the left figure is a data dependency graph, and the right figure is a control flow graph. Figure 3 represents the pseudo code for the task graph shown in Figure 2.

Both a data dependency graph, and a control flow graph are directed acyclic graphs (DAGs). In a task graph, each node represents a meaningful part of the input code. Nodes in white are codes in the preceding process, and nodes in gray are codes in the successive process. The nodes with the same number represents the same meaningful part in the input source code, and the corresponding line in the pseudo code shown in Figure 3. The nodes with the same number in the same color indicate that the particular part of the input source code is runnable in parallel. Here, nodes in a control flow graph have numbers. Each number represents execution cost of the corresponding node. Each array in a data dependency graph indicates a data dependency. If an arrow comes up from node A to node B, the arrow indicates that node B relies on the data generated by node A. Similarly, each array in a control flow graph represents the order of the

execution between nodes. If an arrow comes up from node A to node B, the arrow indicates that node A has to be finished before node B starts. The nodes in the same level are possible to be executed in parallel. The nodes with the same number indicate that the particular line in the pseudo code is runnable in parallel. There are two major difficulties for describing stream mining applications with task graphs. One problem is that the concrete parallelism strongly relies on the characteristics of the input data. The other problem is that a cost for a node varies according to the input data. That is, a task graph for a stream mining application is impossible without the input data modeling.

2.3. Related Work

There are several studies on task graph generation, mainly focusing on generation of random task graphs. A few projects reported task graphs generated based on the actual well-known applications; however, those applications are from numerical applications such as Fast Fourier Transformation, familiar to high performance computing community for years.

Task Graphs for Free (TGFF) provides pseudo-random task graphs [38, 39]. TGFF allows users to control several parameters, however, generates only directed acyclic graphs (DAGs) with one or multiple start nodes, and sink nodes. A period, and deadline is assigned to each task graph based on the length of the maximum path in the graph, and the user specified parameters.

GGen is another random task graph generator proposed by Cordeiro et al. [40]. GGen generates random task graphs according to the well-known random task generation algorithms. GGen also provides a graph analyzer, which characterizes randomly generated task graphs, based on the longest path, the distribution of the out-degree, and the number of edges.

Task graph generator provides both random task graphs, and task graphs extracted from the actual implementations such as Fast Fourier Transformation, Gaussian Elimination, and LU Decomposition [41]. Task graph generator also provides a random task graph generator which supports a variety of network topologies including star, and ring. Task graph generator also provides scheduling algorithms as well.

Tobita et al. proposed Standard Task Graph Set (STG), evaluated several scheduling algorithms, and published the optimal schedules for STG [42, 43]. STG is basically a set of random task graphs. Tobita et al. also provided task graphs from numerical applications such as a robot control programs, a sparse matrix solver, and SPEC fpppp [34].

Besides the studies on task graph generation, Cordeiro et al. pointed out that randomly generated task graphs can create biased scheduling results, and that the biased results can mislead the analysis of scheduling algorithms [40]. According to the experiments by Cordeiro et al., a same scheduling algorithm can obtain a speedup of 3.5 times for the performance evaluation only by changing the random graph generation algorithm. Random task graphs contribute for evaluation of scheduling algorithms, however, do not perfectly cover all the domains of parallel, and distributed applications as Cordeiro et al. figured out in their work. Especially for stream mining applications, the characteristic of the application behaviors is quite different from the characteristic of the applications familiar to the conventional high performance computing community [32]. Task graphs generated from the actual stream mining applications have profound significance in the better optimization of stream mining applications.

These all projects point out the importance of fair task graphs, and seek the best solution for this problem. These projects, however, focus only on data dependencies, control flows, and computational costs of each piece of an implementation. The computational costs are often decided in a random manner, or based on the measurements through speculative executions. No project pays attention to the relationships between the variation of the computational costs, and the characteristics of the input data.

3. ALGORITHMS

3.1. Frequent Pattern Mining

This paper focuses on two algorithms for frequent pattern mining. Frequent pattern mining was originally introduced by Agrawal et al. [44], and the baseline is the mining over the store items purchased on a per-transaction-basis. The goal of the mining is finding out all the association rules between sets of items with some minimum specified confidence. One of the examples of the association rules is that 90% of transactions purchasing bread, and butter purchase milk as well. That is, the association rules those appearances are greater than the specified confidence are regarded as frequent patterns. In the rest of this paper, the confidence is called "support".

There are many proposals for frequent pattern mining; however, this paper specifically picks up two algorithm; Apriori algorithm [45], and FP-growth algorithm [46]. Apriori algorithm is the most basic, but standard algorithm proposed by Agrawal et al.. Many frequent mining algorithms are also developed based on Apriori algorithm. FP-growth algorithm is another algorithm, and is considered as more scalable, and faster than Apriori algorithm. The rest of this section briefly introduces summary of each algorithm.

3.2. Apriori

Figure 4 gives Apriori algorithm, and there are several assumptions as follows. The items in each transaction are sorted in alphabetical order. We call the number of items in an itemset its “size”, and call an itemset of size k a $k\text{-}itemset$. Items in an itemset are in alphabetical order again. We call an itemset with minimum support a "large itemset".

The first pass of Apriori algorithm counts item occurrences in order to determine the $large$ 1-$itemsets$ ((1) in Figure 4). A subsequent pass consists of two phases. Suppose we are in $k\text{-}th$ pass. First, the $large\ itemsets\ L_{k-1}$ found in the $(k-1)\text{-}th$ pass are used for generating the candidate itemsets C_k (potentially large itemsets), using the $apriorigen$ function, which we describe later in this section ((2) in Figure 4). Next, the database is scanned, and the support of candidates in C_k is counted ((3) and (4) in Figure 4). These two phrases prepare for the $large\ itemsets\ L_k$ ((5) in Figure 4), and the subsequent pass is repeated until L_k becomes empty. The $apriorigen$ function takes L_{k-1}, and returns a superset of the set of all the $large\ k\text{-}itemsets$. The $apriorigen$ function consists of the join step (Figure 5), and the prune step (Figure 6).

The distinctive part of Apriori algorithm is the $apriorigen$ function. The $apriorigen$ function reduces the size of candidate sets, and this reduction contributes for speed-up of the mining. The $apriorigen$ function is designed based on Apriori heuristic [45]; if any length k pattern is not frequent in the database, its length $k+1$ super-pattern can never be frequent.

(1) $L_1 = large\ 1\text{-}itemsets$
for $k \geq 2$ and $L_{k-1} \neq \emptyset$ **do**
  (2) $C_k = apriorigen(L_{k-1})$
  for all transactions $t \in database$ **do**
    (3) $C_i = subset(C_k, t)$
    for all candidates $c \in C_i$ **do**
      (4) c.count++
    end for
  end for
  (5) $L_k = \{c \in C_k | \, c.count \geq minsup\}$
end for
(6) $Answer = \cup_k L_k$

Figure 4. The pseudo-code for Apriori algorithm.

for all itemsets $c \in C_k$ **do**
  for all $(k-1)$-$subsets\ s$ of c **do**
    if $s \notin L_{k-1}$ **then**
      delete c **from** C_k
    end if
  end if
end if

Figure 6. The pseudo-code for the prune step of apriorigen function.

insert into C_k
select $p.item_1, p.item_2, \ldots, p.item_{k-1}, q.item_{k-1}$
from $L_{k-1\,p}, L_{k-1\,q}$
where $p.item_1 = q.item_1, \ldots, p.item_{k-2} = q.item_{k-1}, p.item_{k-1} < q.item_{k-1}$

Figure 5. The pseudo-code for the join step of apriorigen function.

3.3. FP-growth

FP-growth algorithm is proposed by Han et al. [46], and they point out Apriori algorithm suffers from the cost for handling huge candidate sets, or repeated scanning database with prolific frequent patterns, long patterns, or quite low support. Han et al. advocated the bottleneck exists in candidate set generation, and proposed frequent pattern tree (FP-tree). Here, we see the construction process of FP-tree, utilizing the example shown on Table 1. In this example, we set the support as "three times", instead of appearance ratio in order to simplify the explanation. The FP-tree developed from this example is shown in Figure 7.

Table 1. The input example for FP-growth algorithm.

TID	item series	frequent items (for reference)
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

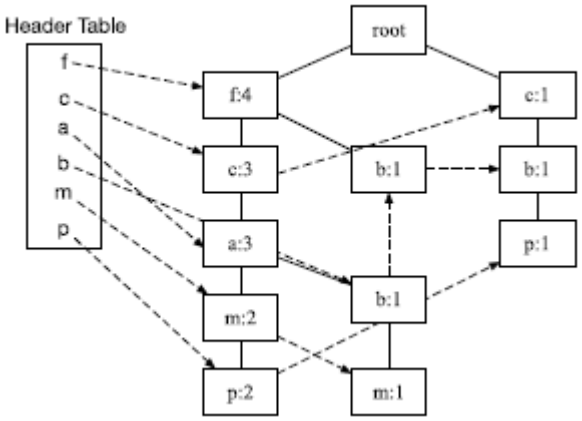

Figure 7. An example of FP-tree.

First, the first data scan is conducted for generating the list of frequent items. The obtained list looks like as follows.

$$< (\text{f}: 4), (\text{c}: 4), (\text{a}: 3), (\text{b}: 3), (\text{m}: 3), (\text{p}: 3) >$$

Here, $(I: c)$ represents item I appears c times in the database. Notice the list is ordered in frequency descending order. The ordering is important as each path of a tree follows this ordering. The rightmost column on Table 1 represents the frequent items in each transaction.

Next, we start generating a tree with putting the root of a tree, labeled with "null". The second data scan is conducted for generating the FP-tree. The scan of the first transaction constructs the first branch of the tree:

$$< (\text{f}: 1), (\text{c}: 1), (\text{a}: 1), (\text{m}: 1), (\text{p}: 1) >$$

Notice the frequent items in the transaction are ordered in the same order to the list of frequent items. For the second transaction, as its frequent item list $< f, c, a, b, m >$ shares a common prefix $< f, c, a >$ with the existing path $< f, c, a, m, p >$, we simply increment the counts of the common prefix, and create a new node (b: 1) as a child of (a: 2). Another new node (m: 1) is created as a child of (b: 1). For the third transaction, its frequent item list $< f, b >$ shares only $< f >$ with f-prefix subtree. Therefore, we increment the count of node $< f >$, and create a new node (b: 1) as a child of this node (f: 3). The scan for the fourth transaction introduces a completely new branch as follows, because no node is shared with existing prefix trees.

$$< (c: 1), (b: 1), (p: 1) >$$

The frequent items in the last transaction perfectly overlaps with the frequent items in the first transaction. Therefore, we simply increment the counts in the path $< f, c, a, m, p >$. Figure 7 is a complete FP-tree developed from this example. In the left part of Figure 7, there is an item header table, and this table contains head of node-links. This header table is utilized for tree traversal when extracting all the frequent patterns.

4. EXPERIMENTS

4.1. Setup

The purpose of this paper is to interpret the relationships between the characteristics of the input data, and behaviors of frequent pattern mining algorithms such as Apriori algorithm, and FP-growth algorithm. This section explains the methodology for the experiments.

We utilize the implementations of Apriori algorithm [47], and FP-growth algorithm [48] in C, which are distributed by Borgelt. We compiled, and run the programs as they are for fairness. No optimization is applied. In order to observe changes of the behaviors of these programs, three kinds of data are prepared as input. The overall feature of each data is summarized as Table 2. The details of each data are as follows.

Table 2. Summary of the input data.

	number of transactions (A)	number of distinct items (B)	(B) / (A)
census	48,842	135	0.0028
papertitle	2,104,240	925,151	0.440
shoppers	26,496,646	836	0.000032

Table 3. Number of found sets.

support [%]	census	papertitle	shoppers
1.25	134,780	49	426
2.5	49,648	21	69
5.0	15,928	8	12
10.0	4,415	3	1
20.0	904	0	0
40.0	117	0	-

- *census*: This is national population census data, and the data are distributed with Apriori algorithm implementation [47], and FP-growth implementation [48] by Borgelt as test data. As shown on Table 2, the ratio of the number of distinct items to the number of transactions is moderate among the three input data, and the ratio is 0.28%. We also see the number of transactions in *census* is the smallest.

- *papertitle*: This is the data set for KDD Cup 2013 Author-Paper Identification Challenge (Track 1), and available at Kaggle contest site [49]. We extract paper titles from Paper.csv,

and create the list of titles. As shown on Table 2, the ratio of the number of distinct items to the number of transactions is extremely high compared to the other two data sets, and the ratio is 44.0%. We also see the number of transactions is moderately high.

- *shoppers*: This is the data set for Acquire Valued Shoppers Challenge, and available at Kaggle contest site as well [50]. For the experiment, we extract categories of purchased items from each transaction data in transactions.csv, and create the list of purchased item category. As shown on Table 2, the ratio of the number of distinct items to the number of transactions is extremely small compared to the other two data sets, and the ratio is 0.0032%. We also see the number of transactions is huge, and the biggest

From the viewpoint of the application, the meaning of frequent mining over *census* is finding out typical portraits of the nation. Similarly, frequent mining over *papertitle* derives popular sets of words (not phrases) in paper titles, and frequent mining over *shoppers* extracts frequent combinations of item categories in the purchased items.

Table 3 shows the number of found sets for each data set when support increases from 1.25% to 40.0%. Both Apriori algorithm, and FP-growth algorithm found the same number of sets for the same support. In case of shoppers with support of 40.0%, both Apriori algorithm, and FP-growth algorithm could not list the candidate items, and quit the executions; therefore, the experiment could not gather meaningful data.

4.2. Results

Figure 8 compares the execution times in seconds of Apriori algorithm, and FP-growth algorithm for each support with *census* data. Figure 9 shows the details of the execution times of Apriori algorithm for each support. Similarly, Figure 10 shows the details of the execution times of FP-growth algorithm for each support. Figure 11 shows the frequency graph of the found pattern length for each support. Here, both algorithms found the same patterns; the frequency shown in this graph is common to both Apriori algorithm, and FP-growth algorithm.

This census input data has the characteristics as follows.

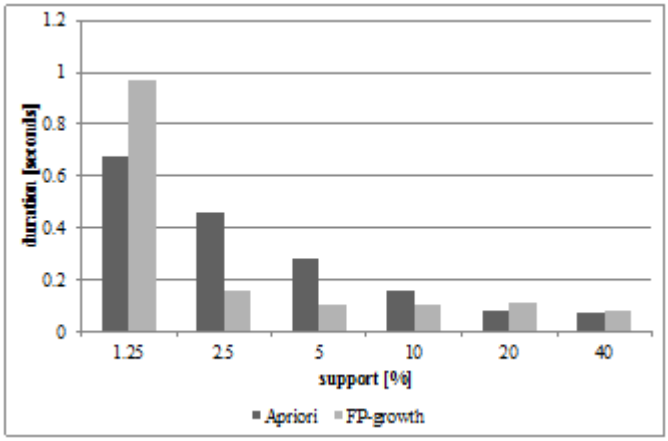

Figure 8. Comparison of durations between Apriori and FP-growth (*census*).

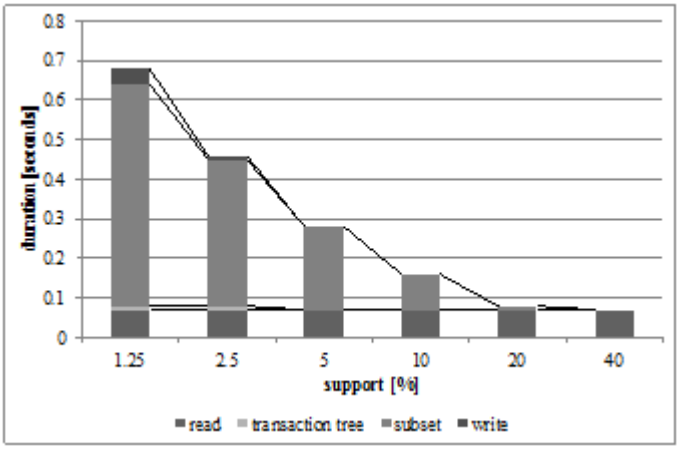

Figure 9. Details of Apriori (*census*).

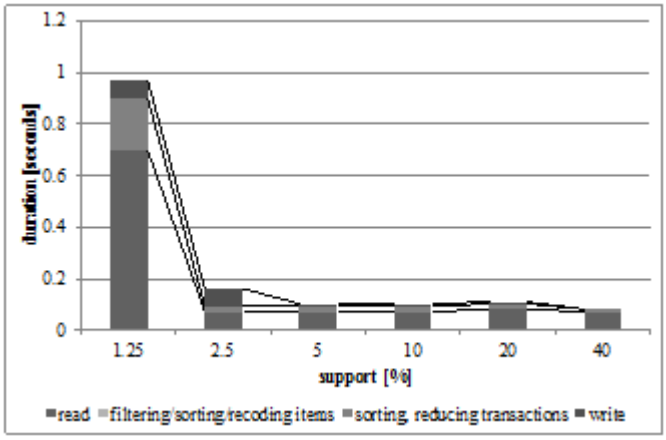

Figure 10. Details of FP-growth (*census*).

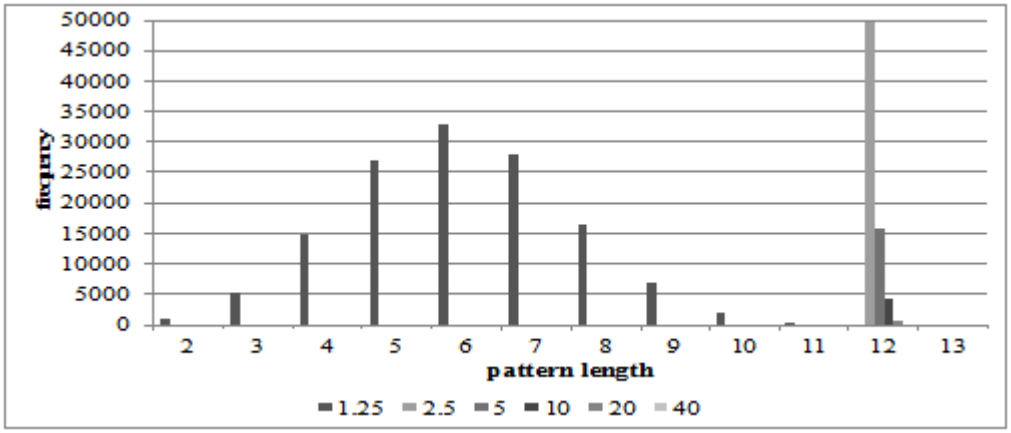

Figure 11. Pattern length (*census*).

- The number of transactions is the smallest among the three cases (Table 2).
- The ratio of the number of transactions, and distinct items is moderate (Table 2).
- The number of found sets is extremely high, and the highest (Table 3).

One point of view is that FP-growth is expected to be superior to Apriori for *census* as the number of found sets is the highest among all the three data sets. The huge number of found sets will cause the huge number of candidate sets, and Apriori will cost more on scanning the large database of the candidate sets. The other point of view is that Apriori is expected to be superior to FP-growth for *census* as the number of transactions is the smallest among all the three. The moderate number of the ratio of the number of transactions, and distinct items with huge number of found items implies the moderate length of the found patterns. This characteristics does not backs up either Apriori, or FP-growth. Separate from the characteristics of *census*, the lower support generally results in more found sets. This general assumption implicates that FP-growth will be superior to Apriori in the lower support, and the overall execution time will increase according to the reduction of the support.

Figure 8 shows that FP-growth overcomes Apriori only when the support is 2.5%, 5%, and 10%, and that Apriori is superior to FP-growth in 1.25%, 20%, and 40%. As general observation, Apriori increases its execution time according to the reduction of support value. FP-growth also similar tendency in a big picture, however, the execution times when the support is greater than 2.5% are almost in the similar range. Additionally, the biggest difference from our expectation is that Apriori is faster than FP-growth when the support is 2.5%, which is the smallest.

Figure 9, and Figure 10 indicate the reason why FP-growth is not clearly superior to Apriori with *census* data set. Figure 9 shows the breakdown of the execution times with Apriori, and illustrates that the overhead stays almost constant, and the main body of the program increases exponentially as the support value decreases. Contrarily, in FP-growth, the overhead occupies the major part of the execution times when the support is 2.5%, 5%, 10%, 20%, or 40%. Only when the support is 1.25%, the ratio of the main body of the algorithm clearly increases in the whole execution time. In fact, FP-growth is always faster than Apriori, or almost similar if we focus only on the durations for the main part of the algorithm. The observation for *census* indicates that the overhead of FP-growth is too huge for overcoming Apriori with this size of input data.

Figure 11 shows that the length of the found set is mostly centering on 12 when the support is 2.5%, 5%, 10%, 20%, and 40%. The figure also illustrates that the length of the found set is scattered from 2 to 12 when the support is 1.25%, and that the average length of the found sets is supposed to be shorter than the other cases when the support is 1.25%. As a conclusion of census case study, the size of data is too small, and the length of the found pattern is too short for providing benefits for FP-growth to overcome Apriori, even with the huge number of found sets.

Figure 12 compares the execution times in seconds of Apriori algorithm, and FP-growth algorithm for each support with papertitle data. Figure 13 shows the details of the execution times of Apriori algorithm for each support. Similarly, Figure 14 shows the details of the execution times of FP-growth algorithm for each support. Figure 15 shows the frequency graph of the found pattern length for each support. Again, both algorithms found the same patterns; the frequency shown in this graph is common to both Apriori algorithm, and FP-growth algorithm.

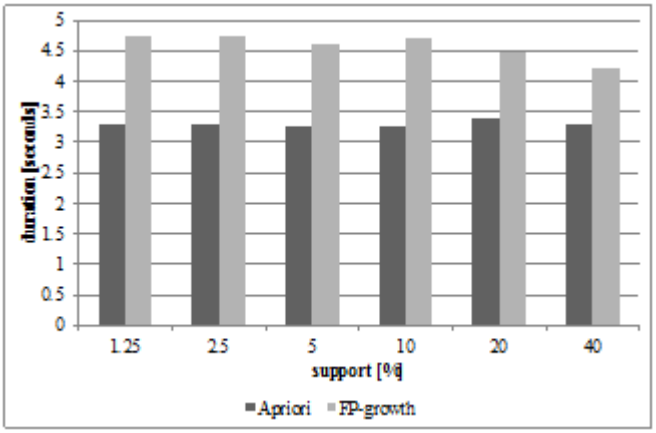

Figure 12. Comparison of durations between Apriori and FP-growth (*papertitle*).

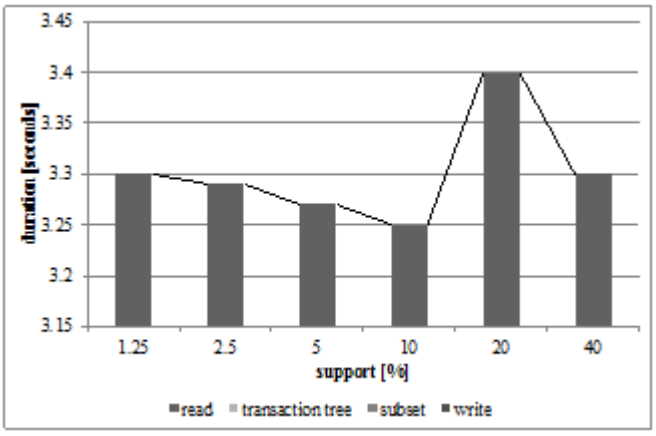

Figure 13. Details of Apriori (*papertitle*).

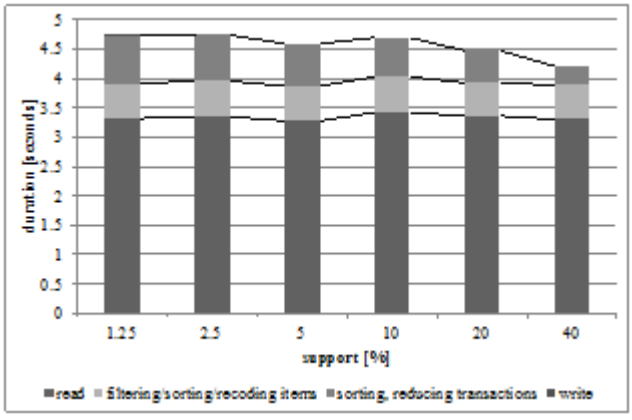

Figure 14. Details of FP-growth (*papertitle*).

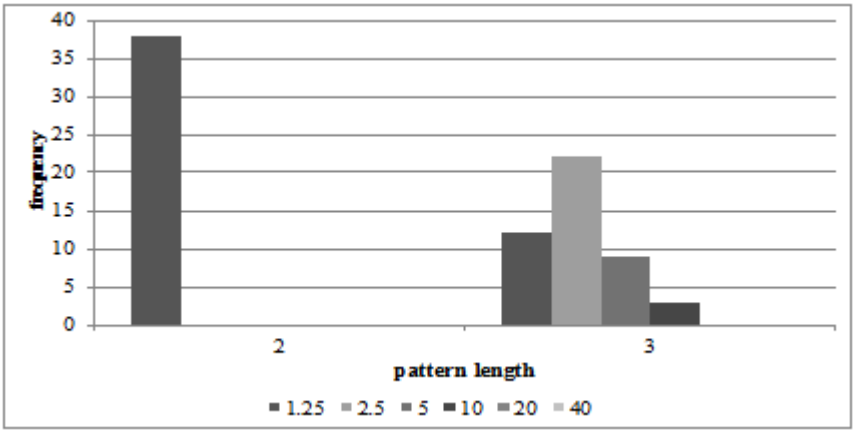

Figure 15. Pattern length (*papertitle*).

This *papertitle* input data has the characteristics as follows.

- The number of transactions is moderate (Table 2).
- The ratio of the number of transactions, and distinct items is the highest (Table 2).
- The number of found sets is the smallest (Table 3).

For *papertitle* case, one point of view is that Apriori is expected to be superior to FP-growth as the number of found sets is the smallest among all the three data sets. The smallest number of found sets will results in the small number of candidate sets, and scanning database by Apriori is expected to be faster than the overhead of construction, and scan of FP-tree by FP-growth. The other point of view is that FP-growth is expected to be superior to Apriori as the number of transactions is moderate, and the larger than *census* data. The biggest number of the ratio of the number of transactions, and distinct items with the smallest number of found items implies that the smaller length of the found patterns. This characteristics backs up either Apriori as well. Separate from the characteristics of *papertitle*, again, the lower support generally results in more found sets. This general assumption implicates that FP-growth will be superior to Apriori in the lower support, and the overall execution time will increase according to the reduction of the support.

Figure 12 shows that the execution times of Apriori, and FP-growth are almost constant, and that Apriori is always superior to FP-growth. In practice, the execution time of Apriori decreases slightly as the support decreases, and the execution time of FP-growth increases slightly as the support decreases. The difference of Apriori, and FP-growth, however, conclusive.

Figure 13, and Figure 14 illustrate the breakdowns of execution times for each algorithm, and these figures also explain why Apriori is clearly superior to FP-growth for *papertitle* data. Figure 13 illustrates that the main part of Apriori is almost ignorable, and the overhead occupies almost all the part of the execution time. The situation is quite similar to FP-growth. Figure 14 shows that the overhead of FP-growth constitutes the substantial part of the execution time, and the main body of FP-growth is extremely small in the overall execution times. When we focus on the actual numbers of main parts of the algorithms in the execution times, the numbers clearly

indicate that the main body of FP-growth consumes more time than the main body of Apriori. The observation here concludes not only FP-tree construction, but also FP-tree scan costs more than the candidate generation by Apriori for *papertitle* data set.

Figure 15 also explains the reason why Apriori always overcomes FP-growth with *papertitle* data set. The pattern length is always two, or three, which is extremely short. As the conclusion of *papertitle* case study, no matter how the number of transactions is large, the number of found set is too small, and the pattern is too short for FP-growth to perform effectively than Apriori.

Figure 16 compares the execution times in seconds of Apriori algorithm, and FP-growth algorithm for each support with *shoppers* data. Figure 17 shows the details of the execution times of Apriori algorithm for each support. Similarly, Figure 18 shows the details of the execution times of FP-growth algorithm for each support. Figure 19 shows the frequency graph of the found pattern length for each support. Again, both algorithms found the same patterns; the frequency shown in this graph is common to both Apriori algorithm, and FP-growth algorithm.

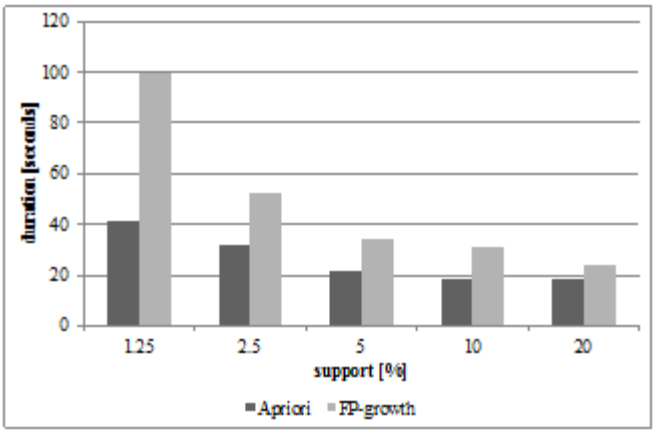

Figure 16. Comparison of durations between Apriori and FP-growth (*shoppers*).

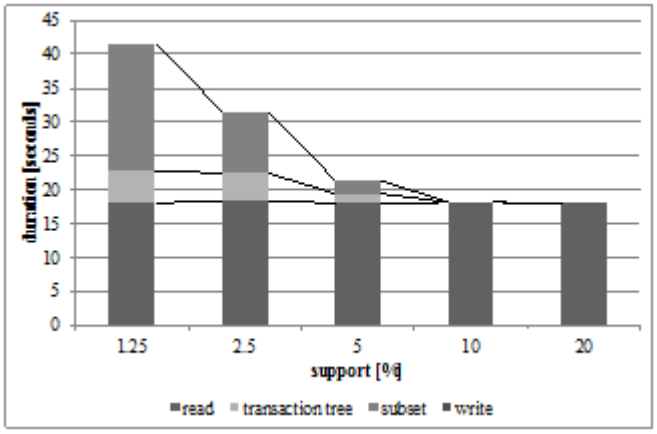

Figure 17. Details of Apriori (*shoppers*).

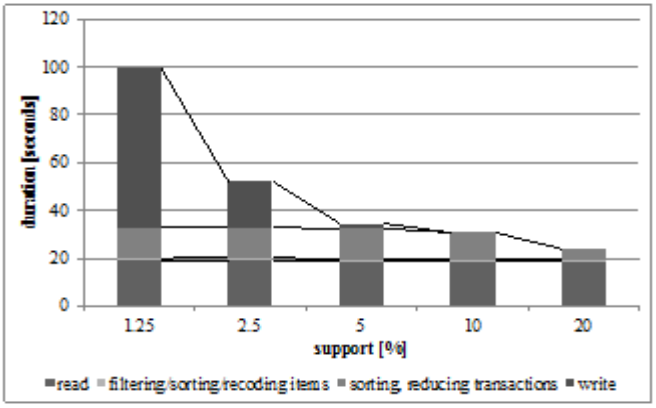

Figure 18. Details of FP-growth (*shoppers*).

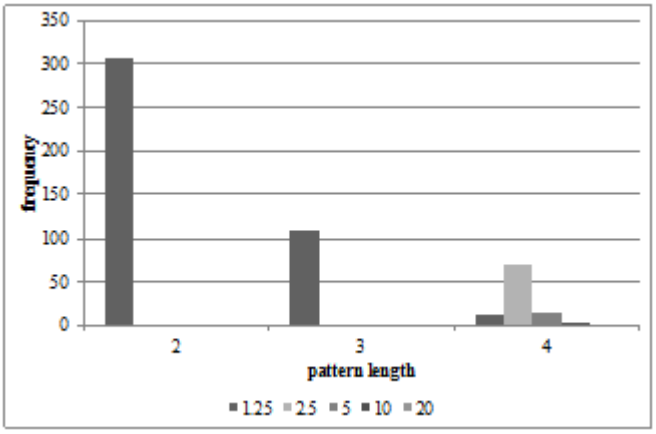

Figure 19. Pattern length (*shoppers*).

This *shoppers* input data has the characteristics as follows.

- The number of transactions is huge, and the biggest (Table 2).
- The ratio of the number of transactions, and distinct items is the smallest (Table 2).
- The number of found sets is moderate (Table 3).

For *shoppers* case, one point of view is that FP-growth is expected to be superior to Apriori as the number of transactions is huge, and larger than *census*, or *papertitle*. The cost for the database scan should be critical when the number of transaction is extremely large. Additionally, Apriori will scan the data repeatedly, the damage for the database scan, therefore, should be serious. The other point of view is that Apriori does not suffer from the database scan so much as the number of found sets for *shoppers* is moderate. The moderate number of found sets implicates that there are not so many candidates for such a huge number of transactions, and which means that the construction, and scanning cost of FP-tree of FP-growth is possible to damage the performance of FP-growth. One more thing is that the ratio of the number of transactions, and distinct items is the smallest among the three data sets, even though the number of found sets is moderate. This fact

suggests that the average length of the found patterns might be moderate. As only the small number of distinct items consist of transactions of *shoppers*, the average length of the found patterns should be long to some extent in order to produce moderate number of found sets. This point, however, does not determine whether Apriori, or FP-growth is superior to the other. Separate from the characteristics of *shoppers*, again, the lower support generally results in more found sets. This general assumption implicates that FP-growth will be superior to Apriori in the lower support, and the overall execution time will increase according to the reduction of the support.

Figure 16 shows that both Apriori, and FP-growth tend to increase the execution time as support decreases. The execution time of Apriori with support of 20% is slightly larger than the execution time with support of 10%, however, this small exception does not change the overall characteristics. The figure also illustrates that Apriori is always superior to FP-growth regardless of support values.

Figure 17, and Figure 18 illustrate the breakdowns of execution times for each algorithm. Similar to *papertitle* case, both of the figures illustrate that the overhead of the execution occupies substantial part of the whole execution times, and the main body of the algorithms is quite small. When we focus on the actual numbers of main parts of the algorithms in the execution times, the main part of FP-growth apparently consumes more time than the main part of Apriori. FP-growth, again, pays expensive cost for both FP-tree construction, and FP-tree scan, and resulted in the worse performance than Apriori.

Figure 19 plots the length of found sets, and explains the reason why Apriori is always superior to FP-growth in *shoppers* case again. The pattern length lies between two and four, and which is short. The figure, and the fact that the moderate number of found sets imply the number of candidates is small. Table 3 also reminds us of the fact that both Apriori and FP-growth quit the job because there is no candidate for *shoppers* with support of 40.0%. These considerations conclude that FP-growth could not exert its advantages for the extremely small number of candidates even with the huge, and the biggest number of transactions.

4.3. Discussion

The overall results clearly indicate that simple one measure such as the total number of transactions, or the total number of distinct items does not contribute for algorithm selection. Actually, Apriori algorithm is considered to be hard to scale, however, Apriori algorithm was faster than FP-growth algorithm in *papertitle* case, and *shoppers* case, even though these two cases handle huge number of transactions compared to *census* case. Apparently, the characteristics of the input data should be determined in order to select appropriate algorithm.

As the first step of the characterization of the input data, we focused on the number of transactions, the number of distinct items, and the ratio of these two numbers. The results showed that these three parameters do contribute for the characterization; however, we need a few more parameters for the better characterization as we saw in the previous section. Of course, the length of the found sets is also the important parameter. The problem is, however, those parameters such as the length of the found sets, or distributions of data, are hard to be anticipated before the target data are processed actually. We need to find a good way to parameterize these characteristics.

5. CONCLUSIONS

Big data quickly comes under the spotlight in recent years, and increases its importance both in socially, and scientifically. Among big data applications, this paper specifically focused on frequent mining, and gave the first step to interpret the relationships between the characteristics of the input data, and behaviors of signature algorithms for frequent pattern mining. The experiments, and discussions backed up that the characteristics of the input data have certain impact on both the performance, and the selection of the algorithm to be utilized. This paper also picked up some parameters for characterizing the input data, and showed these parameters are meaningful, however, revealed some items to be investigated for the better characterization of the input data as well.

REFERENCES

[1] B. Babcock, et al., "Maintaining variance and k-medians over data stream windows," Proc. the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'03), 2003, pp. 234–243.

[2] N. Tatbul, et al., "Load shedding in a data stream manager," Proc. the 29th International Conference on Very Large Data Bases (VLDB'03), 2003, Volume 29, pp. 309–320.

[3] B. Babcock, et al., "Models and issues in data stream systems," Proc. the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'02), 2002, pp. 1–16.

[4] A. C. Gilbert, et al., "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," Proc. of the 27th International Conference on Very Large Data Bases (VLDB'01), 2001, pp. 79–88.

[5] C. C. Aggarwal, et al., "A framework for clustering evolving data streams," Proc. the 29th International Conference on Very Large Data Bases (VLDB'03), Volume 29, 2003, pp. 81–92.

[6] C. C. Aggarwal, et al., "A framework for projected clustering of high dimensional data streams," Proc. the Thirtieth International Conference on Very Large Data Bases (VLDB'04), Volume 30, 2004, pp. 852–863.

[7] C. C. Aggarwal, et al., "On demand classification of data streams," Proc. the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04), 2004, pp. 503–508.

[8] G. Cormode and S. Muthukrishnan, "What's hot and what's not: Tracking most frequent items dynamically," ACM Trans. Database Syst., vol. 30, no. 1, pp. 249–278, Mar. 2005.

[9] G. Dong, et al., "Online mining of changes from data streams: Research problems and preliminary results," Proc. the 2003 ACM SIGMOD Workshop on Management and Processing of Data Streams, 2003.

[10] S. Guha, et al., "Clustering data streams: Theory and practice," IEEE Trans. on Knowl. and Data Eng., vol. 15, no. 3, pp. 515–528, Mar. 2003.

[11] M. Charikar, et al., "Better streaming algorithms for clustering problems," Proc. the Thirty-fifth Annual ACM Symposium on Theory of Computing (STOC '03), 2003, pp. 30–39.

[12] P. Domingos and G. Hulten, "Mining high-speed data streams," Proc. the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00), 2000, pp. 71–80.

[13] G. Hulten, et al., "Mining time-changing data streams," Proc. the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'01), 2001, pp. 97–106.

[14] L. O'Callaghan, et al., "Streaming-data algorithms for high-quality clustering," Proc. the 18th International Conference on Data Engineering (ICDE'02), 2002, pp. 685–.

[15] C. Ordonez, "Clustering binary data streams with k-means," Proc. the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'03), 2003, pp. 12–19.

[16] E. Keogh and J. Lin, "Clustering of time-series subsequences is meaningless: Implications for previous and future research," Knowl. Inf. Syst., vol. 8, no. 2, pp. 154–177, Aug. 2005.

[17] H. Wang, et al., "Mining concept-drifting data streams using ensemble classifiers," Proc. the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03), 2003, pp. 226–235.
[18] V. Ganti, et al., "Mining data streams under block evolution," SIGKDD Explor. Newsl., vol. 3, no. 2, pp. 1–10, Jan. 2002.
[19] S. Papadimitriou, et al., "Adaptive, hands-off stream mining," Proc. the 29th International Conference on Very Large Data Bases (VLDB'03), Volume 29, 2003, pp. 560–571.
[20] M. Last, "Online classification of nonstationary data streams," Intell. Data Anal., vol. 6, no. 2, pp. 129–147, Apr. 2002.
[21] Q. Ding, et al., "Decision tree classification of spatial data streams using peano count trees," Proc. the 2002 ACM Symposium on Applied Computing (SAC'02), 2002, pp. 413–417.
[22] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," Proc. the 28th International Conference on Very Large Data Bases (VLDB'02), 2002, pp. 346–357.
[23] P. Indyk, et al., "Identifying representative trends in massive time series data sets using sketches," Proc. the 26th International Conference on Very Large Data Bases (VLDB '00), 2000, pp. 363–372.
[24] Y. Zhu and D. Shasha, "Statstream: Statistical monitoring of thousands of data streams in real time," Proc. the 28th International Conference on Very Large Data Bases (VLDB'02), 2002, pp. 358–369.
[25] J. Lin, et al., "A symbolic representation of time series, with implications for streaming algorithms," Proc. the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD '03), 2003, pp. 2–11.
[26] D. Turaga, et al., "Resource management for networked classifiers in distributed stream mining systems," Proc. the Sixth International Conference on Data Mining (ICDM2006), 2006, pp. 1102–1107.
[27] D. S. Turaga, et al., "Configuring topologies of distributed semantic concept classifiers for continuous multimedia stream processing," Proc. the 16th ACM International Conference on Multimedia (MM'08), 2008, pp. 289–298.
[28] B. Thuraisingham, et al., "Dependable real-time data mining," Proc. the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC2005), 2005, pp. 158–165.
[29] N. K. Govindaraju, et al., "Fast and approximate stream mining of quantiles and frequencies using graphics processors," Proc. the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05), 2005, pp. 611–622.
[30] K. Chen and L. Liu, "He-tree: a framework for detecting changes in clustering structure for categorical data streams," The VLDB Journal, vol. 18, no. 6, pp. 1241–1260, 2009.
[31] M. M. Gaber, et al., "Mining data streams: A review," SIGMOD Rec., vol. 34, no. 2, pp. 18–26, Jun. 2005.
[32] I. Raicu, et al., "The quest for scalable support of data-intensive workloads in distributed systems," Proc. the 18th ACM International Symposium on High Performance Distributed Computing (HPDC'09), 2009, pp. 207–216.
[33] J. Dongarra, et al., LINPACK Users Guide, SIAM, 1979.
[34] S. P. E. Corporation. Spec benchmarks. http://www.spec.org/benchmarks.html
[35] S. Akioka, et al., "Data access pattern analysis on stream mining algorithms for cloud computation," in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010), Volume 2, 2010, pp. 36–42.
[36] S. Akioka, "Task graphs for stream mining algorithms," in Proc. The First International Workshop on Big Dynamic Distributed Data (BD32013), 2013, pp. 55–60.
[37] K.-M. Schneider, "A comparison of event models for naive bayes anti-spam e-mail filtering," Proc. the Tenth Conference on European Chapter of the Association for Computational Linguistics (EACL'03), Volume 1, 2003, pp. 307–314.
[38] R. P. Dick, et al., "Tgff: Task graphs for free," Proc. the 6th International Workshop on Hardware/Software Codesign (CODES/CASHE'98), 1998, pp. 97–101.
[39] TGFF. http://ziyang.eecs.umich.edu/_dickrp/tgff

[40] D. Cordeiro, et al., "Random graph generation for scheduling simulations," Proc. the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools'10). 2010, pp. 60:1–60:10.
[41] TGG, task graph generator. http://taskgraphgen.sourceforge.net
[42] STG, standard task graph set. http://www.kasahara.elec.waseda.ac.jp/schedule/index.html
[43] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," Journal of Scheduling, vol. 5, no. 5, pp. 379–394, 2002.
[44] R. Agrawal, et al., "Mining association rules between sets of items in large databases," Proc. the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93), 1993, pp. 207–216.
[45] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," Proc. the 20th International Conference on Very Large Data Bases (VLDB'94). 1994, pp. 487–499.
[46] J. Han, et al., "Mining frequent patterns without candidate generation," Proc. the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00), 2000, pp. 1–12.
[47] C. Borgelt. Apriori - association rule induction/frequent item set mining. http://www.borgelt.net//apriori.html
[48] C. Borgelt. FPgrowth - frequent item set mining. http://www.borgelt.net//fpgrowth.html
[49] kaggle. Kdd cup 2013 - author-paper identification challenge (track 1). https://www.kaggle.com/c/kdd-cup-2013-author-paper-identification-challenge
[50] kaggle. Acquire valued shoppers challenge. https://www.kaggle.com/c/acquire-valued-shoppers-challenge

AUTHORS

Sayaka Akioka is an associate professor at Meiji University, Tokyo, Japan. She received her B.E. in Computer Science, Master of Computer Science, and Ph.D. in Computer Science from Waseda University, Tokyo, Japan in 1999, 2001, and 2004 respectively. Previously, she was a researcher at Waseda University, Tokyo, Japan from 2010 to 2013, an assistant professor at The University of Electro-Communications, Tokyo, Japan from 2007 to 2010, a post-doctoral researcher at The Pennsylvania State University from 2005 to 2007, a project researcher at National Institute of Informatics, Tokyo, Japan from 2004 to 2005, and a research associate at Waseda University, Tokyo, Japan from 2001 to 2004. Her research interests include cloud computing, parallel and distributed computing, and data mining.