FastPM: An Approach to Pattern Matching via Distributed Stream Processing

Dingyu Yang[a], Jianmei Guo[b], Zhi-Jie Wang[c,d], Yuan Wang[e], Jingsong Zhang[f], Liang Hu[g], Jian Yin[c], Jian Cao[h]

[a]*School of Electronics and Information, Shanghai Dian Ji University, Shanghai, China*
[b]*Alibaba Group, Hangzhou, China*
[c]*School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China*
[d]*Guangdong Key Laboratory of Big Data Analysis and Processing, Sun Yat-Sen University, Guangzhou, China*
[e]*Department of Industrial System Engineering, National University of Singapore, Singapore*
[f]*Shanghai Institutes for Biological Sciences, Chinese Academy of Sciences, Shanghai, China*
[g]*University of Technology, Sydney & University of Shanghai for Science and Technology, Shanghai, China*
[h]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China*

Abstract

Pattern matching over big data is gaining momentum in recent years. Many real-time applications are involved in pattern matching over a high volume of data to discover potential tendencies, in which real-time responding and concurrent processing are the key performance metrics. However, it is challenging to efficiently matching over live streaming data due to: (i) the high volume of massive data, (ii) the real-time response requirement, and (iii) the concurrent matching queries. To address these challenges, we introduce a pattern model by appending a timestamp set to reduce the number of repeated patterns. We propose FastPM, a distributed stream processing framework to address the high speed real-time data. Our framework combines synchronous and asynchronous mechanisms to deal with multiple matching queries simultaneously, and it develops multiple techniques to enhance the efficiency of pattern matching. We implemented FastPM and evaluated its performance on billions of real-world web-click data. Our empirical results demonstrate the effectiveness of FastPM on matching queries and pattern updates. On average, FastPM responds to a matching query in 0.2 second and to an update request in 0.03 second. Furthermore, FastPM is able to support $5,000$ matching queries simultaneously and the average query latency is 1.3 seconds.

Keywords: Pattern matching, distributed stream processing, dynamic partition, pattern update

1. Introduction

Pattern matching is a hot topic and has wide applications in many domains such as system monitoring, image processing, social networks, and internet of things [1, 2, 3, 4, 5, 6, 7]. Given a pattern sequence, pattern matching is to find sequences that are similar to the given one [8, 9]. In the past, many papers investigated pattern matching problem over *static data*: some researchers developed special data structures and indices to discover useful patterns efficiently (e.g., [8, 9, 10, 11, 12, 13]), while others attempted to improve the efficiency of pattern matching via distributed frameworks (e.g., [14, 15, 16, 17, 18, 19]).

Nowadays, in many real-time applications, such as *web click analysis*, *financial trend analysis*, *social sentiment analysis*, the data is generated quickly and is potentially unbounded in size. Several literatures have been dedicated to find similar sequences or patterns over *live data* (e.g., [20, 21, 22, 23, 24, 25]). However, most of them process pattern matching queries on a single machine, as shown in Figure 1. The matching efficiency of these methods is significantly affected because of executing a single matching query at one time. In practice, it is challenging to support “high-performance” pattern matching due to (i) the high volume of massive data and high velocity of event streams; (ii) the real-time response requirement; and (iii) the concurrent pattern matching queries or tasks.

Email addresses: yangdy@sdju.edu.cn (Dingyu Yang), jianmei.gjm@alibaba-inc.com (Jianmei Guo), wangzhij5@mail.sysu.edu.cn (Zhi-Jie Wang), iseway@nus.edu.sg (Yuan Wang), jasun@dmbio.info (Jingsong Zhang), rainmilk@gmail.com (Liang Hu), issjyin@mail.sysu.edu.cn (Jian Yin), cao-jian@sjtu.edu.cn (Jian Cao)

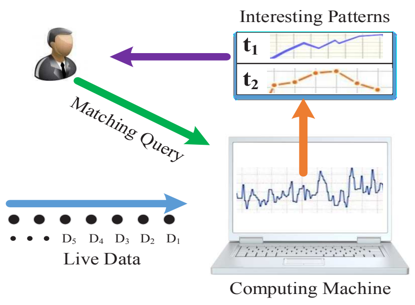


Figure 1: An example of pattern matching over live data. One inputs a matching query, and the computing machine returns matched patterns.

A few researches (e.g., [14, 16, 17]) adopt distributed pattern matching by continuously loading the data in memory. They can handle the high volume of massive data, but they are not well suitable for matching patterns in real-time, since they have to reload the data to find the patterns when the new data is received. There are also some approaches (e.g., [10, 12, 13]) that use various indexing techniques to improve performance. Yet, one can easily find that, the data in real-time applications are generated continuously, and so the indices need to be updated frequently to keep up-to-date; thereby, the maintaining cost (for such updates) is considerable in a high speed stream, which cannot meet the real-time requirement. In addition, some methods (e.g., [20, 22, 26, 21]) are able to achieve real-time pattern matching in streaming environments, however, they are difficult to execute the concurrent pattern matching tasks since they are developed based on the model of "one query at one time".

To achieve the real-time pattern matching over a high volume of live data, we design and implement a framework, called *FastPM*, which enables distributed stream processing for real-time pattern matching. FastPM defines a new pattern model, by appending a list of *timestamps* to record the occurrence time of a certain pattern, and thus in the new model the patterns with the same data values are wrapped into a single pattern with different timestamps. That means if one pattern is received and it has occurred before, we can append it in the old pattern by updating the timestamp instead of constructing a new pattern. Thus, it is able to reduce the number of patterns in memory and improve the efficiency of pattern matching. The timestamp update is processed incrementally and hence it allows us to response real-time queries more conveniently, compared to traditional pattern models (e.g., [11, 27, 13, 28]).

To improve the efficiency of pattern matching, the architecture of *FastPM* consists of three components: (i) the history pattern discovery component, which extracts patterns from history data and deploys these patterns in our distributed system; (ii) the real-time pattern matching component, which is the processing core of FastPM; and (iii) the real-time pattern update component, which is used to process the stream data in an *online* fashion (i.e., new patterns are updated dynamically).

To facilitate the subsequent pattern matching tasks, the first component history pattern discovery integrates three strategies (i.e., hierarchical pattern extraction, cluster-based partition, and frequency-aware deploy). These strategies are developed to replace the random partition approach and the round-robin deploy method in traditional distributed processing systems, by which the performance degrades when the data distribution is non-uniform. Our proposed strategies collaboratively contribute to distributing different patterns according to their features and frequencies.

The second component real-time pattern matching combines asynchronous and synchronous mechanisms together to collaborate with different steps in a pipeline job. It allows us to flexibly deal with multiple matching tasks simultaneously. Particularly, we design two index optimization techniques (i.e., local query index, and branch-prune approach) that significantly improve the efficiency of distributed processing. The local query index can be viewed as the fundament of our optimization, while the branch-prune approach provides a further enhancement, which is designed as an optional choice for specific applications. We examine how to construct the local query index, and present the detailed algorithm for pattern matching query via this index.

The third component real-time pattern update fully exploits the information stored in the master node (of our distributed system), and the key parts in this component are the dynamic partition and the pattern state update. We propose targeted strategies for achieving these goals efficiently. Furthermore , we discuss the applications of FastPM. Particularly, we examine how our framework can be immediately used to handle real-time pattern prediction.

In summary, we made the following main contributions:

Table 1: Notations and Symbols

S	The data stream.
Q	The data sequence in a certain time window Δt.
t	The occurrence time of a pattern.
x_i	The ith data item in a sequence.
T	A set of occurrence time.
h	The number of data items in a pattern.
$NextP$	A set of patterns that have happened after pattern P.

- We define a new pattern model that reduces the storage space and facilitates pattern matching and update.
- We propose a framework, called FastPM, that allows us to achieve the real-time, distributed pattern matching over live data.
- We implement FastPM based on a distributed stream processing platform, and evaluate FastPM using the real-world streaming data including $5.4e+10$ (i.e., 54 *billion*) web click tuples. Our empirical results demonstrate that the effectiveness and efficiency of FastPM, is $8\times$ faster than the baseline method(*Random* partition and $Round-Robin$ deploy). Moreover, FastPM supports $5,000$ matching queries simultaneously.

The rest of the paper is organized as follows. We define basic concepts and problems in Section 2. The details of FastPM are presented in Sections $3 \sim 6$. We also discuss some applications in Section 7. Comprehensive experimental evaluation is conducted in Section 8. A comparison of our work and prior works is presented in Section 9, and we conclude this paper in Section 10.

2. Problem Formulation

In this section, we first describe some concepts and then state our problem formally. For ease of reference, the notations are summarized in Table 1.

Definition 2.1 (Data stream). *A data stream S is an unbounded sequence of data used to send and receive information, during the process of transmission [29].*

In a data stream, the data is continuously produced from data sources and varied with time, and so it is challenging to discover patterns from S. Usually, we use the *time-dependent sequence* to analyze and discover patterns from the data stream.

Definition 2.2 (Time-dependent sequence). *A time-dependent sequence Q is a n-length sequence of data items that appears in S within time window Δt. That is, $Q = \{x_1, x_2, \cdots, x_n\}$, where each x_i ($i \in [1,n]$) is a data item happened in Δt.*

Note that, in a time window Δt, the variety in certain successive data can be formed as a tendency, or an inclination, e.g., increment or decrement.

Definition 2.3 (Time-dependent pattern). *A* time-dependent pattern *P is a tuple $\langle pId, Data\{x_1,\cdots,x_h\}, T\{t_1,t_2,\cdots\}, NextP\{P',P'',\cdots\}\rangle$, where h is the pattern length that indicates the number of data items contained in the pattern, pId is the identity of the pattern, $Data$ is the value set of all data items, T denotes the timestamp set that records all the occurrence time for $Data$, and $NextP$ is a set of patterns that have happened after the pattern P.*

The *Time-dependent pattern* is proposed to extract patterns in a data sequence with timestamps, e.g., time series data. It not only keeps the tendency of the data, but also records the occurrence time of this pattern and its following pattern, which is significant for the applications such as web click analysis, financial trend prediction. The occurrence time is a significant feature in such applications. Figure 2 is an example which shows a trading pattern in stock market data. Figure 2(a) and 2(b) are the prices of one stock and pattern P appears at different timestamps. From our pattern definition, the patterns with the same $Data$ have been wrapped into a single *pattern* in which different timestamps are

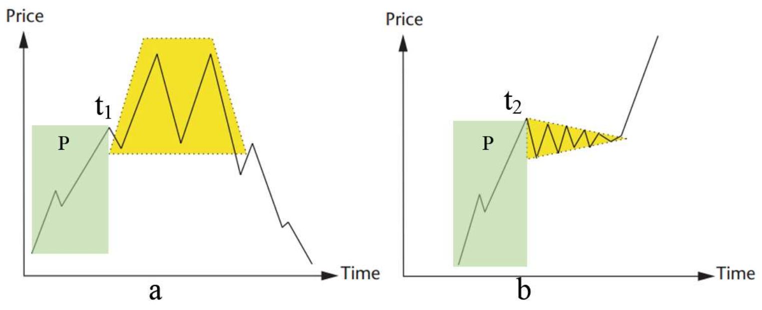


Figure 2: A trading pattern example in stock market data

used to differentiate patterns. The pattern P can be formalized as $\langle pId, Data, T\{t_1, t_2\}, NextP\{P', P''\}\rangle$ to reduce the number of patterns in memory.

Given two patterns P and P^*, we use the *synthetic distance* metric [2] to measure the similarity. The synthetic distance metric combines Euclidean distance and tendency distance. Specifically, let $d(P,P^*)$ denote the similarity between P and P^*, it is computed as

$$d(P,P^*) = \sqrt{\sum_{i=1}^{h}(x_i - x_i^*)^2 + \gamma \times (\sum_{i=1}^{h}|x_i - x_i^*|)} \tag{1}$$

where x_i is the $i-$th attribute in the set $P.Data$, $|x_i - x_i^*|$ is the tendency distance between patterns P and P^* in term of the $i-$th attribute, h is the pattern length, and γ is an adjustable weight. Remark that, although Euclidean distance is widely used to measure the distance of points or patterns, it is unable to recognize the pattern direction or tendency, which is important to discovery pattern fluctuation for applications such as web-click analysis and financial trading analysis. In contrast, the synthetic distance metric can allows us to measure the trend similarity of patterns, which calculates the internal directions (e.g., increment, decrement) of one pattern. Particularly, we find that the synthetic distance equals to Euclidian distance when $\gamma = 0$. Essentially, one can view this distance metric as a more general version of Euclidian distance metric.

With the above concepts in mind, we now formally define our problems as follows.

Definition 2.4. *Given a "target" pattern $\hat{P}$, and a pattern set $\mathbb{P}$ that includes all candidate patterns, we want to retrieve a set $\mathbb{P}^{sim}$ of k patterns, which are most similar to $\hat{P}$, that means $\mathbb{P}^{sim} = \arg\min_{p\in P}{}^{k} d(P,P^*)$*

Definition 2.5. *Given a pattern set $\mathbb{P}$ and a new pattern P, we want to update $\mathbb{P}$ as follows: If pattern P does not appear in $\mathbb{P}$, P will be inserted into set $\mathbb{P}$; otherwise, the pattern $P^* \in \mathbb{P}$ whose Data is same to that of P will be updated by adding only a new timestamp t in the timestamp set T of pattern P^*.*

Our goal is to develop a distributed processing framework that can (i) finish pattern matching and update with time as less as possible, and (ii) handle multiple matching tasks as many as possible, assuming the number of nodes in the distributed system is given (e.g., 10).

3. System Overview

To support time-dependent pattern matching and incremental pattern update, we design a framework, dubbed as FastPM. The architecture of FastPM is shown in Figure 3. It is composed of three major components.

▷ *History pattern discovery component.* It is used to extract patterns from history data, and to deploy these patterns in our distributed system. This component essentially serves as the subsequent pattern matching tasks, and can be regarded as our preprocessing unit. In this component a *hierarchical pattern extraction* method is developed

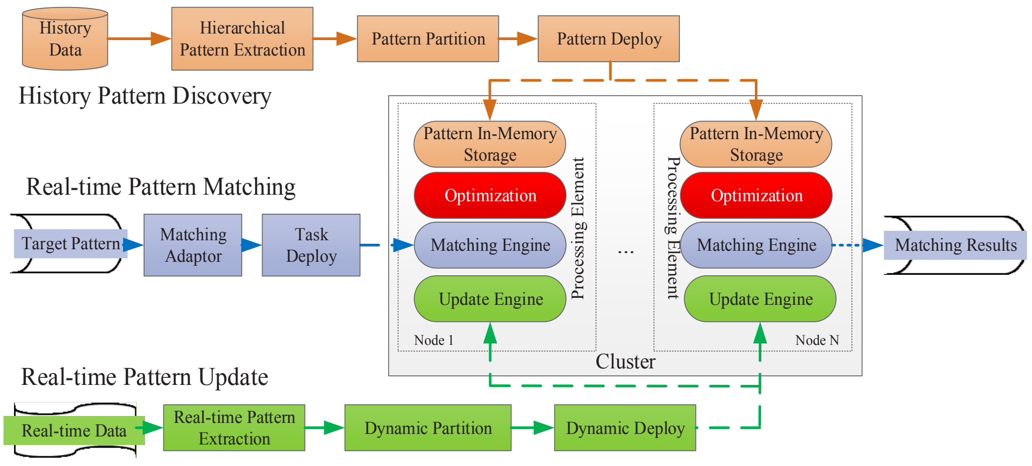


Figure 3: Architecture of FastPM. It is mainly composed of three components: (i) the history pattern discovery component, see the top part of the figure; (ii) the real-time pattern matching component, see the middle part of the figure; and (iii) the real-time pattern update component, see the bottom part of the figure.

to find persistent patterns from different granularities (e.g., every second, every minute, and every hour). Furthermore, it proposes two efficient algorithms for pattern partition and deployment to facilitate the matching efficiency, respectively.

▷ *Real-time pattern matching component.* It is the core of our framework, which mainly serves as searching similar patterns from history patterns. Once a new target pattern is received, a pattern matching job is generated instantly. Next, the *Matching Adaptor* analyzes the job via the system parameters (e.g., the number of partitions/machines), and parses it into multiple matching tasks. Our framework deploys these matching tasks to their corresponding partitions by *Task Deploy*, where they can be executed in parallel. The *Matching Engine* combines synchronous and asynchronous mechanisms together to execute the real-time pattern matching. It first searches the patterns stored in memory and calculates the local top-k patterns, and then merges all the local similar patterns, obtaining the global top-k similar patterns, i.e., the matching results. Particularly, our framework enhances the *Matching Engine* by developing targeted optimization techniques (e.g., local query index, branch-prune approach), which will be expatiated in Section 5.2. Note that, although this paper mainly focuses on pattern matching in distributed systems, we would like to point out that, the matching results can immediately serve as many other applications (e.g., pattern prediction), as discussed in Section 7.)

▷ *Real-time pattern update component.* It is mainly used to process streaming data in an *online* fashion. In this component, patterns from streaming data are extracted successively; the *Dynamic Partition* unit assigns partition (more specifically, identifier of the partition) to the pattern. This is done in the master node of our distributed system, in which the global partition information is stored. Afterwards, the *Dynamic Deploy* unit deploys the pattern to the machine containing the "specified" partition. Essentially, this step is to find a machine or node in which the update operation is to be executed. Finally, the *Update Engine* follows some rules to update the pattern's status, by either inserting the pattern, or appending the pattern's timestamp to an existing pattern (recall Definition 2). For our framework, the most key and relatively complicated parts in the real-time pattern update component are: *Dynamic Partition* and *Update Engine*; Section 6 examines these two parts in more details.

4. History Data Processing

In this section, we first describe the *hierarchical pattern extraction* method, and then discuss how to partition these "extracted" patterns and deploy the partitions on the processing machines (i.e., nodes in distributed systems).

4.1. Hierarchical pattern extraction

The granularity of the sequence generated directly by the original data items is fine-grained. Naturally, the pattern generated based on such a sequence can express only the data tendency in a short time. This implies that, more hidden

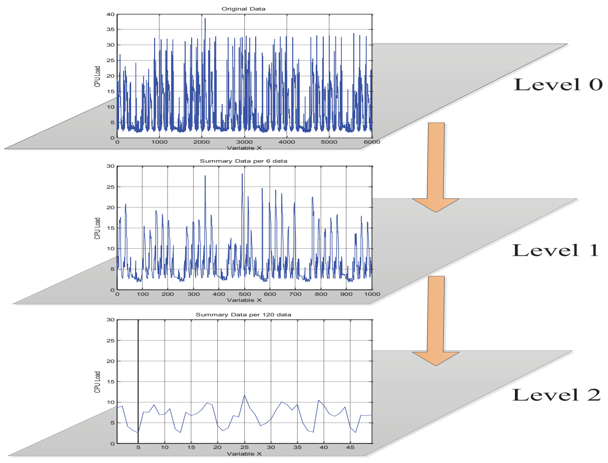


Figure 4: An example of hierarchical pattern extraction. Level 1 is a summary of Level 0, and Level 2 is a further summary of Level 1.

patterns such as *long-term* patterns or *summary* patterns, cannot be acquired if one directly uses the fine-grained sequences.

To detect hidden patterns, we propose a *hierarchical pattern extraction* method. The basic idea of our method is to compress the original sequence(e.g., every second) into a coarse-grained data sequence (e.g., one minute or even one hour). The compressing process essentially is to merge (original) multiple consecutive data into one value. This can be achieved by employing an operator function, say $Operator()$. Here $Operator()$ can be $Average()$, which is calculating the average value as the coarse-grained data, or other user-defined methods.

Figure 4 illustrates an example of the hierarchical pattern extraction method. In this figure, *Level 0* is an original sequence containing 6000 data items, and the fundamental time unit is one second; *Level 1* is a "compressed" sequence by averaging every subsequence containing 6 data items in original sequence; and *Level 2* is a further compressed sequence by averaging more data items from original sequence. From this figure, one can easily understand that, a "high-level" pattern (e.g., at Level 2) essentially implies a longer data tendency, which is important for pattern prediction in a long-term.

Let X be a sequence with n-length data items, l be the number of items of each subsequence to be compressed, and Y be the output sequence. The pseudo-codes of our method are shown in Algorithm 1.

Algorithm 1 *hierarchicPatternExtraction*

```
Input: $X = \{x_1, x_2, \cdots, x_n\}$ and $l$
Output: $Y$
1: for $\{ i = 1; i <= (n-l); i = i+l \}$ do
2:     $T \leftarrow \{x_i, \cdots, x_{i+l}\}$;
3:     $j \leftarrow \lceil i/l \rceil$;
4:     $Y_j \leftarrow Operator(T)$;
5:     $Y = Y \cup Y_j$;
```

4.2. *Cluster-based partition*

In order to facilitate pattern matching in a distributed system, we need to *partition* the patterns into multiple disjoint parts, which shall be *deployed* in different processing machines.

An immediate method for pattern partition is to separate the patterns randomly, e.g., using a hash function. Although this method is easy-to-understand, and is widely used in distributed processing systems (e.g., Spark [30], Storm [31]), its performance could be affected by the distribution of patterns. Imagine if one uses the random partition method, it is highly possible that most of similar patterns could be assigned to different partitions. In this case, it could incur many repetitive traversals in the subsequent pattern matching tasks.

To alleviate the above issue, we propose using the cluster-based partition method. Our method not only separates the patterns into disjoint parts, but also achieves good performance, as demonstrated in Section 8. The rationale behind this method is that, the similar patterns are grouped into one cluster, and each pattern will be assigned to a cluster. Specifically, we use the k-means algorithm [32] to group the patterns into clusters . Here each cluster can be viewed as a partition, and in each partition we use the centroid (of the corresponding cluster) to represent the partition. By this way, all patterns can have their corresponding partitions, and each pattern in a partition is closer to its centroid than other centroids. The pseudo-codes of the cluster-based partition are shown in Algorithm 2.

Algorithm 2 *clusterBasedPartition*

```
Input: $\mathbb{P} = \{P_1, P_2, \cdots, P_n\}$ and $k$
Output: $U^{t+1}, Q^{t+1}$
1:  $t \leftarrow 0$;
2:  $U^t \leftarrow$ Select randomly $k$ centroids from $\mathbb{P}$
3:  while ($True$) do
4:      for { $j = 1; j <= n; j = j + 1$} do
5:          for { $i = 1; i <= k; i = i + 1$} do
6:              $Q^{t+1}(j) \leftarrow \arg\min_i d(U_i^t, P_j)$;
7:      $U^{t+1} \leftarrow Centroid(Q^{t+1})$;
8:      if $U^t == U^{t+1}$ then
9:          Return;
10:     $t++$;
```

4.3. Frequency-aware deploy

After the patterns have been partitioned, we have to deploy them on the processing machines. The widely used method is using the round-robin algorithm [33] to deploy the partitions to each node in distributed systems (e.g., Storm [31], Spark [30], Kafka [34]). Yet, its performance could be significantly affected by the frequency of the patterns. That is, some patterns could occur frequently while others appear occasionally. In this case, the "round-robin deploy method" possibly incur the unbalanced deployment, damaging the performance.

To address the above limitation, we present a *frequency-aware algorithm* for deploying the partitions, which can make the patterns located in each node as balanced as possible. Our method is based on the following intuition: the frequency of patterns in each partition reflects the number of tasks to be handled, and it clearly affects the workload of a node. (Note that, it is very convenient to calculate the frequency of pattern, since we have a timestamp set T to record the occurrence time of one pattern, recall Definition 2.3). Given a partition, we use a simple heuristic: Scanning the workloads of all nodes and deploying the partition on a node that is with the minimum workload currently.

Specifically, assume, without loss of generality, that the number of the nodes (in a distributed system) is M. Let k be the number of partitions (obtained in the previous section), Q_i be the pattern set in the ith partition, $Number(Q_i)$ be the number of patterns in Q_i, Num be an array used to record the number of patterns currently deployed in a node, and $QNode$ be the node index for Q. The details of our method are shown in Algorithm 3.

Algorithm 3 *frequencyAwareDeploy*

```
Input: $Q = \{Q_1, Q_2, \cdots, Q_k\}, M$
Output: $QNode[k]$
1:  $Num[M] \leftarrow 0$;
2:  $QNode[k] \leftarrow 0$;
3:  for { $i = 1; i <= k; i++$} do
4:      $index \leftarrow$ Select the Node with $Minimize(Num[M])$;
5:      $QNode[i] \leftarrow index$;
6:      $Num[index] \leftarrow Num[index] + Number(Q_i)$;
```

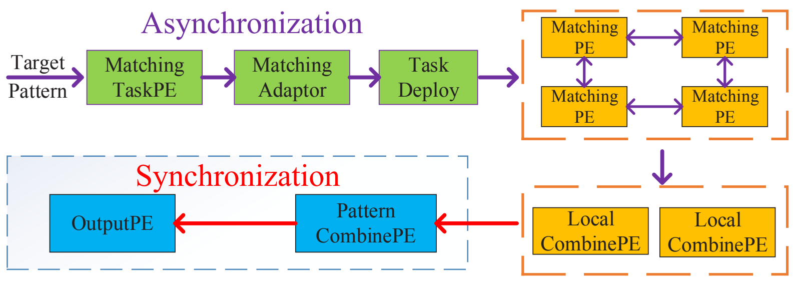


Figure 5: The data flow of a matching task. In this figure the brown arrow refers to the asynchronous communications; asynchronous messages are continuously received and processed in each PE with no latency. In contrast, the red arrow refers to the synchronous mechanism, and the corresponding PE has to wait, until all the upstream subtasks are finished.

Note that, after partitions are deployed in their specific nodes. Each partition creates a *processing element (PE)*, which stores patterns in memory (cf., Figure 3), and is responsible for the subsequent pattern matching.

5. Distributed Pattern Matching

In this section, we first introduce the basic model included in the "real-time pattern matching" component (Section 5.1), and then present our optimization strategies to further improve the efficiency (Section 5.2).

5.1. Basic model for multiple matching tasks

▷ *Overview.* Figure 5 shows the basic model of distributed pattern matching. Our processing model can flexibly deal with *multiple matching tasks* simultaneously. In general, the entire matching process is parsed into a data flow to resolve the complex task, and our model combines asynchronous and synchronous mechanisms together to collaborate with different steps in a pipeline job. The asynchronous mechanism is used to quickly handle "received matching task", "task deployment", and "pattern matching", while the synchronous mechanism is used to gather the "matching results" from all upstream.

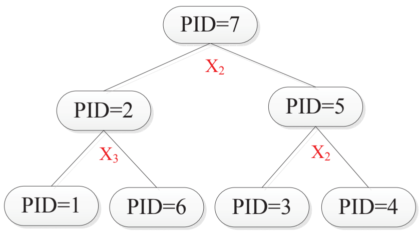
▷ *Processing units.* In what follows, we discuss major processing elements in detail.

- *Matching TaskPE.* When it receives a target pattern $\hat{P}$, it checks the format of $\hat{P}$, generates a matching job, and sends the job to the next PE, i.e., *Matching Adaptor*.
- *Matching Adaptor.* It analyzes the cluster environments to obtain system parameters, and parses the matching job into multiple matching tasks.
- *Task Deploy.* It is used to assign the (multiple) matching tasks to their exclusively partitions.
- *MatchingPE.* It is the core processing element of our system. It finishes pattern matching for each partition and sends similar patterns to downstream PEs. We will discuss *MatchingPE* in more details.
- *Local CombinePE.* It receives similar patterns from *MatchingPE*, merges these results and calculates the top-k similar patterns in each machine.
- *PatternCombinePE.* It is a centralized processing element, which merges all the local results from different machines. We can use the *Merge Sort* algorithm [35] to sort them quickly. Afterwards, the global top-k similar patterns are obtained.

▷ *MatchingPE revisit.* A machine or node can contain multiple *MatchingPE*s and each *MatchingPE* has no intersection with others. Since a node can contain multiple disjoint partitions and each partition can create a PE (recall Section 4). These multiple *MatchingPE*s can allow us to run tasks in parallel, and can receive messages (e.g., temporary results) from their corresponding neighbors. In each *MatchingPE*, patterns are stored in memory and arranged in a hash table T with key-value pairs $< pID, Pattern >$. One can iterate the local hash table T to find top-k similar patterns. That is, it calculates the similarity between the target pattern $\hat{P}$ and each local pattern in T, and then sorts the patterns by their similarities, and finally selects similar patterns with the top-k smallest distances.

PID	X_4	X_3	X_2	X_1
1	4	5	2	7
2	3	5	6	8
3	12	32	32	34
4	23	27	53	48
5	17	25	40	46
6	10	41	10	40
7	12	54	11	55

(a) Patterns to be indexed



(b) Local query index

Figure 6: A local query index construction example: (a) There are seven patterns in a partition to be indexed, and each pattern has four dimensions; (b) the red texts "X_2" or "X_3" denotes the candidate (or splitting) dimension of the "corresponding" node.

5.2. *Distributed processing optimizations*

In this section, we examine the optimization techniques that improve the efficiency of distributed processing. Our main optimization technique is to use local query index to speed up the matching efficiency. Besides, we cover a branch-prune approach to prune some patterns, thereby reducing the storage space and also further improving the matching performance.

5.2.1. *Observation*

Recall Section 5.1, we iterate all the "local" patterns to calculate the distances, in order to find top-*k* similar patterns in each partition. Observe that, in many practical scenarios (e.g., web click monitoring, stock trading), the number of local patterns could be very large. For example, the value of web click is arranged in [0,1000] and the pattern length is 5. Then, the number of all potential patterns could reach $1000^5 = 10^{15}$. In this case, even if we execute these patterns in multiple nodes as mentioned in Section 5.1, it is still very time consuming.

To attack this difficulty, we suggest a k-d tree based method that serves as the local query index, thereby speeding up the matching efficiency. The central observation for our optimization is that, some patterns that are quite different from the target pattern, should not be calculated the distances between these patterns and the target one. This way, the search space can be reduced considerably.

Generally speaking, our method essentially extends the k-d tree algorithm [36] to a distributed stream framework. In what follows, we first examine how FastPM builds the k-d tree for our pattern matching at each node, and then explain how the matching algorithm works on the k-d tree.

5.2.2. *Local query index construction*

To understand the construction process, it is necessary to explain the concept of *the whole distance cost*, in terms of a certain dimension. Assume, without loss of generality, that there are N patterns and each pattern contains h-dimensional data. Then, for any dimension x_i ($i \in [1,h]$), the whole distance cost, denoted by $cost(x)$, is computed as

$$\begin{aligned} cost(x_i) &= \sqrt{\sum_{m=1}^{N}\sum_{n=1}^{N}(p_m(x_i)-p_n(x_i))^2} \\ &+ \gamma\times(\sum_{m=1}^{N}\sum_{n=1}^{N}|p_m(x_i)-p_n(x_i)|) \end{aligned} \tag{2}$$

where γ is an adjust weight, $p_m(x_i)$ denotes the value of the $m-$th pattern in dimension x_i . Following conventional k-d tree construction algorithms, FastPM builds a k-d tree in a greedy, top-down manner, by recursively splitting the current node into two sub-nodes as follows:

- *Step 1.* Compute the whole distance *cost* for each dimension;
- *Step 2.* Choose a dimension whose *whole distance cost* is maximal as the candidate dimension;
- *Step 3.* Sort the values of patterns in terms of the candidate dimension, and pick the pattern, whose value in this candidate dimension is at the middle, as the splitting node (notice that, hereafter, this candidate dimension is referred to as the *splitting dimension* of this node);

- *Step 4.* Distribute the rest patterns whose values are less (resp., larger) than the above "middle" value to the left (resp., right) child.

Notice that, picking the pattern (whose value is at the middle) as the splitting node can allow us to construct a "balanced" k-d tree, in which the patterns are indexed. Thus, the height of the tree is bounded in $O(\log N)$, where N is the number of all patterns. The followings show an example to construct a k-d tree in the context of our concern.

Example 1. *Consider 7 patterns with 4 dimensions* $< X_4, X_3, X_2, X_1 >$ *shown in Figure 6. First, to find the candidate dimension, we calculate the whole distance cost for each dimension based on Equation* (2)*. These costs are* $< 48.9, 440.3, 444.3, 430.3 >$ *when* λ *is set to 10. In this case, dimension* X_2 *is the largest one, and thus it is picked as the candidate dimension. Next, we sort the values of patterns in dimension* X_2*, and acquire the "middle" value* 11*. Thus, we select the pattern* $pId = 7$ *as the splitting node. To balance the tree, patterns* $(pId = 1,2,6)$ *(resp.,* $(pId = 3,4,5)$*) are assigned to the left (resp., right) subtree. We iterate the above steps for each subtree until each subtree is empty.*

5.2.3. *Pattern matching via local query index*

Given a target pattern $\hat{P}$ (also, known as a search point later), FastPM first starts at the root node of the k-d tree, using the standard *depth-first search* (DFS) algorithm to find a candidate node. This node is a leaf node and can be easily found through the splitting dimension at each branch node. Once the algorithm reaches a leaf node, it saves that node point as the current node n_d.

To facilitate the search, FastPM maintains a list of top-k nearest nodes during the searching process. Once the algorithm reaches a node, it determines whether the current node is closer to $\hat{P}$, compared to the nodes in the list. If so, it saves the node and also updates the list (notice: at the initial stage, the number of nodes in the list is less than k; and we always use the notation d_{max} to denote the maximal distance from $\hat{P}$ to the nodes in the list, for short). Otherwise, it does nothing.

After traversing the current node n_d, the distance between a current pattern and the target pattern $\hat{P}$ has been calculated. FastPM checks whether there exist any nodes, on either side of the branches, that are closer to $\hat{P}$ (compared to the nodes currently in the list). This can be done by intersecting *the splitting hyperplane* with *a hypersphere* around the search point and with a radius equal to d_{max}, which can be achieved using the *splitting distance* explained later. Many subtrees far away from $\hat{P}$ will not be traversed, which thus efficiently eliminates some portions of the search space and speeds up the matching efficiency.

Remark that, the splitting distance is somewhat similar to Equation 2, yet it is mainly used to compute the distance between the target pattern and a pattern in a node, and it considers only the splitting dimension (mentioned earlier). Specifically, assume, without loss of generality, assuming that a node stores the pattern P, and the splitting dimension of the node is κ ($\kappa \in [1,h]$). Let $d_{spit}(\hat{P},P)$ denote the splitting distance between P and the target $\hat{P}$, it is computed as

$$d_{spit}(\hat{P},P) = (\hat{P}(x_\kappa) - P(x_\kappa)) + \gamma \times (|\hat{P}(x_\kappa) - P(x_\kappa)|) \tag{3}$$

The pseudo-codes of the matching search algorithm are shown in Algorithm 4. We directly use n_d as the current node to denote the pattern in the tree. Similarly, we use n_d^l (resp., n_d^r) to denote the left (resp., right) child of n_d. In addition, L_{rst} is a sorted list used to store the matching patterns, and N_{vst} is a set of nodes having been visited. Lines 1-4 are used to determine whether a node n_d has been visited, and Lines 5-8 are used to handle whether the pattern in the node should be put into the result list L_{rst}, where d_{cur} is used to store the distance between $\hat{P}$ and the pattern in node n_d, and d_max is the maximal distance in L_{rst}. In addition, Lines 9-12 are used to process the left child, while Lines 13-16 are for handling the right child.

Correctness of Algorithm 4. Suppose n_d be the current node of the k-d tree, n_d^l is the left child of n_d, $d_{n_d^l}$ is the distance between node n_d^l and target pattern $\hat{P}$, d_{split} is the splitting distance between node n_d^l and target pattern $\hat{P}$, and d_{max} is the maximal distance in the result list L_{rst}. If $d_{split} > d_{max}$, assume that n_d^l is a candidate nearest pattern and $d_{n_d^l} < d_{max}$. According to Equations 1 and 3, since the splitting distance is calculated based on one dimension(e.g., κ), it is easy to infer that $d_{split} < d_{n_d^l}$, which leads to a contradiction as $d_{split} > d_{max}$ and $d_{n_d^l} < d_{max}$, that is, there exists no candidate nearest pattern if $d_{split} > d_{max}$. Therefore, our algorithm traverses the left branch only if it is unvisited and the splitting distance is less than d_{max} (Lines $9-12$). Similar to traversing left child, the right child is searched only if the splitting distance is more than d_{max} (Lines $13-16$). $\square$

Algorithm 4 *matchingSearch*

```
Input: $\hat{P}$, $n_d$, $k$, $L_{rst}$, $N_{vst}$
Output: $L_{rst}$
1: if $N_{vst} \cap n_d == \emptyset$ then
2:     $N_{vst} \leftarrow N_{vst} \cup n_d$;
3: else
4:     Return;
5: $d_{cur} \leftarrow d(n_d, \hat{P})$;
6: if $(|L_{rst}| < k) \; || \; (d_{cur} < d_{max})$ then
7:     put pattern in $n_d$ into $L_{rst}$;
8:     $d_{max} = Max(d_{cur}, d_{max})$;
9: if $n_d^l \neq$ null & $N_{vst} \cap n_d^l \neq \emptyset$ then
10:     $N_{vst} \leftarrow N_{vst} \cup n_d^l$;
11:     if $d_{spit}(\hat{P}, n_d^l) <= d_{max}$ then
12:         matchingSearch($\hat{P}, n_d^l, k, L_{rst}, N_{vst}$);
13: if $n_d^r \neq$ null & $N_{vst} \cap n_d^r \neq \emptyset$ then
14:     $N_{vst} \leftarrow N_{vst} \cup n_d^r$;
15:     if $d_{max} <= d_{split}(\hat{P}, n_d^r)$ then
16:         matchingSearch($\hat{P}, n_d^r, k, L_{rst}, N_{vst}$);
```

5.2.4. *Branch-prune optimization*

Normally, if a pattern has occurred in the history sequence, it should be stored as a node in the tree. This implies no matter the pattern is frequent or rare, it shall occupy one if we employ the conventional way. Image if the number of rare patterns (a.k.a., the patterns that happen occasionally) is large, the storage space increases rapidly. This could incur a performance slowdown of the system. To alleviate the above issue, we introduce a *branch-prune approach*. Our approach is simple and easy-to-understand, yet definitely efficient, as evaluated in Section 8.

The approach is depending upon the following observation: in many applications such as web click analysis and financial trending analysis, the "rare" pattern (a.k.a., the pattern that happens occasionally) usually cannot represent the actual characteristics of the data, and has little influence on data analysis. Note that, pattern matching usually serves as the backbone of these applications. The observation above immediately inspires us to re-consider the storage of the patterns. The rationale behind our approach is to remove some branches that are with rare patterns, while retaining the main features of the entire data.

Owe to that the (time dependent) pattern has recorded the occurrence time, we can count the frequency of each pattern easily. Specifically, we discard some rare patterns based on the pattern frequency. The criteria for pruning is that, *the pattern with less frequency is more likely to be removed, and the pattern with more frequency should be reserved as much as possible.*

Let ε $(0 \leq \varepsilon \leq 1)$ be an approximation ratio, and f be the summarized frequency of patterns. The above criteria can be easily achieved: one can sort the patterns by the frequency, preserve $f * \varepsilon$ patterns and remove the rest $f * (1 - \varepsilon)$ patterns. In this way, some patterns ranked behind and their corresponding branches in the tree are to be continuously removed until reaching the watershed. In the end, the k-d tree is largely compressed, consuming less storage. Naturally, the matching query can benefit with less search space, since the number of the overall nodes (stored in the tree) decreases.

6. Real-time pattern update

Recall Section 3, we have mentioned that *Dynamic Partition* and *Update Engine* are two key and relatively complicated parts in the real-time pattern update component. In what follows, we examine how our framework achieves dynamic partition and the pattern state update in detail.

▷ *Dynamic pattern partition.* Since our framework stores the global partition information in master node (cf., Section 3) and each partition is represented by the centroid of the patterns in the partition (cf., Section 4.2), our

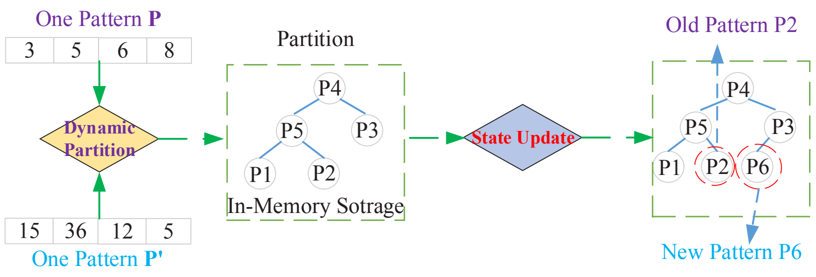


Figure 7: Example of real-time pattern update.

dynamic pattern partition approach fully exploits these available information.

Assume that there are n partitions in total. This implies that there are also n centroids. Our framework organizes these centroids using a spatial indexing structure R-tree, and this structure is maintained in the master node of the distributed system. It mainly serves as the speedup of dynamic pattern partition. With this structure, one can find the nearest centroid for the real-time pattern. The partition corresponding to the "found" centroid shall be used as the "specified" partition. That is, pattern state update in the subsequent step shall be done in this partition (more specifically, it is done in the k-d tree corresponding to this partition). Note that, after new patterns join in the partition, the original centroid might be not optimal any more. Theoretically speaking, we may need to update the centroid whenever a new pattern is received. However, this manner may cause a very high cost. Instead, in our framework we update the centroid periodically. This update manner reduces the cost, and can still achieve the dynamic partition task, since a real-time pattern is still roughly near to the centroid of the original partition for a relatively long time.

▷ *Pattern state update.* Given a real-time pattern P, we assume that its partition has already assigned and has been dynamically deployed on a *machine* (usually, such a machine is a slave node of the distributed system), the operation process of the pattern state update is essentially to search on the k-d tree, and to see whether there is pattern that is same to P (except the timestamp). If found, it means this pattern has occurred; we only need to update this node (in the k-d tree) by appending the pattern P's timestamp to the pattern on this node. Otherwise, we need to insert this pattern as a node in the k-d tree.

The search algorithm (for checking whether P has happened or not) is similar with Algorithm 4. A major difference is that, we here search the top-1 similar pattern in the tree. In brief, the search starts also at the root node of the tree, and it moves to either the left branch of the root node or the right one. This is determined by checking whether the pattern to be inserted is on the left or right side of the *splitting dimension* (recall Section 5.2.3). In the process of searching, if we find a node (in the tree) has "0" distance with the real-time pattern P, we append the timestamp to the node. On the other hand, when we need to move to the left (resp., right) branch of the current node, yet the left (resp., right) branch is null; in this case, it implies that the pattern P is new. The 'null' child of the current node is the potential position for storing P, thereby we add P as a new node here. Note that, adding the real-time pattern P as a new node may cause the tree to be unbalanced, which could lead to a slowdown in terms of the performance of the k-d tree. For this issue, we periodically detect whether the tree is balanced. If not, we trigger a task to re-balance the tree. This way, the performance of the k-d tree can be guaranteed appropriately.

Example 2. *Consider two patterns P and P' and a partition with 5 nodes shown in Figure 6. Assume that this partition is the "specific" partition found by the dynamic partition method. The search algorithm finds a pattern P_2 (in the tree) that is same to P, the system updates P_2 by appending the occurrence time of P. In contrast, for searching P', the search algorithm wants to move to the left node P_6 of P_3 (according to the splitting dimension), yet it is empty. The system updates the tree by inserting P' as a new node P_6.*

7. Applications

Our framework, FastPM, has many immediate applications, e.g., *pattern prediction* (i.e., predict future patterns), *anomaly detection*, *social feeling analysis*, and so on. These applications can be easily applied in our framework. In what follows, we use two examples to illustrate how they can be applied in our framework.

▷ *Application example 1: pattern prediction.* Pattern Prediction is trying to forecast the future tendency based on the target pattern. Given a target pattern, if the target pattern or some similar patterns has occurred before, then these potential information can guild us to finish the prediction. Essentially, once the similar patterns are found, some existing pattern prediction algorithms can be straightforwardly used for pattern prediction, such as *pattern weighting strategy* (PWS) [2], random walk (RW) [37], Bayesian data analysis (BDA) [38], and so on. Consider the PWS algorithm as an example. The prediction can be calculated from the similar patterns by Equation 4 (known as the *weight equation*) and Equation 5 (known as the *prediction equation*) below.

$$\omega_i = 1 - \frac{t_{\hat{P}} - t_{P_i}}{\sum_{j=1}^{k} (t_{\hat{P}} - t_{P_j})} \tag{4}$$

$$P' = \sum_{i=1}^{k} (\omega_i \times NextP_{P_i}) \tag{5}$$

where $t_{\hat{P}}$ (resp., t_{P_i}) is the occurrence time of the target pattern $\hat{P}$ (resp., a similar pattern P_i), and $NextP_{P_i}$ is the next pattern of P_i. Note that, our pattern contains some attributes such as *NextP* and the occurrence time T (recall Definition 2.3), and so these values are easy-to-obtain.

▷ *Application example 2: anomaly detection.* Anomaly detection refers to the problem of discovering patterns in history data that do not conform to the expected behaviors. It has extensive use in a wide variety of application domains, including fraud detection for credit cards, insurance or health care, intrusion detection for cyber-security, fault detection in safety critical systems, and military surveillance for enemy activities [39]. We can achieve anomaly detection in real-time, based on FastPM. For example, when we receive a pattern P, we can search P on the local query index, using top-1 similar search with "0" distance (which is the same as that in 6). If P is not found, then it can be regarded as an outlier pattern (notice: the branch-based prune approach in Section 5.2.4 only removes the patterns happened occasionally; those patterns themselves are also outlier patterns). Furthermore, if P is found but its occurrence time (e.g., 8:00 pm) is inconsistent with its occurrence time set T's distribution (e.g., 6:00 am $\sim$ 12:00 am), it can be also an anomaly pattern. Last but not least, if P is found (here we use P^* to denote the pattern has occurred before P, for clearness), but P does not conform to the expected pattern $P^*.NextP$, it can be also supposed to an anomaly pattern.

8. Experimental Evaluation

In this section, we cover our experimental results in detail. Specifically, Section 8.1 presents experimental settings. Sections 8.2 and 8.3 evaluate the performance of pattern matching and of pattern update, respectively. Finally, as extra experiments, Section 8.4 examines the performance of FastPM from another perspective, using *pattern prediction* as a sample of applications. Note that, besides pattern prediction, FastPM has many other immediate applications, e.g., anomaly detection, social feeling analysis (recall Section 7). In this paper, we do not exhaustively conduct all these experiments, since they are not the focuses of this paper.

8.1. Experimental settings

We cover our experimental settings from several perspectives such as implementation, dataset, method and metric.

▷ *Implementation. Apache S4* [40] is a general-purpose, distributed, scalable, fault-tolerant, and pluggable platform. It allows programmers to develop applications for processing continuous unbounded streaming data easily. Same to our prior work [41], we implemented FastPM also on top of *Apache S4*. Our FastPM is event-driven, and the query processing in each PE (i.e., *processing element*) can be triggered periodically, or by an incoming event. (It is worth noting that, although we implemented FastPM on Apache S4, it can be also implemented on other stream platforms such as Twitter Storm [31], Spark Streaming [30] and Flink [42].) Our distributed system we deployed is a cluster with ten nodes. Each node has one Xeon E5607 Quad Core CPU (2.27GHz), 32GB memory, running CentOS 6.2. We select one node for Zookeeper as our master node and data source adapter in the customization of S4.

▷ *Dataset.* The data used in our experiments is the real-world web click data produced by a game company. The dataset consists of 54 billion tuples generated in one month, namely, it has about 20 million visits every day. We calculate the visiting data every 5 seconds for each page. In particular, we use 80% of the data as our historical data and 20% as the streaming data. The visit bound of each page within a time window is $0 \sim 10^3$, where the type of visit is integer. Notice that, it is hard to find an optimal value for the pattern length h; its determination is usually problem-specific and data-sensitive [43, 44]. That is, the pattern length is determined by domain experts and experienced data analyzers. In our paper, we also follow this rule. Specifically, the length of a pattern is set to 5 in our experiments. This way, the number of potential patterns could reach 10^{3*5}, i.e., 10^{15}.

▷ *Methods.* As we know, traditional pattern matching methods are designed based on sophisticated indexing techniques to achieve good performance, which have to update the indices with tremendous overhead in a high speed stream. In view of this, we adopt the following more competitive methods for comparison. They are: RR (random partition+round-robin deploy), CR (cluster partition+round-robin deploy), CF (cluster partition+frequency-aware deploy), CFL (i.e., CF + local query index), and CFLB (CFL + branch-prune approach). Here RR is a widely used method to process distributed pattern matching in big data platforms (e.g., Storm[31], Spark[30], Kafka[34]) and thus we apply RR as our baseline method. Other four methods optimize the method RR in terms of partition, deployment, local index and pattern pruning to improve the efficiency and scalability; and all these methods employ the hierarchical pattern extraction. Furthermore, we also implement a method called CFLB-H, which does not use the hierarchical pattern extraction, and shall be evaluated in Section 8.4.

▷ *Metrics.* We measure the performance of *FastPM* in terms of several metrics. The first metric is the running time, which is the time interval between the time when a target pattern arrives at the system and the time when *FastPM* outputs the matching result. For the update tasks, it refers to the time of both updating the status of pattern and maintaining the local query index. In our experiments we report the average running time and update time. The second metric is *throughput*, which is measured by the number of tasks processed in a time unit. In our setting, the time unit is one minute. The third metric is used for our extra experiments (i.e., pattern prediction), and is related to the prediction accuracy. Specifically, we use the *mean relative error* (MRE) and the *mean average error* (MAE) to evaluate the prediction accuracy. Support P and $\tilde{P}$ are the real pattern and prediction pattern respectively, h is the pattern length, $\{x_1, x_2, \cdots, x_h\}$ is the data set in a pattern, and N is the number of pattern prediction, then we can calculate *MRE* and *MAE* through the following formulas:

$$MRE = \frac{1}{N} \times \sum_{i=1}^{N} \left(\frac{1}{h} \times \sum_{i=1}^{h} \frac{|P(x_i) - \tilde{P}(x_i)|}{P(x_i)} \right) \tag{6}$$

$$MAE = \frac{1}{N} \times \sum_{i=1}^{N} \left(\frac{1}{h} \times \sum_{i=1}^{h} |P(x_i) - \tilde{P}(x_i)| \right) \tag{7}$$

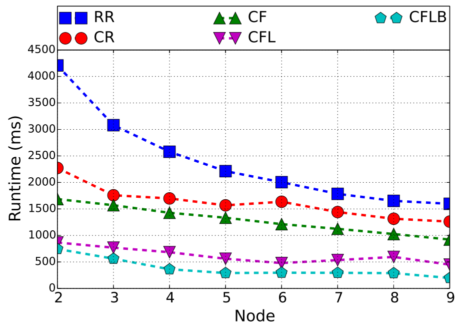
▷ *Other settings.* Since experimental parameters often affect the performance of *FastPM*, in our experiments we evaluate the performance of FastPM by varying several parameters.

- N_{sn}: the number of slave nodes in a cluster. It is ranging from 2 to 9, and 9 is the default number.
- N_{pp} the number of pattern partitions. It is assigned with $[10, 100, 500, 1000]$, and 100 is the default number.
- N_{mt}: the number of matching tasks. It is assigned with $[100, 500, 1000, 5000, 10000]$, and 100 is the default number.

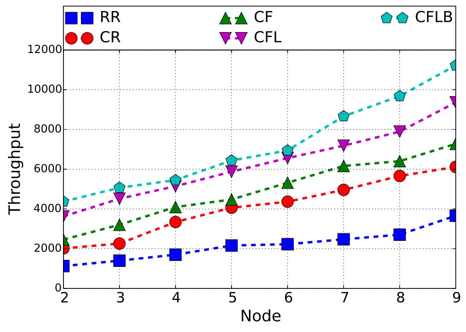
8.2. Performance of pattern matching

To conduct a comprehensive study on the performance of pattern matching, we vary N_{sn}, N_{pp}, N_{mt}, respectively. Notice that, in this paper, when we explicitly mention the performance of FastPM, we refer to CFLB, since it employs all the optimizations proposed in the paper.

▷ *Varying N_{sn}.* We examine the performance of pattern matching by varying the number of (slave) nodes firstly. The experimental results are reported in Figure 8. It can be seen from Figure 8(a) that, CR is superior to RR (which is a common implementation in distributed processing systems such as Spark, Storm, Flink). This is mainly because CR adopts the cluster-based partition, which groups similar patterns together, benefiting to searching k similar patterns. Yet, CR is inferior to CF. This is due to that CR uses the round-robin algorithm to deploy the partitions on physical nodes, it may incur unbalanced deploy, while CF overcomes this limitation by considering the frequency of patterns. Also, one can see that, CFL and CFLB perform better than CF. This is because both of them adopt the optimization technique (i.e., *local query index*) to improve the matching efficiency. In particular, CFLB requires 0.2 seconds for a matching task (notice: there are about $0.8 * 10^{15}$ potential patterns in the history data). On the other hand, one can see that, with more available nodes (i.e., more computing resources) , the average running time decreases (cf., Figure 8(a)) and the throughput increases (cf., Figure 8(b)).



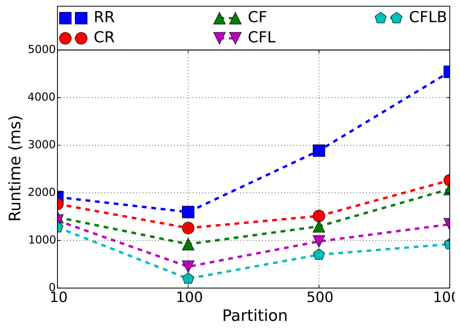
(a) Running time



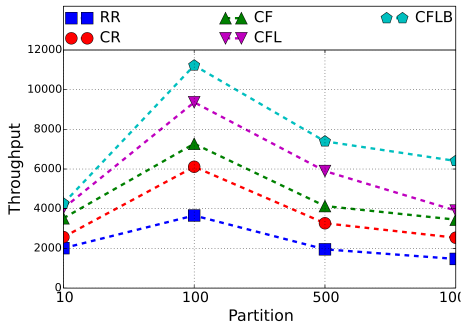
(b) Throughput

Figure 8: Matching performance: varying N_{sn}

▷ *Varying* N_{pp}. We study the performance of pattern matching by varying the number of pattern partitions. Figure 9 reports the results. It can be seen that, among the set of numbers $[10, 100, 500, 1000]$, the best number of partitions is 100. In this case, it gives rise to the least runtime and the highest throughput. When the number of partitions is too small (e.g., $N_{pp} = 10$), each of these methods has the poor performance, this is mainly because the number of patterns in each single partition is relatively large, damaging the parallelism ability. On the other hand, as the number of partitions increases, the number of patterns deployed in each partition is decreased. Naturally, the performance of parallelism is improved. Yet, when the number of partitions is too large, it has also the poor performance as shown in this figure. This is mainly due to that the communication cost among different PEs (i.e., processing elements) increases significantly, incurring the poor performance. In addition, the results also show that CFLB achieves the best performance, regardless of the runtime or the throughput.



(a) Running time



(b) Throughput

Figure 9: Matching performance: varying N_{pp}

▷ *Varying* N_{mt}. This set of experiments studies the performance of pattern matching by varying the number of (matching) tasks. We send all the tasks to our system at one time to test the parallelism ability in depth. The experimental results are shown in Figure 10. We find that, CFLB achieves the best performance, respecting both the runtime (cf., Figure 10(a)) and the throughput (cf., Figure 10(b)). Particularly, when the number of matching tasks is equal to 10000, the runtime of CFLB is about 5 seconds. In addition, as we expected, the throughput improves when there are more matching tasks. This implies that FastPM is scalable to more tasks.

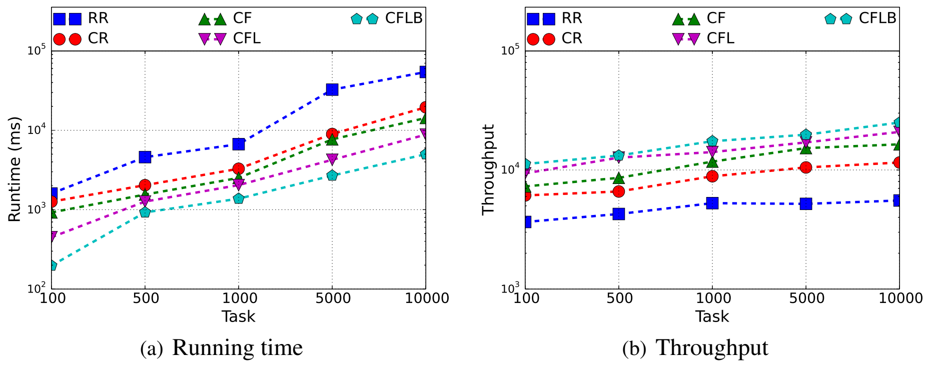


Figure 10: Matching performance: varying N_{mt}

8.3. *Performance of pattern update*

The results reported in the previous section have shown CFLB has the best performance. In this section, we mainly focus on studying CFLB in terms of the update performance. Yet, in the rest of experiments, we also report the results related to CFL, for ease of presentation. We study the update performance by varying the parameters N_{sn} and N_{pp}, respectively. Note that, the parameter N_{mt} used in previous experiments is the number of *matching tasks*, and thus cannot used here.

▷ *Varying N_{sn}*. We study the update performance by varying N_{sn} firstly. Figure 11 shows the experimental results. It can be seen from Figure 11(a) that, the update time of FastPM decreases when more nodes are available. Particularly, it updates a pattern in less than 0.03 seconds (see CFLB with 9 slave nodes, which is the default value of our FastPM). This satisfies the common requirement for real-time applications. Compared to FastPM (i.e., CFLB), the update time of CFL is inferior; this is because the local query index in CFL manages more entries (i.e., nodes), and thus it takes more time (for searching the place to be inserted and for inserting the new pattern). On the other hand, one can see from Figure 11(b) that, the throughput of FastPM increases when more resources are available. This demonstrates the saleability of FastPM from another perspective.

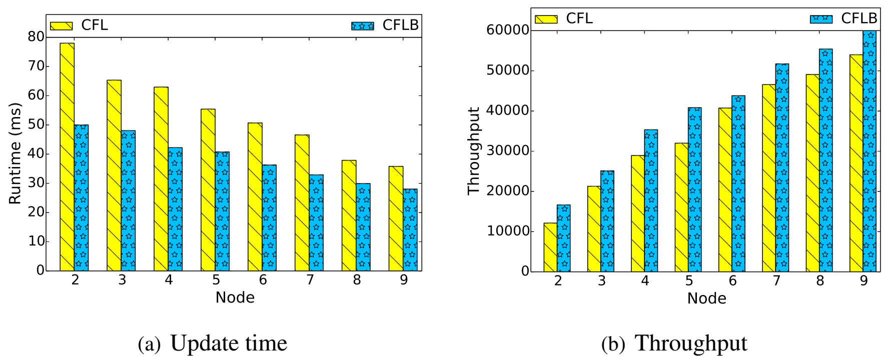
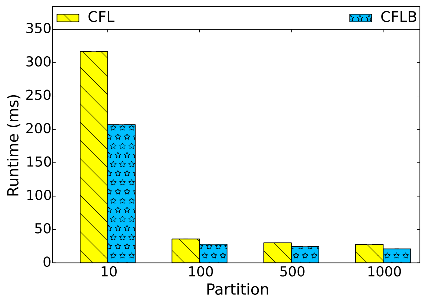
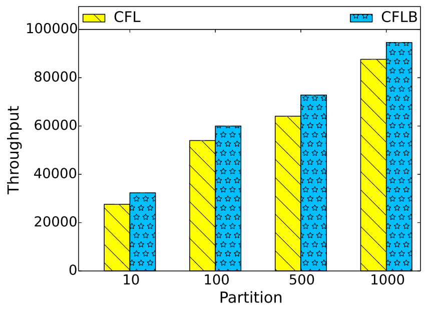


Figure 11: Update performance: varying N_{sn}

▷ *Varying N_{pp}*. We vary the number of pattern partitions to study the update performance. Figure 12 reports the experimental results. On one hand, Figure 12(a) shows that, more partitions benefit to the update performance. When N_{pp} is small (e.g., 10), the update performance is pretty poor (larger than 200 ms even for the *CFLB* method). And we find that "$N_{pp} = 100$" is a turning point, its update time is less than 50 ms. Interestingly, when $N_{pp} >$ 100, the improvement in terms of update performance is minor. (Note that, our update task first finds the patterns through traversing the k-d tree and then updates the content of this pattern, thus the update time should consider both aspects. With more partitions, although the traversal can be improved, the time for updating the content is almost static, i.e., about 10 ms. That is why the update efficiency does not vary much. Essentially, as shown in our experiments, the update time of *CFLB* at $N_{pp} = 1000$ is almost 30 ms.) On the other hand, as shown in Figure 12(b), the throughput increases when more partitions are used, and reaches 10^5 when $N_{pp} = 1000$. (Note that, different from pattern matching, pattern update has no communication cost, and thus the update task can be performed more efficiently with more partitions.)



(a) Update time



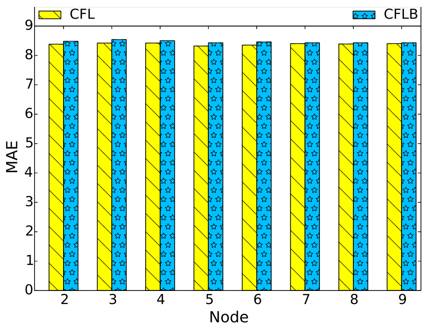
(b) Throughput

Figure 12: Update performance: varying N_{pp}

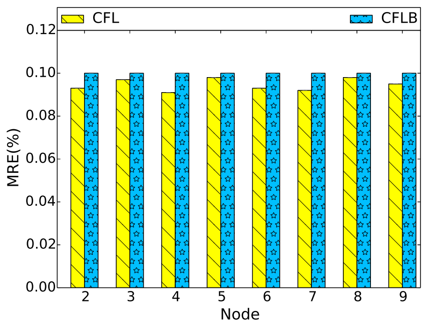
8.4. *Other experimental results*

As mentioned in previous sections, FastPM can be immediately applied to pattern prediction. We study the performance of pattern prediction in this section. We report the results for both CFL and CFLB. Since the results related to runtime and throughput are similar to that of pattern matching, in what follows we do not exhaustively show all results (e.g., varying N_{pp}). Besides, we also test the performance of CFLB-H, which does not employ the hierarchical pattern extraction (recall Section 8.1).

$\triangleright$ *Accuracy of pattern prediction.* Since the accuracy is very important for pattern prediction, we first examine the accuracy. Figure 13 shows the results. From Figure 13(a) and Figure 13(b) we can see that, CFL and CFLB have the (almost) same MAE and MRE value. This implies that, the branch-based prune approach (cf., Section 5.2) has almost no negative impact on the prediction accuracy, on average. Also, from the same figure, we can see that, FastPM achieves a *MRE* of 10% on average, indicating that a prediction accuracy of around 90%. This satisfies the common requirements for most of prediction applications.



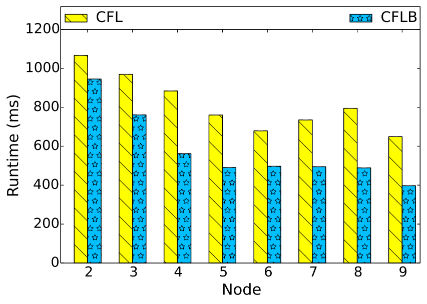
(a) MAE



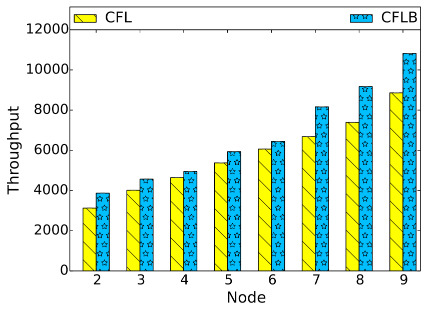
(b) MRE

Figure 13: Prediction performance: accuracy.

▷ *Runtime and throughput of pattern prediction.* To finish pattern prediction, we also need to execute the pattern matching firstly, and then use the PWS algorithm to achieve prediction (recall Section 7). In the following results, the runtime refers to the time interval between the time when a target pattern arrives at the system and the time when the system outputs the prediction result. Also, we report the throughput of the prediction. Figure 14 reports the experimental results. Similar to the results of pattern matching, when more computing resources are available, the average running time decreases (cf., Figure 14(a)) and the throughput increases (cf., Figure 14(b)). Note, however, that the runtime is slightly longer than that of pattern matching, since we needs to execute the PWS algorithm. Positively, the whole time for finishing a pattern prediction task is still pretty small — just about 0.4 seconds.



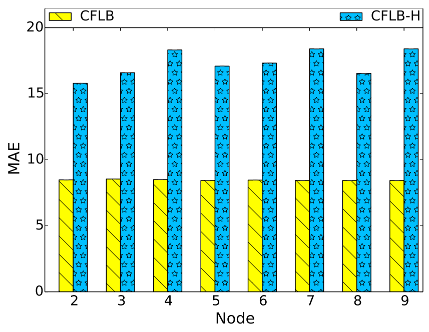
(a) Running time



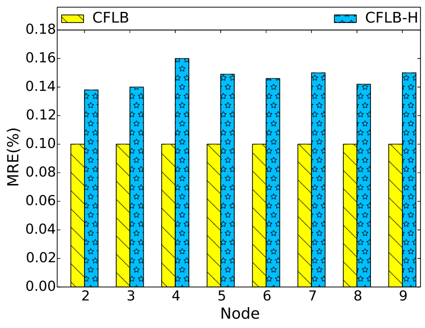
(b) Throughput

Figure 14: Prediction performance: runtime and throughput.

▷ *Comparing CFLB and CFLB-H.* Figure 15 shows the comparison results of CFLB and CFLB-H. It can been seen that, both the MAE and MRE values of CFLB-H are obviously larger than that of CFLB. This essentially reflects the effectiveness of the hierarchical pattern extraction approach, and thus further justifies the effectiveness of FastPM.



(a) MAE



(b) MRE

Figure 15: The results of CFLB and CFLB-H.

9. Related Work

Pattern matching, as a well-known problem, has been investigated extensively in the past decades [8, 9, 10, 11, 12, 13]. For example, Faloutsos et al. [11] proposed a fast pattern matching method based an indexing structure, instead of sequential matching. Argyros et al. [27] proposed a matching algorithm to find similar patterns under the transformation of time and amplitude. Agrawal et al. [28] studied the problem of frequent pattern mining on uncertain data. All these methods focused on matching or mining for the *static data*. They did not consider the *real-time* pattern matching and *dynamic update* (where the new matching tasks and the new patterns are received continuously), which are the focuses of our work. In addition, these works mainly studied pattern matching on *a single machine*.

▷ *Distributed pattern matching.* In order to process large-scale dataset, a few systems have been developed for distributed pattern matching in a cluster. For example, Ahmed and Boutaba [14] formalized the problem of distributed pattern matching and presented a peer-to-peer architecture, called DPMS. DPMS uses a Bloom filter to construct a hierarchy of indices and supports flexible queries involving partial and multiple keywords. Cubit [15] is a scalable peer-to-peer system that can efficiently find the k closest data items for any search key. Liu et al. [16] designed a pattern-finding algorithm based on the MapReduce [45] framework, in order to improve the efficiency in motif detection for prescription compatibility network. DI-matching [17] addressed the problem of incomplete pattern matching in a distributed mobile environment. It matched similar pattern using a distributed weighted Bloom filter. These existing distributed frameworks or systems [14, 15, 16, 45, 17] can perform parallel matching tasks efficiently. Nonetheless, these frameworks are *also* not suitable for *real-time* pattern matching and dynamic update (similar to those methods discussed at the beginning of this section), and thus they are difficult to be extended to real-time applications.

▷ *Real-time pattern matching.* There are already many papers studying the real-time pattern matching for streaming data. For example, Agrawal et al. [20] studied the pattern matching over event streams in RFID-based inventory management. They proposed a formal query evaluation model to offer precise semantics of event pattern queries. Yu et al. [25] proposed efficient algorithms (known as DIMine and CooMine) to discovery frequent co-occurrence patterns across multiple data streams. Vistream [22] supported interactive visual exploration of neighbor-based patterns in data streams. It provided a rich set of visual interfaces and interactions to enable real-time pattern exploration. Zhong et al. [13] developed a pattern discovery technique to extract patterns from text documents and address the pattern evolving problem. They designed a concept-based model to find similar text patterns. CEP^R [21] demonstrated a system to capture the semantic meanings of the matching results. It ranked the results in near real-time as a continuously refreshing view and helped users interact with the system. These existing "real-time methods" are usually processed in a *single machine* and execute *a single matching query* at one time. However, the matching efficiency might be significantly affected, due to the increasingly high velocity of event streams and the rising number of pattern matching queries. Our system is carefully designed to support a large volume of data updates and many simultaneous online matching tasks (e.g., 5000 matching queries).

10. Conclusion

In this paper, we developed a framework, called FastPM, for real-time pattern matching over a high volume of live data. Our framework defines the time dependent pattern, and uses the hierarchical pattern extraction approach to extract the patterns. These patterns are partitioned and deployed via the cluster-based algorithm and the frequency-aware algorithm. To process multiple matching tasks efficiently, FastPM integrates asynchronous and synchronous mechanisms together to collaborate with different steps in a pipeline job. Particularly, FastPM manages patterns using the local query index and employs a branch-based prune approach to enhance the matching efficiency. In addition, FastPM develops targeted strategies for pattern update in real-time. Furthermore, we examined the applications of FastPM. Empirical results based on real-world data (i.e., 54 billion web click tuples) demonstrated the efficiency and effectiveness of our system. Specifically, on average, FastPM responds to a matching query in 0.2 second, which is much more efficient than traditional methods. Moreover, FastPM is able to deal with 5,000 online matching queries simultaneously and still gains an average prediction accuracy of 90%. Also, the pattern update usually takes 0.03 second. In future, we plan to implement FastPM for more applications and verify its performance using more streaming data.

Acknowledgements

We would like to thank anonymous reviewers for their insightful comments. This research was partially supported by National Natural Science Foundation of China (No. 61702320, 61772200, 61602460, 61472453, U1401256, U1501252 and U1611264), Shanghai Pujiang Talent Program (No. 17PJ1401900), Shanghai Municipal Natural Science Foundation (No. 17ZR1406900), Shanghai Municipal Education Commission Funds of Young Teacher Training

Program (No. ZZSDJ17021), China Postdoctoral Science Foundation (No. 2016M600338), Specialized Fund of Shanghai Municipal Commission of Economy and Informatization (No.201602008).

References

[1] E. J. Keogh, P. Smyth, A probabilistic approach to fast pattern matching in time series databases., in: KDD, Vol. 1997, 1997, pp. 24–30.

[2] D. Yang, J. Cao, J. Fu, J. Wang, J. Guo, A pattern fusion model for multi-step-ahead cpu load prediction, Journal of Systems and Software 86 (5) (2013) 1257–1266.

[3] J. Thornton, M. Savvides, B. V. Kumar, A bayesian approach to deformed pattern matching of iris images, IEEE Transactions on Pattern Analysis and Machine Intelligence 29 (4) (2007) 596–606.

[4] M. Li, N. Cao, S. Yu, W. Lou, Findu: Privacy-preserving personal profile matching in mobile social networks, in: Proceedings IEEE INFOCOM 2011, IEEE, 2011, pp. 2435–2443.

[5] W. Fan, Graph pattern matching revised for social network analysis, in: ICDT, ACM, 2012, pp. 8–21.

[6] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, D. Savio, Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services, IEEE transactions on Services Computing 3 (3) (2010) 223–235.

[7] S. An, H. Yang, J. Wang, N. Cui, J. Cui, Mining urban recurrent congestion evolution patterns from gps-equipped vehicle mobility data, Information Sciences 373 (2016) 515–526.

[8] P. Weiner, Linear pattern matching algorithms, in: SWAT'08, IEEE, 1973, pp. 1–11.

[9] J. H. Friedman, J. L. Bentley, R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, ACM Transactions on Mathematical Software (TOMS) 3 (3) (1977) 209–226.

[10] F. Cohen, R. Abarbanel, I. Kuntz, R. Fletterick, Turn prediction in proteins using a pattern-matching approach, Biochemistry 25 (1) (1986) 266–275.

[11] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases, in: SIGMOD, Vol. 23, ACM, 1994.

[12] C. M. Bishop, Pattern recognition, Machine Learning 128.

[13] N. Zhong, Y. Li, S.-T. Wu, Effective pattern discovery for text mining, IEEE Transactions on Knowledge and Data Engineering 24 (1) (2012) 30–44.

[14] R. Ahmed, R. Boutaba, Distributed pattern matching: a key to flexible and efficient p2p search, IEEE Journal on Selected Areas in Communications 25 (1) (2007) 73–83.

[15] B. Wong, A. Slivkins, E. G. Sirer, Approximate matching for peer-to-peer overlays with cubit.

[16] Y. Liu, X. Jiang, H. Chen, J. Ma, X. Zhang, Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network, in: International Workshop on Advanced Parallel Processing Technologies, Springer, 2009, pp. 341–355.

[17] S. Liu, L. Kang, L. Chen, L. M. Ni, How to conduct distributed incompletepattern matching, IEEE Transactions on Parallel and Distributed Systems 25 (4) (2014) 982–992.

[18] D. Deng, G. Li, S. Hao, J. Wang, J. Feng, Massjoin: A mapreduce-based method for scalable string similarity joins, in: ICDE, IEEE, 2014, pp. 340–351.

[19] M. M. Rashid, I. Gondal, J. Kamruzzaman, Dependable large scale behavioral patterns mining from sensor data using hadoop platform, Information Sciences 379 (2017) 128–145.

[20] J. Agrawal, Y. Diao, D. Gyllstrom, N. Immerman, Efficient pattern matching over event streams, in: SIGMOD, ACM, 2008, pp. 147–160.

[21] J. Gu, J. Wang, C. Zaniolo, Ranking support for matched patterns over complex event streams: The cepr system, in: ICDE, IEEE, 2016, pp. 1354–1357.

[22] D. Yang, Z. Guo, Z. Xie, E. A. Rundensteiner, M. O. Ward, Interactive visual exploration of neighbor-based patterns in data streams, in: SIGMOD, ACM, 2010, pp. 1151–1154.

[23] Z. Zhang, J. Jiang, X. Liu, R. Lau, H. Wang, R. Zhang, A real time hybrid pattern matching scheme for stock time series, in: ADC, Australian Computer Society, Inc., 2010, pp. 161–170.

[24] L. Woods, J. Teubner, G. Alonso, Real-time pattern matching with fpgas, in: ICDE, IEEE, 2011, pp. 1292–1295.

[25] Z. Yu, X. Yu, Y. Liu, W. Li, J. Pei, Mining frequent co-occurrence patterns across multiple data streams., in: EDBT, 2015, pp. 73–84.

[26] M. Zihayat, A. An, Mining top-k high utility patterns over data streams, Information Sciences 285 (2014) 138–161.

[27] T. Argyros, C. Ermopoulos, Efficient subsequence matching in time series databases under time and amplitude transformations, in: IEEE ICDM, IEEE, 2003, pp. 481–484.

[28] C. C. Aggarwal, Y. Li, J. Wang, J. Wang, Frequent pattern mining with uncertain data, in: SIGKDD, ACM, 2009, pp. 29–38.

[29] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: PODS, ACM, 2002, pp. 1–16.

[30] Spark, http://spark.incubator.apache.org/.

[31] Storm, https://github.com/nathanmarz/storm.

[32] J. Han, J. Pei, M. Kamber, Data mining: concepts and techniques, Elsevier, 2011.

[33] M. T. Özsu, P. Valduriez, Principles of distributed database systems, Springer Science & Business Media, 2011.

[34] Kafka, http://kafka.apache.org/.

[35] T. H. Cormen, Introduction to algorithms, MIT press, 2009.

[36] M. D. Berg, O. Cheong, M. V. Kreveld, M. Overmars, Computational geometry: algorithms and applications, Third Edition, Springer, Berlin, 2008.

[37] P. Révész, Random walk in random and non-random environments, World Scientific, 2005.

[38] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Rubin, Bayesian data analysis, Vol. 2, Chapman & Hall/CRC Boca Raton, FL, USA, 2014.

[39] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM computing surveys (CSUR) 41 (3).

[40] S4, http://incubator.apache.org/s4/.

[41] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, H. T. Shen, Distributed shortest path query processing on dynamic road networks, VLDB Journal 26 (3) (2017) 399–419.

[42] Flink, http://flink.apache.org/.

[43] J. Lijffijt, P. Papapetrou, K. Puolamäki, Size matters: Finding the most informative set of window lengths, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2012, pp. 451–466.

[44] S. Wang, K. Kam, C. Xiao, S. Bowen, W. A. Chaovalitwongse, An efficient time series subsequence pattern mining and prediction framework with an application to respiratory motion prediction, in: AAAI, 2016.

[45] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: OSDI, 2004, pp. 10–10.