PStream: a Popularity-aware Differentiated Distributed Stream Processing System

Hanhua Chen, *Member, IEEE,* Fan Zhang, *Student Member, IEEE,* Hai Jin, *Fellow, IEEE*

Abstract—Real-world stream data with skewed distributions raises unique challenges to distributed stream processing systems. Existing stream workload partitioning schemes usually use a "one size fits all" design, which leverages either a shuffle grouping or a key grouping strategy for partitioning the stream workloads among multiple processing units, leading to notable problems of unsatisfied system throughput and processing latency. In this paper, we show that the key grouping based schemes result in serious load imbalance and low computation efficiency in the presence of data skewness while the shuffle grouping schemes are not scalable in terms of memory space.

We argue that the key to efficient stream scheduling is the popularity of the stream data. We propose PStream, a popularity-aware differentiated distributed stream processing system which assigns the hot keys using shuffle grouping while assigns rare ones using key grouping. PStream leverages a novel light-weighted probabilistic counting scheme for identifying the current hot keys in dynamic real-time streams. The scheme is extremely efficient in computation and memory consumption, so that the predictor based on it can be well integrated into processing instances in the system. We further design an adaptive threshold configuration scheme, which can quickly adapt to the dynamical popularity changes in highly dynamical real-time streams. We implement PStream on top of Apache Storm and conduct comprehensive experiments using large-scale traces from real-world systems to evaluate the performance of this design. Results show that PStream achieves a 2.3× improvement in terms of processing throughput and reduces the processing latency by 64% compared to state-of-the-art designs.

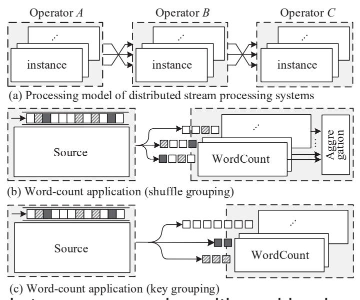
Index Terms—distributed stream processing system; skewness; load balance

1 Introduction

The recent advances in distributed stream processing systems such as Storm [1], Heron [2], Spark Streaming [3], S4 [4], and Samza [5], bring the community great capability to process extremely huge volumes of unbounded and continuous data streams in real-time with clusters [6, 7]. Different applications based on stream processing are widely deployed, *e.g.*, social event detection [8, 9], on-demand ride-hailing [10], and real-time risk detection [11].

Distributed stream processing systems achieve efficient pipeline parallelism [12] for the stream applications. Specifically, in distributed stream processing systems, to achieve high task parallelism and pipeline parallelism, an application is commonly modeled as a directed acyclic graph. In this graph, a vertex is a processing element, which represents an operator for processing a specific user-defined logic in a stream operation. An edge is a channel that routes data flow between operators. The data flow, in the form of a sequence of tuples, traverses along the edges and forms a stream. To further improve the system throughput, distributed stream processing systems achieve data parallelism [13] by creating multiple instances for an operator and making them work in parallel (see Fig. 1(a)). Each of these instances executes the same processing logic. Accordingly, all the tuples from the upstream processing element will be partitioned into multiple sub-streams and scheduled to different instances.

Existing workloads partitioning strategies in a distributed stream processing system include shuffle grouping and key grouping [14]. With shuffle grouping, an instance of an upstream processing element partitions the workloads of the stream among all the downstream instances in a round robin style. It thus generates an even distribution of the workloads. However, such a scheme suffers the problem of scalability in terms of memory. Figure 1(b) shows an example of the word-count application processed with shuffle grouping. The source node assigns the stream of tuples to the downstream instances in a round robin style, irrespective of the keys of the words. Each downstream instance potentially needs to keep counters for all the keys in the stream before achieving the aggregated terms frequencies. It is clear that the consumed memory for the application grows linearly with the parallelism level, *i.e.*, the number of instances of the downstream processing element. Given N unique keys and M word-count instances in the system, the amount of required memory size is $O(MN)$. Such a shuffle grouping partitioning scheme is hard to scale for large-scale workloads. Moreover, as shown in Fig. 1(b), shuffle grouping needs an additional M-way aggregation step to merge the counts of each key. The frequent connections to all the M instances are also expensive and limit the scalability of a real-time processing system.


Operator A
Operator B
Operator C
instance
instance
instance
(a) Processing model of distributed stream processing systems
Source
WordCount
Aggregation
(b) Word-count application (shuffle grouping)
Source
WordCount
(c) Word-count application (key grouping)
Fig. 1. Distributed stream processing with workloads partition strategies

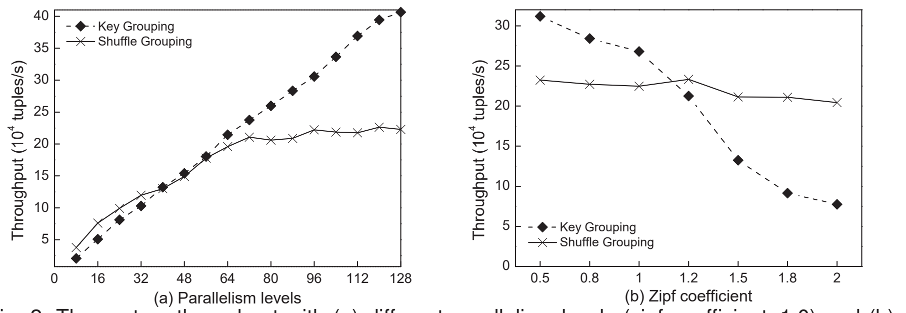


Fig. 2. The system throughput with (a) different parallelism levels (zipf coefficient=1.0) and (b) different skewness (parallelism level=64)

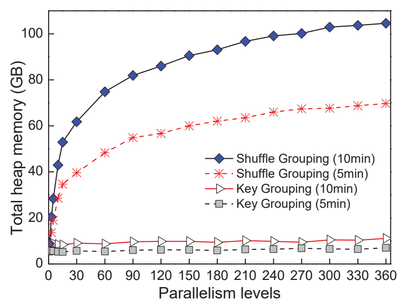


Fig. 3. The heap memory consumption

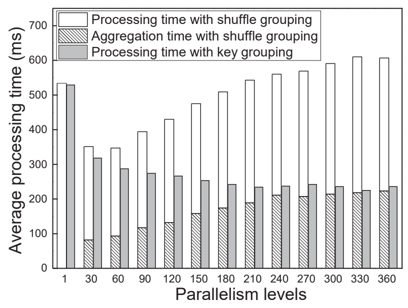


Fig. 4. The processing and aggregation time

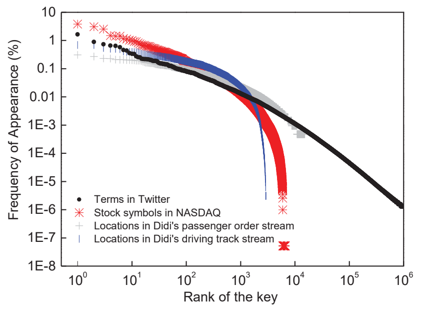


Fig. 5. Skewed distribution in real-time datasets

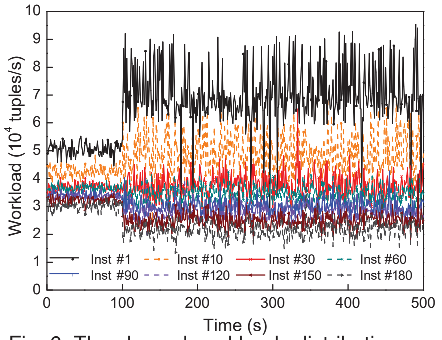


Fig. 6. The skewed workloads distribution

We examine the performance of the shuffle grouping scheme in greater detail with an experiment. In the experiment, we vary the parallelism levels by creating different numbers of downstream word-count instances. With each parallelism level, the system adjusts the source's tuple emitting speed so that it can put the examination of the system to its performance limit. The result in Fig. 2(a) shows that when the parallelism level is low (*i.e.*, there is a small number of instances), the system throughput increases with the parallelism level. However, when the number of instances continues to increase, the system throughput stops increasing. Then we fix the tuple emitting speed and examine the total memory consumption of all the instances. We vary the parallelism levels and record the heap memory of each instance after different time intervals. Figure 3 shows that the memory consumption of shuffle grouping significantly increases with the parallelism level.

We also examine the average processing time and the average aggregation time of all the tuples in each parallelism level during shuffle grouping. Figure 4 shows that the aggregation cost of shuffle grouping increases with the parallelism level, accounting for 30.48% to 60.46% of the total processing time. It is clear the aggregation costs and the memory consumption result in the scalability problem of the shuffle grouping strategy.

A second partition strategy is key grouping (Fig. 1(c)), which uses a hashing based scheme to partition the key space nearly evenly among all the downstream instances. It thus guarantees that the tuples with the same key are scheduled to the same downstream instance. It is clear, such a partition scheme needs an amount of $O(N)$ memory size, where N is the number of unique keys in the data stream. Figure 2(a) shows when the parallelism level increases, the system throughput with key grouping keeps increasing. However, such a scheme leads to load imbalance due to the skewed distribution in various real-world datasets [15].

We collect real-world traces from different systems to show common data skewness. First, we collect two months' traces from Twitter (Dec. 2015 to Jan. 2016). We plot the distribution of term frequency in Fig. 5 (stop words removed). The result shows that an extremely small fraction of 3‰ words accounts for more than 80% total term frequency. Second, we collect the stock exchange stream data from NASDAQ [16] (during Apr., 2017). Figure 5 also shows the distribution of the stock symbols in the NASDAQ stock exchange stream. The statistics show that the NASDAQ data also follows a highly skewed distribution, in which 80% exchanges happen among a fraction of 8% stock symbols. Third, we collect a large-scale dataset of Didi Chuxing [17] (in Chengdu, China, during Nov. 2016), which is the most popular on-demand ride-hailing platform in China. The dataset consists of two kinds of data streams. One stream contains three billion driving track records of taxis. The other stream contains seven million query orders of passengers. Figure 5 shows that 80% passenger orders happen at about 20% locations, while 80% taxi driving tracks appear at 24% of the locations. With such a striking feature of skewness, a stream processing system with key grouping strategy (Fig. 1(c)), assigns much higher loads to a few downstream instances than other instances. This results in poor computation resource utilization and thus greatly degrades system performance.

To examine the problem, we conduct the experiment with the word counting application on the Twitter dataset. In the experiment, we generate 180 word-count instances and use key grouping for data partitioning. We count the number of tuples of each instance in every second. Figure 6 shows the workloads of different instances are highly skewed. To more carefully examine the performance of the key grouping strategy in the presence of data skewness, we conduct extended experiments for key grouping by synthetically generating seven datasets following Zipf distributions with the coefficients varying from 0.5 to 2.0. Figure 2(b) shows that the throughput decreases significantly with the increase of the level of skewness of the stream data during key grouping. The system load imbalance due to data skewness leads to significant performance degradation. In contrast, the performance of shuffle grouping is much more stable in the presence of skewness.

Based on the above analysis, we argue that the key for efficient distributed stream processing is to differentiate the popularity of keys. The key grouping scheme is memory efficient for a large number of rare keys. However, it suffers from the serious problem of load imbalance caused by the hot keys. On the contrary, the shuffle grouping scheme is able to balance the heavy workloads caused by hot keys. However, it does not scale due to the memory cost and the additional aggregation cost because of the large number of rare keys. Based on this insight, we propose a novel popularity-aware differentiated stream processing system called PStream. PStream identifies the popularity of keys in the stream data and uses a differentiated partitioning scheme. For hot keys, PStream chooses the shuffle grouping strategy, while for rare keys, it selects key grouping.

However, identifying the popularity of the keys in a distributed stream processing system is challenging. It is difficult to meet the rigorous requirements of both computation and memory efficiency needed by a distributed stream processing system. To address this issue, in PStream, we design a novel light-weighted predictor for identifying the current hot keys in the real-time data streams. The predictor is based on a proposed novel computation and memory efficient probabilistic counting scheme. It can be well integrated into a processing element instance. We further design an efficient adaptive threshold configuration scheme, which adaptively configures the threshold to optimize system performance by balancing the processing time and memory usage.

We implement PStream on top of Apache Storm [1] and conduct comprehensive experiments using large-scale traces from real-world systems. The results show that PStream achieves a 2.3× improvement in terms of processing throughput and reduces the processing latency by 64% compared to existing schemes.

The main contributions of this work are threefold:

- We design and implement PStream, a novel popularity-aware differentiated stream processing system.
- We design a light-weighted predictor in PStream which efficiently identifies the current hot keys in the dynamic real-time data streams.
- We conduct comprehensive experiments using large-scale real-world data set to demonstrate the efficiency of this design.

The rest of the paper is structured as follows. Section 2 reviews the related work. Section 3 presents the system design of PStream. Section 4 analyzes the hot key predictor. We present the implementation of PStream on top of Apache Storm in Section 5. Section 6 evaluates the performance of PStream. Section 7 concludes the paper.

2 Related Work

In this section, we first review the background of distributed stream processing systems. Then, we introduce existing stream workloads partition designs as well as previous popularity estimation techniques.

Distributed Stream Processing Systems. With the emergence of Big Data applications, stream processing becomes popular, where an unbounded sequence of data tuples generated in real-world applications are pushed to servers for real-time processing. Traditional centralized solutions are replaced by distributed stream processing systems [1–5, 18]. In a distributed stream processing system, a processing element receives a sequence of data tuples from its input queue, performs specific operations on the tuples, and produces the output to the output queue. The processing elements are connected to each other to form a directed acyclic graph to represent the processing logic for a stream application. The data flows are modeled as edges between processing elements. Each processing element can have a set of instances running in parallel. Thus, an upstream processing element instance divides the output tuples among multiple downstream instances to improve the system throughput using different partitioning strategies.

Stream Partition Strategies for Data Parallelism. Recently, the issue of stream workloads partitioning has attracted much research interest in the community [14, 19]. Castro *et al.* [20] propose an instance state management mechanism to keep monitoring all the processing element instances. When an instance becomes the performance bottleneck, *i.e.*, its CPU utilization is higher than a threshold for a period of time, the system dynamically creates a replica instance for it. Such a scheme raises heavy overhead for monitoring and replicating. Balkesen *et al.* [21] propose a hash based stream data partition scheme to achieve load balance. In their scheme, the tuples associated with a hot key are evenly split into a number of parts, where the number of parts is proportional to the frequency of the key. They leverage sampling techniques to obtain the frequency of a hot key. However, the sampling based scheme has unsatisfied precision for distributed processing systems. Moreover, their design cannot adapt to the dynamical changes of the frequency of a key in real-world streams.

Gedik [22] proposes an index-based stream workloads partition scheme. Initially, an upstream instance uses key grouping to partition all the keys to the downstream processing element instances. During stream processing, the system keeps tracking the highly frequent keys and maintains an index for optimizing the mapping of these keys to the downstream instances. However, with the changes of the key popularity in a real-time stream, the index needs to be adjusted frequently. Moreover, each time when adjusting the index, the system needs to briefly suspend the stream processing to migrate the state associated with the keys whose mapping has changed. Thus, it incurs extra high processing latency for the application. Rivetti *et al.* [19] further propose a mapping function for the highly frequent keys. However, the re-computation of the mapping functions is costly. At the same time, their design needs expensive synchronizations for computing the mapping function for each key among multiple upstream instances.

To address the load imbalance problem caused by hot keys in the stream, Nasir *et al.* propose the partial key grouping scheme which splits the workloads associated with a key [14]. In their design, every upstream instance keeps a vector to record the number of tuples it has sent to different downstream instances. For an arriving tuple, the upstream instance selects two downstream instances, and then sends the tuple to the one with a lighter workload estimated according to the local vector. Their key splitting scheme attempts to leverage the famous principle of "power of two choices" [23], which is initially described as a supermarket model. Assuming the customers stream arrives following a Poisson distribution with $\lambda < 1$ at a collection of n counter servers. Each customer independently chooses two servers uniformly at random and waits for service at the one with the fewer customers. The principle shows that for any fixed period of time T, the length of the longest queue in the interval $[0, T]$ is $\frac{\log \log n}{\log 2} + O(1)$ with high probability. Thus, the loads are much better balanced than the case that all the customers choose a server randomly, where the length of the longest queue is $O(\log n)$. Note in the principle to achieve load balance, two servers should be randomly and uniformly selected as candidates for any custom. However, they fix the set of two downstream instances for all the tuples associated with a given key. Although the heavy workloads caused by a hot key can be split into two parts, the key splitting scheme cannot achieve satisfying load balance due to the violation of the assumptions of the principle. Indeed, the difference between the maximum load and the average could be $O(k/n)$ among instances, where k is the total number of tuples.

Popularity Estimation in Streams. The key for efficient stream data scheduling is to efficiently estimate the popularities of the data in a stream and especially identify *heavy hitters*. Recently, finding heavy hitters in the stream has attracted a lot of research efforts. Existing schemes can be classified into two types [24], the counter based schemes [25, 26] and the sketch based schemes [27, 28].

A counter based scheme relies on counting the appearances of tuples associated with each key [24]. The straightforward counting scheme keeps a counter vector for every key. Such a scheme needs an amount of $O(N)$ memory size for the scheduler of each upstream instance, where N is the number of unique keys in the stream. Due to the large popularity of rare keys in real-world streams, maintaining and updating the counter vector of all the keys in a real-time style is prohibitively costly in memory and computation consumption. To save memory cost, an effective strategy is to let the counter vector exclude the rare keys which have a low possibility to become hot ones. Following this way, Metwally *et al.* [26] propose a scheme called SpaceSaving, which keeps a fixed number of K keys with the highest accumulative count values in the counter vector. When a tuple associated with a new key out of the K keys arrives, the scheme replaces the key with the lowest value in the counter vector by the new key. With such a scheme, it is proved that a key with an accumulated count higher than $\frac{W}{K}$ will be kept in the counter vector, where W is the total number of tuples [26]. However, such an estimation does not necessarily reflect the recency of the popularity of a key.

To track the most recent frequent keys, Ran *et al.* propose a sliding-window based scheme [29]. Specifically, they partition the stream of tuples into frames. They use a SpaceSaving structure to identify the most frequent keys in the current frame and reset all the counters once a new frame starts. They further partition a frame into blocks, and save the hot keys as well as the block id when the key is identified to be a popular one in each frame. Such a scheme will discard the accumulated count of a key if it is not identified to be a hot key. Therefore, for keys that are unpopular in the past but currently becoming popular (*e.g.*, newly generated hot topics), their accumulated counts will be discarded before they are identified. Then these keys will be never identified or be identified with a long delay.

A sketch based scheme leverages a set of hash functions to project a large number of keys to a few counters. Cormode *et al.* propose count-min sketch [27], which employs a number of d hash functions. Each hash function can project the keys to a number of w values. The count-min sketch uses $d \times w$ counters, while each key corresponds with d of them. The estimated count of each key is the minimal value of the corresponded d counts which may be slightly larger than the accuracy count due to the hash collisions. To further support the heavy hitter queries, Cormode *et al.* leverage a dyadic ranges method [27]. Specifically, if there are at most N keys, it builds $\log N + 1$ count-min sketches counting for different ranges. The scheme uses the grouping test method to search heavy hitters with $O(\log N)$ time. Such a scheme requires that the entire key set is known in advance.

Recently, Shrivastava *et al.* proposed Ada-CMSketch [28] to support querying for the most recent frequencies of keys. Ada-CMSketch is a count-min sketch that leverages a digital Dolby noise reduction mechanism to reduce the influence of the history information. However, such a design needs to combine the original key and the time-stamp for hashing. This leads an original key to be projected to much more counters and thus the collisions will be more critical. When dealing with the large number of incoming tuples, the accuracy decreases significantly.

3 Differentiated Stream Processing

In this section, we first describe the overview of the PStream design. Then, we present the main idea of how PStream predicts the hot keys in the dynamic data streams.

3.1 Design Overview

Figure 7 shows the overview of PStream. The main idea of the system is to design and deploy a light-weighted hot key predictor to support the scheduling strategy selection. PStream consists of two components: 1) an independent predicting component for identifying real-time hot keys, and 2) a scheduling component in each processing element instance. The predicting component is a standalone component, which collects the information of all the tuples globally and uses this information to identify the real-time hot keys. The scheduling component is embedded in each processing element instance, which stores the identified hot keys and supports fast hot key querying and efficiently scheduling.

Specifically, in the predicting component, we propose a novel probabilistic counting scheme to precisely identify real-time hot keys. The probabilistic counter is computationally efficient because only a very slight cost for generating digits "0" or "1" is incurred by every tuple. With the simple operation, the predictor achieves probabilistic counting of the tuples associated with a key. The probabilistic counting operation can be performed with multiple instances in parallel. All the probabilistic counting instances work independently with the instances of the user logic. Therefore, the predicting component will not affect the original stream processing procedure. Only a very small fraction of keys that are detected to be potential hot ones will be sent and recorded in a synopsis structure.

Usually, keys' popularities in the stream change over time. Thus, we also design a popularity decline scheme in the synopsis structure. As time flies, the synopsis probabilistically decreases the estimated frequency of a key, until the key is evicted from the synopsis. When a key is evicted from the synopsis, it will be regarded as an out-dated hot key.

The scheduling component consists of a memory efficient hot key filter and a differentiated scheduler. The hot key filter is made of a *Counting Bloom filter* (CBF) [30]. When a key in the synopsis of potential hot keys is determined to be a current hot key, it will be inserted into the CBFs of all the involved processing elements. Accordingly, when a tuple is determined to be an out-dated hot key, it will be deleted from the CBFs. With the help of the hot key filter, the differentiated scheduler can quickly check whether the key of an arriving tuple is popular or not. If the key is contained in the CBF, the scheduler assigns the tuple with a shuffle strategy; otherwise, the scheduler assigns the tuple with a hash strategy. Such an operation incurs almost no latency to the original stream processing. To ensure the correctness of the processing results, the partial processing results of each shuffled tuple will be aggregated by an additional aggregator (the aggregator can also have multiple instances) after processing. Specifically, a shuffled tuple will be attached with a Boolean "shuffle" mark. According to the mark, each downstream processing element instance determines to forward the processing result to either an aggregator or a next downstream processing element. When the differentiated scheduler in the upstream instance switches the scheduling strategy for a certain key from hashing to shuffling, it will inform the corresponding processing element instance to forward the processing result as a partial result to the aggregator. On the contrary, when the strategy switches from shuffling to hashing, the aggregator forwards the aggregated result as a new tuple to the corresponding instance by hashing. This incurs a slight overhead for each switching, where only one more connection is added and one extra tuple is processed for a certain instance.

With the above design, PStream is computation and memory efficient. By integrating the differentiated scheduler in a processing element instance, PStream can greatly improve the system throughput. The differentiated scheduler assigns the tuples with hot keys using shuffle grouping, while it assigns the tuples with rare keys using key grouping. On one hand, for hot keys, the heavy workloads incurred by them can be evenly shared by all the downstream instances. Such a scheme effectively avoids possible straggles caused by hot keys, which is the root cause of system throughput degradation as shown in Fig. 2(b). On the other hand, for the rare keys, the hash based scheme partitions the key space among multiple downstream instances. Each instance only needs little memory to store the states of the fraction of the rare keys assigned to it. Hence, we can upgrade the level of data parallelism by generating more processing instances to improve system throughput.

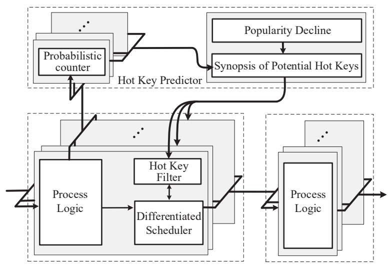


Fig. 7. Architecture of PStream

3.2 Hot Keys Predictor

Due to the rigorous requirements in system efficiency for a distributed streaming processing system, identifying the current hot keys in a highly dynamic real-time stream has three major difficulties. First, the representation of the set of hot keys needs to be space efficient so that each processing element instance can load the set in memory to support strategy selection for tuple scheduling. Second, the predictor should support quickly checking the key of the current tuple in real-time stream against the set of hot keys to guarantee the processing efficiency. Third, the predictor needs to be aware of the dynamic changes of the popularities of the keys over time, which is common in real applications [31].

To meet the above requirements, the predictor of PStream uses a two-level architecture (Fig. 8). The first level identifies a set of potential hot keys by leveraging a novel probabilistic counting scheme. The probabilistic counting values of the potential hot keys are stored in a synopsis. The second level of the predictor stores the identified hot keys in a space efficient CBF. By using such a succinct set representation structure, the hot keys filter can be efficiently loaded in the memory of a processing element instance.

An important component of the hot key predictor is the probabilistic counter. It performs the coin flipping experiments for each arriving tuple. Specifically, for a key k_i, the predictor keeps flipping a coin until it sees the first tail. It counts the times it sees heads before the first tail appears, and saves this count in a value t_i. Intuitively, the key k_i frequently appears in a stream, it will do more coin flipping experiments than a rare key. Thus, its expected experimental value t_i will be larger than the expected value of a rare key. Moreover, the frequency with k_i is roughly equal to 2^{t_i}. Following this intuition, the predictor can achieve an estimated count of tuples associated with k_i. A large value of t_i indicates that k_i is likely to be a hot key. Note that for each tuple the coin flipping experiment is conducted independently. Therefore, a rare key also has a probability to obtain a large experiment value. To alleviate the false positive, only when k_i obtains large experiment values more than once, the predictor determines that k_i is a current hot key. This is because a rare key seldom performs the experiment, and it is hard to meet the low probability event more than once. Thus, we can achieve a much more precise prediction of hot keys. We will give the formal proof of the lower bound of the precision in Section 4.1.

The synopsis shown in Fig. 8 consists of a set of bit vectors. Each bit vector represents a probabilistic counter of a potential hot key. When the data flow arrives, the

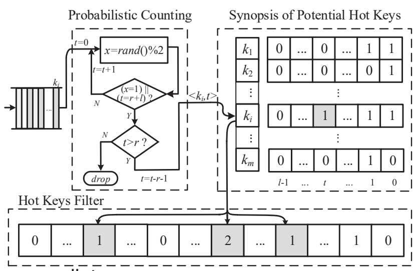


Fig. 8. Hot keys predictor

predictor performs a probabilistic counting operation for k_i. If the experiment value t_i is above a given threshold r, the predictor checks whether the key k_i is in the synopsis. If k_i is not in the synopsis, the predictor generates a new bit vector with length of l, $bitvec\_k_i$, with all the l bits initially set to "0". It then sets the $(t_i$-r-$1)^{th}$ bit of $bitvec\_k_i$ to "1" and inserts the key-value pair (k_i, $bitvec\_k_i$) into the synopsis. If k_i is already in the synopsis, the predictor simply sets the $(t_i$-r-$1)^{th}$ bit of $bitvec\_k_i$ to "1". In the design, the predictor only flips the coin up to (l+r) times, i.e, the maximal value of t_i is limited to (l+r). Algorithm 1 shows the details of the procedure. In Algorithm 1, "submit" means forwarding the tuple to the downstream processing elements according to the stream processing topology.

We store the experiment values of the possible hot keys in the synopsis of potential hot keys. Each bit vector in the synopsis has at least one bit set to "1". To achieve precise prediction, the predictor only inserts an identified hot key k_i, *i.e.*, the associated $bitvec\_k_i$ with more than one "1" bits, into the CBFs. The space efficient hot keys filter is loaded in the memory of the processing element instances. At the same time, due to the computation efficiency of a CBF, the hot key filter supports fast set membership testing within constant time, *i.e.*, the predictor can quickly decide whether the key of the current tuple is hot or not. In Section 4.1, we theoretically analyze the precision of the predictor.

Algorithm 1 Hot Key Predictor

```
1:  constant int $r$;
2:  **ProbabilisticCounting** (Tuple <Key $k$, Value $v$ >)
3:      counter $t \leftarrow 0$;
4:      coin $x \leftarrow 0$;
5:      **while** ($x == 0$ && $t$++ $< r + 1$)
6:          $x \leftarrow rand()\%2$;
7:      **if** ($t > r$)
8:          **submit** tuple $<k, t$-$r$-$1>$;
9:
10: SynopsisHashMap<Key, BitVector> $map$;
11: **Synopsis** (Tuple <Key $k$, Value $v$ >)
12:     **if** ($bitvec\_k \leftarrow map$.get($k$) == null)
13:         $bitvec\_k \leftarrow$ an $l$-length bit vector;
14:     set the $v^{th}$ bit of $bitvec\_k$ to 1;
15:     $map$.put($k$, $bitvec\_k$);
16:     **if** (the number of "1"s in $bitvec\_k \geq 2$)
17:         **submit** tuple $<k$, "append"$>$;
18:         choose a fixed number of <Key, BitVector> pairs
            from $map$ randomly;
19:     **for** each chosen $<ki, bi>$
20:         $bi \leftarrow bi >> 1$;
21:         **if** (the number of "1"s in $bi$ == 1)
22:             **submit** tuple $<ki$, "delete"$>$;
```

4 Theoretical Analysis

In this section, we first theoretically analyze the precision, reliability, and stableness of the proposed hot keys predictor. Then, we present how to cope with the dynamic changes of the popularities of the keys in real-time streams.

4.1 Precision of the Hot Key Prediction

With the hot key predictor, a key is identified as a hot key if and only if it appears more frequently than a given threshold in the stream. It is clear that in the PStream predictor design, the probability for a coin flipping experiment to obtain a result value $t > r$ (r is a given threshold) is as low as $(\frac{1}{2})^r$. However, if we repeat the experiment N_0 times, the expected probability is $1 - (1 - (\frac{1}{2})^r)^{N_0}$, monotonically increasing with N_0. As aforementioned, obtaining a large experiment value does not necessarily mean finding a hot key because the coin flipping experiments of tuples are conducted independently. Thus, in our exponential counting scheme, the predictor detects a key to be a hot key if and only if the key obtains a large experiment value more than once. With this design, the coin flipping results are dependent on the keys. If a key has obtained large experiment values more than once, it has a very high probability to be a hot key. In the following, we formally present the lower bound of the recall rate and the upper bound of the false positive rate of the exponential counting scheme in the PStream predictor design.

Theorem 1. *If a key appears N_0 times, the probability that only one or none of the obtained experiment value is larger than a given threshold r is upper bounded by $(1 + \frac{N_0-1}{2^r}) \times e^{-\frac{N_0-1}{2^r}}$.*

Proof. For a hot key appearing N_0 times, the coin flipping experiment is performed N_0 times. The probability that all its obtained experiment values are no larger than r is $(1 - \frac{1}{2^r})^{N_0}$. The probability that only one obtained experiment value is larger than r is $C^1_{N_0} \times (1 - \frac{1}{2^r}) \times \frac{1}{2^r}$. Thus, the probability that this key is identified as a rare key can be computed by the following equation.

$$\begin{aligned} P(N_0) =& (1 - \frac{1}{2^r})^{N_0} + C^1_{N_0} \times (1 - \frac{1}{2^r}) \times \frac{1}{2^r} \\ =& (1 + \frac{N_0 - 1}{2^r}) \times (1 - \frac{1}{2^r})^{N_0 - 1} \\ <& (1 + \frac{N_0 - 1}{2^r}) \times e^{-\frac{N_0-1}{2^r}} \end{aligned} \tag{1}$$

Theorem 1 is proved. ■

From Theorem 1, we can see that the upper bound value monotonically decreases with the value of $\frac{N_0-1}{2^r}$. Therefore, the probability that an actually hot key is identified as a rare key is very low. Table 1 shows the lower bound of the recall rate with different values of $\frac{N_0-1}{2^r}$. For example, for the key appearing more than 2^{r+3} times, the lower bound of the recall rate is higher than 99.7%.

Theorem 2. *If there are a number of M keys with each of them appearing at most N_1 times, the probability that at least one key*

TABLE 1
Lower bound of the recall rate

Value of $\frac{N_0-1}{2^r}$	4	5	6	7	8
Lower bound of recall rate	90.85%	95.96%	98.26%	99.27%	99.70%

obtain two or more experiment values which are larger than a given threshold r is upper bounded by $1-(1-(\frac{N_1}{2^r})^2)^M$.

Proof. From Eq. (1), for a rare key appearing N_1 times, the probability that only one or none of the obtained experiment value is larger than r is $P(N_1)$. For all the M rare keys, the probability that all of them obtain one or no large experiment value is $P(N_1)^M$. Thus, the probability that at least one rare key is identified as hot can be computed by:

$$\begin{aligned} P(M,N_1) =& 1-P(N_1)^M \\ =& 1-((1+\frac{N_1-1}{2^r})\times(1-\frac{1}{2^r})^{N_1-1})^M \\ <& 1-((1+\frac{N_1-1}{2^r})\times(1-\frac{N_1-1}{2^r}))^M \\ =& 1-(1-(\frac{N_1-1}{2^r})^2)^M < 1-(1-(\frac{N_1}{2^r})^2)^M \end{aligned} \quad (2)$$

Theorem 2 is proved. ■

From Theorem 2, we can obtain that the upper bound value monotonically decreases with the value of $\frac{N_1}{2^r}$. Therefore, with our exponential counting technique, the probability that a rare key is identified as a hot key is also very low. For the key appearing less than 2^{r-3} times, the upper bound of the false positive rate is lower than $P(1,2^{r-3}) < 1.56\%$.

We consider the case that there are a total number of 2^r tuples with rare keys performing the coin flipping experiments, we obtain $M \times N_1 = 2^r$. Thus, the the upper bound of $P(M,N_1)$ in Eq. (2) can be computed by,

$$P(M,N_1) < 1-(1-(\frac{N_1}{2^r})^2)^M = 1-(1-(\frac{N_1}{2^r})^2)^{\frac{2^r}{N_1}} \quad (3)$$

Table 2 shows the upper bound of the false positive rate with different $\frac{2^r}{N_1}$. For instance, if there are more than 2^7 rare keys and each rare key appears less than 2^{r-7} times, the upper bound of the false positive rate is less than 0.78%.

TABLE 2
Upper bound of the *false positive* (FP) rate

Value of $\frac{2^r}{N_1}$	2^3	2^4	2^5	2^6	2^7
Upper bound of FP rate	11.84%	6.07%	3.08%	1.55%	0.78%

4.2 Adaption to Dynamic Popularity Changes

In a real-time stream, the popularities of the keys in the stream dynamically change over time [31]. For efficiently processing the dynamic streams, PStream examines the current arriving rate of a key rather than an accumulated count of a key as the previous design [14]. Considering a given key in the stream, if it rarely appears in a current period of time, it will not incur a heavy workload, no matter how popular it used to be. The basic idea of our popularity adaption strategy is to probabilistically decrease the counters in the synopsis each time when the synopsis is updated. Thus, with time flying, even a hot key's value in the bit vector will keep decreasing unless it continues to appear frequently in the arriving stream. Specifically, to adapt to the dynamic popularities changing of keys in the stream, we design a decline mechanism on top of the synopsis over time.

Given a specified declining rate for all the keys in the synopsis, if a certain key's arriving rate is less than the decline rate, its value stored in the bit vector will keep decreasing until all the bits are set to "0". As aforementioned in Section 4.1, the synopsis uses the total number of "1" bits to identify the hot keys. PStream decreases the value of the bit vector without increasing the total number of "1" bits each time. Specifically, PStream performs the bit-wise right shift operation on each bit vector in the synopsis at a certain rate. It is clear that after the bit-wise right shift operation, the number of "1" bits in the bit vector is monotonically non-increasing. If no new bits are set to "1" in a key's bit vector, after at most l bit-wise right shifts (where l is the length of the bit vector), this associated key will be evicted from the synopsis. To control the declining rate, we set a probability p ($0 < p < 1$) for performing the right shift operation on each bit vector, whenever the bit vector of any key is generated or updated. Specifically, each time the synopsis selects P bit vectors independently and uniformly at random from all the m bit vectors and thus the declining rate is $p = \frac{P}{m}$. To adapt to the frequent change of popularity, we also design an adaptable decline rate adjusting scheme to adjust the value of p during the processing procedure. We will discuss more details in Section 4.4. In the following, we prove that the decline strategy is sensitive to the arriving rate.

Theorem 3. *For a key, if its arriving rate f is larger than $2p$ (p is the declining rate), it will not be evicted from the synopsis with high probability, or it will be evicted within $(\frac{1}{p})^{log_2(\frac{f}{p})+O(1)}$ synopsis updates with high probability.*

Proof. Each key in the synopsis has a probability p to perform the bit-wise right shift operation each time when the synopsis is updated with new experiment values larger than the threshold r. Thus, for each key the expected interval between a right shift operation is $\frac{1}{p}$ synopsis updates or $(\frac{1}{p}) \times 2^r$ coin flipping experiments. In each interval, the expected times of the appearances of a key with arriving rate f is $N_f = (\frac{1}{p}) \times 2^r \times f$. For this key, the expected largest experimental value is,

$$\begin{aligned} V(f) =& \sum_{i=0}^{\infty} i((1-\tfrac{1}{2^{i+1}})^{N_f} - (1-\tfrac{1}{2^i})^{N_f}) \\ =& \lim_{i\to\infty}(i(1-\tfrac{1}{2^{i+1}})^{N_f}) - \sum_{i=0}^{\infty}(1-\tfrac{1}{2^i})^{N_f} \\ =& \log_2 N_f + O(1) \\ =& \log_2(\frac{f}{p}+r) + O(1) \end{aligned} \quad (4)$$

Thus, in each interval, the $(\log_2(\frac{f}{p})+O(1))^{th}$ bit of the bit vector has high probability to be set to "1". If $f > 2p$, we have $\log_2(\frac{f}{p}) > 1$. That is to say, before one right shift operation, there will be a bit set to "1" at a position other than the 0^{th} bit with high probability. Thus the key will not be evicted from the synopsis. On the contrary, for a key with arriving rate $f \leq 2p$, during each interval, the probability for it to set a bit to "1" at a position other than the 0^{th} bit is very low. Thus, the "1" bit at the highest position keeps right shifting until it is evicted from the synopsis. This needs $(\frac{1}{p})^{\log_2(\frac{f}{p})+O(1)}$ synopsis updates. Theorem 3 is proved. ■

4.3 The Stableness of the Synopsis

As the synopsis plays an important role in the hot key predictor, it is vital to achieve a stable state of the synopsis. Intuitively, if the declining rate is too low, new potential hot keys will be inserted into the synopsis before the previously hot but currently rare keys are evicted. With a fixed memory size, future hot keys may fail to be inserted. On the contrary, if the declining rate is too high, keys will be evicted out of the synopsis quickly. The synopsis will become empty rapidly and the current hot keys may be evicted, resulting in poor performance of the predictor. In the following, we prove that for any given declining rate p, the number of keys in the synopsis is stable with explicit upper and lower bounds. The synopsis will never become empty or full.

Theorem 4. *The number of keys in the synopsis will be stable in $(H, \frac{1}{p})$ with high probability, where H is the number of keys whose arriving rates are higher than $2p$ in the stream.*

Proof. From Theorem 3, with high probability the synopsis will contain the keys with arriving rates higher than $2p$ in the stream. We only consider the keys with arriving rates less than $2p$. In each synopsis update, the probability that the newly arriving key is a rare key is not greater than $(1 - H \times (2p))$, while the probability that the 0^{th} bit is set to "1" is $\frac{1}{2}$. After a number of $\frac{1}{p}$ synopsis updates, every bit vector has a high probability to right shift. Thus, at most $\frac{1}{2(1-H\times(2p))} \times \frac{1}{p}$ new rare keys remain in the synopsis.

On the other hand, assume there are a number of X_τ keys in the synopsis at a given time τ. With Theorem 3, we have $X_\tau > H$ with high probability, while $(X_\tau - H)$ keys are rare keys. After $\frac{1}{p}$ synopsis updates, at least $\frac{1}{2(X_\tau - H)}$ old rare keys will be evicted. Thus, we can obtain,

$$X_{\tau+\frac{1}{p}} < X_\tau - \frac{1}{2}(X_\tau - H) + \frac{1}{2}(1-pH) \times \frac{1}{p} = \frac{1}{2}X_\tau + \frac{1}{2p} \tag{5}$$

Since the synopsis is initially empty, it is clear that for any time τ, $X_\tau < \frac{1}{p}$ holds. Theorem 4 is proved. ■

4.4 Adaptive Threshold Configuration

In this section, we analyze how the frequency threshold for differentiating hot keys and rare keys affects the system performance, and how to configure the threshold to minimize the system processing cost dynamically and adaptively.

According to Theorems 1-4, for a certain dataset with a frequency threshold f, we can set $r = \log_2 \frac{1}{p}$ to ensure the precision of the hot key prediction, and $p = \frac{1}{2}f$ to ensure the stableness of the synopsis. However, in practice, it is difficult for a system administrator to configure the threshold for differentiating hot keys and rare keys. Indeed, the key distribution of different data streams are always different, and the distribution may change over time. For example, in a stream S_1 with thousands of keys, the average frequency of a key is around 0.1%; while in a stream S_2 with billions of keys, the peak frequency of a key can be lower than 0.1%. If the threshold of the hot key frequency is set too low (e.g., far lower than 0.1% in S_1), a large fraction of the keys will be identified as popular ones, and the differentiated scheduling is almost the same as shuffle grouping scheme, which leads to a large fraction of unnecessary memory storage. If the threshold of the hot key frequency is set too high (e.g., higher than 0.1% in S_2), nearly no keys will be regarded as hot keys, and the differentiated scheduling is almost the same as key grouping scheme. This will lead to a high load imbalance and low system throughput. To adapt to various stream data distribution, we further design an adaptable threshold adjusting scheme in PStream. Here, we first discuss how to define the optimal frequency threshold in PStream and then propose a method to compute the optimal frequency threshold using lightweight statistics and quickly adjust the parameters to approach an optimal frequency threshold.

In a certain stream with N unique keys, for a frequency threshold f, we assume there are $H(f)$ unique hot keys, each having an appearance frequency higher than f. There are also $L(f)$ unique rare keys with appearance frequency less than f. We further assume the accumulative frequency of all hot keys is $F_H(f)$ and that of all rare keys is $F_L(f)$. To take full advantages of the differentiated scheduling design and optimize system performance, the optimal frequency threshold should satisfy the following conditions: 1) there are only a small fraction of keys which can be identified as hot keys (*i.e.,*, the value of $H(f)$ should be low), so that using shuffle grouping will not incur heavy memory cost; 2) the accumulative frequency of all the identified rare keys (*i.e.,* $F_L(f)$) should be low, so that using key grouping will not incur significant load imbalance.

With the proposed differentiated scheduling scheme, the identified hot keys will be shuffled to all the M downstream instances. Thus, each instance will store all the $H(f)$ keys. Each instance will evenly get $\frac{F_H(f)}{M} \times T$ tuples with hot keys for processing in a unit time T. The rare keys are hashed to all the instances. Thus, each instance approximately stores $\frac{L(f)}{M}$ unique rare keys. The number of rare keys in each instance for processing is upbounded by $F_L(f) \times T$. Therefore, the upbounded processing time for an instance is proportional to $(\frac{F_H(f)}{M} + F_L(f))$. The memory usage is proportional to $(H(f) + \frac{L(f)}{M})$.

To optimize both time and space efficiency, we use the following equation to compute the system processing cost of a frequency threshold f,

$$C(f) = w_T \times (\frac{F_H(f)}{M} + F_L(f)) + w_S \times (H(f) + \frac{L(f)}{M}) \tag{6}$$

In the above equation, w_T and w_S are two normalized weights of the processing time and the memory usage which can be assigned by users. The system administrator can adjust them according to the differences of computing and storage resources of the system. Unlike the threshold f, these two parameters are only related to the system resources and they are not related to the distribution of datasets. The system administrator can try to adjust these two parameters by running a sample application several times, and obtain satisfied fixed values by using some greedy or heuristic strategies as we will discuss later.

From the definition of the processing cost, we have $F_H(f) + F_L(f) = 1$ and $H(f) + L(f) = N$. Thus, we obtain,

$$C(f) = \frac{w_T + w_S \times N}{M} + \frac{M-1}{M} \times (w_T \times F_L(f) + w_S \times H(f)) \tag{7}$$

That is to say, the optimal frequency threshold is the value of f which minimizes $(w_T \times F_L(f) + w_S \times H(f))$. Our target is to adjust our frequency threshold to approach the optimal value periodically. However, there are two challenges here. First, usually, we do not know the data distribution in advance. Hence, we can hardly compute $F_L(f)$ and $H(f)$ for any possible f. Second, the data distribution may change over time. Hence, we should adjust the frequency threshold to an optimal value as quickly as possible.

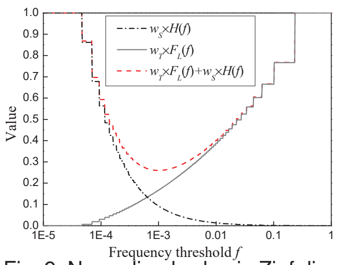


Fig. 9. Normalized value in Zipf distribution (α=1.2)

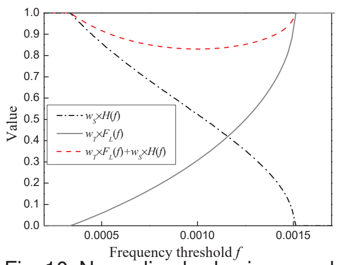


Fig. 10. Normalized value in normal distribution (δ=1)

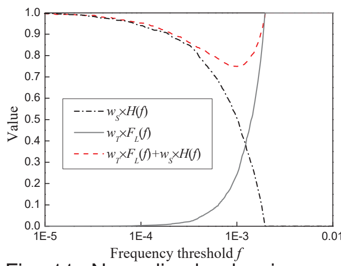


Fig. 11. Normalized value in random distribution

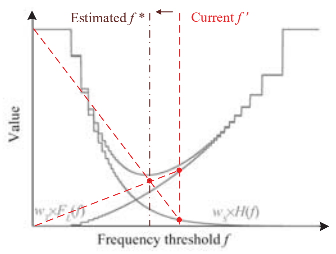


Fig. 12. Adaptively threshold adjusting

To solve the problems, we propose a method to approximately estimate the values of $F_L(f)$ and $H(f)$ for the current frequency threshold f. Specifically, we periodically track the newly arriving records in the synopsis structure. In a time interval, in addition to the aforementioned synopsis updating process, the synopsis also performs the following steps. Each time when a new record (which contains a potential hot key and its experiment value) arrives, the synopsis checks whether it is an actually hot key. Then the synopsis counts for the total number of newly arriving records (v_1), and figures out how many records contain actually hot keys (v_2). If a newly arriving record contains an actually hot key, the synopsis gives the corresponding bit vector a mark. At the end of the time interval, the synopsis computes $\frac{v_2}{v_1}$ and the number of marked bit vectors (v_3). Then it initials the two counters to zero and removes all the marks. The total number of tuples can be estimated by $v_1 \times 2^r$, while the total number of hot keys can be estimated by $v_2 \times 2^r$. Thus, $F_L(f)$ can be estimated by $\frac{v_1 \times 2^r - v_2 \times 2^r}{v_1 \times 2^r} = 1 - \frac{v_2}{v_1}$. The value of v_3 is essentially the number of the identified hot keys in this time interval. Thus, $H(f) = v_3$.

To achieve lower processing cost $C(f)$, we need to minimize both the values of $w_T \times F_L(f)$ and $w_S \times H(f)$. It is not difficult to see that, the value of $F_L(f)$ monotonically increases with f, while $H(f)$ monotonically decreases with f. Thus, our proposal is to find a moderate value for f where $w_T \times F_L(f)$ and $w_S \times H(f)$ are close to each other.

To show the efficiency of our scheme, we generate three datasets with different distributions. Each dataset has 1,000 unique keys, and the frequencies of all the keys follows Zipf distribution, normal distribution, and random distribution, respectively. We configure 1,000 different frequent threshold values f. For each frequent threshold, we compute the exact value of $F_L(f)$ and $H(f)$. We set the value of w_T and w_S to normalize the value of $w_T \times F_L(f)$ and $w_S \times H(f)$ between the interval [0,1]. Then we plot the values in Figs. 9, 10 and 11. Although the functions of $F_L(f)$ and $H(f)$ are quite different with each other in the three datasets, the value of $C(f)$ changes in a similar way. The optimal value of f which minimizes $C(f)$ appears near the point where $w_T \times F_L(f) = w_S \times H(f)$.

To approach the optimal value of f, in each time interval, when we estimate the value of $F_L(f)$ and $H(f)$, we can compare the value of $w_T \times F_L(f)$ with that of $w_S \times H(f)$. If the former is larger, we try to increase the value of f; otherwise, we try to decrease the value of f. We use an adaptive scheme to quick compute a better value of f, which can get closer to the optimal value. Figure 12 shows how we adjust the threshold. For the current frequency threshold f', we estimate the value of $F_L(f')$ and $H(f')$. We draw two lines as follows. If $F_L(f') > H(f')$, one line connects the point (0,$w_T \times F_L(0)$) (*i.e.*, (0,0)) and the point (f', $w_T \times F_L(f')$). The other one connects (0, $w_S \times H(0)$) (*i.e.*, (0,w_S)) and (f', $w_S \times H(f')$). We can compute the intersection of these two lines. We use f^* to denote the x-coordinate value of the achieved intersection. It is obvious that f^* is closer to the point where $w_T \times F_L(f) = w_S \times H(f)$ rather than f'. If $F_L(f') < H(f')$, one line will connect (1,$w_T \times F_L(1)$) (*i.e.*, (1,w_T)) and (f', $w_T \times F_L(f')$), while the other one connects (1, $w_S \times H(1)$) (*i.e.*, (1,0)) and (f', $w_S \times H(f')$), accordingly.

After the synopsis computes the value of f^*, it accordingly computes the values of r and p to make the frequency threshold close to f^*. Then it notifies all the coin flipping instances to adjust the threshold r. After that, a new interval starts, and the synopsis estimates the value of $F_L(f^*)$ and $H(f^*)$ again. The above process repeats periodically. If the distribution is stable, the value of f^* will converge to the optimal value. When the distribution changes, this scheme can also quickly and adaptively change the thresholds.

With such an adaptive thresholds configuration scheme, we can further optimize the setting of w_S and w_T. Specifically, we can set a random initial value of w_T and keep $w_T + w_S = 1$. In the first step, we run a sample application and obtain the system throughput $TP(w_T)$. In the second step, we choose a random value δ from interval [-min$\{w_T, w_S\}$, min$\{w_T, w_S\}$]. We set $w'_T = w_T + \delta$ and run the same application again to obtain the system throughput $TP(w'_T)$. In the third step, we compare the value of $TP(w_T)$ and $TP(w'_T)$. If the later is higher, we set $w_T = w'_T$, otherwise, we keep the value of w_T and make $\delta = \frac{1}{2}\delta$. Then we repeat the three steps in iterations until the difference of $TP(w_T)$ and $TP(w'_T)$ in the third step is low enough (*e.g.*, less than 1%). To avoid the above greedy algorithm trapping into a local optimal area, we can also use some heuristic algorithms such as simulated annealing [32]. Specifically, in the third step, if $TP(w_T)$ is higher, we still set $w_T = w'_T$ with a low probability of $e^{(TP(w'_T) - TP(w_T))/i}$, where i is the number of iterations. Such a design may need more iterations to converge, but it usually achieves higher solution quality.

5 Implementation

We implement PStream on top of Apache Storm [1] and make the source code publicly available[1].

In the PStream implementation, we build a standalone component that is independent of the original user logic topology to execute the tasks of the hot key predictor. The component consists of a coin bolt and a global synopsis bolt (a bolt is the basic processing element of Storm). The

[1] Sourcecode available: https://github.com/CGCL-codes/PStream

coin bolt receives tuples from the bolts which need to use the differentiated scheduler. The coin bolt performs the coin flipping experiment, identifies the potential hot keys, forwards the identified potential hot keys to the global synopsis bolt, and discards all the other keys. The synopsis bolt maintains a HashMap in memory. The HashMap consists of a set bit vectors with a length of l. Each bit vector corresponds to an identified potential hot key.

During the stream processing procedure, each time when an instance of the user logic bolt forwards a tuple to the downstream bolt, the instance generates a signal tuple associated with the same key to the coin bolt. To prevent the coin bolt from becoming the bottleneck of the whole system, PStream creates a large number of coin bolt instances, which is no less than the number of instances of the user logic bolt.

Following the processing framework described in Section 4, each time when the synopsis bolt receives an identified potential hot key from the coin bolt, it updates the corresponding bit vector and checks whether the key is a real hot key (*i.e.* there are more than one "1" bits in the bit vector). After each updating, the synopsis bolt executes a decline phase (*i.e.*, it performs the right shift operation on each bit vector with the probability of p), and then kicks the keys with an empty bit vector out of the HashMap. When the synopsis bolt detects a new real hot key or kicks out a rare key, it generates a signal tuple whose value represents "append" or "delete" for this key. The synopsis bolt broadcasts the signal tuple to all the instances which need to use the differentiated scheduler. Although the synopsis bolt is a centralized component, it will not become the bottleneck of the distributed system. On one hand, with the help of the coin bolt, the vast majority of arriving tuples are previously filtered. On the other hand, the frequency of identifying a new current hot key or an out-dated hot key in the synopsis is much less than that of processing arriving tuples in the user logic bolt. Thus, the broadcasting will not raise the heavy workload for the processing system. Moreover, there are usually multiple instances of the user logic bolt processed in the same worker. To avoid redundant data transmissions from the synopsis to the multiple instances hosted on the same worker, we develop a batch-and-dispatch mechanism which packages the signal tuple for each worker in a single packet, and send the tuple only once. The receiving worker then dispatches the tuple to hosted instances locally. Such a design greatly reduces the communication overhead.

Apart from the standalone predict component, each instance of the user logic bolt maintains a succinct CBF as the hot key filter. When receiving the "append" or "delete" signal from the synopsis bolt, the instance updates the CBF. When the instance is going to send out a tuple to the download processing element, it checks whether the tuple is hot or not against the CBF. According to the results, the scheduler chooses a suitable scheduling scheme from key grouping or shuffle grouping for this tuple.

6 Performance Evaluation

We evaluate the performance of PStream using comprehensive experiments with large-scale real-world datasets.

6.1 Experiment Setups

We deploy PStream on a cluster of 33 machines, each equipped with an Intel Xeon E5-2670 CPU (octa-core 2.4GHz), 64.0GB RAM, and a 1,000Mbps Ethernet interface card. One machine serves as the master node to host the Storm Nimbus. The other machines run Storm Supervisors.

Datasets. In the experiment, we use four large-scale traces collected from real-world systems. The first is a set of tweets crawled from Twitter, which contains 658 million words associated with 5.7 million unique keys. The second is the stock exchange dataset collected from NASDAQ [16] during April 2017. The dataset contains 274 million exchange records associated with 6,649 stock symbols. The third is the hashtag dataset collected from Twitter during Nov. 2012 [33]. The dataset includes 43 million hashtags. The fourth is the on-demand ride-hailing dataset collected from [17], in Chengdu during Nov. 2016. The dataset contains three billion driving track records of taxis, each associated with a taxi id, the GPS location, and a timestamp. To evaluate the performance of PStream with different degrees of skewness, we also conduct extended experiments with synthetic datasets. Each dataset has one billion tuples and 10 million keys, which follow Zipf distributions with coefficients varying from 0.5 to 2.0. A higher coefficient indicates a more skewed distribution. Table 3 summarizes the dataset statistics. We leverage Kafka [34] for the stream input.

Applications. We evaluate PStream with four applications including word-count, *Volume Weighted Average Prize* (VWAP), Hashtag statistics, and road condition detection.

The word-count and the hashtag applications count the number of each word or hashtag in the tweets stream. First, the Kafka Spout feeds tweet messages to the split bolt which splits a tweet into simple words or extracts the hashtags. Second, a count bolt counts for each distinct word or hashtag in a given time interval. Third, an aggregation bolt aggregates and outputs the results.

The VWAP application computes the ratio of the total value of a stock symbol to its total trading volume. Each exchange record is represented as a tuple with the stock symbol as the key. The exchange amount and the price associated with this key are represented as values. The statistic bolt multiples the exchange amount by the price as the intermediate result. An aggregation bolt collects the results and computes the final average price for each symbol.

The road condition detection application counts the distinct taxi ids in the same location area in a short time period to approximately indicate the crowding degree. In this application, a tuple contains a taxi track record with its GPS location mapped to grids on the map as the key. The count distinct bolt records each taxi id in a HashMap and counts the size of the HashMap at the ends of each

TABLE 3
Datasets Statistics

Dataset	Symbol	♯ of tuples	♯ of keys
Tweets	TW	658M	5.7M
NASDAQ records	ST	274M	6.7K
Tiwtter Hashtags	HA	43M	3.4M
DiDi Taxi records	TX	3B	8K
Synthetic Zipf	ZF	1B	10M

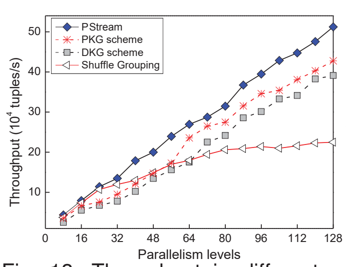


Fig. 13. Throughput in different parallelism levels

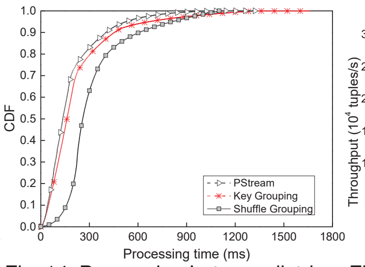


Fig. 14. Processing Latency distribution with ZF dataset

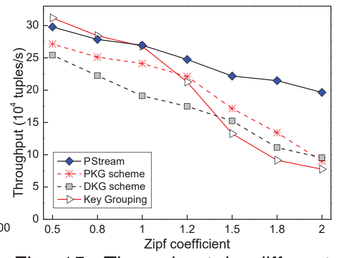


Fig. 15. Throughput in different levels of skewness

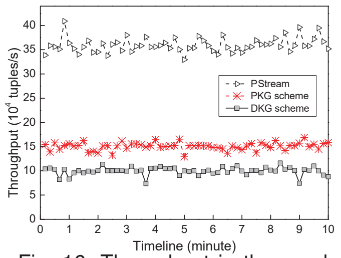


Fig. 16. Throughput in the word-count application

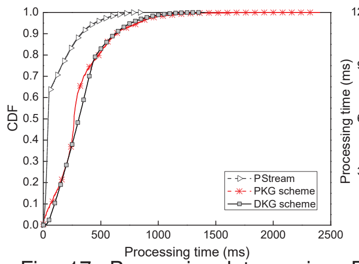


Fig. 17. Processing latency in the word-count application

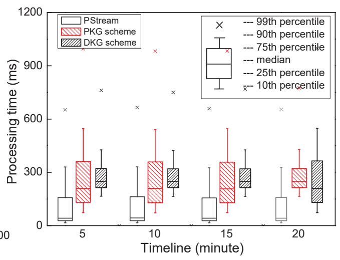


Fig. 18. Processing time variation in the word-count application

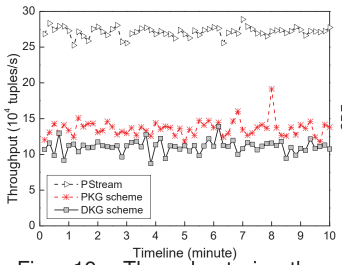


Fig. 19. Throughput in the VWAP application

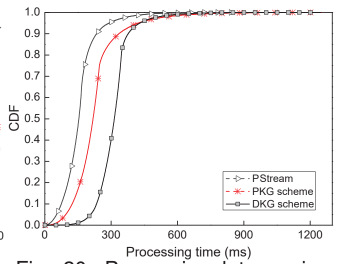


Fig. 20. Processing latency in the VWAP application

time period. If the count surpasses a certain threshold, the detection bolt outputs the results.

Baseline Schemes and Metrics. In the experiment, we first compare the performance of PStream with those of the state-of-the-art designs including the *Partial Key Grouping* [14] (PKG) and *Distribution-aware Key Grouping* [21] (DKG) schemes. We mainly examine system throughput, processing latency, and degree of load imbalance. High throughput and low latency are always desirable in a distributed stream processing system [14]. We define the throughput as the rate of the successfully processed tuples. We define the processing latency as the average processing time for each tuple.

Specifically, we compute the number of tuples in every ten seconds to compute the throughput. We record the total processing latency for every 10,000 tuples which are emitted consecutively and report the average processing latency. To monitor the processing procedure of each tuple, we leverage the acknowledge mechanism in Storm. A tuple is regarded successfully processed if and only if the spout receives the ACK message for this tuple from the last bolt. To achieve high system efficiency, a distributed stream processing system also desires resource efficiency with alleviated load imbalance among processing instances. We measure the number of received tuples in each instance and use the standard deviation of the workloads among all the instances to examine the degree of load imbalance.

We further examine the efficiency of our adaptive threshold configuration scheme. We compare the throughput and the standard deviation of the workloads of PStream with that of fixed threshold configuration. We run the experiment ten times with the word-count and Hashtag statistic applications. We also obtain an experimental optimal threshold manually using large-scale experiments.

Finally, we examine the performance of the proposed hot key predictor compared to existing schemes including count-min sketch [27] and Ada-CMSketch [28]. We fix the memory size of all the structures and examine the precision and the computation cost for hot key identification.

6.2 Results

Overall Performance of PStream. In the experiment, we examine the efficiency of the computation and memory resources of PStream using both the synthetic and real-world datasets. We compare the performance of PStream with those of diverse schemes including key grouping, shuffle grouping, DKG, and PKG.

Figure 13 shows the throughput in different parallelism levels. We compare PStream with shuffle grouping, DKG, and PKG. In the experiment, we fix the popularity distribution with Zipf coefficient = 1.0 and increases the number of processing instances. Specifically, we create 8∼128 instances of the word count bolt and examine the maximum throughput of the system. At each parallelism level, we adjust the speed for tuple emitting to achieve the maximum throughput. The result shows that when the parallelism level reaches 80, the system throughput with shuffle grouping stops increasing, while those of all the other three schemes keep increasing. When the parallelism level reaches 128, the throughput of PStream is 32% and 21% higher than those of DKG and PKG, respectively.

Figure 14 shows the CDF of the processing latency. In the experiment, we fix the popularity distribution with Zipf coefficient = 1.0. The result shows that 72% tuples in PStream have processing latency less than 200ms, while only 19% tuples with shuffle grouping and 61% tuples with key grouping have such low processing latency. The long processing latency of shuffle grouping is mainly caused by the additional aggregation operation, which needs around 300ms for each tuple on average. In PStream, only a small fraction of keys need aggregation and the aggregation latency is around 70ms on average.

Figure 15 plots the system throughput under different degrees of data skewness. We fix the parallelism level at 64 and compare PStream with key grouping and the other two baseline schemes. The result shows that the system throughput of all the existing schemes decreases sharply when the coefficient increases. In contrast, PStream remains a much more stable throughput.

Figure 16 compares the throughput of PStream with

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3019689, IEEE Transactions on Computers

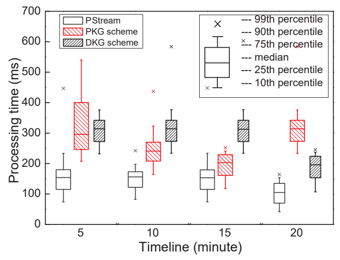


Fig. 21. Processing time variation in the VWAP application

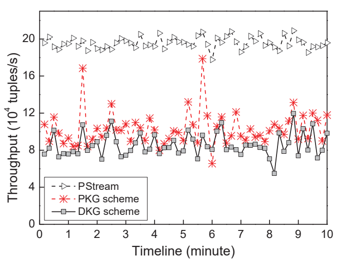


Fig. 22. Throughput in the Hashtag statistic application

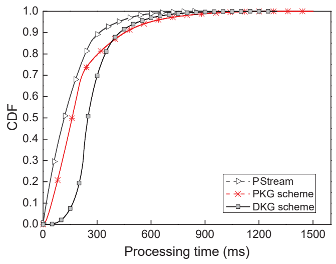


Fig. 23. Processing latency in the Hashtag statistic application

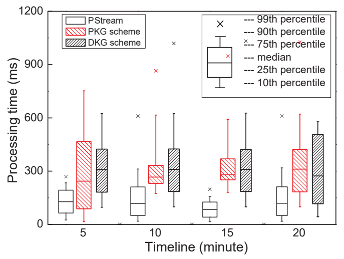


Fig. 24. Processing time variation in the Hashtag statistic application

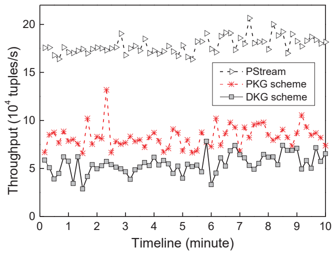


Fig. 25. Throughput in the road condition detection application

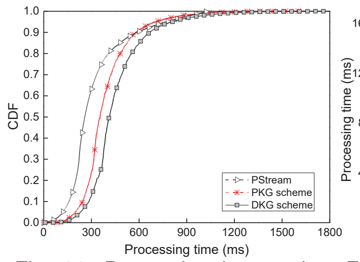


Fig. 26. Processing latency in the road condition detection application

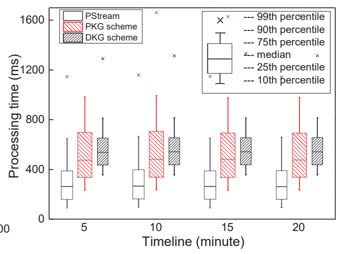


Fig. 27. Processing time variation in the road condition detection application

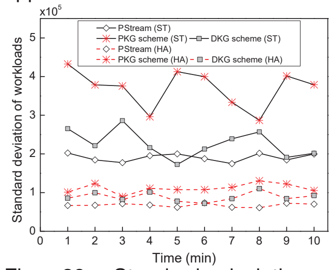


Fig. 28. Standard deviation of workloads among different instances

those of PKG and DKG in the word-count application with the TW dataset. We fix the parallelism level at 128 and record the throughput every 10 seconds. The result shows PStream achieves 2.3× and 3.6× improvements in the system throughput compared to PKG and DKG, respectively.

Figure 17 plots the CDF of the average tuple processing latency for the word-count application. The result shows the latency of PStream is 114ms, while those of PKG and DKG are 322ms and 336ms, respectively. PStream greatly reduces the latency of PKG by 64% and that of DKG by 66%. The result shows that 90% tuples using PStream can be processed in 330ms, while only 63% and 45% tuples of PKG and DKG can have such a short time.

Figure 18 shows the distribution of each tuple's processing time every five minutes. The result shows that PStream has the lowest latency at every percentile. It also achieves a more stable latency compared to the baseline schemes. Between the fifth minute and the 20th minute, both the distributions of PKG and DKG change greatly due to the change of frequency of hot keys.

Figure 19 compares the system throughput of the VWAP application. The result shows that PStream achieves 2.0× and 2.4× improvements of the system throughput compared to PKG and DKG, respectively.

Figure 20 plots the CDF of the average latency of the VWAP application. The result shows that 82% tuples in PStream have the processing latency less than 180ms, while only 8% tuples in DKG and 38% tuples in PKG have such low latency. PStream reduces the average tuple latency of DKG and PKG by 50% and 31%, respectively.

Figure 21 shows the distribution of the processing time of each tuple in the VWAP application. With the dynamic stock records, the distributions of both DKG and PKG schemes vary significantly. In contrast, PStream has a much more stable performance. This reflects that our hot key predictor more precisely predicts the hot keys in the presence of dynamic changes of keys' popularities over time.

Figure 22 compares the throughput of PStream with those of PKG and DKG in the Hashtag statistic application with the HA datasets. The result shows that PStream achieves 1.9× and 2.3× improvements of the average throughput compared to PKG and DKG, respectively.

Figures 23 and 24 plot the CDF of the average latency and the distribution of the processing time in the Hashtag statistic application, respectively. The result shows that 70% tuples in PStream have processing latency less than 130ms, while only 13% tuples in DKG and 54% tuples in PKG have such low latency. The average tuple processing latency of PStream is 148ms, while those of PKG and DKG are 210ms and 280ms, respectively. PStream greatly reduces the latencies of DKG and PKG by 48% and 30%.

Figure 25 compares the throughput of PStream with those of PKG and DKG in the road condition detection application with the TX datasets. The result shows that PStream achieves 2.0× and 2.8× improvements of the average system throughput compared to PKG and DKG.

Figures 26 and 27 plot the CDF of the average latency and the distribution of the processing time in the road condition detection application, respectively. The result shows that 63% tuples in PStream have processing latency less than 300ms, while only 17% tuples in DKG and 21% tuples in PK have such low latency. The average tuple processing latency of PStream is 267ms, which reduces the latencies of DKG and PKG by 39% and 27%, respectively.

We further examine the workloads (*i.e.*, the received tuples) of each instance in an interval of ten seconds in the VWAP and the Hashtag statistic applications. We compute the standard deviations of the workloads in the instances to show the degree of load imbalance. Figure 28 shows that PStream reduces the average standard deviation of DKG by 16% in VWAP and by 25% in Hashtag statistic. PStream reduces the average standard deviation of PKG by 49% in VWAP and by 40% in Hashtag statistic.

We also examine the scalability of PStream. In the experiment, we increase the number of instances (*i.e.*, the parallelism level), and achieve the maximum throughput for each parallelism level to put the examination to the system performance limit. Specifically, for each parallelism

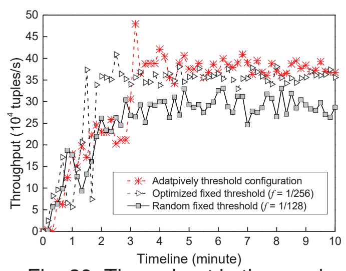


Fig. 29. Throughput in the word-count application with adaptive decline scheme

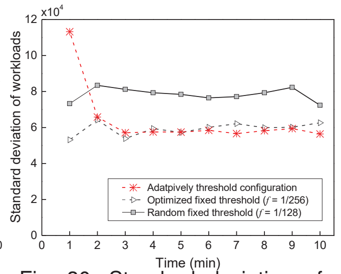


Fig. 30. Standard deviation of workloads among instances in the word-count application

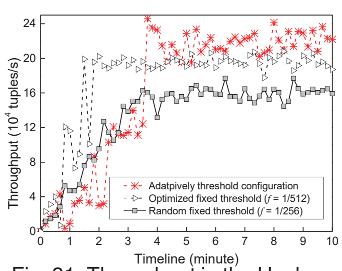


Fig. 31. Throughput in the Hashtag statistic application with adaptive decline scheme

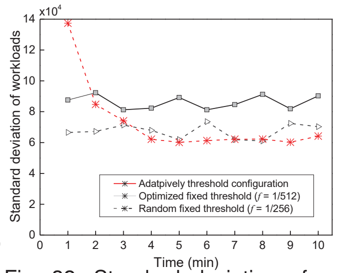


Fig. 32. Standard deviation of workloads among instances in the Hashtag statistic application

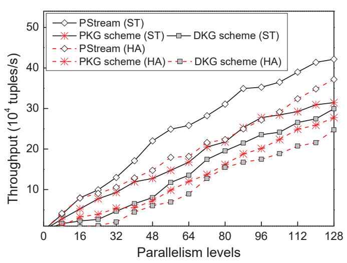


Fig. 33. The optimal throughput of TW and HA

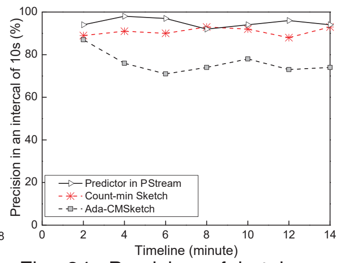


Fig. 34. Precision of hot keys predictor

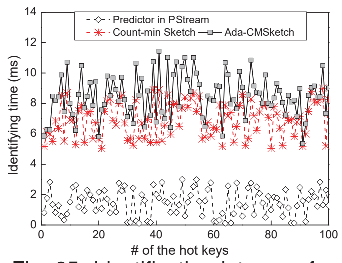


Fig. 35. Identification latency of hot keys predictor

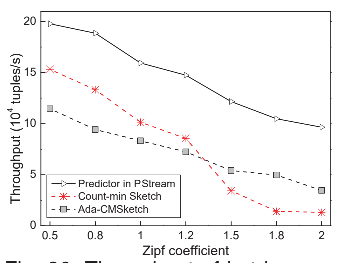


Fig. 36. Throughput of hot keys predictor

level, the Kafka Spout increases the emitting speed until the throughput does not increase anymore. Figure 33 shows that when the parallelism level reaches 128, the throughput of PStream for the VWAP and the Hashtag statistic application is 35% and 33% higher than those of PKG. PStream achieves 40% and 49% higher throughput than those of DKG in the VWAP and the Hashtag statistic application, respectively.

Performance of Adaptive Threshold Configuration. Figure 29 shows the throughput of PStream with the word-count application. We compare the throughput with adaptive threshold configuration and that with randomly set fixed thresholds and manually optimized fixed threshold. We run the experiment with random fixed thresholds ten times and report the median. The result shows that the adaptive threshold configuration scheme of PStream quickly converges to a stable threshold, and the system obtains higher system throughput than that using a manually optimized fixed threshold. PStream with the adaptive threshold configuration scheme achieves 17% to 42% higher throughput compared to those with random thresholds.

Figure 30 shows the standard deviation of workloads among different instances in the word-count application of PStream with adaptive threshold configuration, and that of the scheme with fixed thresholds. The result shows that with the adaptive threshold configuration scheme, loads of PStream instances can become more balanced when the threshold is stable. The adaptive threshold configuration scheme can reduce the average standard deviation of the workloads among all the instances of the optimal fixed threshold setting by 9.1%, and that of the random fixed threshold setting by 24% to 32%.

Figure 31 shows the processing time of the Hashtag statistic application. We compare the performance of PStream with the adaptive threshold configuration and that with optimal fixed thresholds. The adaptive threshold configuration scheme improves the system throughput of an optimal fixed threshold by 19.5% when the threshold becomes stable. Compared to the ten randomly set thresholds, PStream with the adaptive threshold configuration scheme achieves 27% to 46% higher throughput.

Figure 32 shows the standard deviation of workloads in the Hashtag statistic application, where we compare PStream with the adaptive threshold configuration and that with fixed thresholds. The adaptive threshold configuration scheme improves the load balance by 14.5% and 35%, compared to the designs with manually optimized fixed threshold and random fixed threshold.

Performance of the Hot Key Predictor. Figure 34 shows the prediction precision, which is defined as the ratio of the number of identified actually hot keys to the total number of keys identified. The result shows PStream achieves a precision of 96%, outperforming all existing schemes. Our hot key predictor significantly improves the precision of the previous count-min sketch scheme by 23%.

We record the time from the time the predictor receives the tuple to the time the tuple is identified to be a hot key. Figure 35 plots the time of 100 continuous hot keys identified. PStream uses 1.4ms in average, reducing those of count-min sketch and Ada-CMSketch by 80% and 84%.

We also compare the throughput of our hot key predictor. Figure 36 shows that the throughput of our hot key predictor is always higher than those of the other two schemes. When Zipf coefficient = 1.0, our scheme improves the throughput of the count-min sketch by 36.7% and that of the Ada-CMSketch by 47.5%.

7 Conclusion

In this paper, we argue that the key of efficient distributed stream processing is to differentiate the popularities of the keys. We design PStream, a novel differentiated stream processing system. The efficiency of PStream is based on a proposed new hot key predictor for large-scale real-time streams. PStream hot key predictor can accurately identify the current hot keys in real-time streams at very low computation and memory costs. It can adapt to the popularity changes in highly dynamical real-time streams. We implement PStream on top of Apache Storm. Experimental results using large-scale datasets from real-world systems show that PStream greatly outperforms existing designs in terms of throughput and processing latency.

8 Acknowledgements

We thank the anonymous reviewers for the comments with care and insights. The preliminary result of this work was presented in IEEE ICNP 2017 [35].

References

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in: *Proceedings of SIGMOD*, 2014.

[2] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in: *Proceedings of SIGMOD*, 2015.

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in: *Proceedings of SOSP*, 2013.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in: *Proceedings of ICDMW*, 2010.

[5] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *PVLDB*, vol. 10, no. 12, pp. 1634–1645, 2017.

[6] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu, "A general communication cost optimization framework for big data stream processing in geo-distributed data centers," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 19–29, 2016.

[7] K. Hildrum, F. Douglis, J. L. Wolf, P. S. Yu, L. Fleischer, and A. Katta, "Storage optimization for large-scale distributed stream-processing systems," *ACM Transactions on Storage*, vol. 3, no. 4, pp. 5:1–5:28, 2008.

[8] G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, A. Banerjee, B. Heintz, S. Iyer, and A. Jaiswal, "Providing streaming joins as a service at facebook," *PVLDB*, vol. 11, no. 12, pp. 1809–1821, 2018.

[9] H. Wei, H. Zhou, J. Sankaranarayanan, S. Sengupta, and H. Samet, "Detecting latest local events from geotagged tweet streams," in: *Proceedings of SIGSPATIAL*, 2018.

[10] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong, and J. Ye, "A taxi order dispatch model based on combinatorial optimization," in: *Proceedings of KDD*, 2017.

[11] T. Zheng, G. Chen, X. Wang, C. Chen, X. Wang, and S. Luo, "Real-time intelligent big data processing: technology, platform, and applications," *SCIENCE CHINA Information Sciences*, vol. 62, no. 8, pp. 82101:1–82101:12, 2019.

[12] M. I. Gordon, W. Thies, and S. P. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in: *Proceedings of ASPLOS*, 2006.

[13] S. Schneider, M. Hirzel, B. Gedik, and K. Wu, "Safe data parallelism for general streaming," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 504–517, 2015.

[14] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in: *Proceedings of ICDE*, 2015.

[15] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.

[16] NASDAQ website, http://www.nasdaq.com/, 2019.

[17] Didi Chuxing GAIA Initiative, https://gaia.didichuxing.com, 2019.

[18] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: Continuous reliable distributed processing of big data streams," in: *Proceedings of NSDI*, 2016.

[19] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, "Efficient key grouping for near-optimal load balancing in stream processing systems," in: *Proceedings of DEBS*, 2015.

[20] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in: *Proceedings of SIGMOD*, 2013.

[21] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in: *Proceedings of DEBS*, 2013.

[22] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB Journal*, vol. 23, no. 4, pp. 517–539, 2014.

[23] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[24] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *VLDB Journal*, vol. 19, no. 1, pp. 3–20, 2010.

[25] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," *PVLDB*, vol. 5, no. 12, p. 1699, 2012.

[26] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in: *Proceedings of ICDT*, 2005.

[27] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[28] A. Shrivastava, A. C. König, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in: *Proceedings of SIGMOD*, 2016.

[29] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in: *Proceedings of INFOCOM*, 2016.

[30] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[31] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao, "Chi: A scalable and programmable control plane for distributed stream processing systems," *PVLDB*, vol. 11, no. 10, pp. 1303–1316, 2018.

[32] A. Drexl, "A simulated annealing approach to the multiconstraint zero-one knapsack problem," *Computing*, vol. 40, no. 1, pp. 1–8, 1988.

[33] K. McKelvey and F. Menczer, "Design and prototyping of a social media observatory," in: *Proceedings of WWW*, 2013.

[34] Apache Kafka, http://kafka.apache.org/, 2019.

[35] H. Chen, F. Zhang, and H. Jin, "Popularity-aware differentiated distributed stream processing on skewed streams," in: *Proceedings of ICNP*, 2017.

Hanhua Chen received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), in 2010. He is currently a professor at the School of Computer Science and Technology, HUST, China. His research interests include big data processing systems and distributed computing systems.

Fan Zhang is currently a Ph.D candidate in the School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include BigData processing systems.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with HUST, in China. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of IEEE.