Popularity-aware Differentiated Distributed Stream Processing on Skewed Streams

Hanhua Chen, Fan Zhang, Hai Jin
Cluster and Grid Computing Lab
Services Computing Technology and System Lab
Big Data Technology and System Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan 430074, China
Emails: {chen, zhangf, hjin}@hust.edu.cn

***Abstract*—Real-world stream data with skewed distribution raises unique challenges to distributed stream processing systems. Existing stream workload partitioning schemes usually use a "one size fits all" design, which leverage 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 and implement a differentiated distributed stream processing system, call DStream, which assigns the popular keys using shuffle grouping while assigns unpopular ones using key grouping. We design a novel efficient and light-weighted probabilistic counting scheme for identifying the current hot keys in dynamic real-time streams. Two factors contribute to the power of this design: 1) the probabilistic counting scheme is extremely computation and memory efficient, so that it can be well integrated in processing instances in the system; 2) the scheme can adapt to the popularity changes in the dynamic stream processing environment. We implement the DStream system on top of Apache Storm. Experiment results using large-scale traces from real-world systems show that DStream 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**

I. Introduction

The recent advances in distributed stream processing systems such as Storm [23], Heron [13], Spark Streaming [24], S4 [20], and Samza [12], bring the community great ability to process extremely huge volumes of unbounded and continuous data streams in real-time with clusters. Different applications based on stream processing are widely deployed, e.g., event detection in Twitter [23], trends tracking in Google queries [3], stock price tracing and analytics [14], and personalized search advertising in Yahoo! [20].

In distributed stream processing systems, an application is commonly modeled as a directed graph, also called a topology [23]. By deploying different processing stages of a stream application across the machines in a cluster, a distributed stream processing system achieves an efficient processing pipeline for the application. Formally, a vertex of the directed graph is called a processing element, which represents a processing operator. 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, the distributed stream processing system creates multiple instances for an operator which work in parallel (see Fig. 1(a)). Accordingly, the workloads of an upstream processing element instance are partitioned into multiple sub-streams.

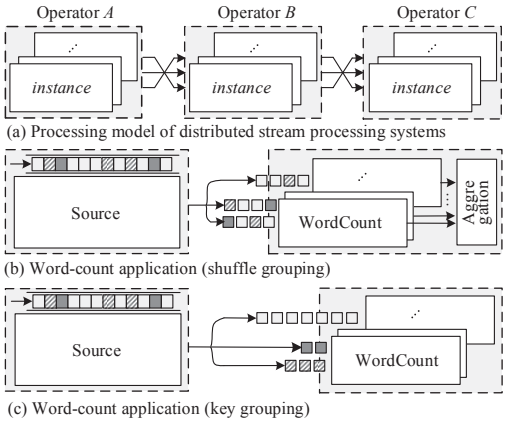

Fig. 1. Distributed stream processing system

Existing workloads partitioning strategies in a distributed stream processing system include shuffle grouping and key grouping [19]. 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 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 the 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

978-1-5090-6501-1/17/$31.00 © 2017 IEEE

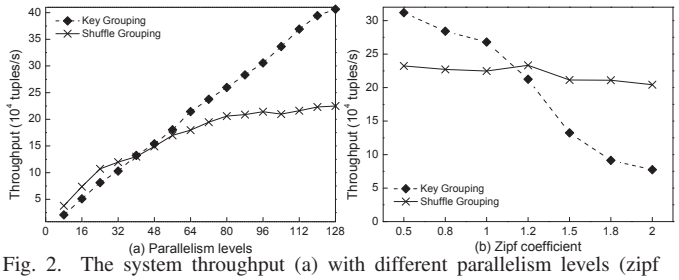

Fig. 2. The system throughput (a) with different parallelism levels (zipf coefficient = 1.0) and (b) with different skewness (parallelism level = 64)

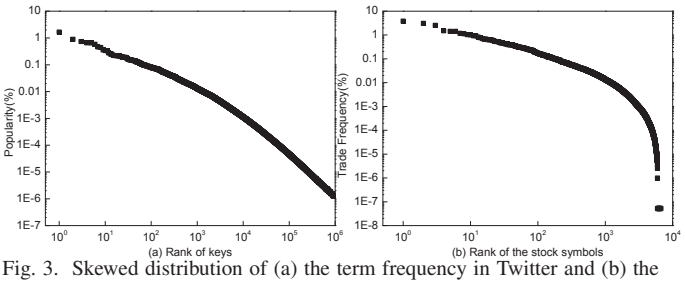

Fig. 3. Skewed distribution of (a) the term frequency in Twitter and (b) the stock trading frequency in NASDAQ

number of instances of the downstream processing element. Given a number of N unique keys and a number of M word-count instances in the system, the amount of required memory space is $O(MN)$. It is clear that such a shuffle grouping partitioning scheme is hard to scale for large-scale workloads.

We examine the performance of the shuffle grouping scheme in greater detail with the experiment shown in Fig. 2(a). In the experiment, the system varies the parallelism level by creating different numbers of downstream word-count instances. With each parallelism level, the system adjusts the source's emitting speed of tuples to 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., the number of instances is small), the system throughput increases with the degree of parallelism level. However, when the number of instances continues to increase, the system throughput stops increasing. The experiment reveals that the memory wall restricts the scalability of the system with shuffle grouping strategy.

The second partition strategy is key grouping (Fig. 1(c)), which uses the 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 assigned to the same downstream instance. It is clear, such a partition scheme needs an amount of $O(N)$ memory space, where N is the number of unique keys in the data stream. The experiment result in Fig. 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 [5]. We have collected two months' traces from Twitter (Dec. 2015 to Jan. 2016) and plot the distribution of term frequency in Fig. 3(a) (stop words removed). The result shows that an extremely small fraction of 3‰ words account for more than 80% total term frequency. We have also collected the stock exchange stream data from NASDAQ [11] (during April, 2017). Figure 3(b) shows the distribution of the popularity 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 happens among a fraction of 8% stock symbols. 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 efficiency and thus greatly degrades the system performance.

To more carefully examine the performance of the key grouping strategy in the presence of data skewness, we conduct extended experiments for key grouping using seven synthetic datasets (the datasets follow zipf distributions with coefficients varying from 0.5 to 2.0). The results in Fig. 2(b) show that the throughput decreases greatly with the increase of the level of skewness of the stream data. This reveals that the system load imbalance due to data skewness leads to significant performance degradation. In contrast, the performance of shuffle grouping remains 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. It however suffers from 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. It however does not scale in terms of memory because of the large number of rare keys. Based on this insight, we propose a novel differentiated stream processing system called DStream in this work. DStream identifies the popularity of keys in the stream data and uses a differentiated partitioning scheme. For hot keys, DStream chooses the shuffle grouping strategy, while for unpopular 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 and cope with the highly dynamic real-time streams. To address this issue, in DStream, we propose a novel light-weighted predictor for identifying the current hot keys in the real-time data streams. Two factors contribute to the efficiency of this design: 1) we propose a novel probabilistic counting scheme, which is computation and memory efficient so that the predictor based on it can be well integrated in an instance of processing element in a distributed stream processing system; 2) the predictor based on the proposed probabilistic counting scheme can adapt to the popularity changes in the dynamic stream processing environment. We implement DStream on top of Apache Storm [23] and conduct comprehensive experiments using large-scale traces from real-world systems. The results show that DStream achieves a 2.3× improvement in terms of processing throughput and reduces the processing latency by 64% compared to existing schemes.

The rest of the paper is structured as follows. Section II

reviews the related work. Section III presents the system design of DStream. Section IV theoretically analyzes the hot key predictor design in DStream. Section V presents the implementation of DStream on top of Apache Storm. Section VI evaluates the performance of DStream. Section VII concludes the paper.

II. Related Work

In this section, we briefly review the background of distributed stream processing systems. We introduce the existing work in stream workloads partition strategies. We also summarize the existing designs for popularity estimation techniques.

A. 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 [3, 20, 23, 24]. In a distributed stream processing system, a processing element receives a sequence of data tuples from its input queue, performs some 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 workload partitioning strategies.

B. Stream Partition Strategies for Data Parallelism

Recently, the issue of stream workloads partitioning has attracted much research interest in the community [19, 21]. Castro *et al.* [9] 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 instances. Moreover, this design cannot adapt to the dynamical changes of the frequency of a key in real-world streams.

Gedik [10] proposes an index-based stream workloads partition scheme. Initially, an upstream instance uses the key grouping strategy 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. Thus, it incurs extra high processing latency for the application. Rivetti *et al.* [21] 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 synchronization for computing the mapping function for each key among multiple upstream instances.

Nasir *et al.* propose the partial key grouping scheme which splits the workloads associated with a key [19]. In their design, every upstream instance keeps a vector to record the number of tuples it has sent to different downstream instances. For a coming tuple, the upstream instance selects two downstream instances, and then sends the tuple to the one with lighter workload estimated according to the local vector. Their key splitting scheme attempts to leverage a famous principle of "power of two choices" [18], which is initially described as a supermarket model. Assuming a customer 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 $\frac{loglogn}{log2} + O(1)$ 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 $O(logn)$. Note in the principle to achieve load balance, two servers should be randomly and uniformly selected as candidates for any customer. However, Nasir *et al.* 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 satisfied 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 number of tuples.

C. Popularity Estimation in Streams

As we have shown in Fig. 3, the key for efficient stream data scheduling is to efficiently estimate the popularities of the data in a stream and especially identify the most frequent keys (aka *heavy hitters*) [10]. Recently, finding heavy hitters in the stream has attracted a lot of research efforts,which can be classified into two types [6], the counter based schemes [15, 17] and the sketch based schemes [7, 22].

A counter based scheme relies on counting the appearances of tuples associated with each key [6]. Metwally *et al,* proposed a scheme called SpaceSaving [17], 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 comes, 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 the tuples [17]. 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 [4]. Specifically, they partition the stream into frames. They use SpaceSaving to identify the most popular keys in the current frame and reset all the counters when the frame ends. They record the approximated count of the popular keys in the previous frame

before reset. The problem is that for a key which was not popular in the past but currently has potentials to become popular (e.g., an emerging new hot topic), such a scheme discards the accumulated count of the key in the previous frame, leading to possible failures in identifying hot keys.

A sketch based scheme leverages a set of hash functions to project a large amount of keys to a few counters. Cormode *et al.* propose Count-min sketch [7], which employs d hash functions. Each hash function can project the keys to w values. The Count-min sketch uses $d \times w$ counters, while each key is corresponded with d of them. The estimated count of each key is the minimal value of the corresponded d counts. To further support the heavy hitter queries, Cormode *et al.* leverages a dyadic ranges methods [7]. Specifically, if there are at most N keys, it builds ($logN$ + 1) Count-min sketches which counts for different ranges (e.g. a range contains the keys whose id is between [0, 2^n]). The scheme uses grouping test methods to search the heavy hitters in the time of $O(logN)$. However, such a scheme needs to know the entire key set in advance. This is difficult for real applications.

III. Differentiated Stream Processing

In this section, we describe the overview of the design of DStream. We present the main idea of how to predict the hot keys in the dynamic data streams effectively and efficiently.

A. Design Overview

Figure 4 shows the overview of DStream. The main idea of the system is to design and deploy a light-weighted popular key predictor to support the scheduling strategy selection. DStream consists of two components: an independent predicting component for detecting potential hot keys, and a scheduling component in each processing element instance. Specifically, in the predicting component, we propose a novel probabilistic counting scheme to precisely identify the current hot keys in a stream. The probabilistic counter is computation efficient because every tuple incurs very slight extra cost of randomly generating digits "0" or "1". With the simple operation, the predictor achieves probabilistic counting of the tuples associated with a key. The keys likely to be hot ones are detected and recorded in a synopsis of potential hot keys.

Each processing element instance is equipped with a memory efficient hot key filter and a differentiated scheduler. The hot key filter is made of a *Counting Bloom filter* (CBF) [8]. 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. When a tuple comes, the differentiated scheduler quickly checks whether the associated key is contained in the CBF or not. If contained, the scheduler leverages the shuffle grouping strategy for this tuple; otherwise, the key grouping strategy is selected. The keys' popularities in the stream may change over time. In the case that a hot key becomes unpopular, the predictor supports probabilistically decreasing the estimated frequency of a key. When a previous hot key becomes unpopular, it will be kicked out from the hot key filter.

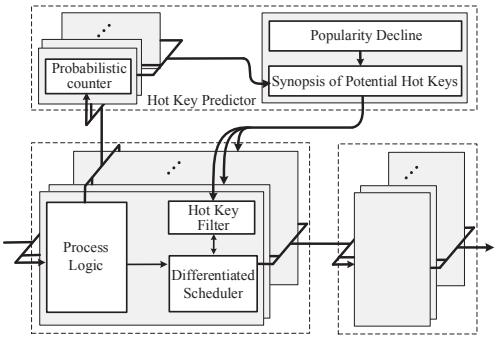

Fig. 4. Architecture of DStream

As we can see the differentiated scheduler is extremely computation and memory efficient. By well integrating the differentiated scheduler in a processing element instance, the system can greatly improve the throughput. The differentiated scheduler partitions the tuples with hot keys using shuffle grouping, while assigns the tuples with unpopular keys using key grouping. On one hand, the heavy workloads incurred by a hot key will be evenly shared by all the downstream processing element 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 unpopular keys, the hash based key grouping scheme partitions the key space among the multiple downstream instances. Each instance only needs a relatively small memory space to store the states of the fraction of the unpopular keys assigned to it. Therefore, we can upgrade the level of data parallelism, i.e., generating more processing element instances, to improve the system throughput.

B. Hot Keys Predictor

Due to the rigorous requirements in system efficiency for a distributed streaming processing system to cope with the highly dynamic real-time streams, identifying the current hot keys mainly has three aspects of difficulties. First, the representation of the set of hot keys needs to be very 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 the 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-world stream applications.

To meet the above requirements, the predictor of the DStream uses a two-level architecture as shown in Fig. 5. The first level identifies the list of hot keys using a novel probabilistic counting scheme. The achieved probabilistic counts 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 Counting Bloom filter. The succinct hot keys filter is loaded in the memory of a processing element instance. The synopsis consists of a set of bit vectors. Each bit vector represents a probabilistic count of a potential hot key. The Counting Bloom filter represents the set of identified hot keys. When the data

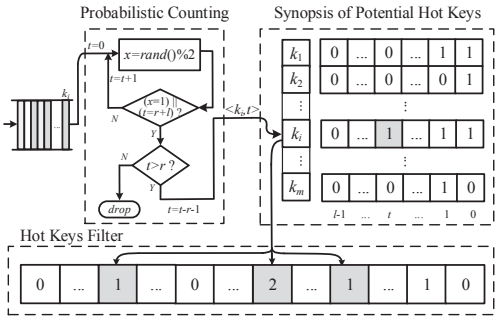

Fig. 5. Hot keys predictor

flow comes, the predictor performs a probabilistic counting operation as following. For a coming tuple with 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. If the value of 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 l-length bit vector, $bitvec\_k_i$, with all the bits initially set to "0". It then sets the $(t_i\text{-}r\text{-}1)^{th}$ 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 $(t_i\text{-}r\text{-}1)^{th}$ bit of the bit vector for k_i to "1". Accordingly, the predictor only flips the coin up to (l+r) times, i.e, the maximal value of t_i is limited to (l+r).

As shown in Fig. 5, an important component of the hot key predictor is the probabilistic counter. It performs the coin flipping experiments. Intuitively, if the key k_i frequently appears in a stream, 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 2^{t_i}. Using this intuition, the predictor can get an estimated count of tuples associated with k_i. A large value of t_i indicates that k_i is likely to be a popular key. We notice that for each tuple the coin flipping experiment is conducted independently. This means an unpopular key also has probability to get 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 popular key. Using such a scheme, we can achieve a precise prediction of hot keys. We will give the formal proof of the lower bound of the precision in Section IV.A.

We store the experiment values of the possible popular keys in the synopsis of potential hot keys. Thus, each bit vector in the synopsis has at least one bit set to "1". To achieve the precise prediction, we only insert an identified hot key k_i, i.e., the associated $bitvec\_k_i$ with more than one "1" bit, into the Counting Bloom filters. The hot key filter is loaded in the memory of the processing element instances. Due to the computation efficiency of a CBF, the hot key filter supports fast set membership checking within a constant time, i.e., the predictor can quickly decide whether the key of the current tuple is hot or not. In Section IV.A, we will show the theory analysis for the precision of the predictor in detail. We will also introduce how to cope with the dynamic changes of popularities of keys in a real-time stream.

IV. Theoretical Analysis

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

A. Precision of the Hot Key Prediction

With the hot key predictor, a key should be identified to be a hot key if and only if it appears more frequently than a given threshold in the stream. Simply counting the actual appearing times of every key needs to maintain a counter for every key, leading to heavy memory cost. To alleviate the overhead, the hot key predictor of DStream leverages a synopsis of potential hot keys and a novel probabilistic counting technique. It is clear that, the probability for a coin flipping experiment to achieve a resulting 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. That is to say the hotter a key is, with the higher probability it can obtain a large experiment value. Note that obtaining a large experiment value does not necessarily mean finding a hot key, because the coin flipping experiments of tuples are conducted independently. We observe, if a key has obtained a large experiment value more than once, it has very high probability to be a hot key. We present the lower bound of the probability in Theorem 1.

Theorem 1. *If a key has performed the coin flipping experiment N_0 times, and obtains no less than ζ ($\zeta > 1$) different experiment values which are larger than a given threshold r, the lower bound of the probability that N_0 is larger than $2^c (c > r)$ is,*

$$P(N_0 \geq 2^c | \zeta) > (1+\frac{\sum_{i=0}^{2^\zeta}(1-(1+(i-\zeta+1)/2^{r+\zeta-2})e^{-((i-\zeta+1)/2^{r+\zeta-2})})}{\sum_{i=2^\zeta+1}^{\infty}(1-e^{-(i/2^{r+\zeta})})^{2^\zeta-1}})^{-1} \quad (1)$$

In practice, the parameter ζ can be configured to different positive integers according to the precision requirement. For example, DStream system hot keys predictor sets $\zeta = 2$ by default. Thus, the inequality can be simplified as,

$$P(N_0 \geq 2^c) > (1+2^{c-r})e^{-2^{c-r}} \quad (2)$$

Before giving the formal proof of Theorem 1, we first present the following lemma.

Lemma 1. *If a key has performed the coin flipping experiment N_0 times, the probability $P(N_0)$ that it can obtain at least ζ ($\zeta > 1$) different experiment values which are larger than a given threshold r satisfies,*

$$\begin{cases} P(N_0,\zeta) > (1-e^{-(N_0/2^{r+\zeta})})^{2^\zeta-1} \\ P(N_0,\zeta) < 1-(1+(N_0-\zeta+1)/2^{r+\zeta-2})e^{-(N_0-\zeta+1)/2^{r+\zeta-2}} \end{cases} \quad (3)$$

Due to the space limit, we omit the proof of Lemma 1. With Lemma 1, we give the proof of Theorem 1 in the following.

Proof of Theorem 1. Assuming there are totally U unique keys $\{k_1, k_2, ..., k_i, .., k_U\}$ in the stream, and they are sorted

in descending order by the times they have performed the coin flipping experiment $\{N_1, N_2, ..., N_i, .., N_U\}$, i.e., $N_1 \geq N_2 \geq ... \geq N_i \geq ... \geq N_U$. Let A_i denote the case that k_i has obtained at least $\zeta(\zeta > 1)$ different experiment values which are larger than r. Thus, the probability in Theorem 1 satisfies,

$$\begin{aligned} P(N_0 \geq 2^c|\zeta) =& \frac{\sum\limits_i P(N_i \geq 2^c \cup A_i)}{\sum\limits_i P(A_i)} \\ =& \frac{\sum\limits_{N_i \geq 2^c} \frac{N_i}{\sum\limits_{j=0}^{U} N_j} P(N_i|\zeta)}{\sum\limits_{i=0}^{U} \frac{N_i}{\sum\limits_{j=0}^{U} N_j} P(N_i|\zeta)} \end{aligned} \tag{4}$$

Combing Eq.(4) with Inequality (3), we obtain,

$$\begin{aligned} & P(N_0 \geq 2^c|\zeta) \\ =& \frac{\sum\limits_{N_i \geq 2^c} N_i P(N_i|\zeta)}{\sum\limits_{i=0}^{U} N_i P(N_i|\zeta)} \\ =& \frac{\sum\limits_{N_i \geq 2^c} N_i P(N_i|\zeta)}{\sum\limits_{N_i \geq 2^c} N_i P(N_i|\zeta) + \sum\limits_{N_i < 2^c} N_i P(N_i|\zeta)} \\ =& (1 + \frac{\sum\limits_{N_i < 2^c} N_i P(N_i|\zeta)}{\sum\limits_{N_i \geq 2^c} N_i P(N_i|\zeta)})^{-1} \\ >& (1 + \frac{\sum\limits_{i=0}^{2^c} (1 - (1 + (i - \zeta + 1)/2^{r+\zeta-2}) e^{-((i-\zeta+1)/2^{r+\zeta-2})})}{\sum\limits_{i=2^c+1}^{\infty} (1 - e^{-(i/2r+\zeta)})^{2^\zeta - 1}})^{-1} \end{aligned} \tag{5}$$

Theorem 1 is thus proved. ■

According to Inequality (2), for any given potential hot key threshold r, if we set $c = r - 2$, we can obtain $P(N_0 \geq 2^r) > 99.28\%$. Table 1 shows the lower bound of the precision if we set r = 10. We find that the precision of the hot keys prediction can be guaranteed with high probability.

TABLE I
PRECISION OF THE HOT KEYS PREDICTION

Threshold 2^c	128	256	512	1024
Lower bound of $P(N_0 \geq 2^c)$	99.28%	97.35%	90.98%	73.53%

B. Adaption to Dynamic Popularity Changes

The popularities of the keys along the stream may change over time. The DStream scheduler is mainly concerned about the current arriving rate of a key rather than the accumulated frequency of a key. It is clear, for any given key, if it rarely appears in a current period of time, it will not incur heavy workload, no matter how popular it used to be. The main idea of the popularity adaption strategy in DStream 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 coming stream. Specifically, to adapt to the dynamic popularities changing of keys in the stream, we design a decline mechanism on top of the synopsis of potential hot keys.

Given the synopsis has a uniform decline rate for every key, if a 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 IV.A, the synopsis uses the total number of "1" bits to identify the hot keys. DStream decreases the value of the bit vector without increasing the total number of "1" bits each time. Specifically, it performs an operation of bit-wise right shift on each bit vector in the synopsis with a stable 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 "1" bit sets to a key's bit vector, after at most a number of l bit-wise right shifts (where l is the length of the bit vector), this associated key will be evicted from the synopsis. For controlling the decline rate, DStream uses a fixed probability p $(0 < p < 1)$ to perform 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 a number of P bit vectors independently and uniformly at random from all the m bit vectors and thus the decline rate is $p = \frac{P}{m}$. In the following, we will prove that the DStream decline strategy is sensitive to the arriving rate.

Theorem 2. *For a key, if its arriving rate f is larger than $2p$ ($p < 1$), 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 $(\frac{1}{p}) * 2^r$ coin flipping experiments. In each interval, the expected times of the appearances of a key with arriving rate f is $N_f = (\frac{1}{p}) * 2^r * f$. For this key, the expected largest experiment value is,

$$\begin{aligned} V(f) =& \sum_{i=0}^{\infty} i((1 - \tfrac{1}{2^{i+1}})^{N_f} - (1 - \tfrac{1}{2^i})^{N_f}) \\ =& \lim_{i \to \infty} (i(1 - \tfrac{1}{2^{i+1}})^{N_f}) - \sum_{i=0}^{\infty} (1 - \tfrac{1}{2^i})^{N_f} \\ =& log_2 N_f + O(1) \\ =& log_2(\frac{f}{p}) + r + O(1) \end{aligned} \tag{6}$$

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, with high probability there will be a higher bit set to "1". 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 higher bit to "1" is very low. Thus, the highest "1" bit keeps right shifting until the key is evicted from the synopsis. This needs $(\frac{1}{p})^{log_2(\frac{f}{p})+O(1)}$ synopsis updates. Theorem 2 is thus proved. ■

C. 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 decline rate is too low, new potential hot keys will be kept inserting into the synopsis before the previously hot but currently unpopular keys being evicted. With fixed memory space, the synopsis will soon become full and the future hot keys may fail to be inserted. On the contrary, if the decline rate is too high, keys will be evicted from the synopsis quickly. The synopsis will become empty rapidly and the currently hot keys may be soon evicted before be identified. This will result in poor space utilization and performance of the predictor. In the following, we prove that in DStream for any given decline rate p, the number of keys in the synopsis is stable with explicit upper and lower bounds. The synopsis never becomes empty or full.

Theorem 3. *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 2, 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 not higher than $2p$. For simplicity, we call these keys rare keys.

Each time when the synopsis stores the experiment value of a new potential hot key detected by the coin flipping experiment, the probability that the 0^{th} bit of the bit vector of this key is set to "1" is $\frac{1}{2}$. The probability that the potential hot key is a rare key is at most $(1 - H * (2p))$. After $\frac{1}{p}$ synopsis updates, every bit vector has high probability to right shift. Thus, at most a number of $\frac{1}{2}(1 - H * (2p)) * \frac{1}{p}$ new rare keys remain in the synopsis.

Assume there are X_τ keys in the synopsis at a given time τ. With Theorem 2, 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 from the synopsis. Thus, we can obtain,

$$X_{\tau+\frac{1}{p}} < X_\tau - \frac{1}{2}(X_\tau - H) + \frac{1}{2}(1 - pH) * \frac{1}{p} = \frac{1}{2}X_\tau + \frac{1}{2p} \quad (7)$$

Since the synopsis is initially empty, from (7), it is clear that for any time τ, $X_\tau < \frac{1}{p}$ holds. Theorem 3 thus proved. ■

V. Implementation

In this section we present the implementation of DStream hot key predictor on top of Apache Storm [23], the most popular streaming processing frameworks.

In the implementation, we build an independent component to execute the task of the hot key predictor, which consists of a coin bolt and a global synopsis bolt (a bolt is the basic processing element of Storm). The coin bolt receives tuples from the original user logic bolt, performs the coin flipping experiment, and then sends the potential hot keys to the synopsis bolt. The synopsis bolt maintains a HashMap in memory. The HashMap consists of a set of l-length bit vectors for each potential hot key.

When each instance of the user logic bolt sends a tuple to the downstream bolt, the system generates a signal tuple associated with the same key to the coin bolt. To avoid the coin bolt to be the processing bottleneck, the system creates a number of instances for the coin bolt, which is not less than the number of instances of the user logic bolt.

Following the logic described in Section IV, the synopsis bolt receives potential hot keys from the coin bolt. The bolt then updates the corresponding bit vector and checks whether there are more than one "1" bits in this bit vector. After updating each time, 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 out the keys with an empty bit vector out of the HashMap. When the bolt detects a current hot key or kicks out an unpopular key, the synopsis bolt generates a tuple with a signal "append" or "delete" for this key. The bolt then broadcasts the tuple to all the associated instances of the user logic bolt. Since the coin bolt filters the vast majority of tuples, the synopsis bolt will not become the system bottleneck.

Each instance of the user logic bolt locally maintains a succinct Counting Bloom filter as the hot key filter. The CBF updates according to the signal tuples sent by the synopsis bolt. When the instance is going to send out a tuple, it declares the tuple is hot or not by querying the CBF. Thus, the system can choose the suitable scheduling scheme for the tuple.

VI. Experiments

We implement DStream [1] atop Apache Storm 1.0.2 [23]. In this section we evaluate the performance of the DStream system using comprehensive experiments with large-scale datasets of real world systems.

A. Methodology

We deploy the DStream system on a cluster consisting of 33 machines. Each machine is equipped with an octa-core 2.4GHz Xeon CPU, 64GB RAM, and a 1000Mbps Ethernet interface card. One machine in the cluster serves as the master node to host the Storm Nimbus. The other machines run Storm supervisors.

In the experiment, we use three large-scale traces collected from real-world systems. The first is a set of tweets crawled from Twitter. This dataset contains 658 million words associated with 5.7 million unique keys. The second is a real-time stock exchange data collected from NASDAQ [11] during April 2017. This dataset contains 274 million exchange records associated with 6,649 stock symbols. The third is the hashtag dataset of Twitter collected during Nov. 2012 [16]. The dataset includes 43 million hashtags. A hashtag is a label with a "#" character at the beginning. We also conduct extended experiments with generated synthetic datasets with one billion tuples and 10 million keys at different levels of skewness. The zipf coefficients of the datasets vary from 0.5 to 2.0. With a

TABLE II
Statistics of the Stream Data

Dataset	Symbol	♯ of tuples	♯ of keys
Tweets	TW	658M	5.7M
Stock exchange records	ST	274M	6.7K
Hashtags	HA	43M	3.4M
Synthetic zipf	ZF	1B	10M

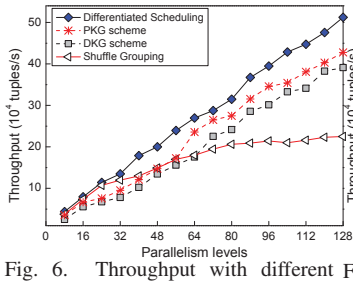

Fig. 6. Throughput with different parallelism levels

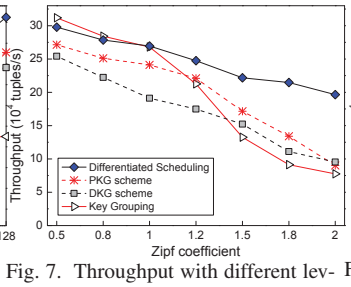

Fig. 7. Throughput with different levels of skewness

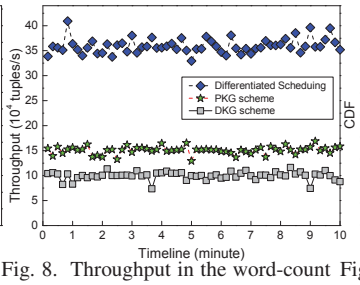

Fig. 8. Throughput in the word-count application

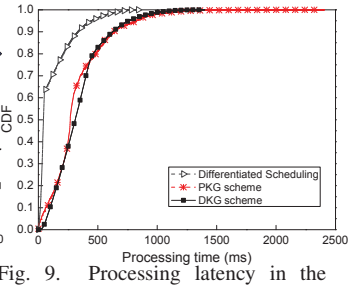

Fig. 9. Processing latency in the word-count application

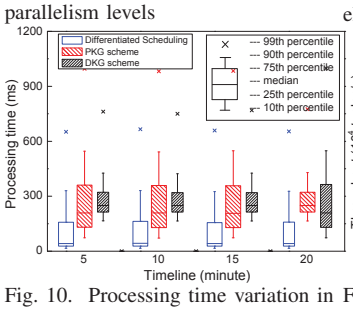

Fig. 10. Processing time variation in the word-count application

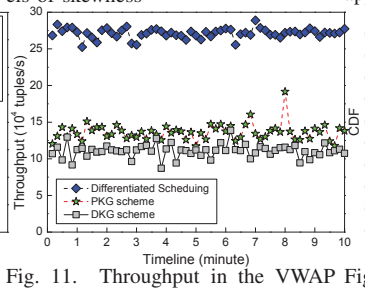

Fig. 11. Throughput in the VWAP application

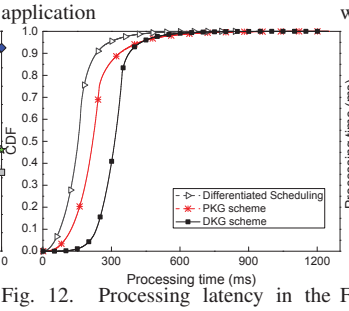

Fig. 12. Processing latency in the VWAP application

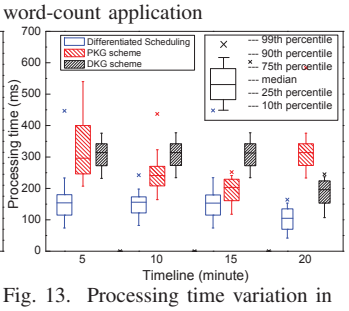

Fig. 13. Processing time variation in the VWAP application

higher zipf coefficient, the data distribution is more skewed. Table 2 summarizes the details of these datasets.

Using these datasets, we evaluate the performance of three stream applications including word-count, *Volume Weighted Average Prize* (VWAP), and Hashtag statistics. We leverage Kafka [2] for the stream input. Accordingly, the source spout in the topology is implemented as a Kafka Spout to keep pulling new messages from the Kafka Brokers.

The word-count and the hashtag applications count the number of each word or hashtag in the tweets stream. For these applications, the Kafka Spout feeds these messages to the split bolt which splits tweets into simple words or extracts the hashtags. The second stage of the topology is a count bolt which counts for each distinct word/hashtag in a given period of time. The third stage is an aggregation bolt which aggregates and outputs the results.

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

We compare the performance of DStream with that of the state-of-the-art *Partial Key Grouping* [19] (PKG) and *Distribution-aware Key Grouping* [21] (DKG) schemes. We mainly examine the metrics including the system throughput, processing latency, and the load imbalance among processing element instances. The throughput is defined as the rate of successful processed tuples over the distributed stream processing system. While the processing latency is defined as the average processing time for a tuple. High throughput and low latency are always desirable in a distributed stream processing system [19]. To achieve high system efficiency, a distributed stream processing system also desires alleviated load imbalance among processing instances. We use the standard deviation to measure the fraction of imbalance.

Specifically, we compute the number of tuples in each interval of ten seconds to obtain the system throughput. We record the total processing latency for each 10,000 tuples which are emitted consecutively and report the average processing latency. To monitor the successfully processing of each tuple, we leverage the acknowledge mechanism in Storm. We regard a tuple to be processed if and only if the Kafka Spout receives the ACK message for this tuple from the last bolt.

B. Results

In the experiment, we first compare the efficiency of the computation and memory resources of DStream with key grouping and shuffle grouping schemes by using the synthetic datasets to run the word-count application. We further compare DStream with PKG and DKG by using large-scale traces from real-world systems.

Figure 6 shows the throughput with different parallelism levels. We compare DStream with the shuffle grouping scheme and the other two baseline schemes. The experiment fixes the popularity distribution with zipf coefficient = 1.0 and increases the number of processing element instances. Specifically, we create 8∼128 instances of the count bolts and examine the maximum throughput of the system. At each parallelism level, we adjust the speed of emitting tuples to achieve the maximum throughput. The result shows that when the parallelism level reaches 80, the system throughput stops increasing with shuffle grouping. All the other three schemes make the system throughput keep increasing. Among these three schemes, DStream has the best performance. When the parallelism level reaches 128, the throughput of DStream is 32% and 21% higher than that of DKG and PKG schemes, respectively.

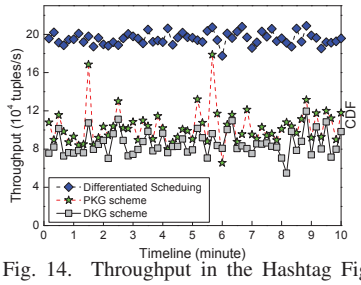

Fig. 14. Throughput in the Hashtag statistic application

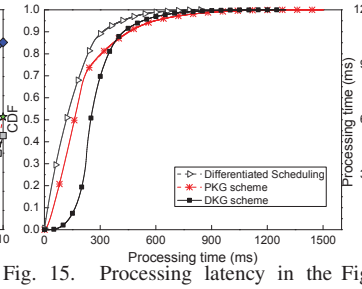

Fig. 15. Processing latency in the Hashtag statistic application

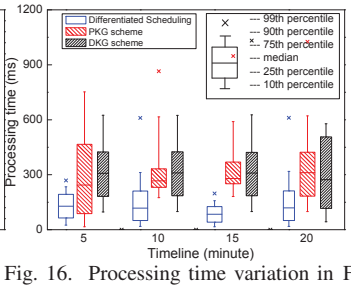

Fig. 16. Processing time variation in the Hashtag statistic application

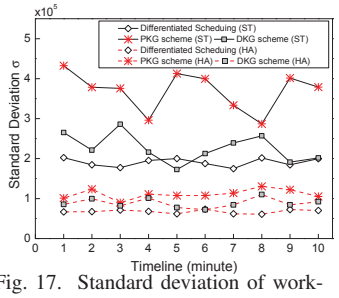

Fig. 17. Standard deviation of workloads among different instances

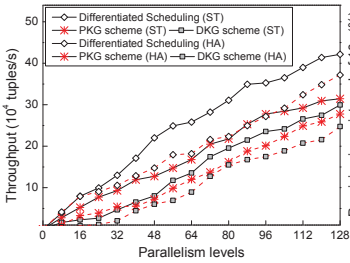

Fig. 18. The optimal throughput of TW and HA

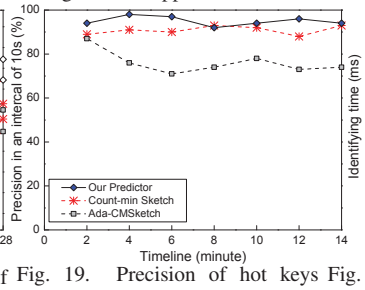

Fig. 19. Precision of hot keys predictor

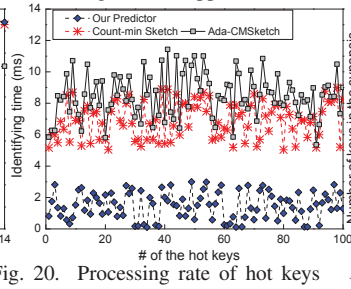

Fig. 20. Processing rate of hot keys predictor

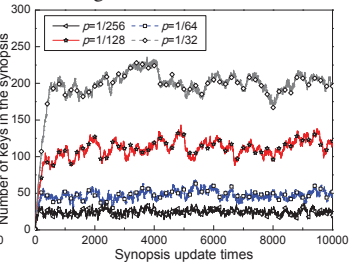

Fig. 21. Load factor of synopsis

Figure 7 plots the system throughput by using synthetic datasets. We compare DStream with the key grouping scheme as well as the other two baseline schemes. We fix the parallelism level to 64. The result shows that the system throughput decreases sharply when the coefficient increases by using key grouping and the other two schemes. The result show clearly that using DStream the throughput does not decrease significantly.

Figure 8 compares the throughput of DStream with those of PKG and DKG in the word-count application by using the TW datasets. In this experiment, we fix the parallelism level to 128, and record the throughput in an interval of ten seconds. The results show that DStream achieves 2.3× and 3.6× improvements of the average system throughput compared to PKG and DKG, respectively.

Figure 9 plots the CDF of the average latency for the word-count application. The result shows that the average tuple processing latency of DStream is 114 ms, while the average tuple processing latency of PKG and DKG is 322 ms and 336 ms, respectively. DStream greatly reduces the latency of PKG by 64% and reduces the latency of DKG by 66%. The results also show that a fraction of tuples using PKG or DKG have much longer latency than the average. The results show that 90% tuples using DStream can be processed in 330 ms, while only 63% and 45% tuples of PKG and DKG can be processed in such a short period of time.

Figure 10 further shows the distribution of each tuple's processing time every five minutes. The results show that DStream achieves the lowest latency at every percentile. DStream also achieves more stable latency for tuple processing compared to the baseline schemes. Between the fifth and twentieth minutes, both the distributions of PKG and DKG change greatly due to the change of frequency of hot keys.

Figure 11 compares the throughput of DStream with those of PKG and DKG in the VWAP application using the ST datasets. The results show that DStream achieves 2.0× and 2.4× improvements of the average system throughput in VWAP application compared to PKG and DKG, respectively.

Figure 12 plots the CDF of the average latency of the VWAP applications. The average tuple processing latency of DStream is 174 ms. The result shows that DStream reduces the latency of PKG and DKG by 31% and 50%, respectively.

Figure 13 shows the distribution of the processing time of each tuple in the VWAP application. Due to the highly dynamic change of the stock exchanging records, the distributions of both DKG and PKG schemes vary significantly. In contrast, the result shows that DStream has a very stable performance. This demonstrates that our hot key predictor can more precisely predict the hot keys in the presence of dynamic changes of keys' popularities over time. The PKG and DKG schemes use the accumulated frequency information to predict the popularity information of keys and are not able to indicate the current hot keys.

Figure 14 compares the throughput of DStream with those of PKG and DKG in the Hashtag statistic application with the HA datasets. The experiment results show that DStream achieves 1.9× and 2.3× improvements of the average system throughput compared to PKG and DKG, respectively.

Figures 15 and 16 plot the CDF of the average latency and the distribution of the processing time in the Hashtag statistic application. The result shows that DStream can achieve very low and stable processing latency. The average tuple processing latency of DStream is 148 ms, while the average tuple processing latency of PKG and DKG is 210 ms and 280 ms, respectively. DStream greatly reduces the latency of PKG and DKG by 30% and 48%.

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 these workloads to show the degree of load imbalance. Figure 17 shows that DStream reduces the average standard deviation of DKG by 16% in VWAP and by 25%

in Hashtag statistic. DStream reduces the average standard deviation of PKG by 49% in VWAP and by 40% in Hashtag statistic. Although the DKG scheme rarely achieves slightly lower standard deviation, DStream has much lower and more stable deviation of workloads.

In the experiment, we further examine the scalability of DStream. The system increases the number of processing element instances (i.e., the parallelism level) and achieves the optimal throughput of the system for each parallelism level. In the experiments, for each parallelism level, the Kafka Spout increases the emitting speed until the throughput no longer increases. Figure 18 shows the optimal throughput for the VWAP and the Hashtag statistic applications. The results show that DStream achieves much higher throughput than the baseline schemes. When the parallelism level reaches 128, the throughput of DStream for the VWAP and the Hashtag statistic application is 35% and 33% higher than those of PKG. DStream achieves 40% and 49% higher the throughput than that of DKG in the VWAP and the Hashtag statistic application, respectively.

Figure 19 shows the precision of our hot key predictor and the existing sketch based schemes. In the experiment, we record all the tuples processed. We count the exact distribution of the keys in these tuples and compute the precision in an interval of ten seconds. The precision is defined as the ratio of the number of the real hot keys to the total number of keys identified. The results show that DStream achieves a prediction precision of 96% which is the best among all the schemes. Our hot key predictor significantly improves the precision of the previous count-min sketch scheme by 23%.

We record the time of a tuple from the time it is received by the predictor to the time it is identified to be a hot key. Figure 20 plots the time of 100 continuous hot keys identified by DStream and the two sketch based schemes. The result shows that DStream identifies a hot key in an average time of 1.4 ms, significantly reducing the time of the count-min sketch scheme and the Ada-CMSketch scheme by 80% and 84%, respectively.

To evaluate the space efficiency of the proposed predictor, we track the total number of items in the synopsis. We examine different decline rate p settings in the experiment. The result in Fig. 21 shows that the total number of the items in the synopsis becomes stable after a short interval. The results show that the average number of items in the synopsis is increasing inversely proportional to the value of p.

VII. Conclusions

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

VIII. Acknowledgements

This research is supported in part by The National Key Research and Development Program of China under grant No.2016QY02D0202, NSFC under grants Nos. 61422202, 61370233, 61433019, Foundation for the Author of National Excellent Doctoral Dissertation of PR China under grant No.201345, and Research Fund of Guangdong Province under grant No.2015B010131001.

References

[1] DStream, http://github.com/CGCL-codes/DStream, 2017.
[2] Apache Kafka, http://kafka.apache.org/, 2017.
[3] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," in *VLDB*, 2013.
[4] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *INFOCOM*, 2016.
[5] 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.
[6] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," in *VLDB*, 2010.
[7] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[8] 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.
[9] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013.
[10] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB J.*, vol. 23, no. 4, pp. 517–539, 2014.
[11] http://www.nasdaq.com/, 2017.
[12] M. Kleppmann and J. Kreps, "Kafka, samza and the unix philosophy of distributed data," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 4–14, 2015.
[13] 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 *SIGMOD*, 2015.
[14] M. Li, X. Hui, M. Endo, and K. Kishimoto, "A quantitative model for intraday stock price changes based on order flows," *J. Systems Science & Complexity*, vol. 27, no. 1, pp. 208–224, 2014.
[15] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB*, 2002.
[16] K. McKelvey and F. Menczer, "Design and prototyping of a social media observatory," in *WWW*, 2013.
[17] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *ICDT*, 2005.
[18] 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.
[19] 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 *ICDE*, 2015.
[20] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW*, 2010.
[21] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, "Efficient key grouping for near-optimal load balancing in stream processing systems," in *DEBS*, 2015.
[22] A. Shrivastava, A. C. König, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *SIGMOD*, 2016.
[23] 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 *SIGMOD*, 2014.
[24] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *SOSP*, 2013.