# A Hybrid Local and Distributed Sketching Design for Accurate and Scalable Heavy Key Detection in Network Data Streams

Qun Huang[a], Patrick P. C. Lee[a]

[a]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*

**Abstract**

Real-time characterization of network traffic anomalies, such as heavy hitters and heavy changers, is critical for the robustness of operational networks, but its accuracy and scalability are challenged by the ever-increasing volume and diversity of network traffic. We address this problem by leveraging parallelization. We propose LD-Sketch, a data structure designed for accurate and scalable traffic anomaly detection using distributed architectures. LD-Sketch combines the classical counter-based and sketch-based techniques, and performs detection in two phases: local detection, which guarantees zero false negatives, and distributed detection, which reduces false positives by aggregating multiple detection results. We derive the error bounds and the space and time complexity for LD-Sketch. We further analyze the impact of ordering of data items on the memory usage and accuracy of LD-Sketch. We compare LD-Sketch with state-of-the-art sketch-based techniques by conducting experiments on traffic traces from a real-life 3G cellular data network. Our results demonstrate the accuracy and scalability of LD-Sketch over prior approaches.

**Note:** An earlier conference version of the paper appeared in IEEE INFOCOM 2014 [15]. We extend our proposed design with the consideration of the ordering of data items (Section 5). We also correct the flaws of the lemmas and theorems in our conference version.

## 1. Introduction

Characterizing traffic anomalies is important for administrators to maintain the robustness of a network. There are two types of anomalies that are of particular interest: flows with persistently large data volume (known as "heavy hitters") and flows with abrupt changes of data volume (known as "heavy changers"), since they may imply the existence of denial-of-service attacks, component failures, or service level agreement violation. Traffic anomalies are detrimental to network robustness and must be detected and suppressed in real-time.

However, today's IP networks continuously grow in size and complexity. Thus, characterizing traffic anomalies in real-time becomes more challenging, particularly in two aspects:

- *Enormous key space.* The complexity of anomaly detection is overwhelmed by the number of keys being monitored. For example, 5-tuple network flows are defined by 104-bit keys (i.e., source/destination IP addresses, source/destination ports, and protocols). Keeping track of $2^{104}$ flow keys can imply huge memory usage.

- *Line-rate packet processing.* Packet processing must keep pace with the increasing line rate to meet the real-time requirement. Conventional single-processor platforms no longer provide enough computational power to achieve this goal. To improve scalability, anomaly detection needs to be performed on *distributed* packet streams in parallel. However, providing both accuracy and scalability guarantees becomes challenging when aggregating the detection results from multiple sources.

Anomaly detection has been extensively studied in the context of data streaming. To deal with the enormous key space, counter-based techniques (e.g., [12, 19–21]) and sketch-based techniques (e.g., [4, 5, 7–11, 13, 16, 17, 23]) propose space-efficient data structures for anomaly detection and derive error bounds. Counter-based techniques use

*Email addresses:* `qhuang@cse.cuhk.edu.hk` (Qun Huang), `pclee@cse.cuhk.edu.hk` (Patrick P. C. Lee)

an associative array to monitor frequent items and are designed for heavy hitter detection; sketch-based techniques project data items into a subset of buckets in a summary called *sketch* and are designed for both heavy hitter and heavy changer detections. Although theoretically sound, such techniques are mainly studied and evaluated in the single-processor paradigm. With the emergence of distributed streaming architectures (e.g., Flume [2], S4 [22], and Storm [25]), an open issue is to seamlessly parallelize such techniques to achieve accurate and scalable anomaly detection.

In this paper, we study the traffic anomaly detection problem from both theoretical and implementation perspectives. We propose *LD-Sketch*, a novel sketching design that combines the counter-based and sketch-based techniques to accurately and scalably detect heavy hitters and heavy changers using distributed architectures. Its main idea is to augment a sketch in which each bucket keeps track of anomaly candidates in an associative array, similar to counter-based techniques. We enhance our counter-based technique to allow the associative array to be dynamically expandable based on the current number of anomaly candidates associated with the bucket. LD-Sketch performs detection in two phases: (i) *local detection*, which guarantees no false negatives and identify the anomaly candidates (including true anomalies and false positives) in a single compute node called *worker*, and (ii) *distributed detection*, which reduces false positives (with a slight increase in the false negative rate) by combining detection results from multiple workers. Thus, not only do we exploit streaming architectures to improve scalability, but we also use their distributed nature to improve the detection accuracy.

In summary, we make the following contributions:

- We design LD-Sketch, which enables accurate and scalable detection of heavy hitters and heavy changers and is seamlessly deployable in distributed architectures. We derive the error bounds, space complexity, and time complexity when LD-Sketch is used in both local detection (i.e., using a single worker) and distributed detection (i.e., using multiple workers).

- We analyze how the ordering of data items influences the memory usage and accuracy of LD-Sketch. By leveraging the ordering property of data items, we propose two enhancement heuristics that respectively reduce the memory usage and the false positive rate of LD-Sketch.

- We implement and compare LD-Sketch with state-of-the-art sketch-based techniques by conducting trace-driven experiments using traces from a real-life 3G cellular network. We show that LD-Sketch achieves higher accuracy than other approaches, and improves accuracy and scalability using multiple workers in a distributed setting.

The rest of the paper proceeds as follows. Section 2 formulates the heavy hitter/changer detection problem. Sections 3 and 4 describe and analyze the local and distributed detection procedures of LD-Sketch, respectively. Section 5 studies the impact of ordering of data items and proposes two enhancement heuristics for LD-Sketch. Section 6 presents our trace-driven evaluation results. Section 7 reviews related work, and finally Section 8 concludes the paper. We also present our proofs of lemmas in Appendix.

## 2. Problem Formulation

We formulate the problem of heavy hitter/changer detection (which we collectively call *heavy key detection*). Table 1 summarizes the major notation in this paper.

### *2.1. Distributed Heavy Key Detection*

We first describe the distributed architecture, as shown in Figure 1, on which our heavy key detection problem is formulated. Our architecture is based on that of [6]. Specifically, we consider an architecture with $p \geq 1$ *remote sites* and $q \geq 1$ *workers*. A remote site is the source of a data stream, while a worker performs heavy key detection based on the streams from multiple remote sites. The architecture has a bipartite structure, in which each remote site can connect to all workers, while there is no communication among the remote sites and among the workers.

We periodically perform heavy key detection on the streams of data items from the $p$ remote sites, and we refer to each time period of detection as an *epoch*. Each data item is represented by a tuple $(x, v_x)$, where $x$ is the key drawn from a domain $[n] = \{0, 1, \cdots, n-1\}$ of size $n$ and $v_x$ is a value associated with $x$. For example, in network traffic

Table 1: Major notation used in the paper.

| **Notation** | **Meaning** |
|---|---|
| | Defined in Section 2 |
| $p$ | number of remote sites |
| $q$ | number of workers |
| $[n]$ | key domain |
| $(x, v_x)$ | data item with key $x$ and value $v_x$ |
| $S(x)$ | true sum for key $x$ in an epoch |
| $D(x)$ | true difference for key $x$ between two adjacent epochs |
| $U$ | total sum of the values of all keys in an epoch |
| $\phi$ | heavy key threshold |
| $H$ | maximum possible number of heavy keys |
| $r$ | number of rows in a sketch |
| $w$ | number of buckets in one row in a sketch |
| $(i, j)$ | the $j$-th bucket in row $i$, where $1 \le i \le r$ and $1 \le j \le w$ |
| $f_i$ | hash function $\{0, 1, \ldots, n-1\} \to \{1, 2, \cdots, w\}$ for row $i$ |
| | Defined in Section 3 |
| $V_{i,j}$ | counter value of bucket $(i, j)$ in the sketch |
| $A_{i,j}$ | associative array in bucket $(i, j)$ ($A_{i,j}[x]$ denotes the counter value of key $x$) |
| $l_{i,j}$ | maximum length of array $A_{i,j}$ |
| $e_{i,j}$ | maximum error for keys in bucket $(i, j)$ |
| $T$ | expansion parameter |
| $k$ | current expansion number (starting from zero) |
| $S_{i,j}^{low}(x)$ | lower estimated sum for key $x$ in bucket $(i, j)$ |
| $S_{i,j}^{up}(x)$ | upper estimated sum for key $x$ in bucket $(i, j)$ |
| $D_{i,j}(x)$ | estimated difference for key $x$ in bucket $(i, j)$ |
| $\epsilon$ | approximation parameter in local detection |
| $\delta$ | upper bound of error probability |
| | Defined in Section 4 |
| $\gamma$ | approximation parameter in distributed detection |
| $d$ | number of workers to which a key is distributed |
| | Defined in Section 5 |
| $\hat{k}_{i,j}$ | controlled expansion number |
| $t_{i,j}$ | number of distinct keys in bucket $(i, j)$ |
| $t_{i,j}^{\geq}$ | number of keys satisfying $S(x) \geq T$ in bucket $(i, j)$ |
| $V_{i,j}^{\geq}$ | total sum of the values of all keys satisfying $S(x) \geq T$ in bucket $(i, j)$ |
| $V_{i,j}^{<}$ | total sum of the values of all keys satisfying $S(x) < T$ in bucket $(i, j)$ |
| $k_{i,j}^{\geq}$ | integer such that $k_{i,j}^{\geq} T \leq V_{i,j}^{\geq} < (k_{i,j}^{\geq} + 1)T$ |
| $k_{i,j}^{<}$ | integer such that $k_{i,j}^{<} T \leq V_{i,j}^{<} < (k_{i,j}^{<} + 1)T$ |

monitoring, $x$ may refer to a 5-tuple flow, and $v_x$ may refer to the byte counts of the flow. In each epoch, let $S(x)$ be the true sum of values of key $x$, and $D(x)$ be the true absolute difference of $S(x)$ of key $x$ in the current epoch and in the last epoch. Also, let $U = \sum_{x \in [n]} S(x)$ be the total sum of the values of all keys in the epoch. Our primary goal is to detect the heavy keys whose true sums or true absolute differences exceed an absolute-value threshold $\phi$ in an epoch. Specifically, a *heavy hitter* is a key $x$ whose $S(x) \geq \phi$, and a *heavy changer* is a key $x$ whose difference $D(x) \geq \phi$. For ease of presentation, we use the same threshold $\phi$ for both types of heavy keys, although it can be defined differently. In practice, it is infeasible to keep track of $S(x)$ and $D(x)$ for every key $x$ given the enormous key space (see Section 1), so we must approximate $S(x)$ and $D(x)$ using some space-efficient data structures.

Let $H$ be the maximum possible number of heavy keys. Thus, for heavy hitter detection, $H = \frac{U}{\phi}$; for heavy changer detection, $H = \frac{2U}{\phi}$ (i.e., twice the maximum possible number of heavy hitters in each epoch). Our space and time complexity results are expressed in terms of $H$, as in prior studies in the literature.

A heavy key detection algorithm typically consists of two procedures: the *update* procedure, which includes the value of each arriving data item into a data structure, and the *detection* procedure, which examines the data structure at the end of each epoch and reports the heavy keys. A good heavy key detection algorithm should introduce (i) small fractions of false positives (i.e., non-heavy keys being treated as heavy keys) and false negatives (i.e., true heavy keys

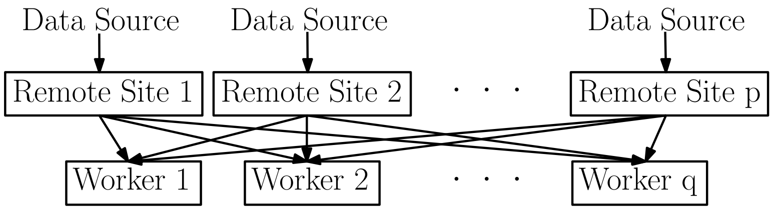


Figure 1: Distributed architecture for heavy key detection.

that are not reported), (ii) low space usage of the data structure, and (iii) low time complexity of both update and detection procedures.

## 2.2. Counter-based and Sketch-based Techniques

We describe two main classes of techniques that identify the heavy keys using space-efficient data structures: counter-based and sketch-based techniques.

*Counter-based techniques* keep individual counters for a subset of keys. For some performance parameter $l$, an associative array $A$ with at most $l-1$ key-value counters is defined. Suppose that we have a stream of data items $\{(x, v_x)\}$ and $v_x = 1$ for all $x$. If $x$ is in $A$, its counter value $A[x]$ is incremented by one; otherwise, if there are empty counters, a new counter is allocated for $x$ with value initialized to one; if no counter is available, each of other existing counters has its value decremented by one. We remove a key from $A$ if its counter value is zero. The value of each counter remaining in $A$ approximates the true sum of the corresponding key. Lemma 1 shows the error bound of the estimated value [21].

**Lemma 1** ([21]). *Consider a stream of data items $\{(x, v_x)\}$ and $v_x = 1$. In counter-based techniques, if $x$ is kept in $A$, then $A[x] \leq S(x) \leq A[x] + \frac{U}{l}$; if $x$ is not in $A$, its estimated value is set to zero and $0 \leq S(x) \leq \frac{U}{l}$.*

Counter-based techniques can identify all heavy hitters (without false negatives) with threshold exceeding $\phi = \frac{U}{l}$ by checking if a key has a positive counter value in the associative array. False positives may exist if some non-heavy hitters remain in the array and are not removed by the end of the epoch. To the best of our knowledge, it remains an open issue how counter-based techniques are applied to heavy changer detection, mainly because the associative array is not linear and we cannot combine two arrays of two epochs to describe the differences of data items.

*Sketch-based techniques* process a data stream in sub-linear space, and can identify both heavy hitters and heavy changers. A sketch is a small summary data structure projected from a large set. Specifically, we consider a sketch with $r$ rows. Each row $i$ ($1 \leq i \leq r$) is associated with $w$ buckets and an independent 2-universal hash function $f_i$ that hashes a key to one of the $w$ buckets. We denote the $j$-th bucket ($1 \leq j \leq w$) in row $i$ by $(i, j)$. Each bucket $(i, j)$ is associated with a counter value $V_{i,j}$ initialized at zero. For each data item $(x, v_x)$, the update procedure hashes key $x$ to one of the buckets in each of the $r$ rows, and increments the counter value by $v_x$. By Markov's inequality, we have the following lemma:

**Lemma 2.** *In sketch-based techniques, $\Pr\{V_{i,j} \geq \nu\} \leq \frac{U}{w\nu}$ for any sufficiently large $\nu > \frac{U}{w}$.*

Sketch-based techniques can identify all heavy hitters (without false negatives) with threshold exceeding $\phi$ by checking if the corresponding buckets of all $r$ rows have values exceeding $\phi$. False positives may exist if non-heavy hitters are hashed to the buckets with values above the threshold in all rows. For heavy changer detection, we compute the differences of counter values of each bucket in the two sketches in two adjacent epochs. By the linear property of a sketch, the difference of each bucket is also the sum of the differences of the keys hashed to the bucket. If a key has differences in the corresponding buckets of all $r$ rows (or a subset of them [7]) exceeding $\phi$, it is reported as a heavy changer. False negatives may also exist in heavy changer detection, for example, when a heavy changer with a positive change greater than $\phi$ and another heavy changer with a negative change less than $-\phi$ are hashed to the same bucket.

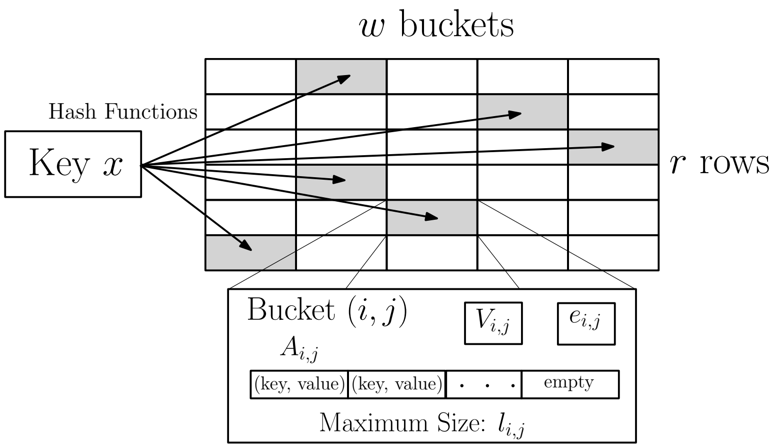


Figure 2: Structure of LD-Sketch.

### *2.3. Our Approach: Overview*

This work proposes LD-Sketch, a design that leverages the distributed architecture to achieve accurate and scalable heavy key detection. It follows a sketching design that combines the classical counter-based and sketch-based techniques. It builds on two phases: local detection and distributed detection, which we describe in Sections 3 and 4, respectively. In addition, in Section 5, we show that LD-Sketch can leverage the ordering of data items to trade off between space complexity and accuracy.

## 3. Local Detection

We present the local detection approach of LD-Sketch that aims to identify the heavy hitters and heavy changers in a single worker. We also analyze the error bounds, space complexity, and time complexity of the local detection approach.

### *3.1. Data Structure*

We first describe the data structure of LD-Sketch, as shown in Figure 2. An LD-Sketch is composed of $r$ rows with $w$ buckets each. Each bucket $(i,j)$ (where $1 \le i \le r$ and $1 \le j \le w$) corresponds to the $j$-th bucket in row $i$, and is associated with a counter value $V_{i,j}$, which will be incremented by $v_x$ for every incoming key $x$. We augment each bucket $(i,j)$ with three additional components, including: (i) $A_{i,j}$, which denotes the associative array used in counter-based detection, (ii) $l_{i,j}$, which denotes the maximum length of $A_{i,j}$, and (iii) $e_{i,j}$, which denotes the maximum estimation error for the true sums of the keys hashed to bucket $(i,j)$.

Our main idea is to use $A_{i,j}$ to keep track of the heavy key candidates that are hashed to the bucket $(i,j)$. Thus, when we restore all heavy keys in the detection procedure, we only inspect the tracked heavy key candidates in all $A_{i,j}$'s, and this significantly improves the accuracy and performance of our heavy key detection. One new extension to the counter-based technique of [21] is that we dynamically increase $l_{i,j}$ (i.e., the maximum size of $A_{i,j}$) as $V_{i,j}$ increases. We call this approach *dynamical expansion*. We define an expansion parameter $T$ as a function of the threshold $\phi$ (see Section 3.2), and the current expansion number $k = \lfloor V_{i,j}/T \rfloor \ge 0$, such that when $k$ is incremented, we increase $l_{i,j}$. The dynamic expansion approach trades memory for accuracy, i.e., by keeping track of more high-valued keys, we can restore the heavy keys more accurately.

Algorithm 1 details how we update a data item to an LD-Sketch. Initially, $V_{i,j}$ and $e_{i,j}$ are set to zero, $l_{i,j}$ is set to one and $A_{i,j}$ contains an empty counter for each bucket $(i,j)$ (where $1 \le i \le r$ and $1 \le j \le w$). For a data item $(x, v_x)$, we hash the key to some bucket $(i,j)$ via a hash function $f_i$ for each row $i$, and call the function UPDATEBUCKET. In essence, the function UPDATEBUCKET builds on counter-based techniques, with an extension of enabling dynamic expansion. Specifically, let us consider bucket $(i,j)$. If key $x$ is in $A_{i,j}$, we increment the counter $A_{i,j}[x]$ (Line 4); or if $A_{i,j}$ is not yet full, we insert $x$ to $A_{i,j}$ (Line 6). Otherwise, if $A_{i,j}$ is full, we either decrement the keys in $A_{i,j}$ or expand $A_{i,j}$. We consider both cases below.

**Algorithm 1** LD-Sketch Update Algorithm

```
Input: data item $(x, v_x)$; expansion parameter $T$
 1: function UpdateBucket($x, v_x, i, j$)
 2:     $V_{i,j} = V_{i,j} + v_x$
 3:     if $x \in A_{i,j}$ then
 4:         $A_{i,j}[x] = A_{i,j}[x] + v_x$
 5:     else if $A_{i,j}$ has less than $l_{i,j}$ counters then
 6:         Insert $x$ to $A_{i,j}$ and set $A_{i,j}[x] = v_x$
 7:     else
 8:         $k = \lfloor V_{i,j}/T \rfloor$
 9:         if $(k+1)(k+2) - 1 \leq l_{i,j}$ then
10:             $\hat{e} = \min(v_x, \min_{y \in A_{i,j}} A_{i,j}[y])$
11:             $e_{i,j} = e_{i,j} + \hat{e}$
12:             for all key $y \in A_{i,j}$ do
13:                 $A_{i,j}[y] = A_{i,j}[y] - \hat{e}$
14:                 if $A_{i,j}[y] \leq 0$ then
15:                     Remove $y$ from $A_{i,j}$
16:                 end if
17:             end for
18:             if $v_x > \hat{e}$ then
19:                 Insert $x$ to $A_{i,j}$ and set $l_{i,j} = v_x - \hat{e}$
20:             end if
21:         else                                        ▷ $l_{i,j} < (k+1)(k+2) - 1$
22:             add $(k+1)(k+2) - 1 - l_{i,j}$ new counters to $A_{i,j}$
23:             $l_{i,j} = (k+1)(k+2) - 1$
24:             Insert $x$ to $A_{i,j}$ and set $A_{i,j}[x] = v_x$
25:         end if
26:     end if
27: end function
28:
29: procedure Update
30:     for data item $(x, v_x)$ do
31:         for row $i = 1, 2, \ldots, r$ do
32:             $j = f_i(x)$
33:             UpdateBucket($x, v_x, i, j$)
34:         end for
35:     end for
36: end procedure
```

We first describe how we decrement the keys (Lines 10-20), and the steps are similar to the original counter-based technique [21]. The decrement value $\hat{e}$ is chosen to be the minimum of $v_x$ and the minimum value of $A_{i,j}$ (Line 10). We first add $\hat{e}$ to $e_{i,j}$ (which will be used in the detection procedure) (Line 11). Then we decrement all keys in $A_{i,j}$ by $\hat{e}$ and remove the keys whose values are no more than zero (Lines 12-17). If $v_x$ is not completely decremented, the residue $v_x - \hat{e}$ can be inserted to $A_{i,j}$, given that the some key must have been removed (Lines 18-20).

We next describe the dynamic expansion of $A_{i,j}$ (Lines 22-24). We keep track of the current expansion number $k = \lfloor V_{i,j}/T \rfloor$ (Line 8). We set $l_{i,j} = (k+1)(k+2) - 1$. When $A_{i,j}$ is full, if $l_{i,j} < (k+1)(k+2) - 1$, which happens when $k$ is incremented, then we add new counters to $A_{i,j}$.

If $T > v_x$ for any data item $x$, then $k$ is incremented by at most one for each new data item $x$. On the other hand, if some data item has a significantly large $v_x$, then $k$ may be aggressively incremented. This implies that we add a large number of new counters to $A_{i,j}$. In Section 5, we further enhance Algorithm 1 to control the expansion of $A_{i,j}$.

We use LD-Sketch to estimate the true sum $S(x)$ of each key $x$. Here, LD-Sketch produces a pair of estimates for each key $x$ and each bucket $(i, j)$: the lower estimate $S^{low}_{i,j}(x)$ and the upper estimate $S^{up}_{i,j}(x)$. If $x$ is in $A_{i,j}$, we set $S^{low}_{i,j}(x) = A_{i,j}[x]$; otherwise $S^{low}_{i,j}(x) = 0$. Also, we set $S^{up}_{i,j}(x) = S^{low}_{i,j}(x) + e_{i,j}$.

## *3.2. Heavy Key Detection*

We now explain how we identify heavy hitters and heavy changers using LD-Sketch.

For heavy hitter detection, we use a single LD-Sketch with expansion parameter $T = \epsilon\phi$, where $\epsilon \in (0, 1]$ denotes the approximation parameter that bounds the error rates. At the end of each epoch, we examine every bucket $(i, j)$ in the sketch. We identify every bucket $(i, j)$ with $V_{i,j} \geq \phi$, and examine the keys kept in $A_{i,j}$. A key $x$ is reported as a heavy hitter if $S^{up}_{i,j}(x) \geq \phi$ for all row $i$, where $1 \leq i \leq r$, and $j = f_i(x)$.

For heavy changer detection, we maintain two LD-Sketches for two adjacent epochs. Both sketches set the expansion parameter $T = \epsilon\phi/2$. At the end of the second epoch, we identify every bucket $(i, j)$ with $V_{i,j} \geq \phi$ in at least one epoch, and examine the keys of $A_{i,j}$ in both sketches. We obtain the lower and upper estimates in the first epoch and those in the second epoch, denoted by $S^{low,1}_{i,j}(x)$, $S^{up,1}_{i,j}(x)$, $S^{low,2}_{i,j}(x)$, and $S^{up,2}_{i,j}(x)$, respectively. The estimated change is given by $D_{i,j}(x) = \max\{S^{up,1}_{i,j}(x) - S^{low,2}_{i,j}(x), S^{up,2}_{i,j}(x) - S^{low,1}_{i,j}(x)\}$. A key $x$ is reported as a heavy changer if $D_{i,j}(x) \geq \phi$ for all row $i$, where $1 \leq i \leq r$, and $j = f_i(x)$.

## *3.3. Analysis*

We present theoretical analysis of the local detection of LD-Sketch as described in Section 3.2. We analyze the error rates, space complexity, and time complexity. Our proofs are presented in Appendix.

### *3.3.1. Error Rates*

First, we analyze the estimated error of a key, based on which we derive the error rates of LD-Sketch. For bucket $(i, j)$ and key $x$ hashed to the bucket (i.e., $f_i(x) = j$), the estimated error of key $x$ in bucket $(i, j)$ depends on the sum of other keys in the bucket, i.e., $\sum_{y \neq x, f_i(y)=j} S(y)$. Lemmas 3 and 4 describe the errors of lower and upper estimated sums with respect to $\sum_{y \neq x, f_i(y)=j} S(y)$, respectively.

**Lemma 3.** *For bucket $(i, j)$ and $x$ with $f_i(x) = j$, if $\sum_{y \neq x, f_i(y)=j} S(y) \leq \frac{(k+1)^2}{k+2}T$,*

$$S(x) - \frac{k+1}{k+2}T \leq S^{low}_{i,j}(x) \leq S(x).$$

**Lemma 4.** *For bucket $(i, j)$ and key $x$ with $f_i(x) = j$, if $kT < \sum_{y \neq x, f_i(y)=j} S(y) \leq (k + 1)T$,*

$$S(x) \leq S^{up}_{i,j}(x) \leq S(x) + (\frac{k+1}{k+2})T.$$

Based on the range of the estimated sum of a key, we now derive the error rates of LD-Sketch. Lemma 5 argues that our local detection has zero false negatives.

**Lemma 5.** *Using LD-Sketch, if key $x$ has $S(x) \geq \phi$, it must be reported as a heavy hitter; if $x$ has $D(x) \geq \phi$, it must be reported as a heavy changer.*

Lemmas 6 and 7 discuss the false positive rates of heavy hitter detection and heavy changer detection, respectively.

**Lemma 6.** *For key $x$ with $S(x) \leq (1 - \epsilon)\phi$, it is never be reported as a heavy hitter. For key $x$ with $(1 - \epsilon)\phi < S(x) < (1 - \epsilon/2)\phi$, it is reported as a heavy hitter with probability at most $(\frac{U}{w\epsilon\phi})^r$.*

**Lemma 7.** *For key $x$ with $D(x) \leq (1 - \epsilon)\phi$, it will never be reported as a heavy changer. For key $x$ with $(1 - \epsilon)\phi < D(x) < (1 - \epsilon/2)\phi$, it is reported as a heavy changer with probability at most $(\frac{6U}{w\epsilon\phi})^r$.*

### *3.3.2. Complexities*

We now perform complexity analysis on LD-Sketch, whose complexity depends on $l_{i,j}$, i.e., the average length of the associative array in each bucket. Lemma 8 shows the worst-case expectation of $l_{i,j}$.

**Lemma 8.** *The expected value of $l_{i,j}$ is given by $E[l_{i,j}] = O(\frac{U^2}{w^2T^2})$.*

Given the average length of associative arrays, Lemma 9 discusses the space and time complexities of LD-Sketch.

**Lemma 9.** *LD-Sketch has space complexity $O(r(w + \frac{U^2}{wT^2}))$. Its time complexity of updating a data item is $O(r(1 + \frac{U^2}{w^2T^2}))$, and that of detection is $(r(w + \frac{U^2}{wT^2}))$.*

Table 2: Comparisons of Existing Approaches with LD-Sketch: Upper Bounds of the Error Probability[1,2].

| Method | Heavy Hitter Detection | | | | | Heavy Changer Detection | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
| Count-Min [11] | $\delta$ | * | * | 0 | 0 | $\delta$ | * | * | * | $\delta$ |
| Combinatorial Group Testing [10] | $\delta$ | * | * | * | $\delta$ | $\delta$ | * | * | * | $\delta$ |
| Reversible Sketch [23] | $\delta$ | * | * | * | $\delta$ | $\delta$ | * | * | * | $\delta$ |
| SeqHash [4] | $\delta$ | * | * | 0 | 0 | $\delta$ | * | * | * | $\delta$ |
| Fast Sketch [17] | $\delta$ | * | * | * | $\delta$ | $\delta$ | * | * | * | $\delta$ |
| LD-Sketch | 0 | $\delta$ | * | 0 | 0 | 0 | $\delta$ | * | 0 | 0 |

[1] Each key is classified into one of the four intervals in which its true sum/difference lies: $R_1 = (0, (1-\epsilon)\phi)$, $R_2 = [(1-\epsilon)\phi, (1-\epsilon/2)\phi)$, $R_3 = [(1-\epsilon/2)\phi, \phi)$, $R_4 = [\phi, (1+\epsilon)\phi)$, $R_5 = [(1+\epsilon)\phi, +\infty)$.
[2] * means that the algorithm fails to bound the error probability in the range.

Table 3: Comparisons of Existing Approaches with LD-Sketch for Heavy Changer Detection: Space and Time Complexities.

| Method | Space | Update Time | Detect Time |
|---|---|---|---|
| Count-Min [11] | $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$ | $O(\log\frac{1}{\delta})$ | $O(n)$ |
| Combinatorial Group Testing [10] | $O(\frac{H}{\epsilon}\log n\log\frac{1}{\delta})$ | $O(\log n\log\frac{1}{\delta})$ | $O(\frac{H}{\epsilon}\log n\log\frac{1}{\delta})$ |
| Reversible Sketch [23] | $O(\frac{\log n^{\Theta(1)}}{\log\log n})$ | $O(\log n)$ | $O(\frac{H}{\epsilon}n^{\frac{3}{\log\log n}}\log\log n)$ |
| SeqHash [4] | $O(\frac{H}{\epsilon}\log n\log\frac{1}{\delta})$ | $O(\log n\log\frac{1}{\delta})$ | $O(\frac{H}{\epsilon}\log n\log\frac{1}{\delta})$ |
| Fast Sketch [17] | $O(\frac{H}{\epsilon}\log\frac{1}{\delta}\log\frac{n\epsilon}{H\log\frac{1}{\delta}})$ | $O(\log\frac{1}{\delta}\log\frac{n\epsilon}{H\log\frac{1}{\delta}})$ | $O(\frac{H}{\epsilon}\log^3\frac{1}{\delta}\log\frac{n\epsilon}{H\log\frac{1}{\delta}})$ |
| LD-Sketch | $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$ | $O(\log\frac{1}{\delta})$ | $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$ |

## *3.4. Parameter Selection*

We now propose the parameter selection of LD-Sketch. As stated above, we set $T = \epsilon\phi$ and $T = \epsilon\phi/2$ for heavy hitter and heavy changer detections, respectively. We express our results in terms of $H$, where $H = \frac{U}{\phi}$ for heavy hitter detection and $H = \frac{2U}{\phi}$ for heavy changer detection (see Section 2.1). The selection of $r$ and $w$ takes two parameters $\epsilon$ and $\delta$ as inputs, similar to the work [10]. For heavy hitter detection, LD-Sketch selects $w = \frac{2H}{\epsilon}$, $r = \log\frac{1}{\delta}$. For heavy changer detection, LD-Sketch selects $w = \frac{6H}{\epsilon}$ and $r = \log\frac{1}{\delta}$. Note that with these selected parameters, the expected value of $l_{i,j}$ in Lemma 8 will have a constant complexity $O(1)$. In Section 6, we will verify this property via trace-driven evaluation.

With the above lemmas and the selected parameters $w, r$ and $T$, Theorems 1 and 2 summarize the error bounds, space complexity, and time complexity of LD-Sketch in heavy hitter and heavy changer detections, respectively.

**Theorem 1.** *Consider an LD-Sketch with $w = \frac{2H}{\epsilon}$, $r = \log\frac{1}{\delta}$ and $T = \epsilon\phi$. It reports all heavy hitters. A non-heavy hitter with sum less than $(1-\epsilon)\phi$ is never reported. A non-heavy hitter with sum between $(1-\epsilon)\phi$ and $(1-\epsilon/2)\phi$ is reported with probability at most $\delta$. The expected space is $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$. The expected time complexity of updating a data item is $O(\log\frac{1}{\delta})$, and that of detection is $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$*

**Theorem 2.** *Consider two LD-Sketches with $w = \frac{6H}{\epsilon}$, $r = \log\frac{1}{\delta}$, and $T = \epsilon\phi/2$. They report all heavy changers. A non-heavy changer with difference less than $(1-\epsilon)\phi$ is never reported. A non-heavy changer with difference between $(1-\epsilon)\phi$ and $(1-\epsilon/2)\phi$ is reported with probability at most $\delta$. The expected space complexity is $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$. The expected time complexity of updating a data item is $\log\frac{1}{\delta}$, and that of detection is $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$.*

## *3.5. Comparisons with Existing Techniques*

We compare LD-Sketch with state-of-the-art techniques. We focus on the sketch-based techniques, which provide error bounds with two parameters: $0 < \epsilon \leq 1$ and $0 < \delta < 1$ in addition to the threshold $\phi$.

We first compare the accuracy of different approaches. We classify each key $x$ into one of the five intervals in which its true sum $S(x)$ or true difference $D(x)$ lies: $R_1 = (0, (1-\epsilon)\phi)$, $R_2 = [(1-\epsilon)\phi, (1-\epsilon/2)\phi)$, $R_3 = [(1-\epsilon/2)\phi, \phi)$, $R_4 = [\phi, (1+\epsilon)\phi)$, and $R_5 = [(1+\epsilon)\phi, +\infty)$. If $x$ belongs to $R_1$, $R_2$ or $R_3$, we evaluate its error probability of being a false positive (i.e., treated as a heavy key); if $x$ belongs to $R_4$ or $R_5$, we evaluate its error probability of being a false negative (i.e., treated as a non-heavy key). Table 2 shows the upper bounds of the error probability for different approaches. LD-Sketch shows two advantages over existing approaches. First, LD-Sketch guarantees a zero false negative rate for both heavy hitter and heavy changer detections. Second, LD-Sketch guarantees a zero false positive rate in the interval $[0, (1-\epsilon)\phi)$ for both heavy hitter and heavy changer detections.

We next compare the space and time complexities of different approaches. In the interest of space, we only consider heavy changer detection. Table 3 lists the space and time complexity of heavy changer detection for different approaches, in terms of $\phi$, $\epsilon$, $\delta$, and $H$. Count-Min consumes $O(\frac{H}{\epsilon}\log\frac{1}{\delta})$ of memory but requires $O(n)$ time to recover heavy keys. Combinatorial Group Testing, Reversible Sketch, SeqHash, and Fast Sketch aims to reduce the detection time, but they all have a $\log n$ term in the memory complexity to keep track of the heavy keys inside the sketch structure, so that the heavy keys can be recovered from the sketch structure. On the other hand, LD-Sketch has the same memory complexity as Count-Min by keeping the heavy key candidates in the associative arrays, while its update and detection time complexity is comparable to other approaches.

## 4. Distributed Detection

In this section, we elaborate how we perform heavy key detection in a distributed architecture via LD-Sketch.

### *4.1. Design*

We design our distributed scheme as follows. We deploy an LD-Sketch in each of the $q$ workers. Then we partition the data stream in each of the $p$ remote sites using a two-step approach. First, for a data item $(x, v_x)$, a remote site hashes key $x$ to a set of $d$ $(1 \leq d \leq q)$ workers for some design parameter $d$. This selection is deterministic in the sense that the same set of $d$ workers will always be selected for all data items with the same key $x$. Second, the remote site uniformly selects one of the $d$ workers and forwards it the data item. The partitioning functions in both steps are identical in all remote sites. Since each of the $d$ workers receives on average $1/d$ of the total value for each key, we set the heavy key threshold to be $(1-\gamma)\frac{\phi}{d}$ for some approximation parameter $0 \leq \gamma < 1$. Finally, if a key is reported as a heavy key by all the $d$ workers, then we report the key as a heavy key in our final results.

The design parameter $d$ in our distributed scheme determines the accuracy of heavy key detection. As $d$ increases, the false positive rate decreases until some turning point is reached. We also introduce a small false negative rate, since some of the $d$ workers may receive less than $(1-\gamma)\frac{\phi}{d}$ of the total value if the partition is not perfectly even. We study the impact of $d$ on the accuracy in the next subsection.

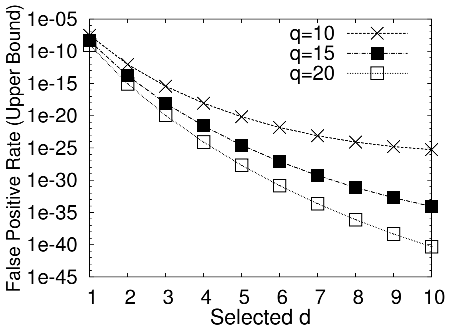
### *4.2. Analysis*

We conduct theoretical analysis on LD-Sketch in a distributed setting. Our results are based on those in Section 3 by substituting new parameters for the distributed scheme.

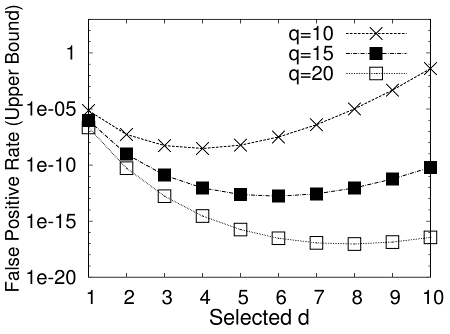
We first consider the space and time complexity. Suppose that each LD-Sketch has $r$ rows and $w$ buckets. For the total sum $U$ and the threshold $\phi$ in local detection, we replace them with $\frac{U}{q}$ and $(1-\gamma)\frac{\phi}{d}$, respectively. We assume that the worker selection for each data item is independent. By similar arguments in Lemma 8, the expectation of the term $U^2$ in the space and time complexity can be replaced with $O(\frac{U^2}{q^2})$, assuming that $q$ is far smaller than the number of available keys $n$. Using the above arguments, the space complexity is $O(r(w + \frac{U^2}{wT^2q^2}))$. The time complexity of the update procedure is $O(r(1+\frac{U^2}{w^2T^2q^2}))$, and that of the detection procedure is $O(r(w+\frac{U^2}{wT^2q^2}))$. We set $T = (1-\gamma)\frac{\epsilon\phi}{d}$ for heavy hitter detection and $T = (1-\gamma)\frac{\epsilon\phi}{2d}$ for heavy changer detection. Since $1 \leq d \leq q$ and $\gamma$ is typically small, we actually reduce both the space and time complexity of local detection.

For the false positive rate, it can be derived from Lemmas 6 and 7 by replacing $U$ and $\phi$ with $\frac{U}{q}$ and $(1-\gamma)\frac{\phi}{d}$, respectively. Note that the distributed scheme further reduces the false positive rate since a non-heavy key needs to be reported by all $d$ workers. Thus, the corresponding false positive rate in the range $[(1-\epsilon)\phi, (1-\epsilon/2)\phi)$ is at most $(\frac{dU}{qw(1-\gamma)\epsilon\phi})^{rd}$ for heavy hitter detection and at most $(\frac{6dU}{qw(1-\gamma)\epsilon\phi})^{rd}$ for heavy changer detection.

We illustrate how the false positive rate varies with different values of $d$. As a case study, we pick $\phi = 0.001U$, so there are at most $H = \frac{U}{\phi} = 1000$ heavy hitters and $H = \frac{2U}{\phi} = 2000$ heavy changers (see Section 2.1). We fix $\gamma = 0$, $\epsilon = 0.8$, $r = 5$, and $w = 4H$. Figures 3(a) and 3(b) show the upper bounds of the false positive rate for heavy hitter and heavy changer detection, respectively. Note that the upper bound of the false positive rate can be viewed as



(a) Heavy Hitter



(b) Heavy Changer

Figure 3: False positive rate of distributed scheme versus $d$.

the function $(cd)^{rd}$, for some constant $c$. As $d$ increases, if $\log(cd) < -1$, the upper bound of the false positive rate decreases; if $\log(cd) > -1$, the upper bound increases exponentially, as shown in Figure 3(b).

For the false negative rate, Lemma 10 shows that it can be bounded in a distributed setting.

**Lemma 10.** *For a heavy key $x$, it will be missed at probability at most* $1 - (1 - e^{-\frac{\phi}{2d}\gamma^2})^d$.

### *4.3. Parameter Selection*

We now propose the parameter selection for distributed detection. The selection involves $H$, $\phi$, $\epsilon$, $\delta$, and $\gamma$ as inputs. For heavy hitter detection, LD-Sketch selects $w = \frac{2dH}{(1-\gamma)q\epsilon}$, $r = \log\frac{1}{d\delta}$, and $T = \frac{(1-\gamma)\epsilon\phi}{d}$; for heavy changer detection, it selects $w = \frac{6dH}{(1-\gamma)q\epsilon}$, $r = \log\frac{1}{d\delta}$, and $T = \frac{(1-\gamma)\epsilon\phi}{2d}$. With the selected parameters, the following theorems summarize the error bounds, space complexity, and time complexity of the distributed scheme of LD-Sketch.

**Theorem 3.** *Consider an LD-Sketch with $w = \frac{2dH}{(1-\gamma)q\epsilon}$, $r = \log\frac{1}{d\delta}$, and $T = \frac{(1-\gamma)\epsilon\phi}{d}$. A heavy hitter is missed with probability at most $1 - (1 - e^{-\frac{\phi}{2d}\gamma^2})^d$. A non-heavy hitter with the true sum less than $(1-\gamma)(1-\epsilon)\phi$ is never reported. A non-heavy hitter with sum between $(1-\gamma)(1-\epsilon)\phi$ and $(1-\gamma)(1-\epsilon/2)\phi$ is reported with probability at most $\delta$. The expected space complexity is $O(\frac{dH}{q\epsilon(1-\gamma)}\log\frac{1}{d\delta})$. The expected time complexity of updating a data item is $O(\log\frac{1}{d\delta})$, and that of detection is $O(\frac{dH}{q\epsilon(1-\gamma)}\log\frac{1}{d\delta})$.*

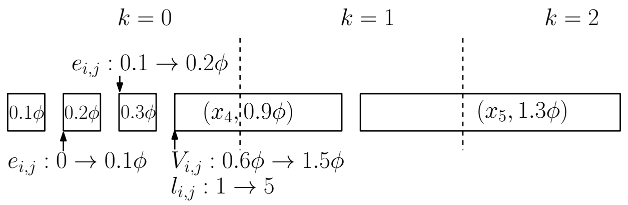
**Theorem 4.** *Consider two LD-Sketches with $w = \frac{6dH}{(1-\gamma)q\epsilon}$, $r = \log\frac{1}{d\delta}$, and $T = \frac{(1-\gamma)\epsilon\phi}{2d}$. A heavy changer is missed with probability at most $1 - (1 - e^{-\frac{\phi}{2d}\gamma^2})^d$. A non-heavy changer with the true difference less than $(1-\gamma)(1-\epsilon)\phi$ is never reported. A non-heavy changer with difference between $(1-\gamma)(1-\epsilon)\phi$ and $(1-\gamma)(1-\epsilon/2)\phi$ is reported with probability at most $\delta$. The expected space complexity is $O(\frac{dH}{q(1-\gamma)\epsilon}\log\frac{1}{d\delta})$. The expected time complexity of updating a data item is $\log\frac{1}{d\delta}$, and that of detection is $O(\frac{dH}{q(1-\gamma)\epsilon}\log\frac{1}{d\delta})$.*

### *4.4. Discussion*

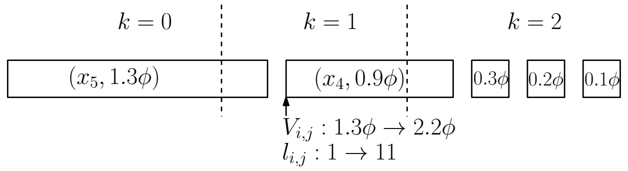
The distributed detection of LD-Sketch builds on its local detection approach. However, it is also possible to extend existing local detection approaches for distributed detection. Such extensions can be classified into two groups.

The first group is to detect heavy keys in each worker independently and merge the detected results. This group of approaches reduces both space and time complexities in workers, but it may introduce a higher error rate as the errors of individual workers are accumulated.

The second group of approaches maintains a local data structure (e.g., an associative array or a sketch) in each worker. It then merges the local data structures into a global one and then performs heavy key detection based on the merged global structure. For example, Mergeable Summary [1] merges two associative arrays without increasing errors. This group of approaches boosts the processing throughput as each worker only needs to process a partial data stream. However, the memory consumption in each worker is not reduced. To keep the same error rate, the size of the



(a) Ascending arrival sequence



(b) Descending arrival sequence

Figure 4: Two arrival sequences of data items, and they lead to different expansions of $A_{i,j}$ and numbers of false positives.

global structure must be the same as that in single-host detection. To make the local structures are mergeable, each worker need to maintain the same size of a local structure as that of the global structure, implying that the memory complexity of each worker is the same as that in single-host detection.

To summarize, the above extensions either increase the error rate or bring no memory reduction. On the other hand, LD-Sketch reduces both the error rate and memory consumption in distributed detection.

## 5. Ordering of Data Items

Our analysis in Sections 3 and 4 focuses on the sum or difference of a key as a whole over an entire epoch, and the complexity results only address the worst-case scenario. However, the actual performance and accuracy of LD-Sketch is in fact sensitive to the ordering of data items *within* an epoch. Specifically, during the update procedure, the values of the data items determine how much we decrement the keys of the associative arrays and how much we expand the associative arrays. The actual number of counters allocated in LD-Sketch, and hence the detection accuracy, varies depending on the ordering of data items and their values received by LD-Sketch.

To our knowledge, the impact of ordering is not considered in prior work. Counter-based techniques always keep fixed-length arrays, so the memory usage is unaffected, although it remains an open issue how the ordering of data items affects decrements of counters and the resulting detection accuracy. On the other hand, sketch-based techniques always add values to buckets, and the summation results are unaffected by the ordering of data items.

In this section, we demonstrate via examples the impact of the ordering of data items on LD-Sketch. We then explain how we leverage the ordering property to enhance LD-Sketch.

### *5.1. Motivating Examples*

We consider the following scenario to illustrate the impact of ordering of data items. Suppose that we perform heavy hitter detection, with the detection threshold $\phi$, approximation parameter $\epsilon = 1$ and the expansion parameter $T = \phi$. Five keys, denoted by $x_1, x_2, x_3, x_4, x_5$, are hashed to the same bucket $(i, j)$, and their true sums $S(x_i)$ ($1 \leq i \leq 5$) are $0.1\phi$, $0.2\phi$, $0.3\phi$, $0.9\phi$, and $1.3\phi$, respectively. To simplify our discussion, we assume that each key has a single data item whose value is equal to the true sum (i.e., $v_x = S(x)$ for key $x$).

Referring to Algorithm 1, since the total sum of all keys in bucket $(i, j)$ is $V_{i,j} = 2.8\phi$, we see that $k$ is at most 2 (Line 8), and the maximum length of the associative array $A_{i,j}$ is $l_{i,j} = 11$ (Line 23). We now consider two specific arrival sequences of the data items, as shown in Figure 4, and discuss how $A_{i,j}$ is actually expanded based on Algorithm 1.

Figure 4(a) shows that the keys arrive in ascending order of their true sums, i.e., in the order $x_1$, $x_2$, $x_3$, $x_4$, and $x_5$. When $x_1$ arrives, it is inserted to $A_{i,j}$ (Line 6). When $x_2$ arrives, $A_{i,j}$ is now full with $l_{i,j} = 1$ counter. Since

```
Initialization (before each epoch): $\hat{k}_{i,j} = 0$
1: $\hat{k}_{i,j} = \hat{k}_{i,j} + 1$
2: add $(\hat{k}_{i,j} + 1)(\hat{k}_{i,j} + 2) - 1 - l_{i,j}$ new counters to $A_{i,j}$
3: $l_{i,j} = (\hat{k}_{i,j} + 1)(\hat{k}_{i,j} + 2) - 1$
4: Insert $x$ to $A_{i,j}$ and set $A_{i,j}[x] = v_x$
```

Figure 5: Controlled expansion: modifications to Lines 22-24 of Algorithm 1.

$k = 0$, Lines 10-20 will be executed, and $x_1$ is removed and $x_2$ is stored in $A_{i,j}$. When $x_3$ arrives, Lines 10-20 will be similarly executed, and $x_2$ is removed and $x_3$ is stored in $A_{i,j}$. At this moment, $e_{i,j}$ is incremented to $0.2\phi$. When $x_4$ arrives, $V_{i,j}$ is incremented from $0.6\phi$ to $1.5\phi$ and we now have $k = 1$. Thus, $A_{i,j}$ expands and $l_{i,j} = 5$, and $x_4$ is stored in $A_{i,j}$ (Lines 22-24). When $x_5$ arrives, there are empty counters, so it is inserted to $A_{i,j}$ (Line 6). Note that $x_4$ remains in $A_{i,j}$. Its estimated sum is $S^{up}_{i,j}(x_4) = A_{i,j}[x_4] + e_{i,j} = 0.9\phi + 0.2\phi = 1.1\phi$. Thus, $x_4$ is treated as a heavy hitter, but it is a false positive. In short, this ascending arrival sequence expands $A_{i,j}$ to have $l_{i,j} = 5$ counters, and returns one false positive.

Figure 4(b) shows another sequence in which the keys arrive in descending order of their true sums, i.e., in the order $x_5$, $x_4$, $x_3$, $x_2$, and $x_1$. When $x_5$ arrives, it is inserted to $A_{i,j}$ (Line 6). When $x_4$ arrives, $A_{i,j}$ is now full with $l_{i,j} = 1$ counter. Since $V_{i,j}$ becomes $2.2\phi$, we have $k = 2$. Thus, $A_{i,j}$ expands to have $l_{i,j} = 11$ counters, and $x_2$ is inserted (Lines 22-24). Afterwards, the following keys $x_3$, $x_2$, and $x_1$ can be inserted directly (Line 6). This scenario keeps $e_{i,j} = 0$, so no false positive is reported. In short, the descending arrival sequence expands $A_{i,j}$ to have $l_{i,j} = 11$ counters, but does not return any false positive.

**Implication.** The above two examples provide preliminary insights that the ordering of data items affects memory usage and detection accuracy in different ways. Both memory usage and detection accuracy are critical metrics for evaluating a real-time heavy key detection scheme. The insights motivate us to conduct more in-depth studies on the impact of data ordering, including both theoretical analysis (see Section 5.2) and trace-driven evaluation (see Section 6).

### *5.2. Analysis*

We now conduct theoretical analysis on the impact of ordering, in particular, when the arrival of data items follows either ascending order or descending order of their true sums over an epoch. We call them an *ascending arrival sequence* and a *descending arrival sequence*, respectively. An ascending (resp. descending) arrival sequence means that the data items of key $x_a$ always arrive before the data items of $x_b$ if $S(x_a) < S(x_b)$ (resp. $S(x_a) > S(x_b)$).

Before we present our analysis, we first modify Algorithm 1 by controlling the expansion of the associative array $A_{i,j}$ for a bucket $(i, j)$. Our observation is that if a data item $x$ has a significantly large $v_x$, then $k$ will be incremented by a large value, thereby making $A_{i,j}$ expand by a large number of additional counters than it actually uses. For instance, in the descending arrival sequence discussed in the previous subsection, when $x_2$ is added, $k$ increases from 0 to 2 and hence $l_{i,j}$ increases from 1 to 11. The reason is that $l_{i,j}$ is incremented based on the expansion number $k$, which is determined by the current counter value $V_{i,j}$ (see Line 8 and Lines 22-24 of Algorithm 1). To handle the problem, we introduce a *controlled expansion number* $\hat{k}_{i,j}$ for each bucket $(i, j)$, such that $\hat{k}_{i,j}$ is incremented by exactly one for each expansion and make $l_{i,j}$ increase with respect to $\hat{k}_{i,j}$ instead of $k$. Thus, $A_{i,j}$ is expanded independently of the value of a data item.

Figure 5 details the modifications, in which we replace Lines 22-24 of Algorithm 1 with Lines 1-4 in Figure 5. Specifically, the controlled expansion number $\hat{k}_{i,j}$ is initialized to zero before each epoch. At each expansion, we increment $\hat{k}_{i,j}$ by one (Line 1) and add $(\hat{k}_{i,j} + 1)(\hat{k}_{i,j} + 2) - 1 - l_{i,j}$ new counters (Line 2). Then we update $l_{i,j}$ and insert the incoming key $x$ (Lines 3-4).

The modifications ensure that $\hat{k}_{i,j} \le k$. When $\hat{k}_{i,j} = k$, we must have $l_{i,j} = (\hat{k}_{i,j} + 1)(\hat{k}_{i,j} + 2) - 1 = (k + 1)(k + 2) - 1$, so Lines 10-20 Algorithm 1 will be executed. Then $\hat{k}_{i,j}$ stops growing until $k$ is incremented again (Line 8 of Algorithm 1).

It is important to note that all analysis results in Section 3.3 still hold, since the results depend on the expansion number $k$, which provides an upper bound on $\hat{k}_{i,j}$.

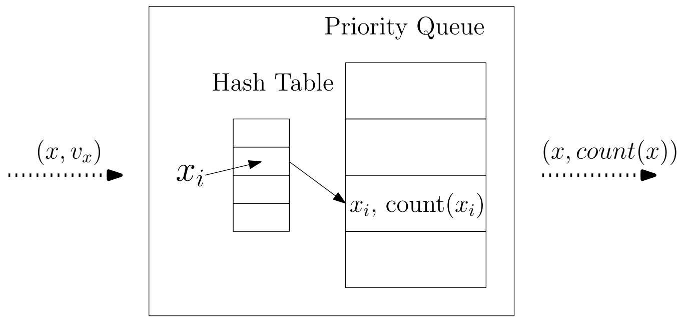


Figure 6: Framework for the enhancement heuristics.

We now formally analyze the impact of ordering based on the modified algorithm, and define the following notation. Consider a bucket $(i, j)$, which is updated with $t_{i,j}$ distinct keys. Their total sum $V_{i,j}$ satisfies the condition $k_{i,j}T \leq V_{i,j} < (k_{i,j} + 1)T$ for some $k_{i,j}$, where $T = \epsilon\phi$ for heavy hitter detection and $T = \epsilon\phi/2$ for heavy changer detection. Among the $t_{i,j}$ keys, we assume that $t^{\geq}_{i,j}$ of them have true sum at least $T$. For ease of presentation, we refer to them as *large keys* and the rest as *small keys*. Let $V^{\geq}_{i,j}$ and $V^{<}_{i,j} = V_{i,j} - V^{\geq}_{i,j}$ be the total sums of large and small keys satisfying $k^{\geq}_{i,j}T \leq V^{\geq}_{i,j} < (k^{\geq}_{i,j} + 1)T$ and $k^{<}_{i,j}T \leq V^{<}_{i,j} < (k^{<}_{i,j} + 1)T$ for some $k^{\geq}_{i,j}$ and $k^{<}_{i,j}$, respectively.

We analyze two issues: (i) how an ascending arrival sequence affects memory usage of LD-Sketch and (ii) how a descending arrival sequence affects detection accuracy of LD-Sketch. The following two lemmas describe the impact of the two types of arrival sequences on $l_{i,j}$ and $e_{i,j}$, respectively.

**Lemma 11.** *For an ascending arrival sequence,* $l_{i,j} = O({k^{<}_{i,j}}^2 + t^{\geq}_{i,j})$.

**Lemma 12.** *For a descending arrival sequence,* $e_{i,j} = O(\frac{T}{k^{\geq}_{i,j}})$.

**Discussion.** From Section 3.1, we know that $l_{i,j} = O(k^2_{i,j})$, which may be very large, especially when a bucket includes a key with a very large value. However, Lemma 11 shows that for a special case of ascending arrival sequences, $l_{i,j}$ only depends on the total sum of small keys and the number of large keys. In real-life networks, the key distribution of network traffic is typically highly skewed [3, 24, 26], such that a few number of large keys account for most of the traffic. This suggests that $k^{<}_{i,j}$ and $t^{\geq}_{i,j}$ are small in practice. Thus, the memory usage is much smaller for ascending arrival sequences.

In terms of accuracy, Lemma 12 considers the maximum estimation error $e_{i,j}$, which determines the false positive rate. It shows that descending arrival sequences bound $e_{i,j}$ to $O(\frac{T}{k^{\geq}_{i,j}})$. Since network traffic is highly skewed in practice [3, 24, 26] and few large keys can generate large $k^{\geq}_{i,j}$, $e_{i,j}$ (and hence the false positive rate) is much smaller for descending arrival sequences.

### 5.3. *Enhancements for LD-Sketch*

Although Lemmas 11 and 12 provide observations on the impact of the ordering of data items, it is non-trivial to leverage the analysis to enhance LD-Sketch. The difficulty comes from the fact that it is infeasible to sort data items without tracking all keys; otherwise, the heavy key detection can be immediately solved by simply sorting keys in descending order and reporting the first few keys.

To take advantage of the ordering property of data items, our idea is to sort data items at a *coarse granularity* rather than produce an exact ordering. We propose two heuristics, namely (i) the *memory-oriented* heuristic, which leverages the ascending ordering of data items to reduce memory usage, and (ii) the *accuracy-oriented* heuristic, which leverages the descending ordering of data items to reduce the false positive rate.

Both heuristics employ the same framework as shown in Figure 6. The framework is composed of a hash table (denoted by $\mathcal{H}$) and a priority queue (denoted by $\mathcal{Q}$), both of which store the same number of items that is pre-configured in advance. The entries of $\mathcal{H}$ are indexed by keys of data items, and each of the entries is a pointer to an entry of $\mathcal{Q}$, which maintains a counter for the corresponding key. $\mathcal{Q}$ can return the key with the largest or the smallest counter value in $\Theta(1)$ time. If multiple keys have the same counter values in $\mathcal{Q}$, we assume that the one that is accessed earlier will be returned.

Algorithm 2 elaborates how the framework is used. It consists of two basic functions, namely UPDATEFRAMEWORK and EVICTENTRY. UPDATEFRAMEWORK includes a data item $(x, v_x)$ into the framework. If key $x$ in $\mathcal{H}$, we look up $\mathcal{H}$ and increment the counter by $v_x$ in $\mathcal{Q}$ (Lines 2-4). Otherwise, if key $x$ is not in $\mathcal{H}$ and both $\mathcal{H}$ and $\mathcal{Q}$ are full, then the framework will evict one entry to make room for the new key (Lines 6-8), and we insert key $x$ into $\mathcal{H}$ and $\mathcal{Q}$ and initialize the counter in $\mathcal{Q}$ to be $v_x$ (Line 9). EVICTENTRY always evicts the highest-priority key and its counter value and removes it from both $\mathcal{H}$ and $\mathcal{Q}$ (Lines 14-16). Based on these two functions, Algorithm 2 works as follows. In each epoch, both $\mathcal{H}$ and $\mathcal{Q}$ are initialized with empty entries (Lines 20-21). We then update the framework with each incoming data item (Lines 22-24). At the end of an epoch, all remaining data items are evicted (Lines 25-27).

Algorithm 2 applies to both heuristics in the same way, with the only difference in the priority definition. For the memory-oriented heuristic, the highest-priority key is the one that has the smallest counter value in $\mathcal{Q}$, while for the accuracy-oriented heuristic, the highest-priority key is the one with the largest counter value in $\mathcal{Q}$. All evicted data items will be the inputs to LD-Sketch.

**Discussion.** We discuss some practical issues for the enhancement heuristics. The first issue is where the heuristics are deployed. Instead of putting the heuristics in the workers (where LD-Sketch is deployed), we deploy the heuristics in each of the remote sites. Such a placement has two advantages. First, it shifts some workload from workers to remote sites. Second, it reduces the data transmission overhead between the remote sites and workers, as the priority queue may aggregate the values of a data item before evicting it.

The second issue is the number of data items being stored in the hash table and the priority queue. Increasing the capacity improves the sorting accuracy, but at the expense of more memory and processing overhead for maintaining the framework. In this paper, we configure the capacity to be 1000 items. Based on our traces, this number is only around $0.1\%$ of the total number of keys observed in an epoch, yet it suffices to significantly improve the detection algorithm as shown in our evaluation (see Section 6).

The final issue is the problem of uneven partitioning, which introduces false negatives in distributed detection. Since the framework can aggregate the values of a key, the uneven partitioning problem can be more severe. Consider an extreme case in which a true heavy hitter is never evicted from the priority queue until the end of the epoch, with its counter value in the priority queue equal to its true sum. When the remote site emits the key to one of $d$ workers, the selected worker will behave as if it receives all values, while the other $d-1$ workers receive none. This uneven partitioning makes LD-Sketch fail to detect the key as a heavy hitter since it requires all $d$ workers to report the key. To handle this problem, we explicitly split an evicted data item into $d$ equal pieces, each holding the counter value divided by $d$, and emit exactly one piece to each worker.

## 6. Experiments

In this section, we present results based on trace-driven experiments. Our goal is to demonstrate the accuracy and scalability of LD-Sketch in a distributed setting.

**Implementation.** We implement a distributed streaming architecture in C++ with remote sites and workers, each of which runs as a software process. To eliminate network I/O overhead in our evaluation, we deploy our architecture in a multi-core testbed, where a remote site and a worker is connected via an in-memory ring buffer. We implement LD-Sketch in each worker.

**Testbed and evaluation methodologies.** Our testbed is a multi-core server with 12 physical 2.93GHz CPU cores and 50GB RAM. The server runs Linux 2.6.32. We compile our code using GCC 4.6 with the -O3 option.

We run our evaluation on real IP packet header traces from a commercial 3G UMTS network in China in December 2010 (the same traces as in our prior work [14]). We leverage the real-life traffic patterns of the traces to evaluate LD-Sketch in practical scenarios. Here, we only focus on the IP traffic in the data plane, but do not consider the traffic in the control plane, which is specific for 3G networks. We select the most heavy-loaded six hours of traces for our evaluation. The traces contain 1.1 billion packets that account for a total of around 600GB of traffic. We configure $p = 3$ remote sites and a number of $q$ workers (varied in our experiments). We divide the traces into three sub-traces in a round-robin manner, and each sub-trace is replayed by one of the three remote sites. Each data item corresponds to

**Algorithm 2** Enhancement Heuristics for LD-Sketch

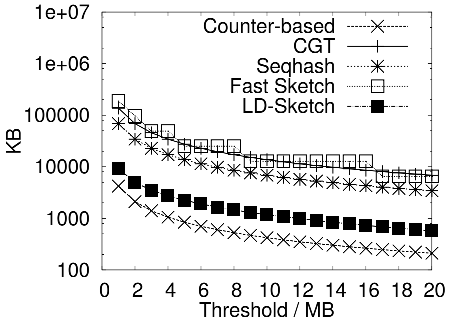
```
Input: data item $(x, v_x)$
1:  function UpdateFramework($x, v_x, \mathcal{H}, \mathcal{Q}$)
2:      if $x \in \mathcal{H}$ then
3:          Lookup $\mathcal{H}$ and the corresponding entry in $\mathcal{Q}$
4:          Increment the counter value of the entry in $\mathcal{Q}$
5:      else
6:          if $\mathcal{H}$ is full then
7:              EvictEntry($\mathcal{H}, \mathcal{Q}$)
8:          end if
9:          Insert $x$ to $\mathcal{H}$ and $\mathcal{Q}$, and initialize its counter in $\mathcal{Q}$ as $v_x$
10:     end if
11: end function
12:
13: function EvictEntry($\mathcal{H}, \mathcal{Q}$)
14:     Find the highest-priority entry $e$ from $\mathcal{Q}$
15:     Emit $(e.key, e.counter)$
16:     Remove $e.key$ from $\mathcal{H}$ and $\mathcal{Q}$
17: end function
18:
19: procedure Main
20:     hash table $\mathcal{H} \leftarrow \emptyset$
21:     priority queue $\mathcal{Q} \leftarrow \emptyset$
22:     for data item $(x, v_x)$ do
23:         UpdateFramework($x, v_x, \mathcal{H}, \mathcal{Q}$)
24:     end for
25:     while $\mathcal{Q}$ is not empty do
26:         EvictEntry($\mathcal{H}, \mathcal{Q}$)
27:     end while
28: end procedure
```

a packet, where the key is set as the 64-bit source and destination IP address pair and the value is set as the IP payload size. We set the epoch length for heavy key detection as 10 minutes, while we also verify the results for different epoch lengths and make consistent observations. Our results are averaged over all 36 epochs of the six-hour traces.

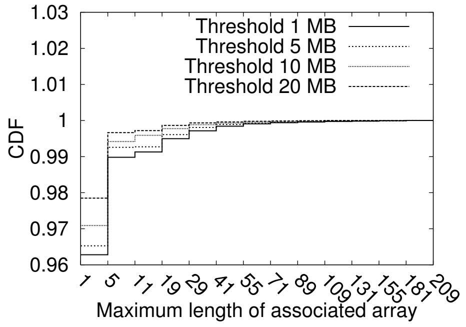
We compare LD-Sketch with several state-of-the-art sketch-based techniques, including Combinational Group Testing (CGT) [10], SeqHash [4], and Fast Sketch [17] in both heavy hitter and heavy changer detections. We also compare LD-Sketch with the counter-based technique in [21]. As the counter-based technique only supports heavy hitter detection (see Section 2.2), we subtract the counters of a key of the associative arrays at two adjacent epochs to estimate the change. Note that all the above sketch-based and counter-based techniques, except LD-Sketch, are only designed for local detection (i.e., using one worker).

**Metrics.** We consider the following metrics:

- *Memory usage:* It is the number of bytes consumed by the data structure. We omit the memory space for storing the hash functions of the sketch, assuming that they consume negligible space.
- *Recall:* It is the ratio of the number of true heavy keys returned to the actual number of true heavy keys. A higher recall means a lower false negative rate.
- *Precision:* It is the ratio of the number of true heavy keys returned to the total number of all heavy keys returned (including true heavy keys and false positives). A higher precision means a lower false positive rate.
- *Update throughput:* It is the total IP payload size divided by the total time to update the data items. Before measurements, we load the traces to memory to eliminate disk overhead. We also bind the worker processes to CPU cores to avoid context switching.



(a) Memory usage



(b) Length distribution of associative arrays in LD-Sketch

Figure 7: Experiment 1 (Memory usage).

We do not present the throughput results of the detection procedure. Our experience is that the detection time is significantly less than that of updating all data items in one epoch.

**Experiment 1 (Memory usage).** We first examine the actual memory usage of different approaches. The results depend on three variables $\delta, \epsilon$, and $\phi$. Given that there are many combinations, we consider a special setting where we fix $\delta = \frac{1}{4}, \epsilon = 1$, and vary $\phi$.

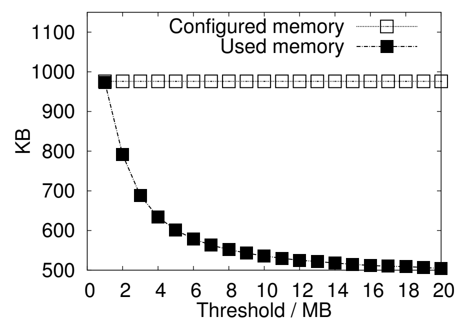
Given the values of $\delta, \epsilon$ and $\phi$, we configure the parameters for different approaches as follows. We set $H = U/\phi$ for heavy hitter detection and $H = 2U/\phi$ for heavy changer detection as defined in Section 2. For previous ones, we employ the default parameter selection mechanisms proposed in their respective studies. Specifically, for CGT, we configure two rows and $2H$ buckets, and each bucket contains $\log(n) + 1$ counters to recover heavy keys, where $log(n) = 64$ is the key length in bits. For SeqHash, we partition the 64-bit key into 25 words with the first word containing 16 bits and two bits for each of the remaining words. We allocate 66 rows in total, including four rows for the first sub-key space, 16 rows for the last sub-key space, and two rows for others. We allocate $H$ buckets to each row, and the allowed miss is set to one specifically for heavy changer detection. For Fast Sketch, we use two hash functions and set the hash table length as $4H$ to reduce hash collisions. For LD-Sketch, we set $r = 2$, $w = 2H$, and $T = \phi$.

Figure 7(a) compares the amount of memory (in units of KB) consumed by different approaches. We see that LD-Sketch uses twice as much memory as counter-based approach. On the other hand, it consumes much less memory than prior sketch-based approaches. For example, LD-Sketch only consumes one-sixth of memory of SeqHash, and its fraction is even less compared to CGT and Fast Sketch. The main reason is that prior approaches require enough space to restore heavy keys from their sketch structures. In contrast, LD-Sketch employs dynamically expandable associative arrays to keep the heavy keys. The average length of each array is bounded within a constant complexity.

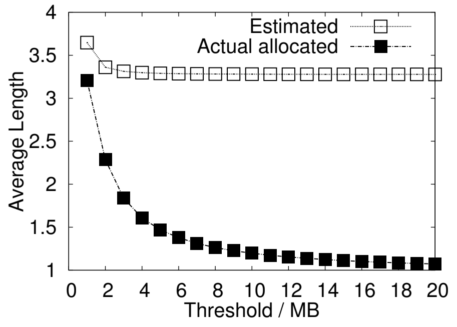
To examine how the length of each associative array in LD-Sketch varies, Figure 7(b) shows the length distribution of the associative arrays in LD-Sketch for different values of threshold $\phi$. We see that more than 96% associative arrays are of length one. Also, the average length of associative arrays is less than two for all cases. The results confirm the theoretical analysis of Lemma 8 that the average length has a constant complexity $O(1)$ (see Section 3.3).

**Experiment 2 (Accuracy of local detection of LD-Sketch).** We compare the accuracy of local detection of LD-Sketch with the detection accuracy of other approaches. For fair comparison, we configure all the detection approaches with the same amount of memory. We fix $M = 10^6$ bytes in the data structure for each approach. For CGT, SeqHash, and Fast Sketch, we fix $\delta = \frac{1}{4}$ and allocate the $M$ bytes to the buckets accordingly. For LD-Sketch, the actual memory usage is not fixed due to dynamic expansion. To handle this, we estimate the length of an associated array to be $2\frac{U^2}{w^2T^2}$ based on Lemma 8 (see Section 3) and configure $w$ that satisfies the memory requirement.

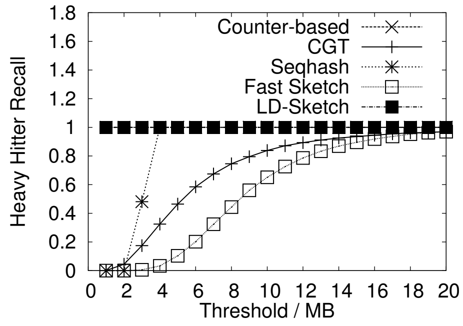
We first validate the memory usage of LD-Sketch, which is dynamic. Figure 8(a) shows that the actual amount of allocated memory does not exceed the configured amount for all cases. The actual amount is even smaller when $\phi$ increases. For example, the actual amount is only 50% of configured amount for $\phi = 20$MB. Figure 8(b) plots the actual average length of the associative arrays versus $\phi$. The results show that the actual average length approaches



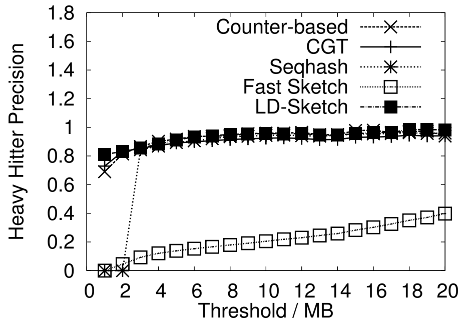
(a) Memory usage



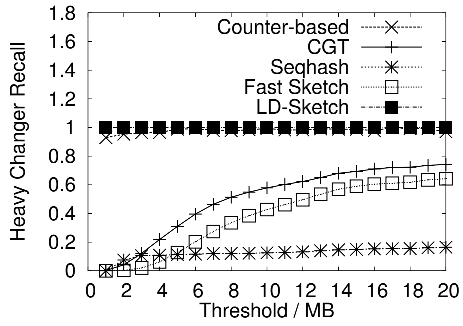
(b) Average associative array length



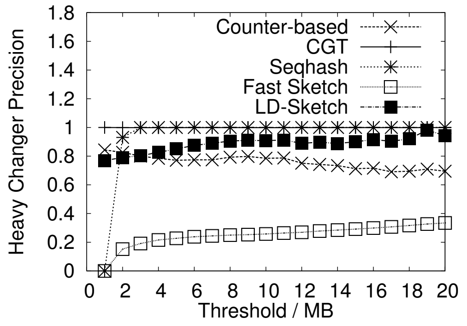
(c) Heavy hitter, Recall



(d) Heavy hitter, Precision



(e) Heavy changer, Recall



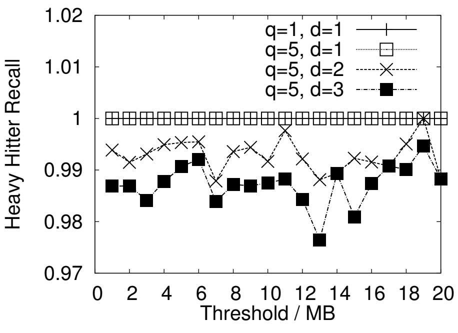
(f) Heavy changer, Precision

Figure 8: Experiment 2 (Accuracy of local detection of LD-Sketch).

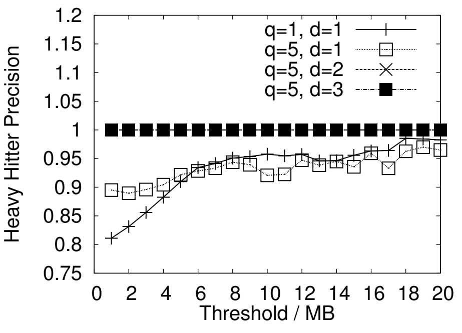
one when $\phi$ increases. For example, when $\phi = 20$MB, our configured length is $3.277$, yet the actual average length is only $1.072$.

Figures 8(c)-(f) show the results of detection accuracy. We see that all CGT, SeqHash, and Fast Sketch have low recall (see Figures 8(c) and 8(e)), because they need more memory to recover all heavy keys (see Table 3). With insufficient memory, they have high false negative rates. Fast Sketch has low precision, since it identifies heavy quotients (i.e., prefixes of heavy keys) and may return many false positives that match the quotients. The counter-based technique has around 90% recall and 70% precision for heavy changer detection. The reason is that the counter values are below the actual values (as in Lemma 1) in both epochs so that we cannot determine whether the estimated change is larger or smaller than the true change. On the other hand, LD-Sketch uses associative arrays to keep track of frequent items, and have recall exactly equal to 100%. Its precision is as low as 80% (when $\phi = 1$MB). However, we can increase its precision with distributed detection (see below).

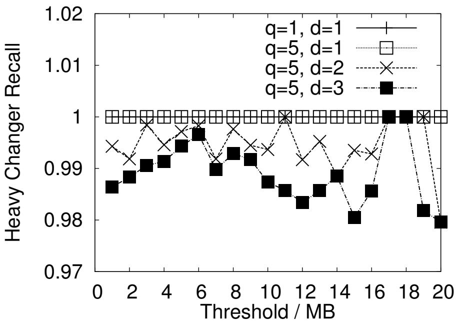
**Experiment 3 (Accuracy of distributed detection of LD-Sketch).** We now study the accuracy of LD-Sketch



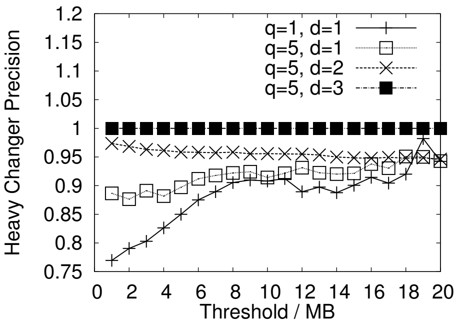
(a) Heavy hitter, Recall



(b) Heavy hitter, Precision



(c) Heavy changer, Recall



(d) Heavy changer, Precision

Figure 9: Experiment 3 (Accuracy for distributed detection of LD-Sketch).

using distributed detection. We deploy LD-Sketch in $q = 5$ workers, each of which is configured with $\frac{M}{q}$ bytes for heavy hitter detection and $\frac{2M}{q}$ for heavy changer detection, where $M = 10^6$ as in Experiment 2. We fix $\gamma = 0$ and vary both $d$ and $\phi$. Other parameters are the same as in Experiment 2.

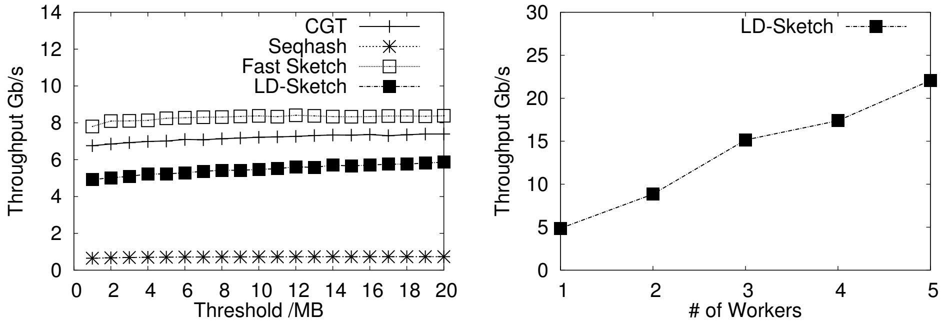
Figure 9 shows the results of different values of $d$. When $d = 1$, the result is similar to the local detection in Experiment 1. When $d > 1$, the heavy hitters can be detected with 100% recall and precision. For heavy changers, choosing $d = 2$ improves the precision of local detection algorithm from 76% to 97% for threshold 1MB, and choosing $d = 3$ can even achieve 100% of precision. However, false negatives will be introduced when $d > 1$ since we may not partition the data items of a key evenly across the workers. Nevertheless, our evaluation shows that the false negative rate is less than 2%.

**Experiment 4 (Update throughput).** In this experiment, we measure the update throughput using a single worker (local detection) and multiple workers (distributed detection). The parameters are the same as in Experiment 2 and 3.

Figure 10(a) presents the update throughput versus the number of true heavy keys when a single worker is used. The throughput changes marginally as the number of true heavy keys grows. Note that the main bottleneck of the sketch-based algorithms is the hash operations, and the number of hash operations remains the same as it depends on the fixed parameter $\delta$. Thus, the throughput is similar as the number of true heavy keys changes. From the figure, SeqHash is the slowest because it employs multiple sketches, in which the hash operations slow down the overall throughput. LD-Sketch, CGT, and Fast Sketch perform the same number of hash operations, but LD-Sketch requires additional time to update the associative arrays. For example, when $\phi$ = 1MB, LD-Sketch achieves only 72% and 63% of the throughput of CGT and Fast Sketch, respectively.

Nevertheless, we can boost the throughput of LD-Sketch via parallelization. Figure 10(b) shows the update throughput of LD-Sketch using multiple workers with $d = 1$. We see that the throughput increases almost linearly as the number of workers grows. When using five workers, LD-Sketch can reach over 20Gb/s of throughput. This shows the scalability of LD-Sketch in a distributed architecture.

**Experiment 5 (Effectiveness of enhancement heuristics).** We study the effectiveness of our proposed enhance-



(a) Single worker (b) Multiple workers

Figure 10: Experiment 4 (Update throughput).

ment heuristics (see Section 5.3). We compare them with the baseline LD-Sketch, which does not take into account the ordering property of data items. In the interest of space, we only report the results of distributed detection with $q = 5$ and $d = 2$, while we verify that the results of other settings are similar. The parameter selection is the same as in Experiment 3.

Figure 11(a) shows the actual memory usage of LD-Sketch for both baseline and two heuristics. We see that the actual amount of memory usage of all algorithms is less than the configured amount, and decreases as $\phi$ increases. In addition, the memory-oriented heuristic consumes only $75\%$ of memory of the baseline algorithm. We further examine the reason behind. Figure 11(b) plots the length distribution of the associative arrays in LD-Sketch when the memory-oriented heuristic is used for $\phi = 1MB$. It shows that more than $99.9\%$ of the associative arrays have length not exceeding five. Note that the accuracy-oriented heuristic uses nearly the same memory as the baseline algorithm.

Figures 11(c)-(f) present the accuracy results. The memory-oriented heuristic loses less than $2\%$ of precision in both heavy hitter and heavy changer detections. On the other hand, the accuracy-oriented heuristic maintains high accuracy. In particular, it improves the precision of heavy changer detection to above $99.6\%$ (see Figure 11(f)), while keeping the false negative rate less than $2\%$ for all cases (see Figures 11(c) and 11(e)).

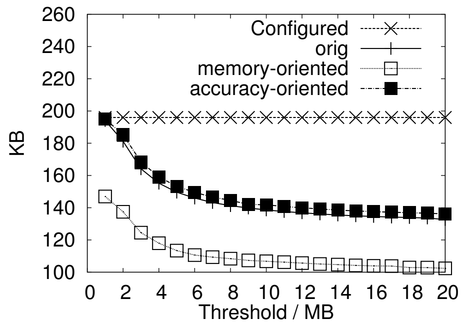
**Summary of results.** LD-Sketch consumes less memory and achieves higher accuracy than existing sketch-based algorithms for heavy key detection. A trade-off is that the speed of LD-Sketch is slightly slower than other sketch-based algorithms. Using distributed detection improves the precision of LD-Sketch significantly while losing no more than $1\%$ recall, and in the meantime, increases the update throughput. The enhancement heuristics further improve LD-Sketch by trading off between memory and accuracy.

## 7. Related Work

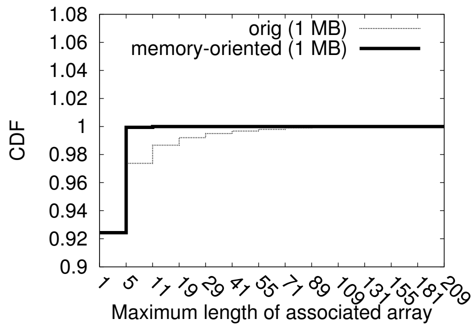
**Counter-based techniques.** The Misra-Gries algorithm [21] maintains an associative array of counters and keeps tracks of frequent items (see Section 2.2). Lossy Counting [19] extends the Misra-Gries algorithm by tracking the estimation error for each key. Space-Saving [20] improves the time and space efficiency with some specialized data structure. Probabilistic Lossy Counting [12] provides probabilistic guarantees on accuracy and consumes less memory by allowing errors in Lossy Counting. The above algorithms only work for heavy hitter detection, but do not address heavy changer detection.

**Sketch-based techniques.** Count-Sketch [5, 7] and Count-Min [11] are two well-known sketch algorithms for frequency estimation. The two algorithms differ in space requirement and error bounds. A comprehensive study [8] shows that both algorithms have similar performance in practice. Sketches have been used in heavy hitter detection (e.g., [13]) and heavy changer detection (e.g., [16]).

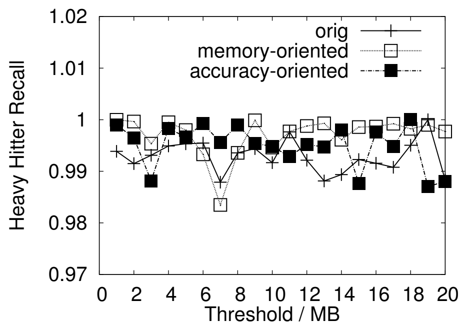
Some studies propose to efficiently restore heavy keys. Cormode et al. [9] consider a hierarchical structure of keys and formalize the problem of finding hierarchical heavy hitters. Combinational Group Testing (CGT) [10] exploits group testing to construct heavy keys from the extra information of buckets. Reversible Sketch [23] generalizes this approach by searching smaller divided keys and constructing heavy keys from the searched results. SeqHash [4] constructs a sequence of subspaces and searches each subspace based on the results of prior ones. Fast Sketch [17]



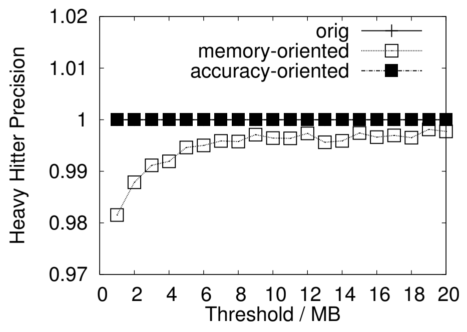
(a) Memory usage



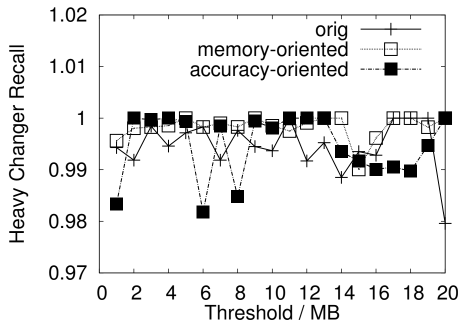
(b) Length distribution of associative arrays in LD-Sketch



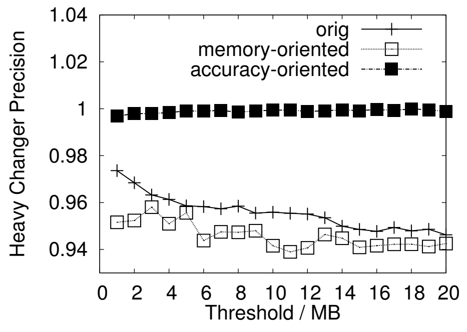
(c) Heavy hitter, Recall



(d) Heavy hitter, Precision



(e) Heavy changer, Recall



(f) Heavy changer, Precision

Figure 11: Experiment 5 (Effectiveness of enhancement heuristics).

optimizes the space and time complexity of group testing by a quotient technique. Group testing assumes that there is exactly one heavy key in a bucket. It leads to a high false negative rate when multiple heavy keys are hashed to the same bucket due to constrained memory, as shown in our evaluation. In contrast, LD-Sketch queries the heavy key candidates in the buckets and guarantees no false negatives.

**Distributed detection.** Cormode et al. [6] exploit the linear property of sketches for distributed detection, and reduce I/O using a prediction model. Manjhi et al. [18] organize all workers into a tree structure, in which each level has its own error parameter to control the detection accuracy. Yi et al. [27] address the worst-case communication complexity in distributed streaming. Mergeable Summary [1] extends the counter-based technique [21] and proposes a technique to merge two associative arrays. However, the above studies only address heavy hitter detection, while the distributed detection of LD-Sketch addresses heavy changer detection as well.

## 8. Conclusions

We present LD-Sketch, a novel distributed sketching design that aims to achieve real-time detection of traffic anomalies, including heavy hitters and heavy changers, in today's IP networks. It combines the classical counter-based and sketch-based techniques, and leverages parallelization of the emerging distributed streaming architectures to achieve both accurate and scalable detection. It is composed of local detection and distributed detection, and for both components we derive the error bounds, space complexity, and time complexity. We also examine the impact of the ordering of data items on LD-Sketch and propose two enhancement heuristics for LD-Sketch. We show via extensive trace-driven experiments that LD-Sketch provides more accurate detection than existing sketch-based techniques, and has both accuracy and scalability improvements in a distributed setting.

## Acknowledgments

This work was supported by (i) GRF CUHK413711 from the Research Grant Council of Hong Kong and (ii) TH127485 from Huawei Technologies.

## References

[1] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable Summaries. In *Proc. of PODS*, 2012.
[2] Apache Flume. `http://flume.apache.org`.
[3] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of IMC*, Nov. 2010.
[4] T. Bu, J. Cao, A. Chen, and P. P. Lee. Sequential Hashing: A Flexible Approach for Unveiling Significant Patterns in High Speed Networks. *Computer Networks*, 54(18):3309–3326, 2010.
[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3–15, 2004.
[6] G. Cormode and M. Garofalakis. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *Proc. of VLDB*, 2005.
[7] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Now Publishers Inc., Jan. 2012.
[8] G. Cormode and M. Hadjieleftheriou. Methods for Finding Frequent Items in Data Streams. *VLDB Journal*, 19(1):3–20, 2010.
[9] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Data Streams. In *Proc. of VLDB*, 2003.
[10] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. In *Proc. of INFOCOM*, 2004.
[11] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[12] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters. *SIGCOMM Comput. Commun. Rev.*, 38(1):5–5, Jan. 2008.
[13] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting : Focusing on the Elephants , Ignoring the Mice. *TOCS*, 21(3):270–313, 2003.
[14] X. He, P. P. C. Lee, L. Pan, C. He, and J. C. S. Lui. A panoramic view of 3G data/control-plane traffic: Mobile device perspective. In *Proc. of IFIP/TC6 Networking*, 2012.
[15] Q. Huang and P. P. C. Lee. LD-Sketch: A Distributed Sketching Design for Accurate and Scalable Anomaly Detection in Network Data Streams. In *Proc. of INFOCOM*, 2014.
[16] B. Krishnamurthy, S. Sen, Y. Zhang, F. Park, and Y. Chen. Sketch-based Change Detection : Methods , Evaluation , and Applications. In *Proc. of IMC*, 2003.
[17] Y. Liu, W. Chen, and Y. Guan. A Fast Sketch for Aggregate Queries over High-Speed Network Traffic. In *Proc. of INFOCOM*, 2012.
[18] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *Proc. of ICDE*, 2005.
[19] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. of VLDB*, 2002.
[20] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams . In *Proc. of ICDT*, 2005.
[21] J. Misra and D. Gries. Finding Repeated Elements. In *Science of Comput Program 2*, 1982.
[22] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *KDCloud*, Dec. 2010.
[23] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. Dinda, M. Y. Kao, and G. Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. *IEEE/ACM Transactions on Networking*, 15(5):1059–1072, 2007.
[24] M. Z. Shafiq, L. Ji, A. X. Liu, and J. Wang. Characterizing and Modeling Internet Traffic Dynamics of Cellular Devices. In *Proc. of SIGMETRICS*, June 2011.
[25] Storm. `https://github.com/nathanmarz/storm`.
[26] K. Thompson, G. J. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *Netwrk. Mag. of Global Internetworking*, 11:10–23, 1997.
[27] K. Yi and Q. Zhang. Optimal Tracking of Distributed Heavy Hitters and Quantiles. In *Proc. of PODS*, 2009.

## Appendix: Proofs

*Proof of Lemma 3*

*Proof.* From Algorithm 1, $S(x) \geq A_{i,j}[x] = S_{i,j}^{low}(x)$ since $A_{i,j}[x]$ is never incremented due to other items not belonging to $x$.

Now we prove the lower bound of $S_{i,j}^{low}(x)$. The main idea of the proof is to sum the maximum possible errors introduced for all expansion numbers. For each expansion number $\kappa$ ($0 \leq \kappa \leq k$), $A_{i,j}$ contains $l_{i,j} = (\kappa+1)(\kappa+2) - 1$ counters, so key $x$ is decremented by at most $\frac{T}{l_{i,j}+1} = \frac{T}{(\kappa+1)(\kappa+2)}$ (by Lemma 1). Note that once key $x$ is decremented by a value, $(\kappa+1)(\kappa+2)-1$ other keys are also decremented by the same value. Therefore, to achieve the maximum decrement of key $x$, the total decrement of other keys in the bucket is $\frac{(\kappa+1)(\kappa+2)-1}{(\kappa+1)(\kappa+2)}T$.

Thus, before the expansion number $k+1$, $A_{i,j}[x]$ is decremented by at most $\sum_{\kappa=0}^{k} \frac{T}{(\kappa+1)(\kappa+2)} = \sum_{\kappa=0}^{k}(\frac{1}{\kappa+1} - \frac{1}{\kappa+2})T = (\frac{k+1}{k+2})T$. To achieve this maximum decrement, it requires the total decrement of other keys to be equal to $\sum_{\kappa=0}^{k}(\frac{(\kappa+1)(\kappa+2)-1}{(\kappa+1)(\kappa+2)}T = (k+1)T - \frac{k+1}{k+2}T = \frac{(k+1)^2}{k+2}T$. Since $\sum_{y \neq x, f_i(y)=j} S(y) \leq \frac{(k+1)^2}{k+2}T$, the results follow. □

*Proof of Lemma 4*

*Proof.* From Algorithm 1, $A_{i,j}[x]$ is decremented by at most $e_{i,j}$, so $A_{i,j}[x] \geq S(x) - e_{i,j}$ and hence $S(x) \leq S_{i,j}^{up}(x)$.

We now prove the upper bound of $S_{i,j}^{up}(x)$. The main idea of the proof is again to sum the maximum possible errors introduced for all expansion numbers, similar to the proof of Lemma 3. Let $e'_{i,j} = S(x) - S_{i,j}^{low}(x)$ be the decrements of $A_{i,j}[x]$. Thus, $e_{i,j} - e'_{i,j}$ corresponds to the decrements of $A_{i,j}[y]$ for any $y \neq x$ when $A_{i,j}[x]$ is not decremented, and is contributed by sum of other keys not $x$ in the bucket, whose maximum value is $\sum_{y \neq x, f_i(y)=j} S(y)$. Thus, $e_{i,j} - e'_{i,j}$ is at most $\sum_{\kappa=0}^{k} \frac{T}{(\kappa+1)(\kappa+2)} = \frac{k+1}{k+2}T$. Since $S_{i,j}^{up}(x) = S_{i,j}^{low}(x) + e_{i,j} = S(x) + (e_{i,j} - e'_{i,j})$, the results follow. □

*Proof of Lemma 5*

*Proof.* The main idea of the proof is that the false negative rate is directly derived from Lemmas 3 and 4.

By Lemma 3, if $S(x) \geq \phi$, then for any bucket $(i,j)$ associated with $x$, we must have $A_{i,j}[x] = S_{i,j}^{low}(x) > S(x) - T \geq 0$ (note that $T = \epsilon\phi$ for heavy hitter detection and $T = \epsilon\phi/2$ for heavy changer detection). For heavy changer detection, if $D(x) \geq \phi$, there must be at least one epoch where $S(x) \geq 0$. Therefore, $x$ must be kept in the associative array of its corresponding buckets.

By Lemma 4, we know $S(x) \leq S_{i,j}^{up}$ for every bucket $(i,j)$. If $S(x) \geq \phi$, then $S_{i,j}^{up}(x) \geq \phi$, so $x$ must be reported as a heavy hitter. Also, $D(x) \leq D_{i,j}(x)$ for every bucket $(i,j)$. If $D(x) \geq \phi$, then $D_{i,j}(x) \geq \phi$, so $x$ must be reported as a heavy changer. □

*Proof of Lemma 6*

*Proof.* The main idea of the proof is to apply Markov's inequality to derive the false positive rate of heavy hitter detection.

In heavy hitter detection, we set $T = \epsilon\phi$. By Lemma 4, for key $x$ with $S(x) \leq (1-\epsilon)\phi$, $S_{i,j}^{up}(x) \leq S(x) + T \leq \phi$. Thus, key $x$ will never be reported.

For key $x$ with $(1-\epsilon)\phi < S(x) < (1-\epsilon/2)\phi$, there always exists an integer $k \geq 0$ such that $S(x) < (1-\epsilon)\phi + \frac{\epsilon\phi}{k+2}$. If $\sum_{y \neq x, f_i(y)=j} S(y) \leq (k+1)\epsilon\phi$, by Lemma 4, $S_{i,j}^{up}(x) \leq (\frac{k+1}{k+2})\epsilon\phi + S(x) < (\frac{k+1}{k+2})\epsilon\phi + (1-\epsilon)\phi + \frac{\epsilon\phi}{k+2} = \phi$. So $x$ is not reported as a heavy hitter. Thus, $x$ is reported as a heavy hitter only if $\sum_{y \neq x, f_i(y)=j} S(y) \geq (k+1)\epsilon\phi$ for all row $i$. By Markov's inequality, $Pr\{\sum_{y \neq x, f_i(y)=j} S(y) \geq (k+1)\epsilon\phi\} \leq \frac{U}{w[(k+1)\epsilon\phi]}$. Since the $r$ hash functions are independent, the probability that $x$ is reported as a heavy hitter is $(\frac{U}{w(k+1)\epsilon\phi})^r \leq (\frac{U}{w\epsilon\phi})^r$. □

*Proof of Lemma 7*

*Proof.* The main idea of the proof of Lemma 7 is similar to that of Lemma 6.

For heavy changer detection, we set $T = \epsilon\phi/2$. Denote the sum of key $x$ in last and current epochs by $S^1(x)$ and $S^2(x)$, respectively. Without losing generality, we assume $S^2(x) \geq S^1(x)$ and hence $D(x) = S^2(x) - S^1(x)$.

For key $x$ with $D(x) \leq (1-\epsilon)\phi$, by Lemma 3 and Lemma 4, the estimated change $D_{i,j}(x) < [S^2(x) + T] - [S^1(x) - T] = D(x) + 2T \leq (1-\epsilon)\phi + \epsilon\phi = \phi$, implying that $x$ is not reported as a heavy changer.

For key $x$ with $D(x) < (1-\epsilon/2)\phi$, there always exits an integer $k \geq 0$ such that $D(x) < (1-\epsilon)\phi + \frac{\epsilon\phi}{k+2}$. If $\sum_{y \neq x, f_i(y)=j} S^1(y) < \frac{(k+1)^2}{k+2}T$ in the first sketch and $\sum_{y \neq x, f_i(y)=j} S^2(y) < (k+1)T$ in the second one, by Lemma 3 and Lemma 4, both $S^{up,2}(x) - S^2(x)$ and $S^1(x) - S^{low,1}(x)$ are at most $\frac{k+1}{k+2}T$. The estimated change $D_{i,j}(x) \leq D(x) + 2\frac{k+1}{k+2}T < (1-\epsilon)\phi + \frac{\epsilon\phi}{k+2} + \frac{k+1}{k+2}\epsilon\phi \leq \phi$, implying that $x$ is not reported as a heavy changer. Thus, key $x$ is reported as a heavy changer only if $\sum_{y \neq x, f_i(y)=j} S^1(y) \geq \frac{(k+1)^2}{k+2}T$ or $\sum_{y \neq x, f_i(y)=j} S^2(y) \geq (k+1)T$. The probability of at least one of them occurs is at most $Pr\{\sum_{y \neq x, f_i(y)=j} S^1(y) \geq \frac{(k+1)^2}{k+2}T\} + Pr\{\sum_{y \neq x, f_i(y)=j} S^2(y) \geq (k+1)T\} \leq \frac{2(k+2)U}{w(k+1)^2\epsilon\phi} + \frac{2U}{w(k+1)\epsilon\phi} \leq \frac{6U}{w\epsilon\phi}$. Since the $r$ hash functions are independent, the probability that $x$ is reported as a heavy changer is at most $(\frac{6U}{w\epsilon\phi})^r$. □

*Proof of Lemma 8*

*Proof.* The main idea of the proof is to compute the expectation of $l_{i,j}$.

In the worst case, the associative array $A_{i,j}$ is expanded as large as possible, so $l_{i,j} = O(k^2) = O((\frac{V_{i,j}}{T})^2)$. Then we consider $E[V_{i,j}^2]$, where $E[V_{i,j}^2] = E[V_{i,j}]^2 + Var[V_{i,j}]$.

Let $Y_x$ be an indicator variable such that $Y_x = 1$ if key $x$ is hashed to bucket $(i,j)$, or $Y_x = 0$ otherwise. Thus, $V_{i,j} = \sum_{x \in [n]} S(x) Y_x$. Given that the keys are uniformly hashed to the buckets, we have $E[Y_x] = 1/w$ and $Var[Y_x] = \frac{w-1}{w^2}$. Thus, $E[V_{i,j}] = \frac{U}{w}$.

Also, since all keys are hashed independently, $Var[V_{i,j}] = Var[\sum_{x \in [n]} S(x) Y_x] = \frac{w-1}{w^2}\sum_{x \in [n]} (S(x))^2$. Since $w$ is much smaller than the number of available keys $n$, we have $w \sum_{x \in [n]} (S(x))^2 < (\sum_{x \in [n]} S(x))^2 = U^2$. Hence $Var[V_{i,j}] = O(\frac{w-1}{w^3}U^2)$.

Thus, combining the two terms, $E[l_{i,j}] = O(\frac{U^2}{w^2T^2})$. □

*Proof of Lemma 9*

*Proof.* The main idea of the proof is to derive the complexity results based on Lemma 8.

We first examine the space complexity. By Lemma 8, the expected space of an LD-Sketch is the total size of all buckets of the sketch and all associative arrays, and is given by $O(rw(1 + l_{i,j})) = O(r(w + \frac{U^2}{wT^2}))$.

We then discuss the time complexities of both update and detection procedures. To update a data item, we update $r$ buckets of the sketch. For each bucket $(i,j)$, we perform $O(1)$ operations if we insert keys directly or expand its associative array. On the other hand, if we decrement keys, we need to go through the entire associative array and perform $l_{i,j}$ operations. Note that Lemma 8 bounds the average length of an associative array. Thus, the time complexity of updating a data item is $O(r(1 + l_{i,j})) = O(r(1 + \frac{U^2}{w^2T^2}))$. For detection, we enumerate all buckets in an LD-Sketch and the associative arrays. The time complexity of detection is $O(rw(1+l_{i,j})) = O(r(w + \frac{U^2}{wT^2}))$. □

*Proof of Lemma 10*

*Proof.* The main idea of the proof is to apply Chernoff's bound adequately.

Let $X$ be the random variable of the sum of values of the heavy key $x$ received by a worker. We first consider heavy hitter detection. Note that $E(X) = \frac{S(x)}{d} \geq \frac{\phi}{d}$. By Chernoff bound, $Pr\{X < (1-\gamma)\frac{\phi}{d}\} < e^{-\frac{E(X)}{2}(1-(1-\gamma)\frac{d\phi}{E(X)})^2} \leq e^{-\frac{\phi}{2d}\gamma^2}$. This is also the probability of missing the key in a single worker. Thus, the probability of missing the key in at least one worker is at most $1 - (1 - e^{-\frac{\phi}{2d}\gamma^2})^d$.

For heavy changer detection, we denote $X$ as the difference of the heavy key $x$. Since $E(X) \geq \frac{\phi}{d}$. We can apply the same results as above. □

*Proof of Lemma 11*

*Proof.* We partition the update procedure into two phases. The first phase updates all small keys only. It expands $A_{i,j}$ by at most $k^{<}_{i,j}$ times. After this phase, we have $\hat{k}_{i,j} \leq k^{<}_{i,j}$, and hence $l_{i,j} \leq (k^{<}_{i,j}+1)(k^{<}_{i,j}+2)-1$.

The second phase updates the remaining $t^{\geq}_{i,j}$ large keys. By Lemma 5, all large keys must be kept in $A_{i,j}$. This implies that $A_{i,j}$ tracks the $t^{\geq}_{i,j}$ large keys, as well as at most $(k^{<}_{i,j}+1)(k^{<}_{i,j}+2)-1$ small keys in the worst case. To store all these keys, $\hat{k}_{i,j}$ is further incremented until $((\hat{k}_{i,j}-1)+1)((\hat{k}_{i,j}-1)+2)-1 \leq (k^{<}_{i,j}+1)(k^{<}_{i,j}+2)-1+t^{\geq}_{i,j} \leq l_{i,j} = (\hat{k}_{i,j}+1)(\hat{k}_{i,j}+2)-1$ (the leftmost term is $l_{i,j}$ at $\hat{k}_{i,j}-1$). Thus, $l_{i,j} = O(\hat{k}^2_{i,j}) = O({k^{<}_{i,j}}^2 + t^{\geq}_{i,j})$. □

*Proof of Lemma 12*

*Proof.* Similar to the proof of Lemma 11, we also partition the update procedure into two phases. The first phase updates all large keys only. When a new large key arrives and $A_{i,j}$ is full, $A_{i,j}$ must expand because the value of the large key is greater than $T$. Therefore, this phase keeps $e_{i,j}=0$ and the controlled expansion number $\hat{k}_{i,j} \leq k^{\geq}_{i,j}$.

The second phase updates the remaining small keys. When keys are decremented, we must have $\hat{k}_{i,j} \geq k^{\geq}_{i,j}$, and $e_{i,j}$ starts to be incremented. For the expansion number $\kappa$ ($k^{\geq}_{i,j} \leq \kappa \leq k$), $e_{i,j}$ is incremented by at most $\frac{T}{(\kappa+1)(\kappa+2)}$ (by Lemma 1). Thus, $e_{i,j}$ is at most $\sum_{\kappa=k^{\geq}_{i,j}}^{k_{i,j}} \frac{T}{(\kappa+1)(\kappa+2)} = \frac{T}{k^{\geq}_{i,j}+1} - \frac{T}{k_{i,j}+2} = O(\frac{T}{k^{\geq}_{i,j}})$. □