DenseAlert: Incremental Dense-Subtensor Detection in Tensor Streams

Kijung Shin, Bryan Hooi, Jisu Kim, Christos Faloutsos
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
{kijungs,christos}@cs.cmu.edu,{bhooi,jisuk1}@andrew.cmu.edu

ABSTRACT

Consider a stream of retweet events - how can we spot fraudulent lock-step behavior in such multi-aspect data (i.e., tensors) evolving over time? Can we detect it in real time, with an accuracy guarantee? Past studies have shown that dense subtensors tend to indicate anomalous or even fraudulent behavior in many tensor data, including social media, Wikipedia, and TCP dumps. Thus, several algorithms have been proposed for detecting dense subtensors rapidly and accurately. However, existing algorithms assume that tensors are static, while many real-world tensors, including those mentioned above, evolve over time.

We propose DenseStream, an incremental algorithm that maintains and updates a dense subtensor in a tensor stream (i.e., a sequence of changes in a tensor), and DenseAlert, an incremental algorithm spotting the sudden appearances of dense subtensors. Our algorithms are: (1) **Fast and 'any time'**: updates by our algorithms are up to **a million times faster** than the fastest batch algorithms, (2) **Provably accurate**: our algorithms guarantee a lower bound on the density of the subtensor they maintain, and (3) **Effective**: our DenseAlert successfully spots anomalies in real-world tensors, especially those overlooked by existing algorithms.

1 INTRODUCTION

Given a stream of changes in a tensor that evolves over time, how can we detect the sudden appearances of dense subtensors?

An important application of this problem is intrusion detection systems in networks, where attackers make a large number of connections to target machines to block their availability or to look for vulnerabilities [22]. Consider a stream of connections where we represent each connection from a source IP address to a destination IP address as an entry in a 3-way tensor (source IP, destination IP, timestamp). Sudden appearances of dense subtensors in the tensor often indicate network attacks. For example, in Figure 1(c), all the top 15 densest subtensors concentrated in a short period of time, which are detected by our DenseAlert algorithm, actually come from network attacks.

Another application is detecting fake rating attacks in review sites, such as Amazon and Yelp. Ratings can be modeled as entries in a 4-way tensor (user, item, timestamp, rating). Injection attacks maliciously manipulate the ratings of a set of items by adding a large number of similar ratings for the items, creating dense subtensors in the tensor. To guard against such fraud, an alert system detecting suddenly appearing dense subtensors in real time, as they arrive, is desirable.

Table 1: Comparison of DenseStream, DenseAlert, and previous algorithms for detecting dense subtensors.

	M-Zoom [32]	D-Cube [33]	CrossSpot [18]	MAF [23]	Fraudar [16]	DenseStream	DenseAlert
Multi-Aspect Data	✓	✓	✓	✓		✓	✓
Accuracy Guarantees	✓	✓			✓	✓	✓
Incremental Updates						✓	✓
Slowly Formed Dense Subtensors	✓	✓	✓	✓	✓	✓	
Small Sudden Dense Subtensors							✓

Several algorithms for dense-subtensor detection have been proposed for detecting network attacks [23, 32, 33], retweet boosting [18], rating attacks [33], and bots [32] as well as for genetics applications [28]. As summarized in Table 1, however, existing algorithms assume a static tensor rather than a stream of events (i.e., changes in a tensor) over time. In addition, our experiments in Section 4 show that they are limited in their ability to detect dense subtensors small but highly concentrated in a short period of time.

Our incremental algorithm DenseStream detects dense subtensors in real time as events arrive, and is hence more useful in many practical settings, including those mentioned above. DenseStream is also used as a building block of DenseAlert, an incremental algorithm for detecting the sudden emergences of dense subtensors. DenseAlert takes into account the tendency for lock-step behavior, such as network attacks and rating manipulation attacks, to appear within short, continuous intervals of time, which is an important signal for spotting lockstep behavior.

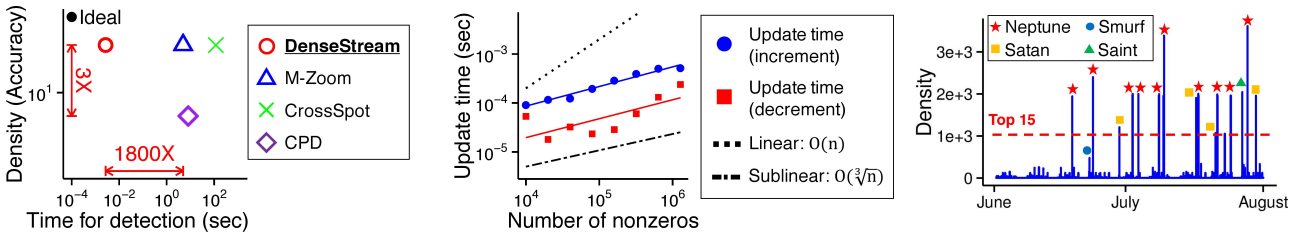
As the entries of a tensor change, our algorithms work by maintaining a small subset of subtensors that always includes a dense subtensor with a theoretical guarantee on its density. By focusing on this subset, our algorithms detect a dense subtensor in a time-evolving tensor up to a million times faster than the fastest batch algorithms, while providing the same theoretical guarantee on the density of the detected subtensor.

In summary, the main advantages of our algorithms are:

- **Fast and 'any time'**: incremental updates by our algorithms are up to *a million times faster* than the fastest batch algorithms (Figure 1(a)).
- **Provably accurate**: our algorithms maintain a subtensor with a theoretical guarantee on its density, and in practice,

KDD'17, August 13–17, 2017, Halifax, NS, Canada.

DOI: 10.1145/3097983.3098087



(a) Speed and accuracy of DenseStream (b) Scalability of DenseStream (c) Effectiveness of DenseAlert

Figure 1: Proposed algorithms are fast, accurate, scalable, and effective. (a) DenseStream, our incremental algorithm, detects dense subtensors significantly faster than batch algorithms without losing accuracy, as seen in the result in Yelp Dataset. (b) The time taken for each update in DenseStream grows sub-linearly with the size of data. (c) DenseAlert, which detects suddenly emerging dense subtensors, identifies network attacks from a TCP Dump with high accuracy (AUC=0.924). Especially, all the 15 densest subtensors revealed by DenseAlert indicate actual network attacks of various types.

its density is similar to that of subtensors found by the best batch algorithms (Figure 1(a)).

- **Effective**: DenseAlert successfully detects bot activities and network intrusions (Figure 1(c)) in real-world tensors. It also spots small-scale rating manipulation attacks, overlooked by existing algorithms.

Reproducibility: The code and data we used in the paper are available at http://www.cs.cmu.edu/~kijungs/codes/alert.

In Section 2, we introduce notations and problem definitions. In Section 3, we describe our proposed algorithms: DenseStream and DenseAlert. In Section 4, we present experimental results. After reviewing related work in Section 5, we conclude in Section 6.

2 NOTATIONS AND DEFINITIONS

In this section, we introduce notations and concepts used in the paper. Then, we give formal problem definitions.

2.1 Notations and Concepts.

Symbols frequently used in the paper are listed in Table 2, and a toy example is in Example 2.1. We use $[y] = \{1, 2..., y\}$ for brevity.

Notations for Tensors: Tensors are multi-dimensional arrays that generalize vectors (1-way tensors) and matrices (2-way tensors) to higher orders. Consider an N-way tensor $\mathcal{T}$ of size $I_1 \times ... \times I_N$ with non-negative entries. Each $(i_1, ..., i_N)$-th entry of $\mathcal{T}$ is denoted by $t_{i_1 \dots i_N}$. Equivalently, each *n-mode index* of $t_{i_1 \dots i_N}$ is i_n. We use $\mathcal{T}_{(n,i_n)}$ to denote the *n-mode slice* (i.e. $(N-1)$-way tensor) obtained by fixing n-mode index to i_n. Then, $Q = \{(n, i_n) : n \in [N], i_n \in [I_n]\}$ indicates all the slice indices. We denote a member of Q by q.

For example, if $N = 2$, $\mathcal{T}$ is a matrix of size $I_1 \times I_2$. Then, $\mathcal{T}_{(1,i_1)}$ is the i_1-th row of $\mathcal{T}$, and $\mathcal{T}_{(2,i_2)}$ is the i_2-th column of $\mathcal{T}$. In this setting, Q is the set of all row and column indices.

Notations for Subtensors: Let S be a subset of Q. $\mathcal{T}(S)$ denotes the subtensor composed of the slices with indices in S, i.e., $\mathcal{T}(S)$ is the subtensor left after removing all the slices with indices not in S.

For example, if $\mathcal{T}$ is a matrix (i.e., $N = 2$) and $S = \{(1, 1), (1, 2), (2, 2), (2, 3)\}$, $\mathcal{T}(S)$ is the submatrix of $\mathcal{T}$ composed of the first and second rows and the second and third columns.

Notations for Orderings: Consider an ordering of the slice indices in Q. A function $\pi : [|Q|] \to Q$ denotes such an ordering where, for each $j \in [|Q|]$, $\pi(j)$ is the slice index in the jth position. That is, each slice index $q \in Q$ is in the $\pi^{-1}(q)$-th position in π. Let $Q_{\pi,q} = \{r \in Q : \pi^{-1}(r) \geq \pi^{-1}(q)\}$ be the slice indices located after or equal to q in π. Then, $\mathcal{T}(Q_{\pi,q})$ is the subtensor of $\mathcal{T}$ composed of the slices with their indices in $Q_{\pi,q}$.

Table 2: Table of symbols.

Symbol	Definition
$\mathcal{T}$	an input tensor
N	order of $\mathcal{T}$
$t_{i_1 \dots i_N}$	entry of $\mathcal{T}$ with index $(i_1, \dots, i_N)$
Q	set of the slice indices of $\mathcal{T}$
q	a member of Q
$\mathcal{T}(S)$	subtensor composed of the slices in $S \subset Q$
$\pi : [\|Q\|] \to Q$	an ordering of slice indices in Q
$Q_{\pi,q}$	slice indices located after or equal to q in π
$sum(\mathcal{T}(S))$	sum of the entries included in $\mathcal{T}(S)$
$d(\mathcal{T}(S), q)$	slice sum of q in $\mathcal{T}(S)$
$d_\pi(q)$	slice sum of q in $\mathcal{T}(Q_{\pi,q})$
$c_\pi(q)$	cumulative max. slice sum of q in $\mathcal{T}(Q_{\pi,q})$
$((i_1, \dots, i_N), \delta, +)$	increment of $t_{i_1 \dots i_N}$ by δ
$((i_1, \dots, i_N), \delta, -)$	decrement of $t_{i_1 \dots i_N}$ by δ
$\rho(\mathcal{T}(S))$	density of a subtensor $\mathcal{T}(S)$
ρ_{opt}	density of the densest subtensor in $\mathcal{T}$
ΔT	time window in DenseAlert
$[y]$	$\{1, 2..., y\}$

Notations for Slice Sum: We denote the sum of the entries of $\mathcal{T}$ included in subtensor $\mathcal{T}(S)$ by $sum(\mathcal{T}(S))$. Similarly, we define *the slice sum* of $q \in Q$ in subtensor $\mathcal{T}(S)$, denoted by $d(\mathcal{T}(S), q)$, as the sum of the entries of $\mathcal{T}$ that are included in both $\mathcal{T}(S)$ and the slice with index $q \in Q$. For an ordering π and a slice index $q \in Q$, we use $d_\pi(q) = d(\mathcal{T}(Q_{\pi,q}), q)$ for brevity, and define the *cumulative maximum slice sum* of q as $c_\pi(q) = \max\{d_\pi(r) : r \in Q, \pi^{-1}(r) \leq \pi^{-1}(q)\}$, i.e., maximum $d_\pi(\cdot)$ among the slice indices located before or equal to q in π.

Notations for Tensor Streams: A tensor stream is a sequence of changes in $\mathcal{T}$. Let $((i_1, ..., i_N), \delta, +)$ be an increment of entry $t_{i_1 \dots i_N}$ by $\delta > 0$ and $((i_1, ..., i_N), \delta, -)$ be a decrement of entry $t_{i_1 \dots i_N}$ by $\delta > 0$.

Example 2.1 (Wikipedia Revision History). Consider the 3-way tensor in Figure 2. In the tensor, each entry t_{ijk} indicates that user i revised page j on date k, t_{ijk} times. The set of the slice indices is

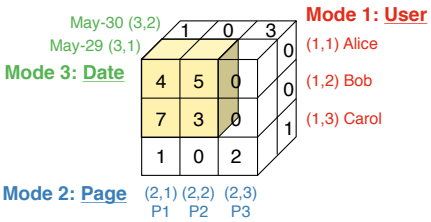


Figure 2: Pictorial depiction of Example 2.1. The colored region indicates subtensor $\mathcal{T}(S)$.

$Q = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2)\}$. Consider its subset $S = \{(1,1),(1,2),(2,1),(2,2),(3,1)\}$. Then, $\mathcal{T}(S)$ is the subtensor composed of the slices with their indices in S, as seen in Figure 2. In this setting, $sum(\mathcal{T}(S)) = 4+5+7+3 = 19$, and $d(\mathcal{T}(S),(2,2)) = 5+3 = 8$. Let π be an ordering of Q where $\pi(1) = (1,3)$, $\pi(2) = (2,3)$, $\pi(3) = (3,2)$, $\pi(4) = (2,2)$, $\pi(5) = (1,1)$, $\pi(6) = (1,2)$, $\pi(7) = (2,1)$, and $\pi(8) = (3,1)$. Then, $Q_{\pi,(2,2)} = S$, and $d_\pi((2,2)) = d(\mathcal{T}(Q_{\pi,(2,2)}),(2,2)) = d(\mathcal{T}(S),(2,2)) = 8$.

2.2 Density Measure.

Definition 2.2 gives the density measure used in this work. That is, the density of a subtensor is defined as the sum of its entries divided by the number of the slices composing it. We let ρ_{opt} be the density of the densest subtensor in $\mathcal{T}$.

Definition 2.2. (Density of a subtensor [32]). *Consider a subtensor $\mathcal{T}(S)$ of a tensor $\mathcal{T}$. The density of $\mathcal{T}(S)$, which is denoted by $\rho(\mathcal{T}(S))$, is defined as*

$$\rho(\mathcal{T}(S)) = \frac{sum(\mathcal{T}(S))}{|S|}.$$

This measure is chosen because: (a) it was successfully used for anomaly and fraud detection [32, 33], (b) this measure satisfies axioms that a reasonable "anomalousness" measure should meet (see Section A of the supplementary document [1]), and (c) our algorithm based on this density measure outperforms existing algorithms based on different density measures in Section 4.5.1.

2.3 Problem Definitions.

We give the formal definitions of the problems studied in this work. The first problem (Problem 1) is to maintain the densest subtensor in a tensor that keeps changing.

Problem 1 (Detecting the Densest Subtensor in a Tensor Stream). ***(1) Given:*** *a sequence of changes in a tensor $\mathcal{T}$ with slice indices Q (i.e., a tensor stream)* ***(2) maintain:*** *a subtensor $\mathcal{T}(S)$ where $S \subset Q$,* ***(3) to maximize:*** *its density $\rho(\mathcal{T}(S))$.*

Identifying the exact densest subtensor is computationally expensive even for a static tensor. For example, it takes $O(|Q|^6)$ even when $\mathcal{T}$ is a binary matrix (i.e., $N = 2$) [15]. Thus, we focus on designing an approximation algorithm that maintains a dense subtensor with a provable approximation bound, significantly faster than repeatedly finding a dense subtensor from scratch.

The second problem (Problem 2) is to detect suddenly emerging dense subtensors in a tensor stream. For a tensor $\mathcal{T}$ whose values increase over time, let $\mathcal{T}_{\Delta T}$ be the tensor where the value of each entry is the increment in the corresponding entry of $\mathcal{T}$ in the last ΔT time units. Our aim is to spot dense subtensors appearing in $\mathcal{T}_{\Delta T}$, which also keeps changing.

Algorithm 1 Dense-subtensor detection in a static tensor

Input: a tensor $\mathcal{T}$ with slice indices Q
Output: a dense subtensor $\mathcal{T}(S_{max})$

```
compute π(·), d_π(·), c_π(·) by D-ordering()
S_max ← Find-Slices()
return T(S_max)
procedure D-ordering():
        ▷ find a D-ordering π(·) and compute d_π(·) and c_π(·)
    S ← Q; c_max ← 0                          ▷ c_max: max. d_π(·) so far
    for j ← 1...|Q| do
        q ← arg min_{r∈S} d(T(S), r)          ▷ q has min. slice sum
        π(j) ← q                              ▷ S = Q_{π,q}
        d_π(q) ← d(T(S), q)                   ▷ d_π(q) = d(T(Q_{π,q}), q)
        c_π(q) ← max(c_max, d_π(q)); c_max ← c_π(q)
        S ← S/{q}
procedure Find-Slices():
    ▷ find slices forming a dense subtensor from π(·) d_π(·), and c_π(·)
    S ← ∅; m ← 0;                             ▷ m: sum(T(S))
    ρ_max ← −∞; q_max ← 0                     ▷ ρ_max: max. density so far
    for j ← |Q|..1 do
        q ← π(j); S ← S ∪ {q}                 ▷ S = Q_{π,q}
        m ← m + d_π(q)                        ▷ m = sum(T(Q_{π,q}))
        if m/|S| > ρ_max then                 ▷ m/|S| = ρ(T(Q_{π,q}))
            ρ_max ← m/|S|; q_max ← q
    return Q_{π,q_max}                        ▷ q_max = arg max_{q∈Q} ρ(T(Q_{π,q}))
```

Problem 2 (Detecting Sudden Dense Subtensors in a Tensor Stream). ***(1) Given:*** *a sequence of increments in a tensor $\mathcal{T}$ with slice indices Q (i.e., a tensor stream) and a time window ΔT,* ***(2) maintain:*** *a subtensor $\mathcal{T}_{\Delta T}(S)$ where $S \subset Q$,* ***(3) to maximize:*** *its density $\rho(\mathcal{T}_{\Delta T}(S))$.*

3 PROPOSED METHOD

In this section, we propose DenseStream, which is an incremental algorithm for dense-subtensor detection in a tensor stream, and DenseAlert, which detects suddenly emerging dense subtensors. We first explain dense-subtensor detection in a static tensor in Section 3.1, then generalize this to DenseStream for a dynamic tensor in Section 3.2. Finally, we propose DenseAlert based on DenseStream in Section 3.3.

3.1 Dense Subtensor Detection in Static Data.

We propose Algorithm 1 for detecting a dense subtensor in a static tensor. Although it eventually finds the same subtensor as M-Zoom [32], Algorithm 1 also computes extra information, including a *D-ordering* (Definition 3.1), required for updating the subtensor in the following sections. Algorithm 1 has two parts: (a) **D-ordering**: find a D-ordering π and compute $d_\pi(\cdot)$ and $c_\pi(\cdot)$; and (b) **Find-Slices**: find slices forming a dense subtensor from the result of (a).

Definition 3.1. (D-ordering). *An ordering π is a* **D-ordering** *of Q in $\mathcal{T}$ if $\forall q \in Q$, $d(\mathcal{T}(Q_{\pi,q}),q) = \min_{r\in Q_{\pi,q}} d(\mathcal{T}(Q_{\pi,q}),r)$.*

That is, a D-ordering is an ordering of slice indices obtained by choosing a slice index with minimum slice sum repeatedly, as in D-ordering() of Algorithm 1.

Using a D-ordering drastically reduces the search space while providing a guarantee on the accuracy. With a D-ordering π, Algorithm 1 reduces the search space of $2^{|Q|}$ possible subtensors to $\{\mathcal{T}(Q_{\pi,q}) : q \in Q\}$. In this space of size $|Q|$, however, there always exists a subtensor whose density is at least 1/(order of the input tensor) of maximum density, as formalized in Lemmas 3.2 and 3.3.

Lemma 3.2. *Let $\mathcal{T}(S^*)$ be a subtensor with the maximum density, i.e., $\rho(\mathcal{T}(S^*)) = \rho_{opt}$. Then for any $q \in S^*$,*

$$d(\mathcal{T}(S^*), q) \geq \rho(\mathcal{T}(S^*)). \tag{1}$$

Proof. The maximality of the density of $\mathcal{T}(S^*)$ implies $\rho(\mathcal{T}(S^* \backslash \{q\})) \leq \rho(\mathcal{T}(S^*))$, and plugging in Definition 2.2 to ρ gives

$$\frac{sum(\mathcal{T}(S^*)) - d(\mathcal{T}(S^*), q)}{|S^*| - 1} = \frac{sum(\mathcal{T}(S^* \backslash \{q\}))}{|S^*| - 1}$$
$$= \rho(\mathcal{T}(S^* \backslash \{q\})) \leq \rho(\mathcal{T}(S^*)) = \frac{sum(\mathcal{T}(S^*))}{|S^*|},$$

which reduces to Eq. (1). ∎

Lemma 3.3. *Given a D-ordering π in an N-way tensor $\mathcal{T}$, there exists $q \in Q$ such that $\rho(\mathcal{T}(Q_{\pi,q})) \geq \rho_{opt}/N$.*

Proof. Let $\mathcal{T}(S^*)$ be satisfying $\rho(\mathcal{T}(S^*)) = \rho_{opt}$, and let $q^* \in S^*$ be satisfying that $\forall q \in S^*$, $\pi^{-1}(q^*) \leq \pi^{-1}(q)$. Our goal is to show $\rho(\mathcal{T}(Q_{\pi,q^*})) \geq \frac{1}{N}\rho(\mathcal{T}(S^*))$, which we show as $N\rho(\mathcal{T}(Q_{\pi,q^*})) \geq d(\mathcal{T}(Q_{\pi,q^*}), q^*) \geq d(\mathcal{T}(S^*), q^*) \geq \rho(\mathcal{T}(S^*))$.

To show $N\rho(\mathcal{T}(Q_{\pi,q^*})) \geq d(\mathcal{T}(Q_{\pi,q^*}), q^*)$, note $N\rho(\mathcal{T}(Q_{\pi,q^*})) = \frac{sum(\mathcal{T}(Q_{\pi,q^*}))N}{|Q_{\pi,q^*}|}$, and since $\mathcal{T}$ is an N-way tensor, each entry is included in N slices. Hence

$$\sum\nolimits_{q \in Q_{\pi,q^*}} d(\mathcal{T}(Q_{\pi,q^*}), q) = sum(\mathcal{T}(Q_{\pi,q^*}))N. \tag{2}$$

Since π is a D-ordering, $\forall q \in Q_{\pi,q^*}$, $d(\mathcal{T}(Q_{\pi,q^*}), q) \geq d(\mathcal{T}(Q_{\pi,q^*}), q^*)$ holds. Combining this and Eq. (2) gives

$$N\rho(\mathcal{T}(Q_{\pi,q^*})) = \frac{sum(\mathcal{T}(Q_{\pi,q^*}))N}{|Q_{\pi,q^*}|}$$
$$= \frac{\sum_{q \in Q_{\pi,q^*}} d(\mathcal{T}(Q_{\pi,q^*}), q)}{|Q_{\pi,q^*}|} \geq d(\mathcal{T}(Q_{\pi,q^*}), q^*).$$

Second, $d(\mathcal{T}(Q_{\pi,q^*}), q^*) \geq d(\mathcal{T}(S^*), q^*)$ is from that $S^* \subset Q_{\pi,q^*}$. Third, $d(\mathcal{T}(S^*), q^*) \geq \rho(\mathcal{T}(S^*))$ is from Lemma 3.2. From these, $\rho(\mathcal{T}(Q_{\pi,q^*})) \geq \frac{1}{N}\rho(\mathcal{T}(S^*))$ holds. ∎

Such a subtensor $\mathcal{T}(S_{max})$ is detected by Algorithm 1. That is, $\mathcal{T}(S_{max})$ has density at least 1/(order of the input tensor) of maximum density, as proved in Theorem 3.4.

Theorem 3.4 (Accuracy Guarantee of Algorithm 1). *The subtensor returned by Algorithm 1 has density at least ρ_{opt}/N.*

Proof. By Lemma 3.3, there exists a subtensor with density at least ρ_{opt}/N among $\{\mathcal{T}(Q_{\pi,q}) : q \in Q\}$. The subtensor with the highest density in the set is returned by Algorithm 1. ∎

The time complexity of Algorithm 1 is linear with $nnz(\mathcal{T})$, the number of the non-zero entries in $\mathcal{T}$, as formalized in Theorem 3.6. Especially, finding S_{max} takes only $O(|Q|)$ given $\pi(\cdot)$, $d_\pi(\cdot)$, and $c_\pi(\cdot)$, as shown in Lemma 3.5.

Lemma 3.5. *Let S_{max} be the set of slice indices returned by Find-Slices() in Algorithm 1, and let $\mathcal{T}(q)$ be the set of the non-zero entries in the slice with index q in $\mathcal{T}$. The time complexity of Find-Slices() in Algorithm 1 is $O(|Q|)$ and that of constructing $\mathcal{T}(S_{max})$ from S_{max} is $O(N|\bigcup_{q \in S_{max}} \mathcal{T}(q)|)$.*

Proof. Assume that, for each slice, the list of the non-zero entries in the slice is stored. In Find-Slices(), we iterate over the slices in Q, and each iteration takes $O(1)$. Thus, we get $O(|Q|)$. After finding S_{max}, in order to construct $\mathcal{T}(S_{max})$, we have to process each non-zero entry included in any slice in S_{max}. The number of such entries is $|\bigcup_{q \in S_{max}} \mathcal{T}(q)|$. Since processing each entry takes $O(N)$, constructing $\mathcal{T}(S_{max})$ takes $O(N|\bigcup_{q \in S_{max}} \mathcal{T}(q)|)$. ∎

Theorem 3.6 (Time Complexity of Algorithm 1). *The time complexity of Algorithm 1 is $O(|Q|\log|Q| + nnz(\mathcal{T})N)$.*

Proof. Assume that, for each slice, the list of the non-zero entries in the slice is stored. We first show that the time complexity of D-ordering() in Algorithm 1 is $O(|Q|\log|Q| + nnz(\mathcal{T})N|)$. Assume we use a Fibonacci heap to find slices with minimum slice sum (line 7). Computing the slice sum of every slice takes $O(nnz(\mathcal{T})N)$, and constructing a Fibonacci heap where each value is a slice index in Q and the corresponding key is the slice sum of the slice takes $O(|Q|)$. Popping the index of a slice with minimum slice sum, which takes $O(\log|Q|)$, happens $|Q|$ times, and thus we get $O(|Q|\log|Q|)$. Whenever a slice index is popped we have to update the slice sums of its dependent slices (two slices are dependent if they have common non-zero entries). Updating the slice sum of each dependent slice, which takes $O(1)$ in a Fibonacci heap, happens at most $O(nnz(\mathcal{T})N)$ times, and thus we get $O(nnz(\mathcal{T})N)$. Their sum results in $O(|Q|\log|Q| + nnz(\mathcal{T})N)$.

By Lemma 3.5, the time complexity of Find-Slices() is $O(|Q|)$, and that of constructing $\mathcal{T}(S_{max})$ from S_{max} is $O(N|\bigcup_{q \in S_{max}} \mathcal{T}(q)|)$.

Since the time complexity of D-ordering() dominates that of the remaining parts, we get $O(|Q|\log|Q| + nnz(\mathcal{T})N)$ as the time complexity of Algorithm 1. ∎

3.2 DenseStream: Dense-Subtensor Detection in a Tensor Stream.

How can we update the subtensor found in Algorithm 1 under changes in the input tensor, rapidly, only when necessary, with the same approximation bound? For this purpose, we propose DenseStream, which updates the subtensor while satisfying Property 1. We explain the responses of DenseStream to increments of entry values (Section 3.2.1), decrements of entry values (Section 3.2.2), and changes of the size of the input tensor (Section 3.2.3).

Property 1 (Invariants in DenseStream). *For an N-way tensor $\mathcal{T}$ that keeps changing, the ordering π of the slice indices and the dense subtensor $\rho(\mathcal{T}(S_{max}))$ maintained by DenseStream satisfy the following two conditions:*

- *π is a D-ordering of Q in $\mathcal{T}$*
- *$\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$.*

3.2.1 Increment of Entry Values. Assume that the maintained dense subtensor $\mathcal{T}(S_{max})$ and ordering π (with $d_\pi(\cdot)$ and $c_\pi(\cdot)$) satisfy Property 1 in the current tensor $\mathcal{T}$ (such π, $d_\pi(\cdot)$, $c_\pi(\cdot)$, and $\mathcal{T}(S_{max})$ can be initialized by Algorithm 1 if we start from scratch). Algorithm 2 describes the response of DenseStream to

Algorithm 2 DenseStream in the case of increment

Input: (1) current tensor: $\mathcal{T}$ with slice indices Q
(2) current dense subtensor: $\mathcal{T}(S_{max})$ with Property 1
(3) current D-ordering: $\pi(\cdot)$ (also $d_\pi(\cdot)$ and $c_\pi(\cdot)$)
(4) a change in $\mathcal{T}$: $((i_1, \ldots, i_N), \delta, +)$
Output: updated dense subtensor $\mathcal{T}(S_{max})$

1: $t_{i_1 \ldots i_N} \leftarrow t_{i_1 \ldots i_N} + \delta$
2: compute j_L and j_H by Eq. (3) and Eq. (4) ▷ [Find-Region]
3: compute R by Eq. (5) ▷ [Reorder]
4: $S \leftarrow \{q \in Q : \pi^{-1}(q) \geq j_L\}$; $RS \leftarrow R \cap S$
5: $c_{max} \leftarrow 0$ ▷ c_{max}: max. $d_\pi(\cdot)$ so far
6: **if** $j_L > 1$ **then** $c_{max} \leftarrow c_\pi(\pi(j_L - 1))$
7: **for** $j \leftarrow j_L \ldots j_H$ **do**
8: $q \leftarrow \arg\min_{r \in RS} d(\mathcal{T}(S), r)$ ▷ q has min. slice sum
9: $\pi(j) \leftarrow q$ ▷ by Lemma 3.7, $S = Q_{\pi,q}$, $RS = R \cap Q_{\pi,q}$
10: $d_\pi(q) \leftarrow d(\mathcal{T}(S), q)$ ▷ $d_\pi(q) = d(\mathcal{T}(Q_{\pi,q}), q)$
11: $c_\pi(q) \leftarrow \max(c_{max}, d_\pi(q))$; $c_{max} \leftarrow c_\pi(q)$
12: $S \leftarrow S/\{q\}$; $RS \leftarrow RS/\{q\}$
13: **if** $c_{max} \geq \rho(\mathcal{T}(S_{max}))$ **then** ▷ [Update-Subtensor]
14: $S' \leftarrow$Find-Slices() in Algorithm 1 ▷ time complexity: $O(|Q|)$
15: **if** $S_{max} \neq S'$ **then** $\mathcal{T}(S_{max}) \leftarrow \mathcal{T}(S')$
16: **return** $\mathcal{T}(S_{max})$

$((i_1, \ldots, i_N), \delta, +)$, an increment of entry $t_{i_1 \ldots i_N}$ by $\delta > 0$, for satisfying Property 1. Algorithm 2 has three steps: (a) **Find-Region**: find a region of the D-ordering π that needs to be reordered, (b) **Reorder**: reorder the region obtained in (a), and (c) **Update-Subtensor**: use π to rapidly update $\mathcal{T}(S_{max})$ only when necessary. Each step is explained below.

(a) Find-Region (Line 2): The goal of this step is to find the region $[j_L, j_H] \subset [1, |Q|]$ of the domain of the D-ordering π that needs to be reordered in order for π to remain as a D-ordering after the change $((i_1, \ldots, i_N), \delta, +)$. Let $C = \{(n, i_n) : n \in [N]\}$ be the indices of the slices composing the changed entry $t_{i_1 \ldots i_N}$ and let $q_f = \arg\min_{q \in C} \pi^{-1}(q)$ be the one located first in π among C. Then, let $M = \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f),\ d_\pi(q) \geq d_\pi(q_f) + \delta\}$ be the slice indices that are located after q_f in π among Q and having $d_\pi(\cdot)$ at least $d_\pi(q_f) + \delta$. Then, j_L and j_H are set as follows:

$$j_L = \pi^{-1}(q_f), \tag{3}$$

$$j_H = \begin{cases} \min_{q \in M} \pi^{-1}(q) - 1 & \text{if } M \neq \emptyset, \\ |Q| \text{ (i.e., the last index)} & \text{otherwise.} \end{cases} \tag{4}$$

Later in this section, we prove that slice indices whose locations do not belong to $[j_L, j_H]$ need not be reordered by showing that there always exists a D-ordering π' in the updated $\mathcal{T}$ where $\pi'(j) = \pi(j)$ for every $j \notin [j_L, j_H]$.

(b) Reorder (Lines 3-12): The goal of this step is to reorder the slice indices located in the region $[j_L, j_H]$ so that π remains as a D-ordering in $\mathcal{T}$ after the change $((i_1, \ldots, i_N), \delta, +)$. Let $\mathcal{T}'$ be the updated $\mathcal{T}$ and π' be the updated π to distinguish them with $\mathcal{T}$ and π before the update. We get π' from π by reordering the slice indices in

$$R = \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\} \tag{5}$$

so that the following condition is met for every $j \in [j_L, j_H]$ and the corresponding $q = \pi'(j)$:

$$d(\mathcal{T}'(Q_{\pi',q}), q) = \min_{r \in R \cap Q_{\pi',q}} d(\mathcal{T}'(Q_{\pi',q}), r). \tag{6}$$

This guarantees that π' is a D-ordering in $\mathcal{T}'$, as shown in Lemma 3.7.

Lemma 3.7. *Let π be a D-ordering in $\mathcal{T}$, and let $\mathcal{T}'$ be $\mathcal{T}$ after a change $((i_1, \ldots, i_N), \delta, +)$. For R (Eq. (5)) defined on j_L and j_H, (Eq. (3) and Eq. (4)), let π' be an ordering of slice indices Q where $\forall j \notin [j_L, j_H]$, $\pi'(j) = \pi(j)$ and $\forall j \in [j_L, j_H]$, Eq. (6) holds. Then, π' is a D-ordering in $\mathcal{T}'$.*

Proof. See Section B of the supplementary document [1]. ■

(c) Update-Subtensor (Lines 13-15): In this step, we update the maintained dense subtensor $\mathcal{T}(S_{max})$ when two conditions are met. We first check $c_{max} \geq \rho(\mathcal{T}(S_{max}))$, which takes $O(1)$ if we maintain $\rho(\mathcal{T}(S_{max}))$, since $c_{max} < \rho(\mathcal{T}(S_{max}))$ entails that the updated entry $t_{i_1 \ldots i_N}$ is not in the densest subtensor (see the proof of Theorem 3.9 for details). We then check if there are changes in S_{max}, obtained by Find-Slices(). This takes only $O(|Q|)$, as shown in Theorem 3.6. Even if both conditions are met, updating $\mathcal{T}(S_{max})$ is simply to construct $\mathcal{T}(S_{max})$ from given S_{max} instead of finding $\mathcal{T}(S_{max})$ from scratch. This conditional update reduces computation but still preserves Property 1, as formalized in Lemma 3.8 and Theorem 3.9.

Lemma 3.8. *Consider a D-ordering π in $\mathcal{T}$. For every entry $t_{i_1 \ldots i_N}$ with index $(i_1, \ldots, i_N)$ belonging to the densest subtensor, $\forall n \in [N]$, $c_\pi((n, i_n)) \geq \rho_{opt}$ holds.*

Proof. Let $\mathcal{T}(S^*)$ be a subtensor with the maximum density, i.e., $\rho(\mathcal{T}(S^*)) = \rho_{opt}$. Let $q^* \in S^*$ be satisfying that $\forall q \in S^*$, $\pi^{-1}(q^*) \leq \pi^{-1}(q)$. For any entry $t_{i_1 \ldots i_N}$ in $\mathcal{T}(S^*)$ with index $(i_1, \ldots, i_N)$ and any $q \in \{(n, i_n) : n \in [N]\}$, our goal is to show $c_\pi(q) \geq \rho(\mathcal{T}(S^*))$, which we show as $c_\pi(q) \geq d_\pi(q^*) \geq d(\mathcal{T}(S^*), q^*) \geq \rho(\mathcal{T}(S^*))$.

First, $c_\pi(q) \geq d_\pi(q^*)$ is from the definition of $c_\pi(q)$ and $\pi^{-1}(q^*) \leq \pi^{-1}(q)$. Second, from $S^* \subset Q_{\pi,q^*}$, $d_\pi(q^*) = d(\mathcal{T}(Q_{\pi,q^*}), q^*) \geq d(\mathcal{T}(S^*), q^*)$ holds. Third, $d(\mathcal{T}(S^*), q^*) \geq \rho(\mathcal{T}(S^*))$ is from Lemma 3.2. From these, $c_\pi(q) \geq \rho(\mathcal{T}(S^*))$ holds. ■

Theorem 3.9 (Accuracy Guarantee of Algorithm 2). *Algorithm 2 preserves Property 1, and thus $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$ holds after Algorithm 2 terminates.*

Proof. We assume that Property 1 holds and prove that it still holds after Algorithm 2 is executed. First, the ordering π remains to be a D-ordering in $\mathcal{T}$ by Lemma 3.7. Second, we show $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$. If the condition in line 13 of Algorithm 2 is met, $\mathcal{T}(S_{max})$ is set to the subtensor with the maximum density in $\{\mathcal{T}(Q_{\pi,q}) : q \in Q\}$ by Find-Slices(). By Lemma 3.3, $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$. If the condition in line 13 is not met, for the changed entry $t_{i_1 \ldots i_N}$ with index $(i_1, \ldots, i_N)$, by the definition of j_L, there exists $n \in [N]$ such that $\pi(j_L) = (n, i_n)$. Since $j_L \leq j_H$, $c_\pi((n, i_n)) = c_\pi(\pi(j_L)) \leq c_\pi(\pi(j_H)) = c_{max} < \rho(\mathcal{T}(S_{max})) \leq \rho_{opt}$. Then, by Lemma 3.8, $t_{i_1 \ldots i_N}$ does not belong to the densest subtensor, which thus remains the same after the change $((i_1, \ldots, i_N), \delta, +)$. Since $\rho(\mathcal{T}(S_{max}))$ never decreases, $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$ still holds by Property 1, which we assume. Property 1 is preserved because its two conditions are met. ■

Theorem 3.10 gives the time complexity of Algorithm 2. In the worst case (i.e., $R = Q$), this becomes $O(|Q| \log |Q| + nnz(\mathcal{T})N)$, which is the time complexity of Algorithm 1. In practice, however, R is much smaller than Q, and updating $\mathcal{T}(S_{max})$ happens rarely. Thus, in our experiments, Algorithm 2 scaled sub-linearly with $nnz(\mathcal{T})$ (see Section 4.4).

Theorem 3.10 (Time Complexity of Algorithms 2 and 3). *Let $\mathcal{T}(q)$ be the set of the non-zero entries in the slice with index q in $\mathcal{T}$. The time complexity of Algorithms 2 and 3 is $O(|R| \log |R| + |Q| + N|\bigcup_{q\in R} \mathcal{T}(q)| + N|\bigcup_{q\in S_{max}} \mathcal{T}(q)|)$.*

Proof. *Assume that, for each slice, the list of the non-zero entries in the slice is stored, and let $q_f = \pi(j_L)$. Computing j_L, j_H, and R takes $O(|R|)$. Assume we use a Fibonacci heap to find slices with minimum slice sum (line 8 of Algorithm 2). Computing the slice sum of every slice in R in $\mathcal{T}(Q_\pi, q_f)$ takes $O(N|\bigcup_{q\in R} \mathcal{T}(q)|)$. Then, constructing a Fibonacci heap where each value is a slice index in R and the corresponding key is the slice sum of the slice in $\mathcal{T}(Q_\pi, q_f)$ takes $O(|R|)$. Popping the index of a slice with minimum slice sum, which takes $O(\log |R|)$, happens $|R|$ times, and thus we get $O(|R| \log |R|)$. Whenever a slice index is popped we have to update the slice sums of its dependent slices in R (two slices are dependent if they have common non-zero entries). Updating the slice sum of each dependent slice, which takes $O(1)$ in a Fibonacci heap, happens at most $O(N|\bigcup_{q\in R} \mathcal{T}(q)|)$ times, and thus we get $O(N|\bigcup_{q\in R} \mathcal{T}(q)|)$. On the other hand, by Lemma 3.5, Find-Slices() and constructing $\mathcal{T}(S_{max})$ from S_{max} take $O(|Q| + N|\bigcup_{q\in S_{max}} \mathcal{T}(q)|)$. Hence, the time complexity of Algorithms 2 and 3 is the sum of all the costs, which is $O(|R| \log |R| + |Q| + N|\bigcup_{q\in R} \mathcal{T}(q)| + N|\bigcup_{q\in S_{max}} \mathcal{T}(q)|)$.* ■

3.2.2 Decrement of Entry Values. As in the previous section, assume that a tensor $\mathcal{T}$, a D-ordering π (also $d_\pi(\cdot)$ and $c_\pi(\cdot)$), and a dense subtensor $\mathcal{T}(S_{max})$ satisfying Property 1 are maintained. (such π, $d_\pi(\cdot)$, $c_\pi(\cdot)$, and $\mathcal{T}(S_{max})$ can be initialized by Algorithm 1 if we start from scratch). Algorithm 3 describes the response of DenseStream to $((i_1, ..., i_N), \delta, -)$, a decrement of entry $t_{i_1 \dots i_N}$ by $\delta > 0$, for satisfying Property 1. Algorithm 3 has the same structure of Algorithm 2, while they are different in the reordered region of π and the conditions for updating the dense subtensor. The differences are explained below.

For a change $((i_1, ..., i_N), \delta, -)$, we find the region $[j_L, j_H]$ of the domain of π that may need to be reordered. Let $C = \{(n, i_n) : n \in [N]\}$ be the indices of the slices composing the changed entry $t_{i_1 \dots i_N}$, and let $q_f = \arg\min_{q\in C} \pi^{-1}(q)$ be the one located first in π among C. Then, let $M_{min} = \{q \in Q : d_\pi(q) > c_\pi(q_f) - \delta\}$ and $M_{max} = \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f),\ d_\pi(q) \geq c_\pi(q_f)\}$. Note that $M_{min} \neq \emptyset$ since, by the definition of $c_\pi(\cdot)$, there exists $q \in Q$ where $\pi^{-1}(q) \leq \pi^{-1}(q_f)$ and $d_\pi(q) = c_\pi(q_f)$. Then, j_L and j_H are:

$$j_L = \min_{q\in M_{min}} \pi^{-1}(q), \tag{7}$$

$$j_H = \begin{cases} \min_{q\in M_{max}} \pi^{-1}(q) - 1 & \text{if } M_{max} \neq \emptyset, \\ |Q| \text{ (i.e., the last index)} & \text{otherwise.} \end{cases} \tag{8}$$

As in the increment case, we update π, to remain it as a D-ordering, by reordering the slice indices located in $[j_L, j_H]$ of π. Let $\mathcal{T}'$ be the updated $\mathcal{T}$ and π' be the updated π to distinguish them with $\mathcal{T}$ and π before the update. Only the slice indices in $R = \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\}$ are reordered in π so that Eq. (6) is met for every $j \in [j_L, j_H]$. This guarantees that π' is a D-ordering, as formalized in Lemma 3.11.

Lemma 3.11. *Let π be a D-ordering in $\mathcal{T}$, and let $\mathcal{T}'$ be $\mathcal{T}$ after a change $((i_1, ..., i_N), \delta, -)$. For R (Eq. (5)) defined on j_L and j_H (Eq. (7) and Eq. (8)), let π' be an ordering of slice indices Q where $\forall j \notin [j_L, j_H]$, $\pi'(j) = \pi(j)$ and $\forall j \in [j_L, j_H]$, Eq. (6) holds. Then, π' is a D-ordering in $\mathcal{T}'$.*

Proof. *See Section B of the supplementary document [1].* ■

Algorithm 3 DenseStream in the case of decrement

Input: (1) current tensor: $\mathcal{T}$ with slice indices Q
(2) current dense subtensor: $\mathcal{T}(S_{max})$ with Property 1
(3) current D-ordering: $\pi(\cdot)$ (also $c_\pi(\cdot)$ and $d_\pi(\cdot)$)
(4) a change in $\mathcal{T}$: $((i_1, \dots, i_N), \delta, -)$
Output: updated dense subtensor $\mathcal{T}(S_{max})$

1: $t_{i_1 \dots i_N} \leftarrow t_{i_1 \dots i_N} - \delta$
2: compute j_L and j_H by Eq. (7) and (8) ▷ [Find-Region]
3: Lines 3-12 of Algorithm 2 ▷ [Reorder]
4: **if** $t_{i_1 \dots i_N}$ is in $\mathcal{T}(S_{max})$ **then** ▷ [Update-Subtensor]
5: $S' \leftarrow$Find-Slices() in Algorithm 1 ▷ time complexity: $O(|Q|)$
6: **if** $S_{max} \neq S'$ **then** $\mathcal{T}(S_{max}) \leftarrow \mathcal{T}(S')$
7: **return** $\mathcal{T}(S_{max})$

The last step of Algorithm 3 is to conditionally and rapidly update the maintained dense subtensor $\mathcal{T}(S_{max})$ using π. The subtensor $\mathcal{T}(S_{max})$ is updated if entry $t_{i_1 \dots i_N}$ belongs to $\mathcal{T}(S_{max})$ (i.e., if $\rho(\mathcal{T}(S_{max}))$ decreases by the change $((i_1, ..., i_N), \delta, -)$) and there are changes in S_{max}, obtained by Find-Slices(). Checking these conditions takes only $O(|Q|)$, as in the increment case. Even if $\mathcal{T}(S_{max})$ is updated, it is just constructing $\mathcal{T}(S_{max})$ from given S_{max}, instead of finding $\mathcal{T}(S_{max})$ from scratch.

Algorithm 3 preserves Property 1, as shown in Theorem 3.12, and has the same time complexity of Algorithm 2 in Theorem 3.10.

Theorem 3.12 (Accuracy Guarantee of Algorithm 3). *Algorithm 3 preserves Property 1. Thus, $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$ holds after Algorithm 3 terminates.*

Proof. *We assume that Property 1 holds and prove that it still holds after Algorithm 3 is executed. First, the ordering π remains to be a D-ordering in $\mathcal{T}$ by Lemma 3.11. Second, we show $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$. If the condition in line 4 of Algorithm 3 is met, $\mathcal{T}(S_{max})$ is set to the subtensor with the maximum density in $\{\mathcal{T}(Q_\pi, q) : q \in Q\}$ by Find-Slices(). By Lemma 3.3, $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$. If the condition is not met, $\rho(\mathcal{T}(S_{max}))$ remains the same, while ρ_{opt} never increases. Hence, $\rho(\mathcal{T}(S_{max})) \geq \rho_{opt}/N$ still holds by Property 1, which we assume. Since its two conditions are met, Property 1 is preserved.* ■

3.2.3 Increase or Decrease of Size. DenseStream also supports the increase and decrease of the size of the input tensor. The increase of the size of $\mathcal{T}$ corresponds to the addition of new slices to $\mathcal{T}$. For example, if the length of the nth mode of $\mathcal{T}$ increases from I_n to $I_n + 1$, the index $q = (n, I_n + 1)$ of the new slice is added to Q and the first position of π. We also set $d_\pi(q)$ and $c_\pi(q)$ to 0. Then, if there exist non-zero entries in the new slice, they are handled one by one by Algorithm 2. Likewise, when size decreases, we first handle the removed non-zero entries one by one by Algorithm 3. Then, we remove the indices of the removed slices from Q and π.

3.3 DenseAlert: Suddenly Emerging Dense-Subtensor Detection

Based on DenseStream, we propose DenseAlert, an incremental algorithm for detecting suddenly emerging dense subtensors. For a stream of increments in the input tensor $\mathcal{T}$, DenseAlert maintains

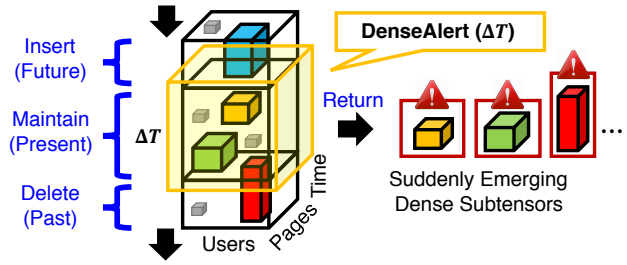


Figure 3: DenseAlert with Wikipedia Revision History (Example 2.1). DenseAlert (yellow box in the figure) spots dense subtensors formed within ΔT time units.

Algorithm 4 DenseAlert for sudden dense subtensors

Input: (1) sequence of increments in $\mathcal{T}$
(2) time window: ΔT
Output: suddenly emerging dense subtensors
1: run Algorithm 1 with a zero tensor
2: wait until the next change happens at time T
3: **if** the change is $((i_1, \ldots, i_N), \delta, +)$ **then**
4: run DenseStream (Algorithm 2)
5: schedule $((i_1, \ldots, i_N), \delta, -)$ at time $T + \Delta T$
6: **else if** the change is $((i_1, \ldots, i_N), \delta, -)$ **then**
7: run DenseStream (Algorithm 3)
8: report the current dense subtensor
9: **goto** Line 2

$\mathcal{T}_{\Delta T}$, a tensor where the value of each entry is the increment of the value of the corresponding entry in $\mathcal{T}$ in last ΔT time units (see Problem 2 in Section 2.3), as described in Figure 3 and Algorithm 4, To maintain $\mathcal{T}_{\Delta T}$ and a dense subtensor in it, DenseAlert applies increments by DenseStream (line 4), and undoes the increments after ΔT time units also by DenseStream (lines 5 and 7). The accuracy of DenseAlert, formalized in Theorem 3.13, is implied from the accuracy of DenseStream.

Theorem 3.13 (Accuracy Guarantee of Algorithm 4). *Let $\Delta\rho_{opt}$ be the density of the densest subtensor in the N-way tensor $\mathcal{T}_{\Delta T}$. The subtensor maintained by Algorithm 4 has density at least $\Delta\rho_{opt}/N$.*

Proof. By Theorems 3.9 and 3.12, DenseAlert, which uses DenseStream for updates, maintains a subtensor with density at least $1/N$ of the density of the densest subtensor. ■

The time complexity of DenseAlert is also obtained from Theorem 3.10 by simply replacing $\mathcal{T}$ with $\mathcal{T}_{\Delta T}$. DenseAlert needs to store only $\mathcal{T}_{\Delta T}$ (i.e., the changes in the last ΔT units) in memory at a time. DenseAlert discards older changes.

4 EXPERIMENTS

We design experiments to answer the following questions:

- **Q1. Speed**: How fast are updates in DenseStream compared to batch algorithms?
- **Q2. Accuracy**: How accurately does DenseStream maintain a dense subtensor?
- **Q3. Scalability**: How does the running time of DenseStream increase as input tensors grow?

Table 3: Summary of real-world tensor datasets.

Name	Size	$\lvert\Omega\rvert$	nnz($\mathcal{T}$)
Ratings: users × items × timestamps × ratings → #reviews			
Yelp [2]	552K × 77.1K × 3.80K × 5	633K	2.23M
Android [24]	1.32M × 61.3K × 1.28K × 5	1.39M	2.64M
YahooM. [12]	1.00M × 625K × 84.4K × 101	1.71M	253M
Wikipedia edit history: users × pages × timestamps → #edits			
KoWiki [32]	470K × 1.18M × 101K	1.80M	11.0M
EnWiki [32]	44.1M × 38.5M × 129K	82.8M	483M
Social networks: users × users × timestamps → #interactions			
Youtube [26]	3.22M × 3.22M × 203	6.45M	18.7M
SMS	1.25M × 7.00M × 4.39K	8.26M	103M
TCP dumps: IPs × IPs × timestamps → #connections			
TCP [22]	9.48K × 23.4K × 46.6K	79.5K	522K

- **Q4. Effectiveness**: Which anomalies or fraudsters does DenseAlert spot in real-world tensors?

4.1 Experimental Settings.

Machine: We ran all experiments on a machine with 2.67GHz Intel Xeon E7-8837 CPUs and 1TB memory (up to 85GB was used by our algorithms).
Data: Table 3 lists the real-world tensors used in our experiments. *Ratings* are 4-way tensors (users, items, timestamps, ratings) where entry values are the number of reviews. *Wikipedia edit history* is 3-way tensors (users, pages, timestamps) where entry values are the number of edits. *Social networks* are 3-way tensors (source users, destination users, timestamps) where entry values are the number of interactions. *TCP dumps* are 3-way tensors (source IPs, destination IPs, timestamps) where entry values are the number of TCP connections. Timestamps are in dates in Yelp and Youtube, in minutes in TCP, and in hours in the others.
Implementations: We implemented dense-subtensor detection algorithms for comparison. We implemented our algorithms, M-Zoom [32], and CrossSpot [18] in Java, while we used Tensor Toolbox [4] for CP decomposition (CPD)[1] and MAF [23]. In all the implementations, a sparse tensor format was used so that the space usage is proportional to the number of non-zero entries. As in [32], we used a variant of CrossSpot which maximizes the density measure defined in Definition 2.2 and uses CPD for seed selection. For each batch algorithm, we reported the densest one after finding three dense subtensors.

4.2 Q1. Speed

We show that updating a dense subtensor by DenseStream is significantly faster than running batch algorithms from scratch. For each tensor stream, we averaged the update times for processing the last 10,000 changes corresponding to increments (blue bars in Figure 5). Likewise, we averaged the update times for undoing the first 10,000 increments, i.e., decreasing the values of the oldest entries (red bars in Figure 5). Then, we compared them to the

[1] Let $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1 \times k}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{I_2 \times k}$, ..., $\mathbf{A}^{(N)} \in \mathbb{R}^{I_N \times k}$ be the factor matrices obtained by the rank-k CP Decomposition of $\mathcal{R}$. For each $j \in [k]$, we form a subtensor with every slice with index (n, i_n) where the (i_n, j)-th entry of $\mathbf{A}^{(n)}$ is at least $1/\sqrt{I_n}$.

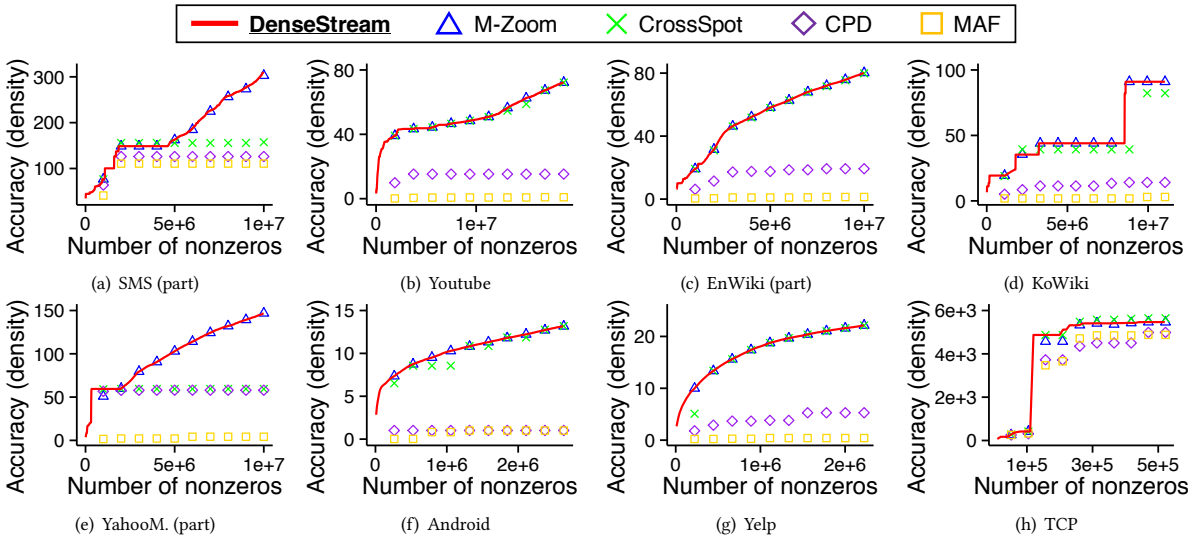


Figure 4: DenseStream is 'any-time' and accurate. While tensors grow, DenseStream maintains and instantly updates a dense subtensor, whereas batch algorithms update dense subtensors only once in a time interval. Subtensors maintained by DenseStream have density (red lines) similar to the density (points) of the subtensors found by the best batch algorithms.

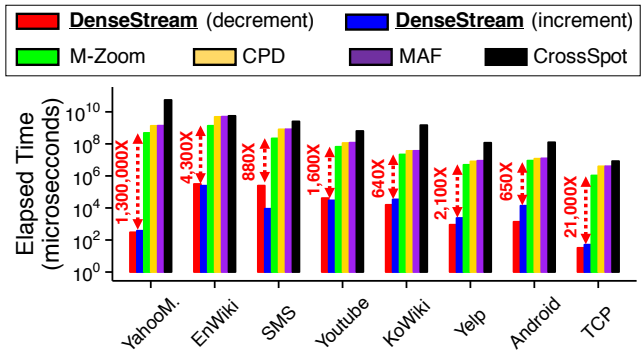


Figure 5: DenseStream outperforms batch algorithms. An update in DenseStream was up to a million times faster than the fastest batch algorithm.

time taken for running batch algorithms on the final tensor that each tensor stream results in. As seen in Figure 5, updates in DenseStream were up to a million times faster than the fastest batch algorithm. The speed-up was particularly high in sparse tensors having a widespread slice sum distribution (thus having a small reordered region R), as we can expect from Theorem 3.10.

On the other hand, the update time in DenseAlert, which uses DenseStream as a sub-procedure, was similar to that in DenseStream when the time interval $\Delta T = \infty$, and was less with smaller ΔT. This is since the average number of non-zero entries maintained is proportional to ΔT.

4.3 Q2. Accuracy

This experiment demonstrates the accuracy of DenseStream. From this, the accuracy of DenseAlert, which uses DenseStream as a sub-procedure, is also obtained. We tracked the density of the dense subtensor maintained by DenseStream while each tensor grows, and compared it to the densities of the dense subtensors found by batch algorithms. As seen in Figure 4, the subtensors that DenseStream maintained had density (red lines) similar to the density (points) of the subtensors found by the best batch algorithms. Moreover, DenseStream is 'any time'; that is, as seen in Figure 4, DenseStream updates the dense subtensor instantly, while the batch algorithms cannot update their dense subtensors until the next batch processes end. DenseStream also maintains a dense subtensor accurately when the values of entries decrease, as shown in Section C of the supplementary document [1].

The accuracy and speed (measured in Section 4.2) of the algorithms in Yelp Dataset are shown in Figure 1(a) in Section 1. DenseStream significantly reduces the time gap between the emergence and the detection of a dense subtensor, without losing accuracy.

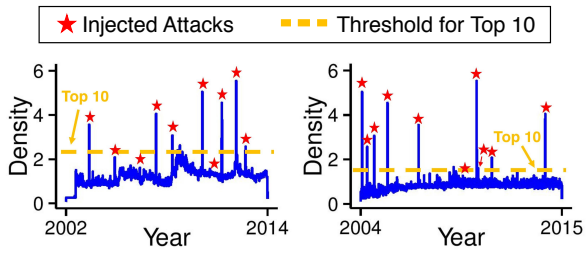
4.4 Q3. Scalability

We demonstrate the high scalability of DenseStream by measuring how rapidly its update time increases as a tensor grows. For this experiment, we used a $10^5 \times 10^5 \times 10^5$ random tensor stream that has a realistic power-law slice sum distribution in each mode. As seen in Figure 1(b) in Section 1, update times, for both types of changes, scaled sub-linearly with the number of nonzero entries. Note that DenseAlert, which uses DenseStream as a sub-procedure, has the same scalability.

4.5 Q4. Effectiveness

In this section, we show the effectiveness of DenseAlert in practice. We focus on DenseAlert, which spots suddenly emerging dense subtensors overlooked by existing methods, rather than DenseStream, which is much faster but eventually finds a similar subtensor with previous algorithms, especially [32].

4.5.1 **Small-scale Attack Detection in Ratings.** For rating datasets, where ground-truth labels are unavailable, we assume an attack scenario where fraudsters in a rating site, such as Yelp, utilize multiple user accounts and give the same rating to the same set of items (or businesses) in a short period of time. The goal



(a) DenseAlert in Yelp (left) and Android (right)

	Recall @ Top-10 in Yelp	Recall @ Top-10 in Android
DenseAlert	**0.9**	**0.7**
Others [16, 18, 23, 32, 33]	0.0	0.0

(b) Comparison with other anomaly detection algorithms

Figure 6: DenseAlert accurately detects small-scale short-period attacks injected in review datasets. However, these attacks are overlooked by existing methods.

of the fraudsters is to boost (or lower) the ratings of the items rapidly. This lockstep behavior results in a dense subtensor of size $\#fake\_accounts \times \#target\_items \times 1 \times 1$ in rating datasets whose modes are users, items, timestamps, and ratings. Here, we assume that fraudsters are not blatant but careful enough to adjust their behavior so that only small-scale dense subtensors are formed.

We injected 10 such small random dense subtensors of sizes from $3 \times 3 \times 1 \times 1$ to $12 \times 12 \times 1 \times 1$ in Yelp and Android datasets, and compared how many of them are detected by each anomaly-detection algorithm. As seen in Figure 6(a), DenseAlert (with ΔT=1 time unit in each dataset) clearly revealed the injected subtensors. Specifically, 9 and 7 among the top 10 densest subtensors spotted by DenseAlert indeed indicate the injected attacks in Yelp and Android datasets, respectively. However, the injected subtensors were not revealed when we simply investigated the number of ratings in each time unit. Moreover, as summarized in Figure 6(b), none of the injected subtensors was detected[2] by existing algorithms [16, 18, 23, 32]. These existing algorithms failed since they either ignore time information [16] or only find dense subtensors in the entire tensor [18, 23, 32, 33] without using a time window.

4.5.2 **Network Intrusion Detection.** Figure 1(c) shows the changes of the density of the maintained dense subtensor when we applied DenseAlert to TCP Dataset with the time window ΔT = 1 minute. We found out that the sudden emergence of dense subtensors (i.e., sudden increase in the density) indicates network attacks of various types. Especially, according to the ground-truth labels, all top 15 densest subtensors correspond to actual network attacks. Classifying each connection as an attack or a normal connection based on the density of the densest subtensor including the connection (i.e., the denser subtensor including a connection is, the more suspicious the connection is) led to high accuracy with AUC (Area Under the Curve) 0.924. This was better than MAF (0.514) and comparable with CPD (0.926), CrossSpot (0.923), and M-Zoom (0.921). The result is still noteworthy since DenseAlert requires only changes in the input tensor within ΔT time units at a time, while the others require the entire tensor at once.

4.5.3 **Anomaly Detection in Wikipedia.** The sudden appearances of dense subtensors also signal anomalies in Wikipedia edit history. Figure 7 depicts the changes of the density of the dense subtensor maintained by DenseAlert in KoWiki Dataset with the time window ΔT = 24 hours. We investigated the detected dense subtensors and found out that most of them corresponded to actual anomalies including edit wars, bot activities, and vandalism. For example, the densest subtensor, composed by three users and two pages, indicated an edit war where three users edited two pages about regional prejudice 900 times within a day.

5 RELATED WORK

Dense Subgraph Detection. For densest-subgraph detection in unweighted graphs, max-flow-based exact algorithms [15, 20] and greedy approximation algorithms [11, 20] have been proposed. Extensions include adding size bounds [3], using alternative metrics [35], finding subgraphs with limited overlap [7, 14], and extending to large-scale graphs [5, 6] and dynamic graphs [9, 13, 25]. Other approaches include spectral methods [27] and frequent itemset mining [29]. Dense-subgraph detection has been widely used to detect fraud or spam in social and review networks [8, 16, 19, 30, 31].

Dense Subtensor Detection. To incorporate additional dimensions and identify lockstep behavior with greater specificity, dense-subtensor detection in multi-aspect data (i.e., tensors) has been considered. Especially, a likelihood-based approach called CrossSpot [18] and a greedy approach giving an accuracy guarantee called M-Zoom [32] were proposed for this purpose. M-Zoom was also extended for large datasets stored on a disk or on a distributed file system [33]. Dense-subtensor detection has been used for network-intrusion detection [23, 32, 33], retweet-boosting detection [18], bot detection [32], rating-attack detection [33], genetics applications [28], and formal concept mining [10, 17]. However, these existing approaches assume a static tensor rather than a stream of events over time, and do not detect dense subtensors in real time, as they arrive. We also show their limitations in detecting dense subtensors small but highly concentrated in a short period of time.

Tensor Decomposition. Tensor decomposition such as HOSVD and CPD [21] can be used to find dense subtensors in tensors, as MAF [23] uses CPD for detecting anomalous subgraph patterns in heterogeneous networks. Streaming algorithms [34, 36] also have been developed for CPD and Tucker Decomposition. However, dense-subtensor detection based on tensor decomposition showed limited accuracy in our experiments (see Section 4.3).

6 CONCLUSION

In this work, we propose DenseStream, an incremental algorithm for detecting a dense subtensor in a tensor stream, and DenseAlert, an incremental algorithm for spotting the sudden appearances of dense subtensors. They have the following advantages:

[2] we consider that an injected subtensor is not detected by an algorithm if the subtensor is not included in the top 10 densest subtensors found by the algorithm or it is hidden in a dense subtensor of size at least 10 times larger than the injected subtensor.

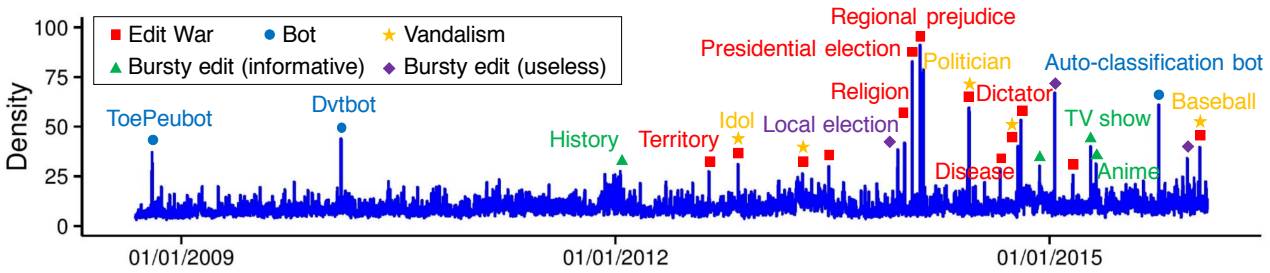


Figure 7: DenseAlert successfully spots anomalies in Korean Wikipedia. The sudden appearances of dense subtensors signal actual anomalies of various types including edit wars, bot activities, and vandalism. The densest subtensor indicates an edit war where three users edited two pages about regional prejudice 900 times within a day.

- **Fast and 'any time'**: our algorithms maintain and update a dense subtensor in a tensor stream, which is up to *a million times faster* than batch algorithms (Figure 5).
- **Provably accurate**: The densities of subtensors maintained by our algorithms have provable lower bounds (Theorems 3.9, 3.12, 3.13) and are high in practice (Figure 4).
- **Effective**: DenseAlert successfully detects anomalies, including small-scale attacks, which existing algorithms overlook, in real-world tensors (Figures 6 and 7).

Reproducibility: The code and data used in the paper are available at http://www.cs.cmu.edu/~kijungs/codes/alert.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1314632 and IIS-1408924. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. Kijung Shin is supported by KFAS Scholarship. Jisu Kim is supported by Samsung Scholarship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, or other funding parties. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] 2017. Supplementary Document. Available online: http://www.cs.cmu.edu/~kijungs/codes/alert/supple.pdf.
[2] 2017. Yelp Dataset Challenge. https://www.yelp.com/dataset_challenge. https://www.yelp.com/dataset_challenge
[3] Reid Andersen and Kumar Chellapilla. 2009. Finding dense subgraphs with size bounds. In *WAW*.
[4] Brett W. Bader, Tamara G. Kolda, et al. 2017. MATLAB Tensor Toolbox Version 2.6. Available online. (2017). http://www.sandia.gov/~tgkolda/TensorToolbox/
[5] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. 2014. Efficient primal-dual graph algorithms for mapreduce. In *WAW*.
[6] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *PVLDB* 5, 5 (2012), 454–465.
[7] Oana Denisa Balalau, Francesco Bonchi, TH Chan, Francesco Gullo, and Mauro Sozio. 2015. Finding subgraphs with maximum total density and limited overlap. In *WSDM*.
[8] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW*.
[9] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *STOC*.
[10] Loïc Cerf, Jérémy Besson, Céline Robardet, and Jean-François Boulicaut. 2008. Data Peeler: Contraint-Based Closed Pattern Mining in n-ary Relations.. In *SDM*.
[11] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*.
[12] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2012. The Yahoo! Music Dataset and KDD-Cup'11. In *KDD Cup*.
[13] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient densest subgraph computation in evolving graphs. In *WWW*.
[14] Esther Galbrun, Aristides Gionis, and Nikolaj Tatti. 2016. Top-k overlapping densest subgraphs. *Data Mining and Knowledge Discovery* (2016), 1–32.
[15] Andrew V Goldberg. 1984. *Finding a maximum density subgraph.* Technical Report.
[16] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. 2016. FRAUDAR: Bounding Graph Fraud in the Face of Camouflage. In *KDD*.
[17] Dmitry I Ignatov, Sergei O Kuznetsov, Jonas Poelmans, and Leonid E Zhukov. 2013. Can triconcepts become triclusters? *Int. J. General Systems* 42, 6 (2013), 572–593.
[18] Meng Jiang, Alex Beutel, Peng Cui, Bryan Hooi, Shiqiang Yang, and Christos Faloutsos. 2015. A general suspiciousness metric for dense blocks in multimodal data. In *ICDM*.
[19] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. 2014. Catchsync: catching synchronized behavior in large directed graphs. In *KDD*.
[20] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *ICALP*.
[21] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.
[22] Richard P Lippmann, David J Fried, Isaac Graf, Joshua W Haines, Kristopher R Kendall, David McClung, Dan Weber, Seth E Webster, Dan Wyschogrod, Robert K Cunningham, et al. 2000. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *DISCEX*.
[23] Koji Maruhashi, Fan Guo, and Christos Faloutsos. 2011. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*.
[24] Julian McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring networks of substitutable and complementary products. In *KDD*.
[25] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T Vu. 2015. Densest subgraph in dynamic graph streams. In *MFCS*.
[26] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC*.
[27] B Aditya Prakash, Mukund Seshadri, Ashwin Sridharan, Sridhar Machiraju, and Christos Faloutsos. 2010. Eigenspokes: Surprising patterns and community structure in large graphs. *PAKDD*.
[28] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense subgraphs with restrictions and applications to gene annotation graphs. In *RECOMB*.
[29] Jouni K Seppänen and Heikki Mannila. 2004. Dense itemsets. In *KDD*.
[30] Neil Shah, Alex Beutel, Brian Gallagher, and Christos Faloutsos. 2014. Spotting suspicious link behavior with fbox: An adversarial perspective. In *ICDM*.
[31] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis - Patterns, Anomalies and Algorithms. In *ICDM*.
[32] Kijung Shin, Bryan Hooi, and Christos Faloutsos. 2016. M-Zoom: Fast Dense-Block Detection in Tensors with Quality Guarantees. In *ECML/PKDD*.
[33] Kijung Shin, Bryan Hooi, Jisu Kim, and Christos Faloutsos. 2017. D-Cube: Dense-Block Detection in Terabyte-Scale Tensors. In *WSDM*.
[34] Jimeng Sun, Dacheng Tao, and Christos Faloutsos. 2006. Beyond streams and graphs: dynamic tensor analysis. In *KDD*.
[35] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *KDD*.
[36] Shuo Zhou, Nguyen Xuan Vinh, James Bailey, Yunzhe Jia, and Ian Davidson. 2016. Accelerating Online CP Decompositions for Higher Order Tensors. In *KDD*.