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 anomalous or even fraudulent behavior 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.

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,
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 the 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{T}$</td>
<td>an input tensor</td>
</tr>
<tr>
<td>$N$</td>
<td>order of $\mathcal{T}$</td>
</tr>
<tr>
<td>$(i_1, ..., i_N)$</td>
<td>entry of $\mathcal{T}$ with index $(i_1, ..., i_N)$</td>
</tr>
<tr>
<td>$Q$</td>
<td>set of the slice indices of $\mathcal{T}$</td>
</tr>
<tr>
<td>$q$</td>
<td>a member of $Q$</td>
</tr>
<tr>
<td>$\mathcal{T}(S)$</td>
<td>subtensor composed of the slices in $S \subset \mathcal{T}$</td>
</tr>
<tr>
<td>$\pi : [</td>
<td>Q</td>
</tr>
<tr>
<td>$Q_{\ell,q}$</td>
<td>slice indices located after equal to $q$ in $\pi$</td>
</tr>
<tr>
<td>$\text{sum}(\mathcal{T}(S))$</td>
<td>sum of the entries included in $\mathcal{T}(S)$</td>
</tr>
<tr>
<td>$d(\mathcal{T}(S), q)$</td>
<td>slice sum of $q$ in $\mathcal{T}(S)$</td>
</tr>
<tr>
<td>$c_S(q)$</td>
<td>cumulative max. slice sum of $q$ in $\mathcal{T}(Q_S,q)$</td>
</tr>
<tr>
<td>$((i_1, ..., i_N), \delta, +)$</td>
<td>increment of $i_1, ..., i_N$ by $\delta$</td>
</tr>
<tr>
<td>$((i_1, ..., i_N), \delta, -)$</td>
<td>decrement of $i_1, ..., i_N$ by $\delta$</td>
</tr>
<tr>
<td>$\rho(\mathcal{T}(S))$</td>
<td>density of a subtensor $\mathcal{T}(S)$</td>
</tr>
<tr>
<td>$\rho_{\text{opt}}$</td>
<td>density of the densest subtensor in $\mathcal{T}$</td>
</tr>
<tr>
<td>$\Delta t$</td>
<td>time window in DenseAlert</td>
</tr>
</tbody>
</table>

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 $I_1 \times \ldots \times I_N$ with non-negative entries. Each $(i_1, ..., i_N)$-th entry of $\mathcal{T}$ is denoted by $t_{i_1, ..., i_N}$. Equivalently, each $n$-mode index of $t_{i_1, ..., i_N}$ 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, $\mathcal{T}(1,i_2)$ 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 $j$th position. That is, each slice index $q \in Q$ is in the $\pi^{-1}(q)$-th position in $\pi$. Let $Q_{\ell,q} = \{r \in Q : \pi^{-1}(r) \geq \pi^{-1}(q)\}$ be the slice indices located after or equal to $q$ in $\pi$. Then, $\mathcal{T}(Q_{\ell,q})$ is the subtensor of $\mathcal{T}$ composed of the slices with their indices in $Q_{\ell,q}$.

Notations for Slice Sum: We denote the sum of the entries of $\mathcal{T}$ included in subtensor $\mathcal{T}(S)$ by $\text{sum}(\mathcal{T}(S))$. 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 $q$. For an ordering $\pi$ 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 $c_S(q) = \max\{d_{\pi}(r) : r \in Q, \pi^{-1}(r) \leq \pi^{-1}(q)\}$, i.e., maximum $d_{\pi}(\cdot)$ among the slice indices located before or equal to $q$ in $\pi$.

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 $t_{i_1, ..., i_N}$ by $\delta > 0$ and $((i_1, ..., i_N), \delta, -)$ be a decrement of entry $t_{i_1, ..., i_N}$ 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...
The density measure is chosen because: (a) it was successfully used for anomaly and fraud detection [32, 33], (b) this measure satisfies axioms that a reasonable "aloneness" 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 \( T \) with slice indices \( Q \) (i.e., a tensor stream) (2) maintain: a subtensor \( T(S) \) where \( S \subset Q \). (3) to maximize: its density \( \rho(T(S)) \).

Identifying the exact densest subtensor is computationally expensive even for a static tensor. For example, it takes \( O(|Q|^6) \) even when \( T \) 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 \( T \) whose values increase over time, let \( \mathcal{J}_{\Delta T} \) be the tensor where the value of each entry is the increment in the corresponding entry of \( T \) in the last \( \Delta T \) time units. Our aim is to spot dense subtensors appearing in \( \mathcal{J}_{\Delta T} \), which also keeps changing.

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

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 \( \pi \) 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 \( \pi \) is a D-ordering of \( Q \) in \( T \) if \( \forall q \in Q, d(T(\pi(q), q)) = \min_{r \in Q} d(T(\pi(q), q)). \)

This 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 \( \pi \), Algorithm 1 reduces the search space of \( 2^{|Q|} \) possible subtensors to \( \mathcal{F}(Q_{\pi}, q) : q \in Q \). 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 \( \mathcal{F}(S^*) \) be a subtensor with the maximum density, i.e., \( \rho(\mathcal{F}(S^*)) = \rho_{opt} \). Then for any \( q \in S^* \),
\[
d(\mathcal{F}(S^*), q) \geq \rho(\mathcal{F}(S^*)).
\]
(1)

Proof. The maximality of the density of \( \mathcal{F}(S^*) \) implies \( \rho(\mathcal{F}(S^*) - d(\mathcal{F}(S^*), q)) \leq \rho(\mathcal{F}(S^*)), \) and plugging in Definition 2.2 to \( \rho \) gives
\[
\frac{\sum(\mathcal{F}(S^*) - d(\mathcal{F}(S^*), q))}{|S^*| - 1} = \frac{\sum(\mathcal{F}(S^*)) - d(\mathcal{F}(S^*), q)}{|S^*| - 1} = \rho(\mathcal{F}(S^*)) = \frac{\sum(\mathcal{F}(S^*))}{|S^*|},
\]
which reduces to Eq. (1). \( \blacksquare \)

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

Proof. Let \( \mathcal{F}(S^*) \) be satisfying \( \rho(\mathcal{F}(S^*)) = \rho_{opt} \), and let \( q^* \in S^* \) be satisfying that \( \forall q \in S^*, \pi^{-1}(q^*) \leq \pi^{-1}(q) \). Our goal is to show \( \rho(\mathcal{F}(Q_{\pi}, q^*)) \geq \frac{1}{N}\rho(\mathcal{F}(S^*)) \), which we show as \( N\rho(\mathcal{F}(Q_{\pi}, q^*)) \geq \sum(\mathcal{F}(Q_{\pi}, q^*)) \geq d(\mathcal{F}(Q_{\pi}, q^*), q^*)$. To show \( N\rho(\mathcal{F}(Q_{\pi}, q^*)) \geq d(\mathcal{F}(Q_{\pi}, q^*), q^*)$, note \( N\rho(\mathcal{F}(Q_{\pi}, q^*)) = \sum(\mathcal{F}(Q_{\pi}, q^*))N, \) and since \( \mathcal{F} \) is an \( N \)-way tensor, each entry is included in \( N \) slices. Hence
\[
\sum_{q \in Q_{\pi}, q} d(\mathcal{F}(Q_{\pi}, q), q) = \sum(\mathcal{F}(Q_{\pi}, q))N.
\]
(2)

Since \( \pi \) is a D-ordering, \( \forall q \in Q_{\pi}, q^*, d(\mathcal{F}(Q_{\pi}, q)), q \geq d(\mathcal{F}(Q_{\pi}, q^*), q^*) \) holds. Combining this and Eq. (2) gives
\[
N\rho(\mathcal{F}(Q_{\pi}, q^*)) = \frac{\sum(\mathcal{F}(Q_{\pi}, q^*))}{|Q_{\pi}, q^*|} \geq \frac{\sum(\mathcal{F}(Q_{\pi}, q^*))}{|Q_{\pi}|} \geq d(\mathcal{F}(Q_{\pi}, q^*), q^*).
\]

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

Such a subtensor \( \mathcal{F}(S_{max}) \) is detected by Algorithm 1. That is, \( \mathcal{F}(S_{max}) \) 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 \( \rho_{opt}/N \).

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

The time complexity of Algorithm 1 is linear with \( \text{nnz}(\mathcal{F}) \), the number of the non-zero entries in \( \mathcal{F} \), as formalized in Theorem 3.6.

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 \( \mathcal{F} \) that keeps changing, the ordering \( \pi \) of the slice indices and the dense subtensor \( \rho(\mathcal{F}(S_{max})) \) maintained by DENSESTREAM satisfy the following two conditions:

- \( \pi \) is a D-ordering of \( Q \) in \( \mathcal{F} \) or \( \rho(\mathcal{F}(S_{max})) \geq \rho_{opt}/N \).

3.2.1 **Increment of Entry Values.** Assume that the subtensor \( \mathcal{F}(S_{max}) \) and ordering \( \pi \) (with \( d_{\pi}(\cdot) \) and \( c_{\pi}(\cdot) \)) satisfy Property 1 in the current tensor \( \mathcal{F} \) (such \( \pi, d_{\pi}(\cdot), c_{\pi}(\cdot), \) and \( \mathcal{F}(S_{max}) \) 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: \( \mathcal{T}(\text{S}_{\text{max}}) \) with Property 1
(3) current D-ordering: \( \pi(i) \) (also \( d_x(i) \) and \( c_x(i) \))
(4) a change in \( \mathcal{T} \): (\( i_1, \ldots, i_N, \delta, + \))

**Output:** updated dense subtensor \( \mathcal{T}(\text{S}_{\text{max}}) \)

1. \( i_1, \ldots, i_N \leftarrow i_1, \ldots, i_N + \delta \)
2. compute \( j_L \) and \( j_H \) by Eq. (3) and Eq. (4) \( \triangleright \) [Find-Region]
3. compute \( R \) by Eq. (5) \( \triangleright \) [Reorder]
4. \( S \leftarrow \{ q \in Q : \pi^{-1}(q) \notin [j_L, j_H] \}; \quad RS \leftarrow R \cap S \)
5. \( c_{\text{max}} \leftarrow 0 \)
6. if \( j_L > 1 \) then \( c_{\text{max}} \leftarrow \arg \min_{q \in \{ q \}} d(T(S), q) \)
7. for \( j = j_L \) to \( j_H \) do
8. \( q \leftarrow \arg \min_{r \in 
(\mathcal{T}(S), q)} d(T(S), q) \) \( \triangleright \) \( q \) has min. slice sum
9. \( \pi(j) \leftarrow q \triangleright \) by Lemma 3.7, \( S = Q \cap q_q; \quad RS = R \cap \{q \} \)
10. \( d_{\pi} = d(T(S), q) \triangleright \) \( d_{\pi}(q) = d(T(S), q) \)
11. \( c_{\pi} = \max(c_{\text{max}}, d_{\pi}(q)); \quad c_{\pi} \leftarrow \arg \min_{q \in \{ q \}} d_{\pi}(q) \)
12. \( S \leftarrow S \cap \{q \}; \quad RS \leftarrow R \cap \{q \} \)
13. if \( c_{\pi} \geq \pi(T(\text{S}_{\text{max}})) \) then \( \triangleright \) [Update-Subtensor]
14. \( S \leftarrow \text{Find-Slices}() \) in Algorithm 1 \( \triangleright \) time complexity: \( O(|Q|) \)
15. if \( \text{S}_{\text{max}} \neq S \) then \( \mathcal{T}(\text{S}_{\text{max}}) \leftarrow \mathcal{T}(S) \)
16. return \( \mathcal{T}(\text{S}_{\text{max}}) \)

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

**Lemma 3.7.** Let \( \pi \) be a D-ordering in \( \mathcal{T} \), and let \( \pi' \) be a D-ordering with a different change \( ((i_1, \ldots, i_N), \delta, +) \). For \( R \) (Eq. (5)) defined on \( j_L \) and \( j_H \) (Eq. (3) and Eq. (4)), let \( \pi' \) be an ordering of slice indices \( \pi' \) which \( \forall v \in [j_L, j_H], \pi'(v) = \pi(v) \) and \( \forall v \notin [j_L, j_H], \pi'(v) \) holds. Then, \( \pi' \) 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 \( \mathcal{T}(\text{S}_{\text{max}}) \) when two conditions are met. We first check \( c_{\text{max}} \geq \pi(T(\text{S}_{\text{max}})) \), which takes \( O(1) \) if we maintain \( \pi(T(\text{S}_{\text{max}})) \), since \( c_{\text{max}} < \pi(T(\text{S}_{\text{max}})) \) entails that the updated entry \( i_1, \ldots, i_N \) is not in the densest subtensor (see the proof of Theorem 3.9 for details). We then check if there are changes in \( \text{S}_{\text{max}} \) obtained by \( \text{Find-Slices}() \). This takes only \( O(|Q|) \), as shown in Section 3.6. Even if both conditions are met, updating \( \mathcal{T}(\text{S}_{\text{max}}) \) is simply to construct \( \mathcal{T}(\text{S}_{\text{max}}) \) from given \( \text{S}_{\text{max}} \) instead of finding \( \mathcal{T}(\text{S}_{\text{max}}) \) from scratch. This conditional update reduces computation but still preserves Property 1, as formalized in Lemma 3.8 and Theorem 3.9.

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

**Proof.** Let \( \mathcal{T}(S) \) be a subtensor with the maximum density, i.e., \( \pi(T(S')) = \rho_{\text{opt}} \). Let \( q' \in S^c \) be satisfying that \( \forall q \in S, \pi^{-1}(q') \leq \pi^{-1}(q) \). For any entry \( i_1, \ldots, i_N \in \mathcal{T}(S') \) with index \( i_1, \ldots, i_N \) and any \( q \in \{i_1, \ldots, i_N \} \), our goal is to show \( \pi_{\text{c}}(q) \geq \pi(T(S')) \), which we show as \( \pi_{\text{c}}(q') \geq \pi(T(S'), q') \geq \rho(T(S')) \).

First, \( \pi_{\text{c}}(q') \geq \pi_{\text{c}}(q') \) is from the definition of \( \pi_{\text{c}}(q) \) and \( \pi^{-1}(q') \leq \pi^{-1}(q) \). Second, from \( S^c \subset Q_{\text{c}}, d_{\mathcal{T}'}(q') \geq d(T(Q_{\text{c}}, q'), q') \geq d(T(S'), q') \) holds. Third, \( d(T(S'), q') \geq \rho(T(S')) \) is from Theorem 3.2. From these, \( \pi_{\text{c}}(q) \geq \pi(T(S')) \) holds.

**Theorem 3.9 (Accuracy Guarantee of Algorithm 2).** Algorithm 2 preserves Property 1, and thus \( \pi(T(\text{S}_{\text{max}})) \geq \rho_{\text{opt}} / N \) 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 \( \pi \) remains to be a D-ordering in \( \mathcal{T} \) by Lemma 3.7. Second, we show \( \pi(T(\text{S}_{\text{max}})) \geq \rho_{\text{opt}} / N \). If the condition in line 13 of Algorithm 2 is met, \( \mathcal{T}(\text{S}_{\text{max}}) \) is set to the subtensor with the maximum density in \( \{Q_{\text{r}}, q \in Q \} \) by \( \text{Find-Slices}() \). By Lemma 3.3, \( \pi(T(\text{S}_{\text{max}})) \geq \rho_{\text{opt}} / N \). If the condition in line 13 is not met, for the changed entry \( i_1, \ldots, i_N \), we let \( \Delta_{\text{L}} \), the density of \( \Delta_{\text{R}} \), be defined by \( (i_1, \ldots, i_N) \) such that \( \pi(\Delta_{\text{L}}) = (n, i_1) \). Since \( \Delta_{\text{L}} \leq \Delta_{\text{R}} \), \( c_{\text{c}}(n, i_1) = c_{\text{c}}(\pi(\Delta_{\text{L}})) \leq c_{\text{c}}(\pi(\Delta_{\text{R}})) = c_{\text{c}} < \rho(T(\text{S}_{\text{max}})) \leq \rho_{\text{opt}} \). Then, by Lemma 3.8, \( i_1, \ldots, i_N \) does not belong to the densest subtensor, which thus remains the same after the change \( (i_1, \ldots, i_N), \delta, + \). Since \( \rho(T(\text{S}_{\text{max}})) \) never decreases, \( \rho(T(\text{S}_{\text{max}})) \geq \rho_{\text{opt}} / N \) 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 \( O(|Q| \log |Q| + nzz(\mathcal{T})) \), which is the time complexity of Algorithm 1. In practice, however, \( R \) is much smaller than \( Q \), and updating \( S_{\text{max}} \) happens rarely. Thus, in our experiments, Algorithm 2 scaled sub-linearly with \( nzz(\mathcal{T}) \) (see Section 4.4).
Theorem 3.10 (Time Complexity of Algorithms 2 and 3). Let $T(q)$ be the set of the non-zero entries in the slice with index $q$ in $T$. The time complexity of Algorithms 2 and 3 is $O(|R| \log |R| + |Q| + N \sum_{q \in R} T(q)) + O(|R| \sum_{q \in T(q)})$.

Proof. Assume that, for each slice, the list of the non-zero entries in the slice is stored, and let $q_f = \pi(j)$. 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 $T(q_{\pi,q_f})$ takes $O(|N| \cup_{q \in R} T(q))$. 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 $T(q_{\pi,q_f})$ takes $O(|R|)$. Popping the index of a slice with minimum slice sum, which takes $O(|R| \log |R|)$, happens $|R|$ times, and thus we get $O(|R| \log |R|)$. Whenever a slice index is popped, we have to update the slice sum 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 $O(|N| \cup_{q \in R} T(q))$ times, and thus we get $O(|N| \cup_{q \in R} T(q))$. On the other hand, by Lemma 3.5, Find-Slices() and constructing $T(S_{\max})$ from $S_{\max}$ take $O(|N| \cup_{q \in S_{\max}} T(q))$. Hence, the time complexity of Algorithms 2 and 3 is the sum of all the costs, which is $O(|R| \log |R| + |Q| + N \sum_{q \in R} T(q)) + O(|N| \cup_{q \in T(q)})$.

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

For a change $((i_1, \ldots, i_N), \delta, \cdots)$, we find the region $[J_L, J_H]$ of the domain of $\pi$ that may need to be reordered. Let $C = \{(n,i) : n \in [N]\}$ be the indices of the slices composing the changed entry $i_1, \ldots, i_N$ and let $q_f = \arg\min_{q \in C} \pi^{-1}(q)$ be the one located first in $\pi$ among $C$. Then, let $M_{\min} = \{q \in Q : d_{\pi}(q) > c_{\pi}(q) - \delta\}$ 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:

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

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

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

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

Algorithm 3 DenseStream in the case of decrement

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

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

is a D-ordering in $T'$.

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

The last step of Algorithm 3 is to conditionally and rapidly update the maintained dense subtensor $T(S_{\max})$ using $\pi$. The subtensor $T(S_{\max})$ is updated if entry $i_1, \ldots, i_N$ belongs to $T(S_{\max})$ (i.e., if $\rho(T(S_{\max}))$ decreases by the change $((i_1, \ldots, i_N), \delta, \cdots)$) and there are changes in $S_{\max}$, obtained by Find-Slices(). Checking these conditions takes only $O(|Q|)$, as in the incremental case. Even if $T(S_{\max})$ is updated, it is just constructing $T(S_{\max})$ from given $S_{\max}$, instead of finding $T(S_{\max})$ 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, $\rho(T(S_{\max})) \geq \rho_{opt}/N$ 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 $\pi$ remains to be a D-ordering in $T$ by Lemma 3.11. Second, we show $\rho(T(S_{\max})) \geq \rho_{opt}/N$. If the condition in line 4 of Algorithm 3 is met, $T(S_{\max})$ is set to the subtensor with the maximum density in $T(\{Q_{\pi, q} : q \in Q\}$ by Find-Slices(). By Lemma 3.3, $\rho(T(S_{\max})) \geq \rho_{opt}/N$. If the condition is not met, $\rho(T(S_{\max}))$ remains the same, while $\rho_{opt}$ never increases. Hence, $\rho(T(S_{\max})) \geq \rho_{opt}/N$ 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 $T$ corresponds to the addition of new slices to $T$. For example, if the length of the $n$th mode of $T$ 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 $\pi$. 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 $\pi$.

3.3 DenseAlert: Suddenly Emerging Dense-Subtensor Detection

Based on DenseStream, we propose DenseAlert, an incremental algorithm for detecting suddenly emerging dense subtenors. For a stream of increments in the input tensor $T$, DenseAlert maintains
Figure 3: DenseAlert with Wikipedia Revision History (Example 2.1). DenseAlert (yellow box in the figure) spots dense subtensors formed within $\Delta T$ time units.

Algorithm 4 DenseAlert for sudden dense subtensors

**Input:** (1) sequence of increments in $T$

(2) time window: $\Delta T$

**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 $((i_1, ..., i_k), \delta, +)$ then
4. run DenseStream (Algorithm 2)
5. schedule $((i_1, ..., i_k), \delta, -)$ at time $T + \Delta T$
6. else if the change is $((i_1, ..., i_k), \delta, -)$ then
7. run DenseStream (Algorithm 3)
8. report the current dense subtensor
9. goto Line 2

$T_{\Delta T}$, a tensor where the value of each entry is the increment of the value of the corresponding entry in $T$ in last $\Delta T$ time units (see Problem 2 in Section 2.3), as described in Figure 3 and Algorithm 4. To maintain $T_{\Delta T}$ and a dense subtensor in it, DenseAlert applies increments by DenseStream (line 4), and undoes the increments after $\Delta T$ 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 $\delta_{\text{opt}}$ be the density of the densest subtensor in the $N$-way tensor $T_{\Delta T}$. The subtensor maintained by Algorithm 4 has density at least $\delta_{\text{opt}}/N$.

**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 $T$ with $T_{\Delta T}$. DenseAlert needs to store only $T_{\Delta T}$ (i.e., the changes in the last $\Delta T$ 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?
- **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 undoing the last 10,000 changes corresponding to increments (blue bars in Figure 5). Likewise, we averaged the update times for processing 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

\[ A^{(i)} \in \mathbb{R}^{I_i \times J_i \times K_i}, \quad A^{(j)} \in \mathbb{R}^{I_j \times J_j \times K_j}, \ldots, A^{(N)} \in \mathbb{R}^{I_N \times J_N \times K_N} \]

be the factor matrices obtained by the rank-$k$ CP Decomposition of $\mathcal{X}$. 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 $A^{(n)}$ is at least $1/\sqrt{k}$.
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 Dense-
Stream when the time interval $\Delta T = \infty$, and was less with smaller 
$\Delta T$. This is since the average number of non-zero entries maintained 
is proportional to $\Delta T$.

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 
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 algo-
rithms in Yelp Dataset are shown in Figure 1(a) in Section 1. Den-
seStream 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 prac-
tice. We focus on DenseAlert, which spots suddenly emerging 
dense subtensors overlooked by existing methods, rather than Den-
seStream, which is much faster but eventually finds a similar sub-
tensor 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, 
use multiple user accounts and give the same rating to the same 
set of items (or businesses) in a short period of time. The goal 

![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.](image)

![Figure 5: DenseStream outperforms batch algorithms. An update in DenseStream was up to a million times faster than the fastest batch algorithm.](image)
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 sub-tensors are formed.

We injected 10 such small random dense sub-tensors 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 $\Delta T=1$ time unit in each dataset) clearly revealed the injected sub-tensors. Specifically, 9 and 7 among the top 10 densest sub-tensors spotted by DENSEALERT indeed indicate the injected attacks in Yelp and Android datasets, respectively. However, the injected sub-tensors 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 sub-tensors 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 sub-tensors in the entire tensor [18, 23, 32, 33] without using a time window.

\(^2\)We consider that an injected sub-tensor is not detected by an algorithm if the sub-tensor is not included in the top 10 densest sub-tensors found by the algorithm or if it is hidden in a dense sub-tensor of size at least 10 times larger than the injected sub-tensor.

\[ \text{Recall} @ \text{Top-10} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Yelp Recall @ Top-10</th>
<th>Android Recall @ Top-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>DENSEALERT</td>
<td>0.9</td>
<td>0.7</td>
</tr>
<tr>
<td>Others [16, 18, 23, 32, 33]</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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

\[ \text{Area Under the Curve} \text{ (AUC)} 0.924. \text{ This was better than MAF (0.514) and comparable with CPD (0.926), CrossSpot (0.923), and M-Zoom (0.921). The result is still noteworthy since DENSEALERT requires only changes in the input tensor within $\Delta T$ time units at a time, while the others require the entire tensor at once.} \]

4.5.3 Anomaly Detection in Wikipedia. The sudden appearances of dense sub-tensors 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 $\Delta T = 24$ hours. We investigated the detected dense sub-tensors and found out that most of them corresponded to actual anomalies including edit wars, bot activities, and vandalism. For example, the densest sub-tensor, 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 sub-tensors in real time, as they arrive. We also show their limitations in detecting dense sub-tensors 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 sub-tensors 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 sub-tensor in a tensor stream, and DENSEALERT, an incremental algorithm for spotting the sudden appearances of dense sub-tensors. They have the following advantages:
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 successfullly 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. Jiwoo 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