D-Cube: Dense-Block Detection in Terabyte-Scale Tensors

Kijung Shin[1], Bryan Hooi[2], Jisu Kim[2], Christos Faloutsos[1]
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA, USA
[1]{kijungs, christos}@cs.cmu.edu, [2]{bhooi, jisuk1}@andrew.cmu.edu

ABSTRACT

How can we detect fraudulent lockstep behavior in large-scale multi-aspect data (i.e., tensors)? Can we detect it when data are too large to fit in memory or even on a disk? Past studies have shown that dense blocks in real-world tensors (e.g., social media, Wikipedia, TCP dumps, etc.) signal anomalous or fraudulent behavior such as retweet boosting, bot activities, and network attacks. Thus, various approaches, including tensor decomposition and search, have been used for rapid and accurate dense-block detection in tensors. However, all such methods have low accuracy, or assume that tensors are small enough to fit in main memory, which is not true in many real-world applications such as social media and web.

To overcome these limitations, we propose D-Cube, a disk-based dense-block detection method, which also can be run in a distributed manner across multiple machines. Compared with state-of-the-art methods, D-Cube is (1) **Memory Efficient**: requires up to *1,600× less memory* and handles *1,000× larger* data (*2.6TB*), (2) **Fast**: up to *5× faster* due to its near-linear scalability with all aspects of data, (3) **Provably Accurate**: gives a guarantee on the densities of the blocks it finds, and (4) **Effective**: successfully spotted network attacks from TCP dumps and synchronized behavior in rating data with the highest accuracy.

Keywords

Dense-Block Detection, Anomaly/Fraud Detection, Tensor

CCS Concepts

•**Information systems** → **Data mining;**

1. INTRODUCTION

Given a tensor which is too large to fit in memory, how can we find dense blocks in it? A common application of this problem is review fraud detection, where we want to detect suspicious lockstep behavior among groups of fraudulent user accounts who review suspiciously similar sets of products. Typically, for each review, we also have a number of additional dimensions such as timestamp, review length, number of stars, review keywords, etc.

Previous work [38, 28, 20] has shown the benefit of incorporating all this information by modeling the data as a tensor. Tensors allow us to consider additional dimensions such as time, in order to identify dense regions of interest more accurately and specifically. Extraordinary dense blocks in the tensor correspond to groups of users with lockstep behaviors both in the products they review and along the additional dimensions (for example, multiple users reviewing the same products at the exact same time). Dense-block detection in tensors has also been found effective for network intrusion detection [38, 28], retweet boosting detection [20], bot activities detection [38], and genetics applications [35, 28].

Based on these facts, several approaches have been developed for rapid and accurate dense-block detection in tensors. One approach is to use tensor decomposition, such as CP Decomposition and HOSVD [28]. Such methods based on tensor decomposition, however, are outperformed by search-based methods [38, 20] in terms of accuracy and flexibility with regard to the choice of density metric. Especially, the latest search method [38] also provides a guarantee on the densities of the blocks it finds.

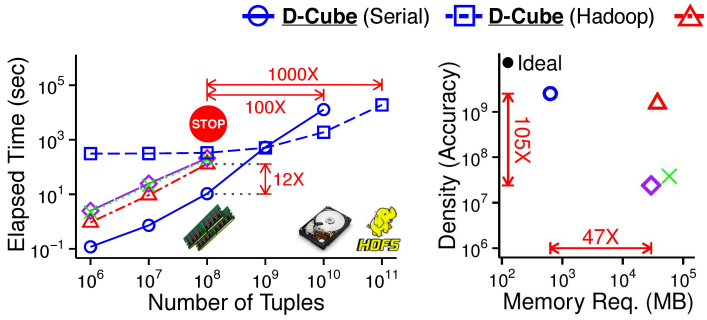
However, all existing search-based methods for dense-block detection in tensors assume that tensors are small enough to fit in memory. Moreover, they are not directly applicable to tensors stored in disk since using them for such tensors incurs too much disk I/O cost due to their highly iterative nature. However, real-world applications, such as social media and web, often involve disk-resident tensors with terabytes or even petabytes, which in-memory algorithms cannot handle. This leaves a growing gap that needs to be filled.

Table 1: Comparison between D-Cube and state-of-the-art dense-block detection methods. ✓represents 'supported'.

	M-Zoom [38]	CrossSpot [20]	MAF [28]	Fraudar [19]	**D-Cube**
High-order Data	✓	✓	✓		✓
Flexibility in Density Measures	✓	✓		✓	✓
Accuracy Guarantees	✓			✓	✓
Out-of-core Computation					✓
Distributed Computation					✓

WSDM 2017, February 06-10, 2017, Cambridge, United Kingdom

DOI: http://dx.doi.org/10.1145/3018661.3018676



(a) Data scalability

(b) Accuracy and Memory Requirements

Dataset	Order	Volume	Mass	Attack Ratio	Attack Type
DARPA	1	738	1.52M	**100%**	Neptune
	2	522	614K	**100%**	Neptune
	3	402	113K	**100%**	Smurf
	4	1	10.8K	**100%**	Satan
	5	156K	560K	30.4%	SNMP
AirForce	1	1	1.93M	**100%**	Smurf
	2	8	2.53M	**100%**	Smurf
	3	6,160	897K	**100%**	Neptune
	4	63.5K	1.02M	94.7%	Neptune
	5	930K	1.00M	94.7%	Neptune

(c) Network intrusion detection using D-CUBE (Densest blocks detected by D-CUBE in TCP dumps)

Figure 1: **D-Cube outperforms its state-of-the-art competitors in all aspects.** The red stop sign denotes 'out of memory'. (a) **Fast & Scalable**: D-CUBE was *12× faster* and successfully handled *1,000× larger data (2.6TB)* than its competitors, (b) **Efficient & Accurate**: D-CUBE required *47× less memory*, and found denser blocks than its competitors from English Wikipedia revision history, (c) **Effective**: D-CUBE successfully spotted dense blocks consisting of network attacks, from TCP dumps, with high accuracy. See Section 4 for the detailed experimental settings.

To overcome these limitations, we propose D-CUBE, a dense-block detection method for disk-resident tensors. D-CUBE works under the W-Stream Model [34], where data are only sequentially read and written during computation. As seen in Table 1, only D-CUBE supports disk-resident (or out-of-core) computation, allowing it to process data too large to fit in main memory. D-CUBE is optimized for this setting by carefully minimizing the amount of disk I/O and the number of steps requiring disk accesses, without losing accuracy guarantees it provides. Moreover, we provide a distributed version of D-CUBE using the MAPREDUCE framework [15], specifically its open source implementation, HADOOP [1].

The main characteristics of D-CUBE are as follows:

- **Memory Efficient:** D-CUBE requires up to *1,600×* less memory and successfully handles *1,000×* larger data (*2.6TB*) than in-memory algorithms (Figure 1(a)).
- **Fast:** D-CUBE detects dense blocks up to *5×* faster in real-world tensors and *12* × faster in synthetic tensors than state-of-the-art methods due to its near-linear scalability with all aspects of tensors (Figure 1(a)).
- **Provably Accurate:** D-CUBE provides a guarantee on the densities of the blocks it finds (Theorem 3), and shows high accuracy similar to state-of-the-art methods on real-world tensors (Figure 1(b)).
- **Effective:** D-CUBE successfully spotted network attacks from TCP dumps, and lockstep behavior in rating data, with the highest accuracy (Figure 1(c)).

Reproducibility: Our open source code and the data used are at **http://www.cs.cmu.edu/~kijungs/codes/dcube/**.

In Section 2, we provide notations and a formal problem definition. In Section 3, we propose D-CUBE, a disk-based dense-block detection method. In Section 4, we present experimental results. After discussing related work in Section 5, we offer conclusions in Section 6.

2. NOTATIONS AND DEFINITIONS

In this section, we introduce notations and concepts used in the paper. After defining density measures, we also give a formal definition of the dense-block detection problem.

2.1 Notations and Concepts

Table 2 lists the symbols frequently used in the paper. We use $[x] = \{1, 2..., x\}$ for brevity. Let $\mathcal{R}(A_1, ..., A_N, X)$ be a relation with N dimension attributes, denoted by A_1, ..., A_N, and a nonnegative measure attribute, denoted by X (see Example 1 for a running example). For each tuple $t \in \mathcal{R}$ and for each $n \in [N]$, $t[A_n]$ and $t[X]$ are used to denote the values of A_n and X, resp., in t. For each $n \in [N]$, we use $\mathcal{R}_n = \{t[A_n] : t \in \mathcal{R}\}$ to denote the set of distinct values of A_n in $\mathcal{R}$. The relation $\mathcal{R}$ is naturally represented as an N-way tensor of size $|\mathcal{R}_1| \times ... \times |\mathcal{R}_N|$. The value of each entry in the tensor is $t[X]$, if the corresponding tuple t exists, and 0 otherwise. Let $\mathcal{B}_n$ be a subset of $\mathcal{R}_n$. Then, a *block* $\mathcal{B}$ in $\mathcal{R}$ is defined as $\mathcal{B}(A_1, ..., A_N, X) = \{t \in \mathcal{R} : \forall n \in [N], t[A_n] \in \mathcal{B}_n\}$, the set of tuples where each attribute A_n has a value in $\mathcal{B}_n$. The relation $\mathcal{B}$ is called a 'block' because it forms a subtensor of size $|\mathcal{B}_1| \times ... \times |\mathcal{B}_N|$ in the tensor representation of $\mathcal{R}$, as in Figure 2(b). We define the mass of $\mathcal{R}$ as $M_{\mathcal{R}} = \sum_{t \in \mathcal{R}} t[X]$, the sum of attribute X in the tuples of $\mathcal{R}$. We denote the set of tuples of $\mathcal{B}$ whose attribute $A_n = a$ by $\mathcal{B}(a, n) = \{t \in \mathcal{B} : t[A_n] = a\}$ and its mass, called *attribute-value mass of a in A_n*, by $M_{\mathcal{B}(a,n)} = \sum_{t \in \mathcal{B}(a,n)} t[X]$.

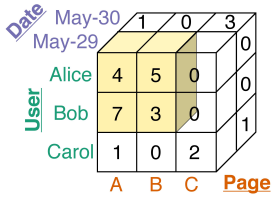
Table 2: Table of symbols.

Symbol	Definition
$\mathcal{R}(A_1, ..., A_N, X)$	relation representing an N-way tensor
N	number of dimension attributes in $\mathcal{R}$
A_n	n-th dimension attribute in $\mathcal{R}$
X	measure attribute in $\mathcal{R}$
$t[A_n]$ (or $t[X]$)	value of attribute A_n (or X) in tuple t in $\mathcal{R}$
$\mathcal{B}$	a block in $\mathcal{R}$
$\rho(\mathcal{B}, \mathcal{R})$	density of block $\mathcal{B}$ in $\mathcal{R}$
$\mathcal{R}_n$ (or $\mathcal{B}_n$)	set of distinct values of A_n in $\mathcal{R}$ (or $\mathcal{B}$)
$M_{\mathcal{R}}$ (or $M_{\mathcal{B}}$)	mass of $\mathcal{R}$ (or $\mathcal{B}$)
$\mathcal{B}(a, n)$	set of tuples with attribute $A_n = a$ in $\mathcal{B}$
$M_{\mathcal{B}(a,n)}$	attribute-value mass of a in A_n
k	number of blocks we aim to find
$[x]$	$\{1, 2..., x\}$

Example 1 (Wikipedia Revision History). *As in Figure 2, assume a relation $\mathcal{R}$(user, page, date, count), where each tuple (u, p, d, c) in $\mathcal{R}$ indicates that user u revised page p on date d, c times. The first three attributes, A_1=user, A_2=page, and A_3=date, are dimension attributes, and the other one, X=count, is the measure attribute. Let $\mathcal{B}_1$={Alice, Bob}, $\mathcal{B}_2$={A, B}, and $\mathcal{B}_3$={May-29}. Then, $\mathcal{B}$ is the set of tuples regarding the revision of page A or B by Alice or Bob on May-29, and its mass $M_{\mathcal{B}}$ is 19, the total number of such re-*

User	Page	Date	Count
Alice	A	May-29	4
Alice	B	May-29	5
Bob	A	May-29	7
Bob	B	May-29	3
Carol	C	May-30	1
⋮	⋮	⋮	⋮

(a) Relation $\mathcal{R}$



(b) Tensor Representation of $\mathcal{R}$

Figure 2: Pictorial description of Example 1. (a) Relation $\mathcal{R}$. The colored tuples compose block $\mathcal{B}$. (b) Tensor representation of $\mathcal{R}$. The block $\mathcal{B}$ forms a subtensor of $\mathcal{R}$.

visions. The attribute-value mass of Alice (i.e., $M_{\mathcal{B}(Alice,1)}$) is 9, the number of revisions on A or B by exactly Alice on May-29. In the tensor representation, $\mathcal{B}$ composes a subtensor in $\mathcal{R}$, as depicted in Figure 2(b).

2.2 Density Measures

We present density measures proven useful for anomaly detection in past studies. We use them throughout the paper although our dense-block detection method, explained in Section 3, is flexible and not restricted to specific measures.

Arithmetic Average Mass (Definition 1) and Geometric Average Mass (Definition 2), which were used for detecting network intrusions and bot activities [38], are the extensions of density measures widely-used for graphs [14, 23].

Definition 1 (Arithmetic Average Mass ρ_{ari} [38]). *The arithmetic average mass of a block $\mathcal{B}$ in a relation $\mathcal{R}$ is*

$$\rho_{ari}(\mathcal{B},\mathcal{R})=\rho_{ari}(M_{\mathcal{B}},\{|\mathcal{B}_n|\}_{n=1}^N,M_{\mathcal{R}},\{|\mathcal{R}_n|\}_{n=1}^N)=\frac{M_{\mathcal{B}}}{\frac{1}{N}\sum_{n=1}^N|\mathcal{B}_n|}.$$

Definition 2 (Geometric Average Mass ρ_{geo} [38]). *The geometric average mass of a block $\mathcal{B}$ in a relation $\mathcal{R}$ is*

$$\rho_{geo}(\mathcal{B},\mathcal{R})=\rho_{geo}(M_{\mathcal{B}},\{|\mathcal{B}_n|\}_{n=1}^N,M_{\mathcal{R}},\{|\mathcal{R}_n|\}_{n=1}^N)=\frac{M_{\mathcal{B}}}{(\prod_{n=1}^N|\mathcal{B}_n|)^{\frac{1}{N}}}.$$

Suspiciousness (Definition 3), which was used for detecting 'retweet-boosting' activities [21], is the negative log-likelihood that $\mathcal{B}$ has mass $M_{\mathcal{B}}$ under the assumption that each entry of $\mathcal{R}$ is i.i.d. from a Poisson distribution.

Definition 3 (Suspiciousness ρ_{susp} [20]). *The suspiciousness of a block $\mathcal{B}$ in a relation $\mathcal{R}$ is*

$$\rho_{susp}(\mathcal{B},\mathcal{R})=\rho_{susp}(M_{\mathcal{B}},\{|\mathcal{B}_n|\}_{n=1}^N,M_{\mathcal{R}},\{|\mathcal{R}_n|\}_{n=1}^N)$$
$$=M_{\mathcal{B}}\left(\log\left(\frac{M_{\mathcal{B}}}{M_{\mathcal{R}}}\right)-1\right)+M_{\mathcal{R}}\prod_{n=1}^N\frac{|\mathcal{B}_n|}{|\mathcal{R}_n|}-M_{\mathcal{B}}\log\left(\prod_{n=1}^N\frac{|\mathcal{B}_n|}{|\mathcal{R}_n|}\right).$$

We slightly abuse the notations to emphasize that all density measures are the functions of the cardinalities of the attributes and the masses of $\mathcal{B}$ and $\mathcal{R}$.

2.3 Problem Definition

Based on the concepts and the density measures defined in the previous sections, Definition 4 gives a formal definition of the problem of detecting dense blocks in a large-scale tensor.

Definition 4 (Large-scale Top-k Densest Block Detection). **(1) Given:** *a large-scale relation $\mathcal{R}$ not fitting in memory, the number of blocks k, and a density measure ρ,* **(2) Find:** *k distinct blocks of $\mathcal{R}$ with the highest density in terms of ρ.*

Algorithm 1: D-Cube

Input : relation: $\mathcal{R}$, number of blocks we aim to find: k, density measure: ρ
Output: k dense blocks

1 $\mathcal{R}^{ori} \leftarrow copy(\mathcal{R})$
2 compute $\{\mathcal{R}_n\}_{n=1}^N$
3 $results \leftarrow \emptyset$ ▷ list of dense blocks
4 **for** $i \leftarrow 1..k$ **do**
5 $\quad M_{\mathcal{R}} \leftarrow \sum_{t\in\mathcal{R}} t[X]$
6 $\quad \{\mathcal{B}_n\}_{n=1}^N \leftarrow find\_single\_block(\mathcal{R},\{\mathcal{R}_n\}_{n=1}^N, M_{\mathcal{R}},\rho)$
7 $\quad$ ▷ see Algorithm 2
8 $\quad \mathcal{R} \leftarrow \{t\in\mathcal{R} : \exists n\in[N], t[A_n]\notin\mathcal{B}_n\}$ ▷ $\mathcal{R}\leftarrow\mathcal{R}-\mathcal{B}$
9 $\quad \mathcal{B}^{ori} \leftarrow \{t\in\mathcal{R}^{ori} : \forall n\in[N], t[A_n]\in\mathcal{B}_n\}$
10 $\quad results \leftarrow results \cup \{\mathcal{B}^{ori}\}$
11 **return** $results$

Even when we restrict our attention to finding one block in a matrix fitting in memory (i.e., $k=1$ and $N=2$), obtaining an exact solution is known to be computationally intractable with large data [18, 24]. Thus, our focus in this work is to design an approximate algorithm with (1) near-linear scalability with all aspects of $\mathcal{R}$, which does not fit in memory, (2) lower bounds on the approximation ratio at least for some density measures, and (3) meaningful results on real-world data.

3. PROPOSED METHOD

In this section, we propose D-Cube, a disk-based dense-block detection method. We describe D-Cube in Section 3.1 and prove its theoretical properties in Section 3.2. In Section 3.3, we present our MapReduce implementation of D-Cube. Throughout this section, we assume the entries of tensors (i.e., the tuples of relations) are stored in disk, and read and written only in a sequential way. All other data (e.g., distinct attribute-value sets and the mass of each attribute value), however, are assumed to be stored in memory.

3.1 Algorithm

D-Cube is a search method that starts with the given relation and removes attribute values (and tuples with the attribute values) sequentially so that a dense block is left. Contrary to previous approaches, D-Cube removes multiple attribute values (and tuples with the attribute values) at a time to reduce the number of iterations and also the amount of disk I/O. In addition to this advantage, D-Cube carefully chooses attribute values to remove to give the same accuracy guarantee as if attribute values were removed one by one, and shows comparable or even higher accuracy empirically.

3.1.1 Overall Structure of D-Cube (Algorithm 1)

Algorithm 1 describes the overall structure of D-Cube. D-Cube first copies and assigns the given relation $\mathcal{R}$ to $\mathcal{R}^{ori}$ (line 1); and computes the sets of distinct attribute values composing $\mathcal{R}$ (line 2). Then, it finds k dense blocks one by one from $\mathcal{R}$ (line 6) using its mass as a parameter (line 5). The detailed procedure for detecting a single dense block from $\mathcal{R}$ is explained in Section 3.1.2. After each block $\mathcal{B}$ is found, the tuples included in $\mathcal{B}$ are removed from $\mathcal{R}$ (line 8) to prevent the same block from being found again. Due to this change in $\mathcal{R}$, blocks found from $\mathcal{R}$ are not necessarily the blocks of the original relation $\mathcal{R}^{ori}$. Thus, instead of $\mathcal{B}$, the block in $\mathcal{R}^{ori}$ formed by the same attribute values forming

Algorithm 2: *find_single_block* in D-CUBE

Input : relation: $\mathcal{R}$, attribute-value sets: $\{\mathcal{R}_n\}_{n=1}^N$, mass: $M_{\mathcal{R}}$, density measure: ρ
Output: sets of attribute values forming a dense block

1 $\mathcal{B} \leftarrow copy(\mathcal{R})$, $M_{\mathcal{B}} \leftarrow M_{\mathcal{R}}$ ▷ initialize the block $\mathcal{B}$
2 $\mathcal{B}_n \leftarrow copy(\mathcal{R}_n)$, $\forall n \in [N]$
3 $\tilde{\rho} \leftarrow \rho(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)$ ▷ $\tilde{\rho}$: max ρ so far
4 $r, \tilde{r} \leftarrow 1$ ▷ r: current order of attribute values, $\tilde{r}$: r with $\tilde{\rho}$
5 **while** $\exists n \in [N], \mathcal{B}_n \neq \emptyset$ **do** ▷ until all are removed
6 compute $\{\{M_{\mathcal{B}(a,n)}\}_{a \in \mathcal{B}_n}\}_{n=1}^N$
7 $i \leftarrow select\_dimension()$ ▷ see Algorithms 3 and 4
8 $D_i \leftarrow \{a \in \mathcal{B}_i : M_{\mathcal{B}(a,i)} \leq \frac{M_{\mathcal{B}}}{|\mathcal{B}_i|}\}$ ▷ set to be removed
9 sort D_i in an increasing order of $M_{\mathcal{B}(a,i)}$
10 **for** *each* $a \in D_i$ **do**
11 $\mathcal{B}_i \leftarrow \mathcal{B}_i - \{a\}$, $M_{\mathcal{B}} \leftarrow M_{\mathcal{B}} - M_{\mathcal{B}(a,i)}$
12 $\rho' \leftarrow \rho(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)$ ▷ ρ when a is removed
13 $order(a,i) \leftarrow r$, $r \leftarrow r+1$
14 **if** $\rho' > \tilde{\rho}$ **then**
15 $\tilde{\rho} \leftarrow \rho'$, $\tilde{r} \leftarrow r$ ▷ update max ρ so far
16 $\mathcal{B} \leftarrow \{t \in \mathcal{B} : t[A_i] \notin D_i\}$ ▷ remove tuples
17 $\tilde{\mathcal{B}}_n \leftarrow \{a \in \mathcal{R}_n : order(a,n) \geq \tilde{r}\}$, $\forall n \in [N]$ ▷ reconstruct
18 **return** $\{\tilde{\mathcal{B}}_n\}_{n=1}^N$

Algorithm 3: *select_dimension* by cardinality

Input : attribute-value sets: $\{\mathcal{B}_n\}_{n=1}^N$
Output: a dimension in $[N]$

1 **return** n with maximum $|\mathcal{B}_n|$

Algorithm 4: *select_dimension* by density

Input : attribute-value sets: $\{\mathcal{B}_n\}_{n=1}^N$ and $\{\mathcal{R}_n\}_{n=1}^N$, attribute-value masses: $\{\{M_{\mathcal{B}(a,n)}\}_{a \in \mathcal{B}_n}\}_{n=1}^N$, masses: $M_{\mathcal{B}}$ and $M_{\mathcal{R}}$, density measure: ρ
Output: a dimension in $[N]$

1 $\tilde{\rho} \leftarrow -\infty$ ▷ $\tilde{\rho}$: max ρ so far
2 $\tilde{i} \leftarrow 1$ ▷ $\tilde{i}$: dimension with $\tilde{\rho}$
3 **for** *each dimension* $i \in [N]$ **do**
4 **if** $\mathcal{B}_i \neq \emptyset$ **then**
5 $D_i \leftarrow \{a \in \mathcal{B}_i : M_{\mathcal{B}(a,i)} \leq \frac{M_{\mathcal{B}}}{|\mathcal{B}_i|}\}$ ▷ set to be removed
6 $M'_{\mathcal{B}} \leftarrow M_{\mathcal{B}} - \sum_{a \in D_i} M_{\mathcal{B}(a,i)}$
7 $\mathcal{B}'_i \leftarrow \mathcal{B}_i - D_i$
8 $\rho' \leftarrow \rho(M'_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n \neq i} \cup \{|\mathcal{B}'_i|\}, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)$ ▷ ρ when all in D_i are removed
9 **if** $\rho' > \tilde{\rho}$ **then**
10 $\tilde{\rho} \leftarrow \rho'$, $\tilde{i} \leftarrow i$ ▷ update max ρ so far
11 **return** $\tilde{i}$

$\mathcal{B}$ is added to the list of k dense blocks (lines 9-10). Due to this step, D-CUBE can detect overlapping dense blocks. That is, a tuple can be included in multiple dense blocks found by D-CUBE.

Based on our assumption that the sets of distinct attribute values (i.e., $\{\mathcal{R}_n\}_{n=1}^N$ and $\{\mathcal{B}_n\}_{n=1}^N$) are stored in memory and can be randomly accessed, all the steps in Algorithm 1 can be performed by sequentially reading and writing tuples in relations (i.e., tensor entries) in disk without loading all the tuples in memory at once. For example, filtering steps in lines 8-9 can be performed by sequentially reading each tuple from disk and writing the tuple to disk only if it satisfies the given condition.

Note that this overall structure of D-CUBE is similar with that of M-ZOOM [38] except the fact that tuples are stored in disk. However, the methods differ significantly in the way each dense block is found from $\mathcal{R}$, which is explained in the following section.

3.1.2 Single Block Detection (Algorithm 2)

Algorithm 2 describes how D-CUBE detects each dense block from the given relation $\mathcal{R}$. It first initializes the block $\mathcal{B}$ to $\mathcal{R}$ (lines 1-2), then repeatedly removes attribute values and tuples of $\mathcal{B}$ with those attribute values until $\mathcal{B}$ becomes empty (line 5). Specifically, in each iteration, D-CUBE first chooses a dimension attribute A_i from which attribute values are removed (line 7), then computes D_i, the set of attribute values whose masses are less than the average (line 8). The way the dimension attribute is chosen is explained in Section 3.1.3.

The tuples whose attribute values of A_i are in D_i are removed from $\mathcal{B}$ at once within a single scan of $\mathcal{B}$ (line 16). However, deleting a subset of D_i may achieve higher value of the metric ρ. Hence, D-CUBE computes the changes in the density of $\mathcal{B}$ (line 11) as if the attribute values in D_i were removed one by one, in an increasing order of their masses. This allows D-CUBE to optimize ρ as if we removed attributes one by one, while still benefiting from the computational speedup of removing multiple attributes in each scan. Note that these changes in ρ can be computed exactly without actually removing the tuples from $\mathcal{B}$ or even accessing the tuples in $\mathcal{B}$ since its mass (i.e., $M_{\mathcal{B}}$) and the number of distinct attribute values (i.e., $\{|\mathcal{B}_n|\}_{n=1}^N$) are maintained up-to-date (lines 11-12). This is because removing an attribute value from a dimension attribute does not affect the mass of the other attribute values of the same attribute. The orders that attribute values are removed and when the density of $\mathcal{B}$ is maximized are maintained (lines 13-15) so that the block $\mathcal{B}$ maximizing density can be restored and returned (lines 17-18), as the result of Algorithm 2.

Note that, in each iteration (lines 5-16) of Algorithm 2, the tuples of $\mathcal{B}$ in disk need to be scanned only twice, once in line 6 and once in line 16. Moreover, both steps can be performed by simply sequentially reading and/or writing tuples in $\mathcal{B}$ without loading all the tuples in memory at once. For example, to compute attribute-value masses in line 6, D-CUBE increases $M_{\mathcal{B}(t[A_n],n)}$ by $t[X]$ for each dimension attribute A_n after reading each tuple t in $\mathcal{B}$ sequentially from disk.

3.1.3 Dimension Selection (Algorithms 3 and 4)

We discuss two policies for choosing a dimension attribute from which attribute values are removed. They are used in line 7 of Algorithm 2 offering different advantages.

Maximum cardinality policy (Algorithm 3): The dimension attribute with the largest cardinality is chosen, as described in Algorithm 3. This simple policy, however, provides an accuracy guarantee (see Section 3.2.2).

Maximum density policy (Algorithm 4): The density of $\mathcal{B}$ when attribute values are removed from each dimension attribute is computed, then the dimension attribute leading to the highest density is chosen. Note that the tuples in $\mathcal{B}$, stored in disk, do not need to be accessed for this computation, as described in Algorithm 4. Although this policy does not provide the accuracy guarantee given by the maximum cardinality policy, this policy works well with various

density measures and tends to spot denser blocks than the maximum cardinality policy in real-world data, especially when ρ_{geo} or ρ_{susp} is used.

3.1.4 Efficient Implementation

We present the optimization techniques used for the efficient implementation of D-CUBE.

Combining Disk-Accessing Steps. The amount of disk I/O can be reduced by combining multiple steps requiring disk accesses. In Algorithm 1, updating $\mathcal{R}$ (line 8) can be combined with computing the mass of $\mathcal{R}$ (line 5) in the next iteration. That is, if we aggregate the values of the tuples in $\mathcal{R}$ while the tuples of $\mathcal{R}$ are written for the update, we do not need to scan $\mathcal{R}$ again for computing its mass in the next iteration. Likewise, in Algorithm 2, updating $\mathcal{B}$ (line 16) can be combined with computing attribute-value masses (line 6) in the next iteration. This optimization reduces the amount of disk I/O in D-CUBE about 30%.

Caching Tensor Entries in Memory. Although we assume that tuples are stored in disk, storing them in memory up to the memory capacity speeds up D-CUBE up to 3 times in our experiments (see Section 4.4). We cache the tuples in $\mathcal{B}$, which are more frequently accessed than those in $\mathcal{R}$ or $\mathcal{R}^{ori}$, in memory with the highest priority.

3.2 Analysis

In this section, we prove the time and space complexity of D-CUBE and the accuracy guarantee provided by D-CUBE.

3.2.1 Complexity Analysis

Theorem 1 states the worst-case time complexity, which is also the worst-case I/O complexity, of D-CUBE. We let $L = \max_{n\in[N]} |\mathcal{R}_n|$.

Theorem 1 (Worst-case Time Complexity)**.** *The worst-case time complexity of Algorithm 1 is* $O(kN^2|\mathcal{R}|L)$.

Proof. Since at least one attribute value is removed in each iteration (lines 5-16) of Algorithm 2, the number of iterations is at most NL, which is the maximum number of distinct attribute values. Executing lines 6 and 16 NL times takes $O(N^2|\mathcal{R}|L)$, which dominates the time complexity of the other parts, since $L \leq |\mathcal{R}|$ by definition. Thus, the worst-case time complexity of Algorithm 2 is $O(N^2|\mathcal{R}|L)$, and that of Algorithm 1, which executes Algorithm 2 for k times, is $O(kN^2|\mathcal{R}|L)$. ■

However, this worst-case time complexity, which allows the worst distributions of the measure attribute values of tuples, is too pessimistic. In fact, we experimentally show that D-CUBE scales linearly with k, N, and $\mathcal{R}$; and even sub-linearly with L (see Section 4.4).

Theorem 2 states the memory requirement of D-CUBE. Since the tuples do not need to be stored in memory all at once in D-CUBE, its memory requirement does not depend on the number of tuples (i.e., $|\mathcal{R}|$).

Theorem 2 (Memory Requirements)**.** *The amount of memory space required by* D-CUBE *is* $O(\sum_{n=1}^{N} |\mathcal{R}_n|)$.

Proof. D-CUBE stores $\{\{M_{\mathcal{B}(a,n)}\}_{a\in\mathcal{B}_n}\}_{n=1}^{N}$, $\{\mathcal{R}_n\}_{n=1}^{N}$, and $\{\mathcal{B}_n\}_{n=1}^{N}$ in memory. Each has at most $\sum_{n=1}^{N} |\mathcal{R}_n|$ values. Thus, the memory requirement is $O(\sum_{n=1}^{N} |\mathcal{R}_n|)$. ■

3.2.2 Accuracy Guarantee

We show that D-CUBE gives the same accuracy guarantee with in-memory algorithms [38] although accesses to tuples (stored in disk) are restricted in D-CUBE to reduce disk I/O cost. Specifically, Theorem 3 states that the block found by Algorithm 2 with the maximum cardinality policy has density at least $1/N$ of the optimum when ρ_{ari} is used as the density measure.

Theorem 3 ($1/N$-Approximation Guarantee)**.** *Let* $\mathcal{B}^*$ *be the block* $\mathcal{B}$ *maximizing* $\rho_{ari}(\mathcal{B},\mathcal{R})$ *in the given relation* $\mathcal{R}$*. Let* $\tilde{\mathcal{B}}$ *be the block returned by Algorithm 2 with* ρ_{ari} *and the maximum cardinality policy. Then,* $\rho_{ari}(\tilde{\mathcal{B}},\mathcal{R}) \geq \frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R})$.

Proof. First, the maximal block $\mathcal{B}^*$ satisfies that for any $i \in [N]$ and for any attribute value $a \in \mathcal{B}_i^*$, its attribute-value mass $M_{\mathcal{B}^*(a,i)}$ is at least $\frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R})$. This is since the maximality of $\mathcal{B}^*$ implies $\rho_{ari}(\mathcal{B}^*-\mathcal{B}^*(a,i),\mathcal{R}) \leq \rho_{ari}(\mathcal{B}^*,\mathcal{R})$, and plugging in Definition 1 to ρ_{ari} gives $\frac{M_{\mathcal{B}^*}-M_{\mathcal{B}^*(a,i)}}{\frac{1}{N}((\sum_{n=1}^{N}|\mathcal{B}_n^*|)-1)} = \rho_{ari}(\mathcal{B}^*-\mathcal{B}^*(a,i),\mathcal{R}) \leq \rho_{ari}(\mathcal{B}^*,\mathcal{R}) = \frac{M_{\mathcal{B}^*}}{\frac{1}{N}\sum_{n=1}^{N}|\mathcal{B}_n^*|}$, which reduces to

$$M_{\mathcal{B}^*(a,i)} \geq \frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R}). \tag{1}$$

Consider the earliest iteration (lines 5-16) in Algorithm 2 where an attribute value a of $\mathcal{B}^*$ is included in D_i, for some $i \in [N]$. Let $\mathcal{B}'$ be $\mathcal{B}$ in the beginning of the iteration. Our goal is to prove $\rho_{ari}(\tilde{\mathcal{B}},\mathcal{R}) \geq \frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R})$, which we show as $\rho_{ari}(\tilde{\mathcal{B}},\mathcal{R}) \geq \rho_{ari}(\mathcal{B}',\mathcal{R}) \geq M_{\mathcal{B}'(a,i)} \geq M_{\mathcal{B}^*(a,i)} \geq \frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R})$.

First, $\rho_{ari}(\tilde{\mathcal{B}},\mathcal{R}) \geq \rho_{ari}(\mathcal{B}',\mathcal{R})$ is from maximality of $\rho_{ari}(\tilde{\mathcal{B}},\mathcal{R})$ among blocks generated in iteration (lines 13-15 in Algorithm 2). Second, applying $|\mathcal{B}_i'| \geq \frac{1}{N}\sum_{n=1}^{N}|\mathcal{B}_n'|$ from the maximum cardinality policy (Algorithm 3) to Definition 1 of ρ_{ari} gives $\rho_{ari}(\mathcal{B}',\mathcal{R}) = \frac{M_{\mathcal{B}'}}{\frac{1}{N}\sum_{n=1}^{N}|\mathcal{B}_n'|} \geq \frac{M_{\mathcal{B}'}}{|\mathcal{B}_i'|}$. And $a \in D_i$ gives $\frac{M_{\mathcal{B}'}}{|\mathcal{B}_i'|} \geq M_{\mathcal{B}'(a,i)}$. So combining these gives $\rho_{ari}(\mathcal{B}',\mathcal{R}) \geq M_{\mathcal{B}'(a,i)}$. Third, $M_{\mathcal{B}'(a,i)} \geq M_{\mathcal{B}^*(a,i)}$ is from $\mathcal{B}' \supset \mathcal{B}^*$. Fourth, $M_{\mathcal{B}^*(a,i)} \geq \frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R})$ is from Eq. (1). Hence $\rho_{ari}(\tilde{\mathcal{B}},\mathcal{R}) \geq \frac{1}{N}\rho_{ari}(\mathcal{B}^*,\mathcal{R})$ holds. ■

3.3 MapReduce Implementation

We present our MAPREDUCE implementation of D-CUBE, assuming that tuples in relations are stored in a distributed file system. Specifically, we describe two MAPREDUCE algorithms corresponding to steps of D-CUBE accessing tuples.

Filtering Tuples. In lines 8-9 of Algorithm 1 and line 16 of Algorithm 2, D-CUBE filters tuples with the given conditions. These steps are done by the following map-only algorithm, where we broadcast the data used in each condition (e.g., $\{\mathcal{B}_n\}_{n=1}^{N}$ in line 8 of Algorithm 1) to mappers using the distributed cache functionality.

- Map-stage: Take a tuple t (i.e., $<t[A_1],...,t[A_N],t[X]>$) and emit t if t satisfies the given condition. Otherwise, the tuple is ignored.

Computing Attribute-value Masses. Line 6 of Algorithm 2 is performed by the following algorithm, where we reduce the amount of shuffled data by combining the intermediate results within each mapper.

- Map-stage: Take a tuple t (i.e., $<t[A_1],...,t[A_N],t[X]>$) and emit N key/value pairs, $\{<(n,t[A_n]),t[X]>\}_{n=1}^{N}$.

Table 3: Summary of real-world datasets.

Name	Volume	#Tuples
Rating data (user, item, timestamp, rating, 1)		
Yelp [4]	552K × 77.1K × 3.80K × 5	2.23M
Android [29]	1.32M × 61.3K × 1.28K × 5	2.64M
Netflix [12]	480K × 17.8K × 2.18K × 5	99.1M
YahooM. [16]	1.00M × 625K × 84.4K × 101	253M
Wiki revision history (user, page, timestamp, #revisions)		
KoWiki [38]	470K × 1.18M × 101K	11.0M
EnWiki [38]	44.1M × 38.5M × 129K	483M
Social networks (user, user, timestamp, #interactions)		
Youtube [30]	3.22M × 3.22M × 203	18.7M
SMS	1.25M × 7.00M × 4.39K	103M
TCP dumps (src IP, dst IP, timestamp, #connections)		
DARPA [27]	9.48K × 23.4K × 46.6K	522K
TCP dumps (protocol, service, src bytes, ..., #connections)		
AirForce [2]	3 × 70 × 11 × 7.20K × 21.5K × 512 × 512	648K

- Reduce-stage: Take $<(n,a), \{t[X] : t \in \mathcal{B}, t[A_n]{=}a\}>$ and emit $<(n,a), \text{sum}(\{t[X] : t \in \mathcal{B}, t[A_n]{=}a\})>$, whose value corresponds to $M_{\mathcal{B}(a,n)}$.

Likewise, the other steps of D-Cube accessing tuples (i.e., lines 2 and 5 of Algorithm 1) can be implemented on MapReduce in a straightforward way. See the supplementary document [3] for details.

4. EXPERIMENTS

We design experiments to answer the following questions:

- **Q1. Memory Efficiency**: How much memory space does D-Cube require for analyzing real-world data? How large data can D-Cube handle?
- **Q2. Speed and Accuaracy**: How fast and accurately does D-Cube spot dense blocks?
- **Q3. Scalability**: Does D-Cube scale linearly with all aspects of data? Does D-Cube scale out?
- **Q4. Effectiveness**: Which anomalies and fraud does D-Cube detect in real-world data?

4.1 Experimental Settings

Machines: We ran all serial algorithms on a machine with 2.67GHz Intel Xeon E7-8837 CPUs and 1TB memory. We ran MapReduce algorithms on a 40-node Hadoop cluster, where each node has an Intel Xeon E3-1230 3.3GHz CPU and 32GB memory.

Data: We used large-scale real-world tensors from various domains, including rating data, Wikipedia revision history, temporal social networks, and TCP dumps. They are summarized in Table 3, and their detailed description is in the supplementary document [3]. We used synthetic tensors for scalability tests. Each tensor was created by generating a random binary tensor and injecting ten random dense blocks, whose volumes are 10^N and densities (in terms of ρ_{ari}) are between 10× and 100× of that of the entire tensor.

Implementations: We implemented D-Cube in Java with Hadoop 1.2.1, and used the Java implementation of open source M-Zoom[1] [38] and CrossSpot[2] [20]. Tensor Toolbox [8] was used for CP Decomposition (CPD) [25] and MAF [28]. For D-Cube, we used the maximum density policy (see Section 3.1.3), which consistently yielded higher accuracy than the maximum cardinality policy. We used CPD as the seed selection method of CrossSpot as in [38].

[1]https://github.com/kijungs/mzoom
[2]https://github.com/mjiang89/CrossSpot

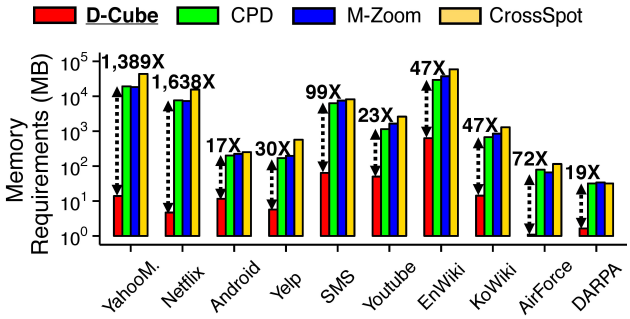


Figure 3: **D-Cube is memory efficient.** D-Cube requires up to **1,600× less memory** than its competitors.

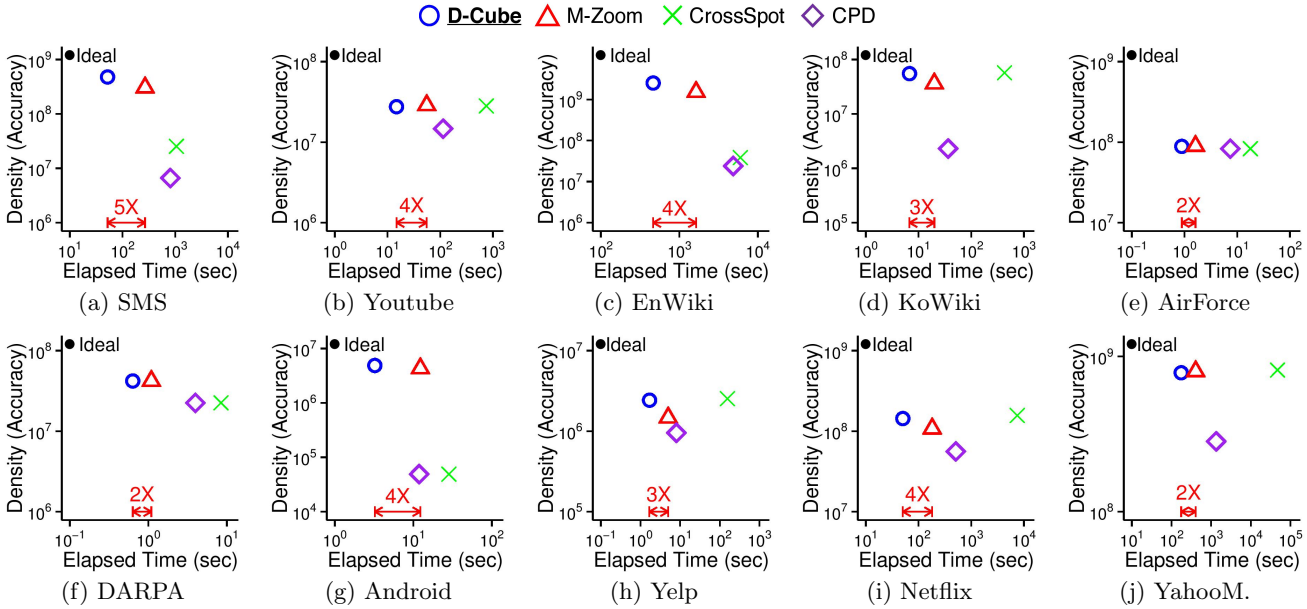
4.2 Q1. Memory Efficiency

We compare the amount of memory required by different methods for handling real-world datasets. As seen in Figure 3, D-Cube, which does not require tuples to be stored in memory, required up to **1,600× less memory space** than the second best method, which stores tuples in memory.

Due to its memory efficiency, D-Cube successfully handled **1,000× larger data** than its competitors within a memory budget. We ran methods on 3-way synthetic tensors with different numbers of tuples (i.e., $|\mathcal{R}|$), with a memory budget of 16GB per machine. In every tensor, the cardinality of each dimension attribute was 1/1000 of the number of tuples, i.e., $|\mathcal{R}_n| = |\mathcal{R}|/1000$, $\forall n \in [N]$. Figure 1(a) in Section 1 shows the result. The Hadoop implementation of D-Cube successfully spotted dense blocks in a tensor with 10^{11} tuples (**2.6TB**), and the serial version of D-Cube successfully spotted dense blocks in a tensor with 10^{10} tuples (**240GB**), which was the largest tensor that can be stored on a disk. However, all other methods ran out of memory even on a tensor with 10^9 tuples (21GB).

4.3 Q2. Speed and Accuracy

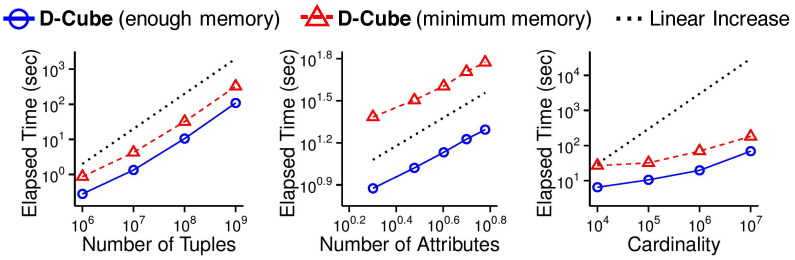
We compare how rapidly and accurately each method detects dense blocks in real-world datasets. We measured the wall-clock time (average over three runs) taken for detecting three blocks by each method, and measured the maximum density of the three blocks found by each method. Note that the serial version of D-Cube was used, and each dataset was cached in memory by D-Cube since they fit in memory (see Section 3.1.4). Figure 4 shows the results when ρ_{susp} [20], which has the soundest theoretical foundation, was used as the density measure. D-Cube provided the best trade-off between speed and accuracy. Specifically, D-Cube was up to **5× faster** than the second fastest method M-Zoom. Moreover, D-Cube consistently spotted high-density blocks, while the accuracy of the other methods varied on data. Especially, D-Cube gave **the densest blocks** in SMS, Android, and EnWiki datasets, and compared with CPD, gave up to **105× denser** blocks. Although MAF does not appear in Figure 4, it consistently provided sparser blocks than CPD with similar speed. In addition, the maximum cardinality policy of D-Cube resulted in 33% sparser blocks on average than the maximum density policy, which was used in this



(a) SMS (b) Youtube (c) EnWiki (d) KoWiki (e) AirForce

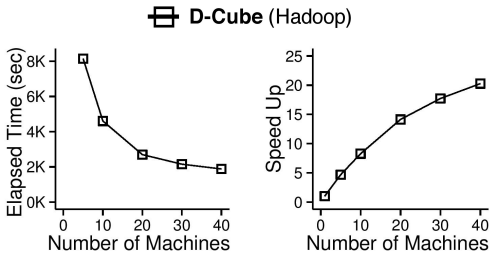
(f) DARPA (g) Android (h) Yelp (i) Netflix (j) YahooM.

Figure 4: **D-Cube achieves both speed and accuracy.** In each plot, points represent the speed of different methods and the highest density (in terms of ρ_{susp}) of three blocks found by the methods. Upper-left region indicates better performance. D-Cube gave the best trade-off between speed and density. Specifically, D-Cube consistently found high-density blocks up to **5× faster** than second best method M-Zoom.



(a) Scalability w.r.t $|\mathcal{R}|$ (b) Scalability w.r.t N (c) Scalability w.r.t $|\mathcal{R}_n|$

Figure 5: **D-Cube is scalable with all the aspects of tensors.** D-Cube scaled (sub-)linearly with every aspect of tensors regardless of the amount of memory space available.



(a) Elapsed Time (b) Speed Up

Figure 6: **D-Cube scales out.** D-Cube was speeded up 8× with 10 machines, and 20× with 40 machines.

experiment, with similar speed. Experiments with the other density measures (i.e., ρ_{ari} and ρ_{geo}) resulted similarly, as described in the supplementary document [3].

4.4 Q3. Scalability

We show that D-Cube scales (sub-)linearly with every aspect of tensor data, i.e., the number of tuples, the number of dimension attributes, and the cardinality of dimension attributes. To measure the scalability with each factor, we started with finding a dense block in a synthetic tensor with 10^8 tuples, and 3 dimension attributes each of whose cardinality is 10^5. Then, we measured the running time as we changed one factor at a time while fixing the other factors. As seen in Figure 5, D-Cube scales linearly with every factor and even sub-linearly with the cardinality of attributes. Moreover, D-Cube also scales linearly with k, the number of blocks we aim to find (see the supplementary document [3]). This supports our claim in Section 3.2.1 that the worst-case time complexity of D-Cube (Theorem 1) is too pessimistic. The linear scalability of D-Cube held both with enough memory (blue solid lines in Figure 5) to store all tuples and with minimum memory (red dashed lines in Figure 5) to barely meet requirements although D-Cube was up to 3× faster in the former case.

We also evaluate the machine scalability of the MapReduce implementation of D-Cube. We measured its running time taken for finding a dense block in a synthetic tensor with 10^{10} tuples and 3 dimension attributes each of whose cardinality is 10^7, as we increased the number of machines running in parallel from 1 to 40. Figure 6 shows the changes in running time and speed-up defined as T_1/T_M where T_M is the running time with M machines. The speed-up increased near linearly when a small number of machines were used, while it flattened as more machines were added due to the overhead in the distributed system.

4.5 Q4. Effectiveness

We demonstrate the effectiveness of D-Cube in two applications using real-world data.

Network Intrusion Detection. D-Cube detected network attacks in TCP dumps with high accuracy by detecting corresponding dense blocks. We consider two TCP dumps with different models. DARPA Dataset is modeled as a 3-way tensor where the dimension attributes are source IP,

Table 4: **D-Cube spotted network attacks fastest with the highest accuracy** from TCP dumps.

Datasets	AirForce		DARPA	
	Elapsed Time (sec)	Accuracy (AUC)	Elapsed Time (sec)	Accuracy (AUC)
CPD [25]	413.2	0.854	105.0	0.926
MAF [28]	486.6	0.912	102.4	0.514
CrossSpot [20]	575.5	0.924	132.2	0.923
M-Zoom [38]	27.7	0.975	22.7	0.923
D-Cube	**15.6**	**0.987**	**9.1**	**0.930**

Table 5: **D-Cube detected fraud fastest with the highest accuracy** in rating datasets.

Datasets	Android		Yelp	
	Elapsed Time (sec)	Recall @ Top-10	Elapsed Time (sec)	Recall @ Top-10
CPD [25]	59.9	0.54	47.5	0.52
MAF [28]	95.0	0.54	49.4	0.52
CrossSpot [20]	71.3	0.54	56.7	0.52
M-Zoom [38]	28.4	0.70	17.7	0.30
D-Cube	**7.0**	**0.90**	**4.9**	**0.60**

destination IP, and timestamp in minutes; and the measure attribute is the number of connections. AirForce Dataset, which does not include IP information, is modeled as a 7-way tensor where the measure attribute is the same but the dimension attributes are the features of the connections, including protocol and service (see the supplementary document [3] for the detailed description of AirForce Dataset). Both datasets include labels indicating whether each connection is malicious or not.

Figure 1(c) in Section 1 lists the five densest blocks (in terms of ρ_{geo}) found by D-Cube in each dataset. We found that the dense blocks are mostly composed of various types of network attacks. Based on this observation, we classified each connection as malicious or benign based on the density of the densest block including the connection (i.e., the denser the block including a connection is, the more suspicious the connection is). This led to high accuracy, and as seen in Table 4, the highest accuracy was achieved when D-Cube was used for dense-block detection in both datasets. For each method, the result with the density measure giving the highest accuracy is reported.

Fraud Detection in Rating Data. We assume an attack scenario where fraudsters in a review site, who aim to boost (or lower) the ratings of the set of businesses, create multiple user accounts and give the same score to the businesses within a short period of time. This lockstep behavior forms a dense block with volume $(\#users \times \#items \times 1 \times 1)$ in the rating dataset, whose dimension attributes are users, businesses (items), timestamps, and rating scores.

We injected 10 such random dense blocks whose volumes vary from 15×15×1×1 to 60×60×1×1 in Yelp and Android datasets. We compared the number of the injected blocks detected by each dense-block detection method. We considered each injected block as overlooked by a method, if the block did not belong to any of the first 10 dense blocks spotted by the method or it was hidden in a natural dense block at least 10 times larger than the injected block. We repeated this experiment 10 times, and the averaged results are summarized in Table 5. For each method, the results with the density measure giving the highest accuracy are reported. In both datasets, D-Cube detected the injected blocks the most. Especially, in Android Dataset, D-Cube detected 9 out of 10 injected blocks, while the second best method detected only 7 injected blocks on average.

5. RELATED WORK

Dense Subgraph Detection. Dense-subgraph detection in graphs has been extensively studied in theory; see [26] for a survey. Exact algorithms [18, 24] and approximate algorithms [14, 24] have been proposed for finding subgraphs with maximum average degree. These have been extended for incorporating size restrictions [7], alternative metrics for denser subgraphs [40], evolving graphs [17], subgraphs with limited overlap [11], and streaming or distributed settings [10, 9]. Dense subgraph detection has been applied to fraud detection in social or review networks [21, 13, 36, 19, 37].

Dense Block Detection in Tensors. Extending dense subgraph detection to tensors [20, 38] incorporates additional dimensions, such as time, to identify dense regions of interest with greater accuracy and specificity. CrossSpot [20] starts from a seed block and adjusts it in a greedy way until it reaches a local optimum, which usually takes long and does not provide any approximation bound. M-Zoom [38], which starts from the entire tensor and only shrinks it by removing attributes one by one in a greedy way, improves CrossSpot in terms of speed and approximation guarantees. Both methods, however, require the tuples to be loaded into memory at once and to be randomly accessed, which limit their applicability to large datasets. Dense-block detection in tensors has been found useful for detecting retweet boosting [20], network attacks [38, 28], bot activities [38], and for genetics applications [35, 28].

Large-Scale Tensor Decomposition. Tensor decomposition such as HOSVD and CP decomposition [25] can be used to spot dense subtensors [28]. Scalable algorithms for tensor decomposition have been developed, including disk-based algorithms [31, 39], distributed algorithms [22, 39], and approximation algorithms based on sampling [32] and count-min sketch [41]. However, dense-block detection based on tensor decomposition has serious limitations: it usually detects blocks with significantly lower density (see Section 4.3), provides less flexibility with regard to the choice of density metric, and does not provide approximation bounds.

Other Anomaly/Fraud Detection Methods. In addition to dense-block detection, many approaches, including those based on egonet features [5], coreness [37], and behavior models [33], have been used for anomaly/fraud detection in graphs. See [6] for surveys.

6. CONCLUSION

In this work, we propose D-Cube, a disk-based dense-block detection method, to deal with disk-resident tensors too large to fit in memory. D-Cube is optimized to minimize disk I/O cost, while it still provides guarantees on the quality of the blocks it finds. Moreover, we provide the distributed version of D-Cube running on MapReduce for terabyte-scale or larger data distributed across multiple machines. To sum up, D-Cube achieves the following advantages over state-of-the-art dense-block detection methods:

- **Memory Efficient:** D-Cube handles *1,000*× larger data (*2.6TB*) by reducing memory usage up to *1,600*× compared with in-memory algorithms (Section 4.2).
- **Fast:** Even when data fit in memory, D-Cube is up

to $5\times$ faster than its competitors (Section 4.3) with near-linear scalability (Section 4.4).
- **Provably Accurate:** D-Cube is one of the methods giving the best approximation guarantee (Theorem 3) and the densest blocks in real-world data (Section 4.3).
- **Effective:** D-Cube gave the best accuracy in two applications: detecting network attacks from TCP dumps and lockstep behavior in rating data (Section 4.5).

Reproducibility: Our open source code and the data used are at **http://www.cs.cmu.edu/~kijungs/codes/dcube/**.

Acknowledgments. 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.

7. REFERENCES

[1] Apache hadoop. http://hadoop.apache.org.
[2] Kdd cup 1999 data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.
[3] Supplementary document. Available online: http://www.cs.cmu.edu/~kijungs/codes/dcube/supple.pdf.
[4] Yelp dataset challenge. https://www.yelp.com/dataset_challenge.
[5] L. Akoglu, M. McGlohon, and C. Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *PAKDD*. 2010.
[6] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.
[7] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *WAW*. 2009.
[8] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.6. Available online.
[9] B. Bahmani, A. Goel, and K. Munagala. Efficient primal-dual graph algorithms for mapreduce. In *WAW*, 2014.
[10] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.
[11] O. D. Balalau, F. Bonchi, T. Chan, F. Gullo, and M. Sozio. Finding subgraphs with maximum total density and limited overlap. In *WSDM*, 2015.
[12] J. Bennett and S. Lanning. The netflix prize. In *KDD Cup*, 2007.
[13] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW*, 2013.
[14] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*. 2000.
[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[16] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. In *KDD Cup*, 2012.
[17] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *WWW*, 2015.
[18] A. V. Goldberg. *Finding a maximum density subgraph.* Technical Report, 1984.
[19] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. Fraudar: Bounding graph fraud in the face of camouflage. In *KDD*, 2016.
[20] M. Jiang, A. Beutel, P. Cui, B. Hooi, S. Yang, and C. Faloutsos. A general suspiciousness metric for dense blocks in multimodal data. In *ICDM*, 2015.
[21] M. Jiang, P. Cui, A. Beutel, C. Faloutsos, and S. Yang. Catchsync: catching synchronized behavior in large directed graphs. In *KDD*, 2014.
[22] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *KDD*, pages 316–324, 2012.
[23] R. Kannan and V. Vinay. *Analyzing the structure of large graphs.* Technical Report, 1999.
[24] S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, pages 597–608. 2009.
[25] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.
[26] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. 2010.
[27] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, et al. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *DISCEX*, 2000.
[28] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, 2011.
[29] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *KDD*, 2015.
[30] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC*, 2007.
[31] J. Oh, K. Shin, E. E. Papalexakis, C. Faloutsos, and H. Yu. S-hot: Scalable high-order tucker decomposition. In *WSDM*, 2017.
[32] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. In *PKDD*, 2012.
[33] R. A. Rossi, B. Gallagher, J. Neville, and K. Henderson. Modeling dynamic behavior in large evolving graphs. In *WSDM*, 2013.
[34] J. M. Ruhl. *Efficient algorithms for new computational models.* PhD thesis, 2003.
[35] B. Saha, A. Hoch, S. Khuller, L. Raschid, and X.-N. Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *RECOMB*, 2010.
[36] N. Shah, A. Beutel, B. Gallagher, and C. Faloutsos. Spotting suspicious link behavior with fbox: An adversarial perspective. In *ICDM*, 2014.
[37] K. Shin, T. Eliassi-Rad, and C. Faloutsos. Corescope: Graph mining using k-core analysis - patterns, anomalies and algorithms. In *ICDM*, 2016.
[38] K. Shin, B. Hooi, and C. Faloutsos. M-zoom: Fast dense-block detection in tensors with quality guarantees. In *ECML/PKDD*, 2016.
[39] K. Shin and U. Kang. Distributed methods for high-dimensional and large-scale tensor factorization. In *ICDM*, 2014.
[40] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *KDD*, 2013.
[41] Y. Wang, H.-Y. Tung, A. J. Smola, and A. Anandkumar. Fast and guaranteed tensor decomposition via sketching. In *NIPS*, 2015.