

# 15-826 Project Report

*Haoming Chen*

Dept. of Computer Science  
Carnegie Mellon University  
haomingc@cs.cmu.edu

*Xinrui He*

Dept. of Computer Science  
Carnegie Mellon University  
hxr@cmu.edu

March 27, 2017

## Abstract

We are living in an information-explosion era where 2.5 Quintillion bytes of data are generated every day. It's an interesting and challenging work to dig deep into these data and mine "gold" out of them to present and take advantage of their intrinsic nature. Moreover, data are usually associated with multiple dimensions (e.g: a person's statistics may be presented with gender, nationality, height, weight, age, etc.).

Tensors, or multi-dimensional data, are naturally expressed and stored as a table in relational databases with columns representing the value of each dimension. However, it's not always feasible to manipulate large-scale multi-dimensional data in memory. In this project, we utilized SQL to directly operating on disk-based data which can not fit in memory. To be more specific, we implemented D-Cube[6] algorithm to detect dense sub-tensors which enables us to discover patterns and anomalies in multi-dimensional data.

## 1 Introduction

### 1.1 Problem Definition

The problem we want to solve can be summarized as follows:

- **GIVEN**: multi-dimensional data along with its attributes, and metrics/methods for the specific D-Cube algorithm presented in Table 1.
- **FIND**:  $k$  dense blocks (sub-tensors) with density in descending order.

More details of the D-Cube algorithm will be covered in *Section 3: Methods*.

Table 1: Given data and methods

<i><b>Notation</b></i>	<i><b>Explanation</b></i>
$R$	Multi-dimensional data (Tensors)
$N$	Number of dimension attributes
$k$	Number of desired dense blocks
$\rho$	Density measure of tensor
$DimSelect$	Dimension selection method

## 1.2 Motivation

A specific entry of multi-dimensional data has a rich representation of meanings. For example, a patient’s record can have dimension attributes of age, gender, allergy history, medical treatment, etc. A TCP dump record in real-life Internet may consists of source IP, destination IP, and forwarding timestamp, etc.

Although it’s trivial to understand what a specific record means, it’s much harder to grasp the big picture of all records. What’s more, there exist similar records in the data which may form dense blocks in high dimensional space with proper manipulation such as reorganization. These dense blocks are in-explicit to simple human inspection and hide in the N-dimension space, but they are associated with important characteristics and properties in the real world. In the example of TCP dump, for instance, if there’re significant number of connections which come from the same source IP address and go to another same destination IP address within a fairly short time frame, then we have a reason to believe that these connections may come from a malicious cyber attack on purpose. In such scenario, these connections form a dense block in the multi-dimensional space as their attributes in different dimensions have great similarity to each other. Therefore, we can take advantage of this property of dense blocks and extract meaningful insights from gigantic amounts of data.

Aside from the example given above, finding dense blocks can also be useful in several pattern recognition schemes, as well as anomaly detection like distinguishing fake reviews written by paid posters on online shopping websites. To realize the idea, we proposed and implemented a dense block detection algorithm, D-Cube[6], using SQL and disk-based data in relational databases to overcome the traditional problem where data are too large to fit in memory. Latter experiments and result shows that D-Cube achieves our goal with both great efficiency and accuracy.

## 1.3 Contributions

The contributions of this project are summarized as follows:

- We implemented the D-Cube algorithm in SQL with a Python adapter named *Psycopg* to present a succinct and logical frame work which can be easily understand. We then

test the theory of dense block detection on different multi-aspect data collected in the real world and analyze their associated characteristics in different real-world problem settings.

- Upon the basis of the algorithm, we proposed and implemented optimization by taking advantage of indexing in relational databases. We tested the optimized algorithm and showed that it generally achieved better performance in multiple aspects.
- We carefully designed several controlled tests and experiments to find out the effect of different real-value variable settings (e.g. different  $k$ ,  $N$ , etc.).
- Likewise, we tested different metrics/methods of dimension selection and density measurement, and compare results in multiple aspects to reach conclusions in terms of computational efficiency, accuracy, etc.

Overall, our work presents a clear implementation and demonstration on the underlying intuition and setting of D-Cube algorithm by running and explaining on several real-world datasets, as well as elaborately investigating the effect and characteristics of different parameter and metrics. Moreover, we proposed possible optimization to speed up the computation and further cutting down the disk usage. We compare the results with the original algorithm and give detailed analysis.

## 2 Survey

Next we list the papers that each member read, along with their summary and critique.

### 2.1 Papers read by Haoming Chen

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

The first paper was the D-Cube paper by Kijung Shin, Bryan Hooi, Jisu Kim, Christos Faloutsos.[6]

- ***Problem definition:***

Detecting fraudulent behavior and anomaly in large scale multidimensional data have a great practical value in real-world applications. However, current approaches all presents the limitation of handling large volumes of such data which cannot fit in memory and their accuracy is far from satisfactory.

- ***Main idea:***

This paper introduces an approach called D-Cube to detect dense blocks in large scale multidimensional data while improving the effectiveness and accuracy.

By modeling the data as a tensor, former techniques like tensor decomposition and search-based methods are dedicated to finding dense blocks in high dimension space, to detect suspicious and potentially fraudulent behaviors like the cyber intrusion, malicious review writing and rating, etc. Though effective in a sense, these methods are limited in scale as they can only handle data that fits in memory. D-Cube, a new model as this paper presents, tackles this problem by allowing data to stay in disk while finding the dense blocks. Moreover, this technique can further speed up the computation by running in a distributed framework like MapReduce.

The algorithm starts by calculating certain statistics of block density and caches them in the memory as they will be frequently accessed. By running iteratively, the algorithm finds the dense blocks one by one and removes corresponding tuples that comprise these blocks with sequential scanning and writing in the disk space. Under this scheme, the algorithm does the computation on the fly, so we don't have to load all tuples into the memory at once. Furthermore, this paper also mentions combining steps that require disk access to reduce disk I/O as an optimization to increase efficiency.

Experiment on real-world datasets and analysis of D-Cube shows that it beats the traditional approaches of dense block detection in four dimensions: (1) requirement of memory; (2) computation speed; (3) accuracy and (4) efficiency in real-world application. Given the same amounts of data and computation resources, D-Cube amazingly achieves 1600x reduction of memory need and 5x speed up, while maintaining or even improving the detection accuracy.

- ***Use for our project:***

D-Cube is the exact algorithm that we are going to implement for our project. In addition, we have limited resource in terms of computation power and memory. Thus, D-Cube has the exact property we desire to detect dense block sitting in disk space.

- ***Shortcomings:***

Inside the iteration of D-Cube algorithm, there's a sub-module which is in charge of determining the removal order of dimensions. The paper presents two heuristic methods of decision of depending on cardinality or density. However, it's not mathematically guaranteed to be the best in terms of efficiency and accuracy. Further research can dive into this sub-problem and try to find a way for wiser deletion of dimensions.

### 2.1.2 Graph Analytics using the Vertica Relational Database

The second paper was the Vertica paper by Jindal, Alekh, Samuel Madden, Mal Castellanos, and Meichun Hsu.[2]

- ***Problem definition:***

Comparing to vertex-centric graph analytic systems, using a relational database like

Vertica to conduct graph analysis through SQL queries presents several aspects of advantages and flexibility, while at the meantime, guarantees comparable runtime performance.

- ***Main idea:***

Nowadays, graph analytics plays a significant part in our real-world applications, including social network analysis, shortest path finding, etc. Under such circumstance, many vertex-centric graph analytics systems like Giraph and GraphLab are invented to facilitate the complex computation. This paper shows that a traditional relational database like Vertica is sufficient to tackle such tasks, and even better, outperforms these vertex-centric systems by exploiting the nature of relational operators and data storage.

In the realm of graph analysis, many computations involve vertex/edge selection, aggregation, projections, weights update, etc. These sub-procedures can be easily done by forming SQL queries with relational operators, which are highly optimized in Vertica to reduce disk I/O and runtime complexity. Moreover, relational databases like Vertica can be easily extended using UDFs to handle modified execution pipelines while graph analytics systems are limited under such circumstance as they're not suitable in nature.

The paper further introduces several approaches to optimize query manipulation in relational databases, including creating temporary tables instead of updating existed ones to take advantage of the sequential writing, etc. From the experiments, we can conclude that conducting graph analysis using Vertica yields comparable runtime performance as it eliminates data loading in the first place. In addition, Vertica's nature of excelling at performing relational operations and its built-in optimization of only accessing qualified records instead of doing full scan repeatedly like Giraph further expedite the computation. Finally, Vertica is also more flexible in a way as users can customize it to tradeoff between memory overhead and disk I/O, and even combine graph analysis with relational analysis. Vertex-centric systems lack the ability to do so as they are limited to certain types of analysis and have no control once the pipeline is settled.

- ***Use for our project:***

Since we are using SQL and relational database to implement our project, this paper is greatly helpful as it not only presents detailed SQL query translations of fundamental graph algorithms, but also points out the potential optimizations which we can exploit to further reduce disk I/O and memory overhead.

- ***Shortcomings:***

Relational databases like Vertica reduce the disk I/O at the cost of using more memory to achieve the comparable runtime performance as vertex-centric systems. However,

memory is much more expensive than hard disks and usually memory is the bottleneck. Thus, future research can focus on optimizing computation procedures to further reduce the memory requirement.

### 2.1.3 PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations

The third paper was the PEGASUS paper by Kang, U., Charalampos E. Tsourakakis, and Christos Faloutsos.[4]

- ***Problem definition:***

This paper introduces a large-scale graph mining library, Pegasus, which is implemented using the Hadoop architecture and its applications on several aspects like PageRank, connected components finding, etc. In addition, it also presents multiple optimizations to further speed up the computation and conducted experiments on real-world graph data.

- ***Main idea:***

Pegasus is a large-scale graph mining system that runs in a distributed manner on top of the Hadoop architecture. On most graph mining problems and applications, matrix-vector multiplications are heavily and repeatedly performed, Pegasus exploits this phenomenon and introduces a generalization called GIM-V that runs iteratively. By customizing the basic operations in GIM-V with SQL, we can perform many frequently-used graph mining algorithms like PageRank, RWR, etc. in parallel.

In addition to the naive implementation of GIM-V, this paper also introduces ways to optimize it, including computation based on block multiplication (GIM-V BL), which is superior in both speed and memory overhead using block encoding. Also, the idea of preprocessed edge clustering combined with block encoding (GIM-V CL) further reduces the computation. Another optimization exploits the nature of diagonal matrix blocks (GIM-V DI) to decrease the number of iterations when performing shuffling and disk I/O in MapReduce.

Experiments of utilizing Pegasus to conduct graph analysis show that it achieves at least 5x faster performance comparing to the naive implementation of algorithms mentioned above, and has the desired property to scale up well with increasing number of worker machines.

- ***Use for our project:***

A traditional problem for most graph mining approaches is that they assume graph data can sit in the memory, which is absolutely not the case in reality. Thus, using large-scale mining tools like Pegasus to take advantage of distributed architecture is a good choice to deal with gigantic graph mining problems, and our project is just

the exact situation. Moreover, Pegasus has already applied several optimizations to its basic operations which beats the naive implementations, thus we can utilize it to achieve faster computation.

- ***Shortcomings:***

Pegasus proposed GIM-V as a basic unit and implemented it with several optimizations. This is beneficial for many graph mining algorithms as they heavily rely on matrix-vector multiplication. However, not all graph mining algorithms are comprised of matrix-vector multiplication, and Pegasus is not helpful under these circumstances.

## 2.2 Papers read by Xinrui He

### 2.2.1 A General Suspiciousness Metric for Dense Blocks in Multimodal Data

The first paper was the CROSSSPOT paper by Jiang Meng, Alex Beutel, Peng Cui, Bryan Hooi, Shiqiang Yang, and Christos Faloutsos. [1]

- ***Problem Definition:***

In research, analysis about dense blocks is very important, because they indicate fraud, cheating, and emerging trends. However, we don't have a good metric to measure whether these dense blocks are worthy of attention. In addition, we lack a method to evaluate the suspiciousness of dense block with a different number of modes.

- ***Main idea:***

In this paper, authors propose a series of axioms, all axioms of which a good metric should be observed. These basic axioms include Density, Size, Concentration, Contrast and Multimodal. Authors propose a new metric, suspiciousness, that in line with all these axioms. The suspiciousness score of a multimodal block is the negative log likelihood of blocks mass under an ERP model. Other competitors, such as Mass, Density, Average Degree and Singular Value, have some shortcomings because they break some of the axioms somehow.

Furthermore, authors propose CROSSSPOT algorithm to solve suspicious block detection problem. This is a local search algorithm that starts from a seed suspicious block. For each iteration, we update by optimally choose a subset of values in a specific mode while holding constant values in other modes. We keep adjusting mode until it converges.

In addition, authors evaluate the CROSSSPOT algorithm with synthetic datasets. The experiment results show that CROSSSPOT outperforms other methods by having higher recall and precision on average. Compared to HOSVD, CROSSSPOT improves recall for finding both dense high-order blocks and low-order blocks. Experiments also prove that the performance of the algorithm is robust when we exceed a moderate number of random seeds, such as 50. The experiment results also indicate that the

convergence happens really fast, usually within 5 iterations. Experiments on retweeting dataset also show CROSSSPOT has better performance because it could catch bigger and denser blocks.

- ***Use for our project:***

This paper introduces a new metric and a new algorithm. We could include this new metric, suspiciousness in our project. And we may try to develop a more efficient algorithm based on CROSSSPOT.

- ***Shortcomings:***

The suspiciousness metric proposed here is based on several axioms presented by authors. However, these axioms may not always hold. When these axioms violated, the suspiciousness metric may have a negative effect on measuring dense blocks. For the CROSSSPOT algorithm, it is a greedy algorithm by choosing the local maximum. Thus, the final result may not be a global maximum. It is very interesting to develop an algorithm for finding the global maximum.

### 2.2.2 M-Zoom: Fast Dense-Block Detection in Tensors with Quality Guarantees

The second paper was the M-Zoom paper by Shin Kijung, Bryan Hooi, Christos Faloutsos.[5]

- ***Problem Definition:***

In this paper, authors focus on the problem of detecting the  $k$  densest blocks in a tensor. They not only consider addressing the problem of finding  $k$  distinct blocks with highest densities, but also the problem of finding  $k$  densest blocks with size bounds. They solve the problems on three density measures: Arithmetic Average Mass, Geometric Average Mass, and Suspiciousness.

- ***Main idea:***

Authors propose a flexible method to solve the problem of finding dense blocks, M-ZOOM (Multidimensional Zoom), which is more efficient, scalable and accurate. Authors also introduce how to implement the basic algorithm to solve various problems faster and better. The basic idea of the M-ZOOM algorithm is that it firstly duplicates the given relation and then finds  $k$  dense blocks one by one. In each iteration, it removes the tuples in the block to avoid same results. In the end, it returns the blocks of the duplication of relation, which have same attribute values with the found blocks. When finding the dense block, it uses a greedy strategy to choose the attribute value that maximizes the density of the block which removes this attribute value. It finally chooses the block with the maximum density among those obey the size limitation. To implement this algorithm in an efficient way, authors introduce the method of using min-heap for this greedy selection.

Compared to other traditional algorithms, M-ZOOM is fast, accurate and effective. Authors evaluate the performance of M-ZOOM with several experiments. On all the



real world datasets, M-ZOOM reaches the highest accuracy and fastest speed. Even when the number of tuples rises significantly, M-ZOOM performance is relatively good as well and the running time scales sub-linearly. Experiments also prove that M-ZOOM could have good performance in real data. It could detect edit wars and bot activities in Wikipedia with high accuracy. Furthermore, compared to other algorithms, M-ZOOM is very flexible because it works on different data, density measures, and features.

- ***Use for our project:***

The M-ZOOM algorithm proposed in this paper is more efficient than CROSSSPOT and several experiments show that this algorithm has high accuracy and efficiency. We could implement this algorithm for our dense block detection task.

- ***Shortcomings:***

The experiment shows that sometimes, the diversity of dense blocks M-ZOOM found is less than the CPD algorithm. We may figure out the reason and improve the M-ZOOM. M-ZOOM algorithm uses the min-heap for greedy selection of each single dense block. However, this method cannot guarantee the density because the algorithm is based on local optimum. We can use method like D-Cube to order the attribute values when we find each dense block.

### 2.2.3 GBASE: an efficient analysis platform for large graphs

The third paper was the GBASE paper by Kang, U., Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. [3]

- ***Problem Definition:***

Aiming at addressing large-scale graph queries in an efficient way, authors try to build a parallel and distributed graph management system. The key problems they need to address are storage, algorithms and query optimization.

- ***Main idea:***

Authors propose a new structure of graph management system, which contains two components: the indexing stage and the query stage. In the indexing stage, a raw graph is partitioned into several clusters firstly. Then, reshuffle the nodes and compress. Finally, the compressed blocks are stored in graph base with metadata. One technology they use to compress block is compressed block encoding. The key idea of this technology is to encode the graph as a binary string to reduce the storage space. They also introduce several ways to implement this technology, such as Zip Compression and Gap Elias Encoding. Another interesting method to reduce storage is that they use the grid placement to store the compressed blocks. This method minimizes the number of input files to answer queries.

In the query stage, authors noticed that many graph mining operations could be transformed to matrix-vector multiplication. This kind of operation is corresponding to a SQL join. Based on this observation, authors point out that we could use optimized join algorithms to address these kinds of graph mining operations. Authors explain the detailed implementation of several graph queries. Because most of these operations are carried out over adjacency matrix, the query execution engine of GBASE is built on top of HADOOP and MapReduce platform to execute queries sufficiently.

To evaluate the performance of the GBASE system, authors conduct several experiments. Results show that compression could reduce the file size and running time significantly. This improvement is more significant on targeted queries.

- ***Use for our project:***

The compression technique is very useful because it could lead to a huge reduction of storage and running time. We could also use the query transformation technology to speed up graph mining operations.

- ***Shortcomings:***

Some graph mining operations cannot be unified as matrix-vector multiplication. In these cases, the optimization authors proposed wont work.

## 3 Proposed Method

This section introduces our implementation of D-Cube algorithm of finding dense block. To be specific, this section describes how to manage data with SQL and how to implement D-Cube algorithms with Python and SQL code.

### 3.1 Notations and Symbols

This following table introduces symbols frequently used in the paper.

Symbol	Description
R	Relation table: an N-way tensor table includes N columns. Each column represents a dimension attribute.
R <sub>n</sub>	Table with 1 column which contains distinct values of the n-th attribute of R
MassR	mass of R (or B)
MassB <sub>a,n</sub>	attribute value mass of a in dimension n

Table 2: Table of arguments

## 3.2 Generic SQL Codes

This section describes general SQL method for general data management and manipulation of tables which are used across different algorithms.

- **TableDropCreate**

If table  $T$  exists, drop it.

DROP TABLE  $T$

Create a new table  $T$  with column format  $F$ .

CREATE TABLE  $T$  ( $F$ )

- **LoadTableFromFile**

Load data in file  $file$  to table  $T$  with column format  $F$ .

COPY  $T$  ( $F$ ) FROM  $file$  DELIMITER AS  $delim$  CSV

- **CopyTable**

Create Table  $T2$  and insert all data from  $T1$ .

CREATE TABLE  $T2$  AS TABLE  $T1$

- **DistinctAttributeValue**

Computes the set of distinct values on attribute  $A$  of table  $T$  and save them into table  $Tn$ .

TableDropCreate( $Tn$ )

INSERT INTO  $Tn(A)$  SELECT DISTINCT ON ( $A$ )  $A$  FROM  $T$

- **Mass**

Compute the mass of table  $T$ . We assume all the tuples in  $T$  has a measure attribute equals to 1.

SELECT count(\*) FROM  $T$

- **DeleteRows**

Delete rows in Table  $T$  with *condition*.

DELETE FROM  $T$  WHERE *condition*

- **InsertRow**

Insert a *newEntry* into Table  $T$ .

INSERT INTO  $T$  VALUES (*newEntry*)

## 3.3 D-Cube Algorithms

This section describes the implementation of D-Cube algorithms in Python and SQL. D-Cube algorithms are efficient and accurate to find dense blocks. This method is widely used in different practical tasks, such as finding malicious attacks in the network transmission and finding abnormal following relationship in social network data.

### 3.3.1 Overall Structure of D-Cube (Algorithm 1)

This algorithm describes the over structure of D-CUBE. It takes command line input as arguments. The table 3 shows description of these arguments.

notation	type	requirement	description
-file	string	Required	Full path to the data file to load from.
-delim	string	default=','	Delimiter that separate the columns in the input file.
-N	integer	Required	Number of dimension attributes.
-k	integer	Required	Number of dense blocks we aim to find.
-density	string	default='arithmetic'	Density measures. Support arithmetic, geometric and suspiciousness.
-selection	string	default='density'	Dimension selection policy. Support density and cardinality.

Table 3: Table of arguments

The main idea of this algorithm is to find each dense block one by one. It keeps a static copy of the original relation table. After each block  $B$  is found, tuples are removed from relation table to avoid duplication. However, when construct the final dense block, it select tuples from the original relation table such that we can get overlaped dense blocks.

**Input** : Arguments  
**Output**:  $k$  dense block tables and a report table  
 TableDropCreate( $R$ )  
 TableDropCreate( $ReportTable$ )  
 LoadTableFromFile( $R$ )  
 CopyTable( $ORI, R$ )  
**for**  $n$  in range( $N$ ) **do**  
     DistinctAttributeValue( $Rn$ )  
**end**  
 TableDropCreate( $ResultTable$ )  
**for**  $i$  in range( $k$ ) **do**  
      $MassR \leftarrow Mass(R)$   
      $Bn \leftarrow find\_single\_block(R, Rn, MassR, density)$  #See Algorithm 2  
     DeleteFromBlock( $R, Bn$ )  
     BlockCreateInsert( $ResultTable\_i, ORI, Bn$ )  
**end**  
 Add dense block information into  $ReportTable$

**Algorithm 1: D-CUBE**

Details of special functions in Algorithm 1 are listed below:

- **DeleteFromBlock**

Delete block  $B$  from relation  $R$ . Delete tuples of  $R$  that all attributes are in  $Bn$ .

```

DELETE FROM  $R$  USING  $B0, \dots, Bn$ 
WHERE  $R.A0=B0.A0$  AND ... AND  $R.An=Bn.An$ 

```

- **BlockCreateInsert**

Create the table of block in  $ORI$  formed by the same attribute values forming  $Bn$ .

```

INSERT INTO  $ResultTable\_i(A)$ 
SELECT  $ORI.A0, \dots, ORI.An$  FROM  $ORI, B0, \dots, Bn$ 
WHERE  $ORI.A0=B0.A0$  AND ... AND  $ORI.An=Bn.An$ 
ORDER BY  $A0, \dots, An$ 

```

### 3.3.2 Single Block Detection (Algorithm 2)

This algorithm describes how to find each dense block from the given relation. The main idea is to order the attribute values based on block density. We can iterate to select dimension and delete attribute values. Because of the order of attribute values, we can get the optimal number of attribute to remove where we could get maximum block density.

**Input** :  $R, Rn, MassR, density$   
**Output**: tables of attribute values forming a dense block  
CopyTable( $B, R$ )  
 $MassB \leftarrow MassR$   
CopyTable( $Bn, Rn$ )  
 $BestDensity \leftarrow DensityMeasure(MassB, Bn, MassR, Rn)$  # See Algorithm 3  
 $r, Best\_r \leftarrow 1$   
TableDropCreate( $OrderTable$ )  
**while** BlockNotEmpty( $B$ ) **do**  
    **for**  $n$  in range( $N$ ) **do**  
        DistinctAttributeValue( $Bn$ )  
    **end**  
    TableDropCreate( $AttValMassTable$ )  
    AttValMass( $AttValMassTable$ )  
     $i \leftarrow SelectDimension()$  # See Algorithm 4  
     $threshold \leftarrow MassB / MassB_i$   
    SelectValuesToRemove( $D_i$ )  
    CopyTable( $D_{static}, D_i$ )  
    **while** TableNotEmpty( $D_i$ ) **do**  
         $a, MassB_{a,i} \leftarrow FetchFirstRow(D_i)$   
        DeleteRows( $D_i$ ) # Delete all rows with same  $a$  and  $MassB_{a,i}$   
        DeleteRows( $B_i$ ) # Delete  $a$  from  $B_i$   
         $MassB \leftarrow MassB - MassB_{a,i}$   
         $density' \leftarrow DensityMeasure(MassB, Bn, MassR, Rn)$   
        InsertRow( $OrderTable$ ) # Insert  $a, i, r$  into  $OrderTable$   
         $r \leftarrow r + 1$   
        **if**  $density' > BestDensity$  **then**  
             $BestDensity \leftarrow density'$   
             $Best\_r \leftarrow r$   
        **end**  
    **end**  
    UpdateBlock( $B, D_i$ )  
**end**  
ReconstructBlock( $Best\_Bn$ )

**Algorithm 2:** find\_single\_block in D-CUBE

Details of special functions in Algorithm 2 are listed below:

- **AttValMass**

Compute attribute value mass  $M_{B(a,n)}$ . For each dimension  $dim$ , we calculate number of the tuples in  $B$  whose attribute value equals to  $a$ .

```
INSERT INTO AttValMassTable
SELECT dim, B.A_dim, COUNT(*) AS AttValMass
FROM B_dim AS A, B WHERE A.A_dim = B.A_dim
GROUP BY B.A_dim"
```

- **SelectValuesToRemove**

Select attribute values to be removed according to table *AttValMassTable* and *threshold*.  
Sort the table in an increasing order of attribute mass.

```
INSERT INTO  $D_i$  SELECT  $a\_value$ ,  $attrVal\_mass$ 
FROM AttValMassTable
WHERE  $dimension\_index = i$  AND  $attrVal\_mass \leq threshold$ 
ORDER BY  $attrVal\_mass$ 
```

- **FetchFirstRow**

Fetch the first row of the  $D_i$  table.

```
SELECT  $a\_value$ ,  $attrVal\_mass::text$ 
FROM  $D_i$  LIMIT 1
```

- **UpdateBlock**

Remove tuples in  $B$  whose attribute value  $A_i$  is in  $D_i$ .

```
DELETE FROM  $B$  USING  $D_i$ 
WHERE  $B.A_i = D_i.a\_value$ 
```

- **ReconstructBlock**

Reconstruct block  $Best\_B$  with attribute values that maximize block density. These attribute values should have order greater than  $Best\_r$ .

```
INSERT INTO  $Best\_Bn$ 
SELECT  $A.A_n$  FROM  $R$  AS  $A$ ,  $OrderTable$  AS  $B$ 
WHERE  $A.A_n = B.a\_value$  AND
 $B.order\_a\_i \geq Best\_r$ 
AND  $B.dimension\_index = n$ 
```

### 3.3.3 Density Measures (Algorithm 3)

This algorithm support three methods to calculate block density: Arithmetic Average Mass, Geometric Average Mass and Suspiciousness.

**Input** :  $MassB, Bn, MassR, Rn, method$   
**Output**: density of the dense block

```

if  $method=Arithmetic$  then
   $sum \leftarrow 0$ 
  for  $n$  in  $range(N)$  do
     $sum = sum + Mass(Bn)$ 
  end
   $density = MassB / sum * N$ 
end
if  $method=Geometric$  then
   $product \leftarrow 1$ 
  for  $n$  in  $range(N)$  do
     $product = product * Mass(Bn)$ 
  end
   $power = pow(product, \frac{1}{N})$ 
   $density = MassB / power$ 
end
if  $method=Suspiciousness$  then
   $ratio \leftarrow MassB / MassR$ 
   $density = MassB * (\log(ratio) - 1)$ 
   $ratioProduct \leftarrow 1$ 
  for  $n$  in  $range(N)$  do
     $ratioProduct = ratioProduct * Mass(Bn) / Mass(Rn)$ 
  end
   $density = density + MassR * ratioProduct - MassB * \log(ratioProduct)$ 
end

```

**Algorithm 3:** DensityMeasure

### 3.3.4 Dimension Selection (Algorithm 4)

This algorithm support two policy to select dimension for attribute value removal in Algorithm 2. They are select\_dimension by cardinality and select\_dimension by density.



**Input** :  $Bn, N, policy$   
**Output**: a dimension in  $[N]$   
**if**  $policy=cardinality$  **then**  
     $dim \leftarrow -1$   
     $maxMass \leftarrow -1$   
    **for**  $n$  **in**  $range(N)$  **do**  
         $currMass = Mass(Bn)$   
        **if**  $currMass > maxMass$  **then**  
             $maxMass = currMass$   
             $dim = n$   
        **end**  
    **end**  
**end**  
**if**  $policy=density$  **then**  
     $BestDensity \leftarrow -\infty$   
     $dim \leftarrow 1$   
    **for**  $i$  **in**  $range(N)$  **do**  
        **if**  $BlockNotEmpty(B_i)$  **then**  
             $threshold \leftarrow MassB / MassB_i$   
             $SelectValuesToRemove(D_i)$   
             $delta \leftarrow DcubeSum(D_i)$   
             $MassB' \leftarrow MassB - delta$   
             $B'_i \leftarrow UpdateBlock(B_i, D_i)$   
            Switch  $B'_i$  and  $B_i$   
             $density' \leftarrow DensityMeasure(MassB, Bn, MassR, Rn)$   
            Switch  $B'_i$  and  $B_i$   
            **if**  $density' > BestDensity$  **then**  
                 $BestDensity \leftarrow density'$   
                 $dim \leftarrow i$   
            **end**  
        **end**  
    **end**  
**end**

**Algorithm 4:** select\_dimension

Details of special functions in Algorithm 2 are listed below:

- **DcubeSum**

Calculate sum of attribute value masses in  $D$

SELECT SUM( $attrval\_mass$ ) FROM  $D$

### 3.3.5 Bucketize Time

Time attribute is very special in finding dense block problem. For the purpose of grouping data into meaningful time slots, we use bucketize time method to construct time buckets

by hour. The main idea is to update relation table after loaded from csv file by ignoring minutes with regular expression.

UPDATE  $R$  SET  $time\_stamp$  = SUBSTRING( $time\_stamp$  from '.\*:')

### 3.3.6 Unit Tests

We construct a test dataset of 798 rows by manually select dense blocks labeled as 'smurf' in 'darpa\_with\_label' and the rest of the tensor have tiny diversity. The table 4 shows description of these arguments.

Function Name	Expected Result
TableDropCreate	Drop old table with same name and create a new empty table.
LoadTableFromFile	Load relation table from csv file. Get a table of 798 rows.
CopyTable	A new table same as the source table.
DistinctAttributeValue	Distinct attribute values of the relation table. Get three tables with 420, 79 and 21 rows.
Mass	Get total row number of the table. Should return 798 for relation table.
DeleteRows	All tuples under condition are removed
InsertRow	A new row is appended in the table
DeleteFromBlock	All tuples with specific attributes are deleted.
BlockCreateInsert	Get result dense block table.
AttValMass	Get a table of attribute value, dimension and attribute value mass.
SelectValuesToRemove	Get a table of attribute value and attribute value mass. No attribute value mass is greater than threshold.
FetchFirstRow	Get attribute value $a$ and corresponding attribute value mass
UpdateBlock	A new block table whose tuples don't have same attribute values in table $D$ .
ReconstructBlock	Get a dense block table.
DensityMeasure	Three density values that consistent with the manual calculation results.
DimensionSelection	Get a dimension for removal. According to our testset, the first selection should return dimension 0.

Table 4: Table of unit tests

## 4 Experiments

*Under Construction...*

### 4.1 Design of experiments

- Manually generate synthetic dense blocks in tensors for sanity check before running on large real-life datasets.
- Design a set of different *Number of dimensions*  $k$  (e.g.:  $k = 1-10$ ), run them on the same dataset, record and analyze performance of the algorithm. Specifically, analyze the qualities of different ranked dense blocks as well as their wall clock time needed.
- Run the algorithm on datasets with different *Number of dimensions*  $N$  (e.g.:  $N = 1, 2, 3, 5, 10$ , etc.), analyze their run-time complexity, accuracy, etc.
- Design controlled tests and conduct experiments on the 6 possible combinations of two dimension selection methods (Density and Cardinality) and three density measurement methods (Arithmetic, Geometric and Suspiciousness). Compare the result and interpret the findings in multiple aspects.
- With the exact same parameter setting, run the optimized algorithm with indexing and the baseline, record *Elapsed Time*, statistics of found dense blocks including mass, density, etc. Compare and analyze the result in terms of computational efficiency, disk space usage, etc.
- Using different real-world data like *Amazon Rating Dataset*, *Yelp Rating Dataset*, *Air-force TCP Dump*, run the D-Cube algorithm for anomaly detection. Analyze the correctness of the found dense blocks with regard to the labeled data and interpret the relation and other observations between them.

## 5 Conclusions

*Under Construction...*

### 5.1 Preliminary Result

We ran our implemented D-Cube algorithm on the Darpa TCP Dump datasets. Also, we bucketize all timestamps by hour. The following tables present several statistics and properties on the found dense blocks, as well as global features like elapsed run time. Further experiments and analysis will be carried out in the final phase of our project. Today, we just simply present the statistics for a bird view of the big picture.

### 5.1.1 Overall Statistics of Dense Blocks

We ran the D-Cube algorithm to find five dense blocks using  $k = 5$  and tested different combinations of methods/metrics.

#### Abbreviation

- **Ari:** "Arithmetic Average Mass"
- **Geo:** "Geometric Average Mass"
- **Susp:** "Suspiciousness"

Table 5: Overall Statistics of Dense Blocks

Density Measure	Dimension Selection	Results			
Ari	Cardinality	block_index	density	entrycount	elapsed_time
		0	366652	733304	508.489068031
		1	175602	175602	442.847835064
		2	249806.5	499613	431.193703175
		3	167476.090909	614079	437.219784975
		4	59406	79208	353.711796999
Ari	Density	block_index	density	entrycount	elapsed_time
		0	366652	733304	820.505131006
		1	218625.75	291501	746.266477823
		2	126192.272727	462705	876.399868011
		3	175602	175602	790.000819921
		4	147491.4	245819	816.553550005
Geo	Cardinality	block_index	density	entrycount	elapsed_time
		0	461952.572786	733304	468.892929077
		1	175602	175602	394.351580858
		2	314736.467751	499613	418.060773849
		3	185975.932754	614079	377.054594994
		4	62867.4312621	79208	349.504483938
Geo	Density	block_index	density	entrycount	elapsed_time
		0	495302.873497	846956	983.86125493
		1	173213.086794	499633	808.176931143
		2	152576.554959	277250	789.166759014
		3	175602	175602	763.752912998
		4	132015	132015	704.104015112

Table 6: Overall Statistics of Dense Blocks (Continued)

Density Measure	Dimension Selection	Results			
Susp	Cardinality	block_index	density	entrycount	elapsed_time
		-----+	-----+	-----+	-----+
		0	19503374.6066	1030835	588.014246941
		1	5300815.96286	277366	527.220749855
		2	10710059.6369	1173432	497.591800928
		3	10792554.5146	1221207	416.026752949
		4	4825039.41155	568282	355.364156008
Susp	Density	block_index	density	entrycount	elapsed_time
		-----+	-----+	-----+	-----+
		0	27044399.2325	1357245	1105.47115088
		1	22717743.9624	2316286	964.843622923
		2	4953487.37897	319169	726.697937965
		3	9390605.60882	1342592	756.514394999
		4	7110081.85708	641552	575.360764027

## References

- [1] Meng Jiang, Alex Beutel, Peng Cui, Bryan Hooi, Shiqiang Yang, and Christos Faloutsos. A general suspiciousness metric for dense blocks in multimodal data. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 781–786. IEEE, 2015.
- [2] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. Graph analytics using vertica relational database. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1191–1200. IEEE, 2015.
- [3] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: an efficient analysis platform for large graphs. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(5):637–650, 2012.
- [4] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM’09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.
- [5] Kijung Shin, Bryan Hooi, and Christos Faloutsos. M-zoom: Fast dense-block detection in tensors with quality guarantees. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 264–280. Springer, 2016.
- [6] Kijung Shin, Bryan Hooi, Jisu Kim, and Christos Faloutsos. D-cube: Dense-block detection in terabyte-scale tensors. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 681–689. ACM, 2017.

# A Appendix

## A.1 Member Contributions

We have finished *Phase1: Literature* and *Phase2: SQL Implementation* so far, and we listed the contributions of each group member as follows.

### Haoming Chen:

- Literature research and analysis of three papers: (1) D-Cube: Dense-Block Detection in Terabyte-Scale Tensors[6]; (2) Graph analytics using vertica relational database[2] and (3) Pegasus: A peta-scale graph mining system implementation and observations[4].
- Implemented sub-modules in D-Cube using SQL and Psycopg including *Algorithm2: find\_single\_block* and two methods of dimension selection in D-Cube, *Algorithm4: select\_dimension by cardinality* and *select\_dimension by density*.
- Debugging and testing of D-Cube implementation, run the algorithm on three different methods of density measurement using *Density* as the dimension selection method, extract the result and statistics from database and reach preliminary conclusions.
- In charge of components of project report, including *Abstract*, *Introduction-Motivation*, half of *Survey*, *Experiment Design* and *Plan of activities*.

### Xinrui He:

- Literature research and analysis of three papers: (1) D-Cube: Dense-Block Detection in Terabyte-Scale Tensors[6]; (2) Graph analytics using vertica relational database[2] and (3) Pegasus: A peta-scale graph mining system implementation and observations[4].
- Implemented sub-modules in D-Cube using SQL and Psycopg including three metrics of density measures *Algorithm1: D-Cube Overall Structure*, *Algorithm3: Density Measures with Arithmetic Average Mass*, *Geometric Average Mass*, *Suspiciousness*, and *Bucketize time*.
- Debugging and testing of D-Cube implementation, run the algorithm on three different methods of density measurement using *Cardinality* as the dimension selection method, extract the result and statistics from database and reach preliminary conclusions.
- In charge of components of project report, including detailed introduction and showcase of D-Cube algorithm and SQL in *Methods* and *Unit tests*.

## A.2 Plan of activities

For the later phases of the project, we plan to perform the following tasks:

- Full inspection and debugging of the implemented D-Cube algorithm to make sure the algorithm works correctly before we step into the experiment process.
- Based on the initial implementation, try to optimize the algorithm in terms of computational efficiency, disk space usage, etc. Possible optimization may involve indexing in relational databases. Test the optimized algorithm with the baseline to check the difference of performance.
- Compare two implementations of generating intermediate tables mentioned in *Project Description T3* and give conclusions upon analysis.
- Implement the evaluation method of *ROC Curve* and report statistics such as mass, density, etc. of the dense blocks. Compared the found blocks with the labeled data and analyze their associated meanings in real-world scenarios. Present different applications of the algorithm on pattern recognition, anomaly detection, etc.
- Design controlled tests to conduct experiments on various parameter settings (e.g. different combinations of  $k$  and metrics). Compare and interpret the findings in multiple aspects including accuracy, computational efficiency, etc.