Dense Subtensor Mining using SQL

Megha Arora
School of Computer Science
CMU
marora@andrew.cmu.edu

Prakruthi Prabhakar
School of Computer Science
CMU
prakrutp@andrew.cmu.edu

June 24, 2017

1 Introduction-Motivation

With the increasing interaction between users and automated systems, detection of dense blocks in big data has several practical applications. One such example would be fraud detection, where it can help to find groups of fraudulent users who review similar set of products in a lockstep behavior. In such a scenario, every user and his attributes like review, time-stamp, etc. could be modeled as a tensor. In this project, we implement D-CUBE [4], a disk-based algorithm, which identifies dense blocks in tensors. Another example application of this algorithm would be detection of network attacks, where lockstep behavior of suspicious connections between different IP addresses in a short amount of time could indicate a network attack. Many earlier solutions to the problem of dense block detection assume that tensors are small enough to fit in memory. With growing amounts of data, it becomes imperative to design algorithms which can handle disk-resident tensors. One of the salient features of D-CUBE is that it is memory efficient and uses disk-resident tensors to allow processing data too large to fit in main memory.

2 Problem Definition

Top-k Dense Block Detection using D-CUBE algorithm:
Given: multi-aspect data R, number of attributes N, number of dense blocks to be obtained k, a density measure ρ, and a dimension selection policy P.
Output: top-k distinct dense blocks in relation R with highest density measured in terms of ρ using an efficient implementation of the D-CUBE dense block detection algorithm.

3 Survey

We list the papers that each member read, along with their summary and critique. Table 1 gives a list of common symbols we used in our summary.

Symbol	Definition
K	number of modes in the data
T	number of iterations
N	K-length vector for the size of each mode
k	number of partitions for GBASE graph partitioning
$l^{(p)}$	size of the p^{th} GBASE partition
R	Relation representing a tensor
N	Number of dimension attributes in relation R
k	Number of dense blocks to be found by the algorithm
L	Number of distinct values of attributes in a dimension
$\|R_n\|$	Number of distinct values of attributes in n^{th} dimension
$\|R\|$	Size of relation
UDF	User Defined Function
I/O	Input / Output

Table 1: Symbols and definitions

3.1 Papers read by Megha Arora

A General Suspiciousness Metric for Dense Blocks in Multimodal Data [2]

The first paper is based on a general suspiciousness metric for dense blocks in multimodal data by Jiang et al.

- ***Problem Definition*:**
  Given a K-mode dataset (tensor) X , with counts of events (that are non-negative integer values), and two subtensors $Y1$ and $Y2$, which is more suspicious and worthy of further investigation?

- ***Main idea*:**
  The paper has three key contributions. It -
  1. establishes a **list of axioms** based on density, size, concentration, contrast and multimodality, which a suspiciousness metric should satisfy.
  2. introduces an easy and fast to compute **suspiciousness metric** that satisfies all the axioms.

3. proposes the **CrossPot algorithm** to identify and compare dense regions in multimodal data.

The axioms exhaustively cover all aspects which can help distinguish dense blocks in tensors. It also shows how the existing metrics like mass, average degree and SVD based techniques fail to satisfy all the axioms. The novel metric they propose **provides scores for blocks** (not just nodes), **works on a subset of modes**, and **significantly improves results over conventional techniques**. The metric for a multimodal block is the negative log likelihood of the block under an ErdősRényi-Poisson model.

CrossPot is a robust and efficient local search based algorithm, which **identifies and ranks dense blocks in tensors**. The authors also conducted experiments on real-world datasets to show the utility of the proposed approaches. They were able to find both high and low-order dense blocks in a synthetically generated dataset and a dataset from the social networking platform Tencent Weibo. CrossPot runs in $O(T \times K \times (E + NlogN))$ time and improved the F1 score by as much as 68% for the latter dataset. E refers to the number of non-zero values in the data and more precisely, N is the maximum number of unique entries for any mode. They observed faster run-time, and high precision and recall values for analysis on the synthetic dataset as well. The algorithm can be **parallelized** to run on different machines at the same time.

- ***Use for our project*:**
  The paper sets the **right motivation** for the problem of identifying dense sub-tensors that we are tackling in the project. The axioms provide an **in-depth understanding of how these dense blocks compare** with the other parts of the dataset, which will help us to improve and efficiently debug our implementation. The CrossPot algorithm establishes an intuitive line of thought which we can pursue while trying to identify dense blocks. The experiments give us an overview of the kind of **tests and analysis we can perform** over our implementation. Their methodology also achieves good run time bounds and is scalable (theoretically), which will also be one of our primary goals.

- ***Shortcomings*:**
  The biggest shortcoming of the approach is that it assumes that the **data will be in main memory**, which is rarely the case in large-scale data analysis. CrossPot uses values for all other modes while doing 'AdjustMode' processing on a particular node. This essentially means that even though the run-time is quasi-linear in N and linear in terms of the non-zero values, running the algorithm on a large dataset will give worse performance as a lot of time would be spent in continuously swapping data between memory and disk. The paper does not talk about these numbers.

GBASE: an efficient analysis platform for large graphs [5]

The second paper presents the GBASE analysis platform for large graphs by Kang et al.

- ***Problem Definition*:**
  How to build a general graph management system in a parallel, distributed settings to support billion-scale graphs for various applications while addressing storage-related and algorithmic challenges?

- ***Main idea*:**
  The paper introduces **GBASE, a storage and analysis platform** for graphs exceeding billions of nodes and edges. It naturally fits in a **distributed** and **parallelized** setting for mining graphs. GBASE also implements widely used node-based and edge-based algorithms including k-step neighbours, obtaining induced graphs, k-step egonets, cross-edges, and K-cores. Even though it stores a compressed version of the adjacency matrix, it also supports queries based on incidence matrix of the graph.

  GBASE uses the idea of first dividing the adjacency matrix into **homogeneous blocks** or **clusters** (i.e. subgraphs corresponding to each block is either very dense or very sparse) for storage and indexing, and then using the Gap Elias-γ encoding for **compression**. The estimated storage space turns out to be $\sum_{1\leq p,q\leq k} l^{(p)}l^{(q)}H(d^{(p,q)})$, where $H(.)$ refers to Shannon's entropy and d refers to the density of the partition. The blocks are placed after combining multiple blocks into files, and storing them using **grid placement** as opposed to vertical or horizontal placement, ensuring only $O(\sqrt{K})$ accesses for relevant files during query processing. Both global and targeted queries are processed using **GBASE's query search engine** built over Hadoop. All queries are internally implemented as matrix multiplication operations. **Grid selection for targeted queries** fetches only relevant files (reduces running time) for decoding and evaluating the result of the query. Induced subgraph queries use incidence matrix instead. The authors have also provided a routine to implement such queries using the adjacency matrix.

  Finally, the paper justifies the claims of improved performance and space by using GBASE on real-world datasets. Size reduction achieved by GBASE's **clustered and compressed block (CCB) encoding** was as high as 43x for a particular dataset. Despite an overhead of decoding compressed information, indexing time and query response time (using grid selection for targeted queries) turned out to be notably better for the CCB approach. The authors also show that the analysis and indexing are both **machine** and **edge scalable**.

- ***Use for our project*:**
  After reading this work, we are more equipped at **using SQL for graph mining** as the exposure we had with SQL was mostly based on conventional databases where data is stored in the form of tables. The paper does a good job at explaining how graph operations can be translated to **matrix multiplication operations**. This will help us to implement dense block identification algorithms in a way such that majority of

operations will be performed under the hood of efficient matrix multiplications. Some of the ideas discussed in the paper like grid selection help in getting us accustomed to analysis at a scale, where **most of the data won't fit in main memory**, which is what we would want to account for in our implementation too.

- ***Shortcomings*:**
  One shortcoming of GBASE is that it **doesn't support graph updates**, which would not affect static analysis, but at the same time most real-world graph based settings are dynamic, where nodes become inactive, and new nodes are added leading to new connections. Improvement for **global queries** in not very significant as **grid selection cannot be utilized** and all blocks have to be decoded and then evaluated. GBASE supports only eleven different types of graph queries; it doesn't provide a direct interface for other types of queries. It **doesn't support heterogeneous and nested queries**. Moreover, it cannot implement queries where translation to matrix multiplication operations is not possible, as that is all the query engine can process internally.

PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations [6]

The third paper introduces PEGASUS, a graph mining system by Kang et al.

- ***Problem Definition*:**
  How to unify seemingly different graph mining tasks, via a generalization of matrix-vector multiplication and come up with an effective implementation of matrix-vector multiplication with several optimizations?

- ***Main idea*:**
  The paper introduces a **Peta-Scale Graph Mining library called PeGaSus** that can handle analysis on graphs with billions of nodes and edges. First, the paper discusses how various graph mining operations can be **unified** and implemented using **Generalized Iterative Matrix-Vector (GIM-V) multiplications**. Most of the prior implementations for these operations required data to reside in shared memory. PeGaSus uses **Hadoop's MapReduce framework** for graph-based computations. The authors shows how GIM-V is implemented to use Hadoop's parallel processing framework. Further, the paper also covers in depth how PageRank, Random Walk with Restart, Diameter Estimation, and Connected Components analysis can be performed using the GIM-V sub-routine.

  The authors elaborately discuss different methods to run GIM-V faster. The first approach uses **block multiplication (BL)**, where each block contains only non-zero elements. Matrices and vectors are joined block-wise, which is much faster than the conventional element-wise implementation. Using blocks also facilitates compression

and faster sorting during the shuffling stage of MapReduce. The second optimization uses **clustered edges** which is useful when **combined with block encoding (BL-CL)** as the amount of data to be stored reduces. The last optimization is **iterating along diagonal blocks (DI)** to reduce the number of iterations (no change in runtime for each iteration). Comparing these optimizations, the authors conclude that BL-CL performs best, running over 2.93 times faster than the naive implementation. The implementation is both machine and edge scalable. The paper also presents analysis and results of using PeGaSus on real-world datasets and showing how it can help in finding interesting patterns and outliers.

- ***Use for our project*:**
  This work is very relevant to our project. Using the **GIM-V sub routine** for graph algorithms can help us in making our dense block mining implementation faster and more effective. The examples shown in the paper enable utilizing parallel frameworks like MapReduce for graph mining tasks and show us the way to build algorithms that can run in **distributed** and **parallelized setups**. The analysis on real-world graphs sets the right motivation for our problem, highlighting the possibilities regarding the kind of insights that can be drawn from our project and some **potential applications**.

- ***Shortcomings*:**
  It is not clear from the paper how **updates in the graphs** can be handled. Updates in the graph would imply decoding the modified blocks, updating the values, encoding, and re-indexing. It could also entail adding new blocks. For the clustering step, updates would mean decoding multiple blocks to re-cluster efficiently, which would be expensive. The paper only covers some particular graph mining algorithms. It will be **hard to implement most other algorithms using the GIM-V subroutine**.

3.2 Papers read by Prakruthi Prabhakar

M-Zoom: Fast Dense-Block Detection in Tensors with Quality Guarantees [3]

- ***Problem Definition*:**
  Detection of k-dense blocks: Given a relation (multi-aspect data) R, the number of blocks k and a density measure ρ, find k distinct dense blocks from R with the highest densities defined in terms of the density measure ρ.

- ***Main idea*:**
  The paper proposes a technique to identify **multiple dense blocks** in tensors with and without size bounds on the blocks identified. The method supports several density measures (in accordance with Density Axiom) and provides a lower bound of $(1/N)$ of maximum density under the assumption that arithmetic average mass is used as a density metric.

The data is modeled as a relation with N dimensional attributes and an associated score value for every tuple in the relation. The algorithm works by repeatedly finding a single dense block which satisfies the size bounds until k required dense blocks have been found. Every dense block is identified by choosing the block with the highest density among blocks generated by removing tuples whose attribute has the least mass. After the dense block is identified, its tuples are removed from the relation before the next iteration in order to prevent the same block being identified again. The returned blocks could have overlapping tuples. The algorithm also provides an efficient way of finding the attribute with minimum mass in a greedy fashion using min-heaps.

The algorithm is **scalable** in both time and space complexity and scales linearly (or sub-linearly) with the number of blocks to be found, the number of attributes, the size of the relation and the cardinality of distinct attributes in the relation ($O(kN|R|log|R_n|)$ time complexity and $O(kN|R|)$ space complexity). This is proven both theoretically and experimentally in the paper and has been shown to be **114x** faster than other state-of-the-art methods. M-Zoom has also been shown to identify **diverse set of dense blocks** (diversity measured in terms of average dissimilarity between different pairs of blocks).

Experimental results have indicated that M-Zoom provides **the best trade-off between run-time and accuracy** of identifying dense blocks on 8 different datasets, in comparison to other state-of-the-art methods. It is also proved to be very effective in identifying dense blocks in **real data** (edit wars and bot activities in Wikipedia and network attacks in Airforce Data).

- ***Use for our Project*:**
  This paper introduces Dense subtensor mining and explains all the relevant concepts to solve the problem. If the datasets we use fit in memory, this could provide an efficient implementation idea for solving the problem. It helped **understand fundamental concepts** like various density and suspiciousness metrics, concepts of size, volume and mass of the relation. It also guided in thinking about **efficient implementation** of the algorithm by providing examples like greedy min-heap approach to identify attributes and efficient storage of order and iteration number to identify the required snapshot rather than storing a list of snapshots. The paper also signifies the need for **exhaustive experimentation** on several datasets and analysis of various metrics like scalability and accuracy, which sets the principles for our implementation.

- ***Shortcomings*:**
  The biggest drawback of M-Zoom is that it assumes that the tensors are **small enough to fit in main memory**. This limits the application of M-Zoom to large datasets. It also doesn't support implementation and execution of the algorithm in a **distributed framework** like Hadoop MapReduce. The accuracy guarantee is provided only for

using a specific density metric and is also not available when size bounds are provided.

D-Cube: Dense-Block Detection in Terabyte-Scale Tensors [4]

- ***Problem Definition*:**
  Top-k Dense Block Detection: Given a relation (multi-aspect data) R, number of dense blocks to be obtained k and a density measure ρ, identify top-k distinct dense blocks in relation R with highest density measured in terms of ρ.

- ***Main idea*:**
  D-Cube is a disk-based algorithm to identify multiple dense blocks in tensors, which can also be made to execute in parallel on multiple machines. D-Cube requires **1600x** less memory and can handle **1000x** larger data than other state-of-the-art in-memory algorithms. Under specific conditions (maximum cardinality policy for choosing attributes to remove and arithmetic average mass as the density metric), D-Cube guarantees that the block found is at least (1/N) of the optimum.

  The data is modeled as a relation with N dimensional attributes and an associated score value for every tuple in the relation. The algorithm works by finding k dense blocks one by one, whose tuples are removed from the relation before the next iteration to prevent identification of the same block again. D-Cube could potentially detect blocks with overlapping tuples in them. In order to identify a single dense block, D-Cube sets a particular dimension to be removed (based on maximum cardinality or density policy), identifies all attributes in that dimension with mass less than the average mass and removes them from the current relation in a **single scan**. This **minimizes the number of disk accesses** in the algorithm as one scan removes **multiple attribute value tuples**.

  The algorithm provides a theoretical worst-case time complexity of $O(kN^2|R|L)$ and space complexity of $O(\sum_{n=1}^{N}|R_n|)$. Experimentally, this has been shown to be pessimistic and it **scales linearly** or sub-linearly for various parameters. The paper also provides a **MapReduce implementation on Hadoop** and has been shown to increase speed-up linearly with more machines.

  Experimental results have shown that the algorithm achieves both **optimal run time** and **good algorithmic accuracy** on ten different datasets in comparison to other state-of-the-art methods. It is also been proven to be effective on **real data** on problems like detection of network attacks in TCP dumps and fraud detection in Ratings data.

- ***Use for our Project*:**
  This paper provides a memory efficient technique for dense tensor mining problem achieving both speed and accuracy. For the large datasets in our project, we will

be mainly exploring this technique. The paper explains the algorithm and its sub-problems in great detail and helped us understand why and how it **minimizes the number of disk I/O** at every step. The ability to execute our **implementation on distributed machines** will also be a key aspect we will be focusing on as described in the paper. A thorough experimentation and analysis on accuracy, speed, scalability and effectiveness has been performed in the paper and we will use this as a benchmark for comparison of our results on all these different metrics.

- ***Shortcomings*:**
  The paper doesn't discuss the **diversity of the blocks** that are detected by this approach (as done by the M-Zoom paper). Since the algorithm could identify dense blocks with overlapping tuples, it would be interesting to see the effectiveness of this approach in detecting diverse blocks. It also assumes that the sets of distinct attribute values of the relation can be stored **in memory** and used by the algorithm. If most of the tuples in the data contain distinct attribute values, this assumption could become a bottleneck for storing as well as randomly accessing elements quickly. The worst case time complexity grows **quadratically in the number of dimensions**, which could become a bottleneck for very high dimensional datasets. The algorithm doesn't take in **size bounds** on the dense blocks as inputs.

Graph Analytics using the Vertica Relational Database [1]

- ***Problem Definition*:**
  Given a relational database, analyze its usefulness as a platform for graph analytics and show examples of conversion of graph analytics into relational database queries. Specifically, analyze Vertica relational database for graph analytics.

- ***Main idea*:**
  This paper presents column-oriented Vertica relational database as an effective platform for graph analytics. It identifies the relevance and benefits of **relational databases** in solving traditional **graph analytics** problems as well as combination of graph and **relational analysis problems**.

  Graph data being collected and stored in relational databases, need for performing operations on smaller subsets of data and presence of many attributes along with vertices and edges in real data facilitate using relational databases for graph analytics. Many operations on graphs like computing statistics on vertices and edges, triangle counting, single source shortest path and the like can be **performed using scans, joins, selection and aggregation operations**, for which relational databases have been customized for.

  The paper presents Giraph execution pipeline as an example for vertex-centric models and analyzes its drawbacks like **fixed query plan**, inflexibility in modifying query

operators and inability to cater to broader multigraph analysis. It then presents Vertica and describes how vertex-centric queries can be translated into **SQL queries** and run. Vertica as a relational database eliminates the need for using messages to maintain consistency. The paper also describes various query optimization techniques like **update-in-place** for smaller set of updates and joins with only updated values in the table to prevent exploring entire data. Vertica also supports **projections and segmentations** to optimize query, **efficient joins** with direct operation on compressed data without decoding and **query pipelining** to avoid repeated disk accesses at every stage of the program. Shared memory extensions of Vertica allows users to implement **special table UDFs** at cost of higher memory footprint and thereby facilitating reduced disk I/O.

The paper presents experimental analysis on three major graph algorithms - PageRank, Single Source Shortest Path and Connected Components using highly optimized SQL queries on Vertica. It is shown that Vetica has very **less memory footprint** in comparison to Giraph and GraphLab, very less read I/O due to efficient operations on compressed datasets, **high write I/O** because of writing output of every operation to disk (which can be avoided using shared memory UDFs as shown in the paper). It also presents results on mixed graph and relational analyses and shows that vertica is **time-efficient** than Giraph on subgraph projection, aggregation and join operations. The paper concludes by showing other benefits of Vertica on problems such as identifying pairs of nodes with large number of common neighbors and finding nodes which act as intermediate node between two disconnected components in graphs.

- ***Use for our Project*:**
  This paper helped us understand how graph analytics and problems on graph based data can be translated to **simple yet effective SQL queries** using aggregations, projections and joins. It also helped me understand the effectiveness of relational databases and how the resilience and other well established features could be exploited for developing powerful graph analytics over them. We will be using SQL on real data for our project and this helped me to think about implementation as well as potential difficulties of the same. **Input and output I/O, efficiency and memory footprint** are some key components which the paper has analyzed and shown optimizations for, which we will also be exploring for our problem.

- ***Shortcomings*:**
  The paper explains the effectiveness of SQL and relational databases for solving graph analytics problems. However, it does not explain the limitations of the same in elaborate detail. There could be some problems which could be **naturally hard** to program as aggregations and joins in SQL and a better understanding of the limitations would help in this regard. If the graph algorithm has **large number of intermediate operations**, there could be potential increase in output I/O, which could pose as a serious limitation for Vertica based systems. The shared memory UDFs will not be able to

help much if the datasets are large due to increased memory footprint.

4 Implementation

In this project, we implement D-CUBE for dense block detection in terabyte-scale tensors [4]. We use PostgreSQL to implement the algorithm. We assume that all the data is stored on disk and is processed using SQL. We use psycopg2 module in python to build a wrapper functionality for our module. Metadata and intermediary computations like attribute value mass and order values are stored and processed from an SQL table. Hence, our implementation has an $O(1)$ **memory footprint**. We also ensure that we drop all intermediary tables which are created in our implementation.

We implement the algorithm in a modular manner. We describe each of the modules below.

4.1 Input and Output

Required input - 1) Database parameters, 2) input and output file paths, and 3) algorithm parameters
Output - 1) k-dense blocks, and 2) their densities

In this project, input is taken through a constants.py file. Database parameters include database name, username, pghost and pgport. Input is provided as a comma-separated file along with the counts (mass of each tuple). The number of attributes in the tensor along with the delimiter is also provided as input parameters in the same file. Execution creates two output files - i) Densities of k-dense blocks obtained (file path provided by OUTPUT_LOCATION parameter) and ii) dense blocks with block number (file path provided by OUTPUT_LOCATION_BLOCKS parameter). The block number ranges from 0 to k-1, with 0 indicating the first dense block found by the algorithm. If the algorithm fails to find as many as k-dense blocks, it returns the number of dense blocks it found along with their densities. The number of desired dense blocks (k) is also provided as an input parameter in the same file. All the associated code is provided in the 'src' directory of our project.

4.2 Algorithm 1 - Main Module

This module implements the overall structure of D-CUBE algorithm. The objective of this module is to obtain top k-dense blocks, given a relation R, a density measure ρ and the number of dense blocks to be found k. The entire relation R is copied to another table $R_{original}$. The distinct values for every attribute is computed from this table. The algorithm then finds dense blocks one at a time. Once we obtain a dense block B, we remove the tuples of B from the relation R. Hence, every subsequent dense block is found on the modified relation R. The actual block is then extracted from $R_{original}$ from the attributes present in

the single dense block. This allows the algorithm to find overlapping blocks. All operations on the relation like copying tables, finding distinct values of each attribute and removing tuples with specific attribute values are performed using SQL statements. We also ensure that if the relation becomes empty before finding the required k dense blocks, we return the total number of dense blocks found by the algorithm.

4.3 Algorithm 2 - Finding Single Dense Block

The objective of this module is to find a single dense block given a relation R, using a density measure ρ and a dimension selection policy P. At the start of the function, the relation is copied to create another relation B. After computing the initial density of the relation using the input density measure, the algorithm removes tuples from B repeatedly by finding attribute values for which mass is less than the average by selecting a specific dimension using the given policy. All the attribute values in this dimension are then ordered in increasing order of mass. These values are removed one after the other by updating the mass and set of distinct values for that attribute. During this update, new density is computed and *order* is updated if the new density is greater than the one previously obtained. Order maintains the attribute values and the iteration when the density is maximized by removing a particular attribute value. This computation is repeated for every attribute value in every dimension. A single dense block is then reconstructed by fetching the tuples from the input relation R, for which the order values of each attribute is greater than or equal to the iteration when the density was maximum. As in algorithm 1, all operations in this module are performed using SQL queries. Sorting of attribute values based on mass is performed using the SQL ORDER BY operation. Dimension selection and density computation are performed by separate modules which are described in the sub-section below.

4.4 Density Computation

In this project, we support three density measures:

- Arithmetic Average Mass (A)
- Geometric Average Mass (G)
- Suspiciousness (S)

These measures are implemented as described in [4]. Arithmetic average mass is computed as the ratio of mass of block B to the arithmetic mean of the cardinalities of all attributes. Geometric average mass is computed as the ratio of mass of block B to the geometric mean of the cardinalities of different attributes. The suspiciousness measure computes the degree of suspiciousness of a block based on ratio of mass of the block to the mass of the entire relation and the ratio of product of cardinalities of attributes for the block to product of cardinalities of the attributes for the entire relation.

4.5 Dimension Selection Policies

In this project, we implement two policies for selecting the dimension from which attribute values are removed:

- Cardinality (C)
- Density (D)

These policies are implemented as described in [4]. According to policy C, the dimension with the highest cardinality is returned. The cardinalities of each dimension is computed in SQL using the DISTINCT and COUNT functionalities. According to policy D, the dimension which results in the largest increase of density when its attribute values are removed is returned. For every dimension, if the cardinality is non-zero, the attribute values with mass less than the average is determined. The new density with these attribute values removed is then computed. The dimension which results in the largest increase in density is then returned. All the required functionalities like removing specific attribute values and computing attribute values masses are performed using SQL commands on tables.

5 Optimizations

In this section, we describe several optimization techniques we experimented with in our implementation. We also analyze all the techniques and report the performance for each in this section.

5.1 Indexing

We create different indexes for the tables in our implementation using PostgreSQL. We start out with no indexing and experiment with indexing one table at a time and proceeding incrementally to see relative performance gain. There are two primary strategies we try for indexing each table: 1. Index on different attributes jointly, 2. Create separate indexes for different attributes in a table. All the results here were obtained by running the code on 5000 random rows from the DARPA dataset. The tables we index and exact methodology is explained below -

- **Block and Original Table:** We observed that the individual attributes of the block and the original table are accessed repeatedly by the algorithm in SQL statements which had a *WHERE* clause. Hence, we create three different indexes on the three attributes for both Block B and the original relation R.
- **Order:** Algorithm 2 uses a table called *order* which maintains the order for different attribute values in each column. The attribute value and the corresponding column is accessed in a *WHERE* clause to identify the attributes to be removed. We create a joint index on attribute value and dimension (column number) for this table, since they are always referenced together.

- **Distinct Value for Attributes:** This table maintains the distinct values for each attribute and their respective masses. We observed that attribute values along with the column number are referenced together. We thus create a joint index on these two attributes of this table.

5.1.1 Results and Analysis

All results in this section pertain to runs on 5000 random rows of the DARPA dataset. We initially observed significant variance in run-time of different methods. Each experiment is repeated three times to capture the difference in performance accurately. Each experiment runs different indexing methods for all combinations of dimension selection policy (Cardinality, C and Density, D) and density computation methods (Arithmetic, A, Geometric, G and Suspiciousness, S). We run the different indexing strategies at the same time (one after the other) to ensure comparable machine load for accurate results.

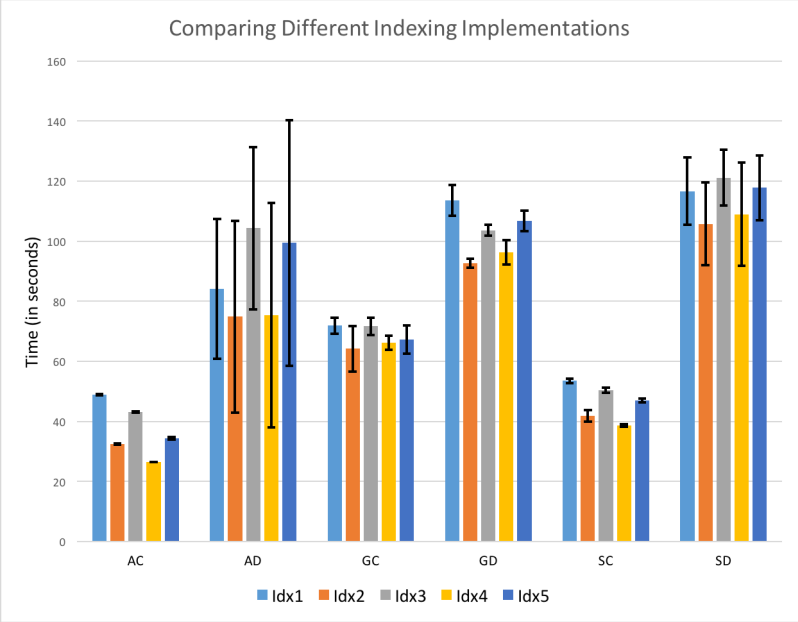

Figure 1: Run-time (in seconds) comparison for different indexing strategies along with deviation (error bars) for different combinations of density measure and dimension selection policy. Idx1 represents no indexing. Idx2 represents separate indexes on the original table and block table for three attributes. Idx3 represents joint index on the three attributes of the original relation and the block table. Idx4 strategy creates separate indexes on the original table and block table for three attributes and also an index on the 'order' table. Idx5 represents indexing only on the 'distinct values for attributes' table.

Figure 1 shows the average run-time along with deviation (error-bars) for the different indexing strategies. The figure also shows the run-time with no indexing for reference. The best run-time is achieved by creating separate indexes on attributes in two tables - the

original table and the block table used by Algorithm 2. These indexes were created for all three attributes in the DARPA dataset. Creating indexes on attribute dimension and value in the 'order' table helps in some settings, but leads to comparable or slightly higher run-time in others. Creating index on 'distinct value for attributes' table does not help as there are frequent updates and deletions on this table. The conclusions from these experiments are below -

- If multiple columns are referenced in the *WHERE* clause, indexing on these attributes helps by creating a B-Tree index to quickly access the specific attribute values.
- If there is no primary key, creating joint indexes on secondary key is not very helpful. Creating separate indexes on each of the attributes in the secondary key leads to better performance.
- If frequent updates and deletions occur in a table, creating an index does not improve performance significantly as the B-Tree has to be rebalanced frequently.

5.2 Copying vs. Marking

We compare two different implementation strategies for using the original relation in Algorithm 2. Algorithm 2 uses the original relation to initialize the block from which entries are removed to obtain the final dense block. We implemented two different ways of performing this operation, which are explained below -

- **COPY Method (T3.1):** In this approach, we copy the entire input relation table into a new table called *Block*. We then remove entries from this new table to obtain a single dense block.
- **MARK Method (T3.2):** In this approach, we add an additional flag column to the original table and initialize the column with value *True*. We then set the flag attribute to *False*, if the corresponding entry is removed from the block. The final dense block is obtained by rows where flag value is *True* at the end of Algorithm 2. In this approach, we also need to initialize the flag value to *True* at the beginning of Algorithm 2 for every call.

5.2.1 Results and Analysis

All results are presented for 5000 random rows of the DARPA dataset. We repeat each experiment three times to account for varying machine load for accurate comparison. We also test both the methods for all combinations of dimension selection policy (Cardinality, C and Density, D) and density computation method (Arithmetic, A, Geometric, G and Suspiciousness, S). We run the different implementations at the same time (one after the other) to ensure comparable machine load for accurate results.

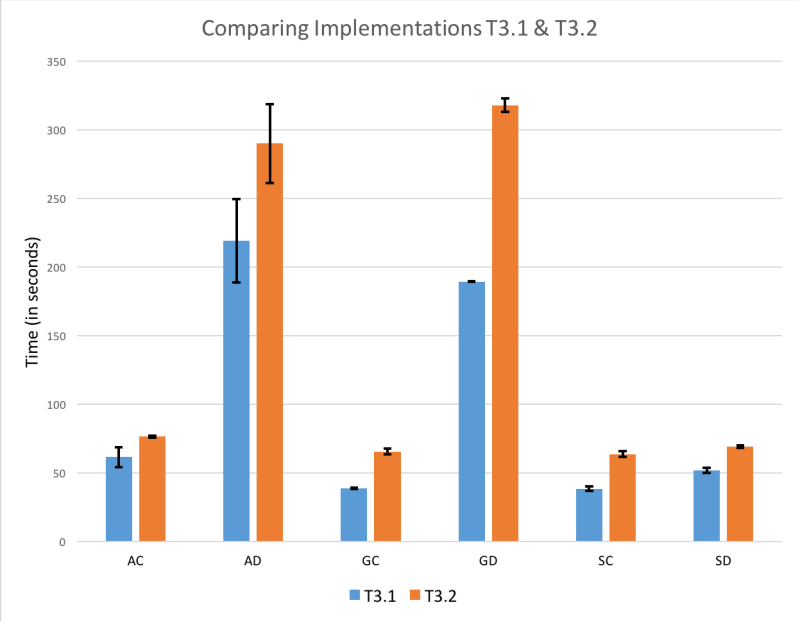

Figure 2: Run-time (in seconds) comparison for the 'copy' and 'mark' strategies along with the deviation (error bars) for different combinations of density measure and dimension selection policy. T3.1 represents the 'copy' strategy where the original relation is copied into a new table called block and subsequent deletions are performed on the new table. T3.2 represents the 'mark' strategy where a new column called flag is added to the original table and deletes are performed by marking the flag to be False.

Figure 2 shows the average run-time along with the deviation for the 'copy' and 'mark' strategies. We observe that the 'copy' strategy performs significantly better than the 'mark' strategy in all the cases. The explanation for the same is below -

- In Algorithm 2, the selected dimension and its specific attributes are removed from the table. As the datasets are large in size, copy method provides the advantage of processing smaller tables in subsequent iterations as the algorithm progresses. If the entries are not deleted from the table, but only marked as deleted using a flag, the algorithm has to process larger tables in every iteration in Algorithm 2.
- Moreover, Algorithm 2 computes the mass of the block and distinct values in each attribute every time a few entries values are removed. In each of these computations, 'mark' method's performance is deteriorated because of the *WHERE* clause for selecting entries with flag value $True$.

We conclude that the 'copy' method performs significantly better than the 'mark' method.

5.3 Custom Optimization

We timed different sections and modules of our implementation and used 'select analyse' on our SQL queries. We observed that most of the time is spent in iterative sections of the

implementation. We then tried converting iterative updates and inserts to bulk updates and inserts. These optimizations are described below -

- **Distinct Value for Attributes table:** In the creation of this table, we select the distinct values of each attribute and create a table with column number and the distinct value. In our original implementation, we use a *SELECT* statement which selects distinct values from every dimension, followed by iterating through these values (using a cursor) and inserting the tuples one after the other into the table. We optimized this by doing *bulk inserts*. We use bulk insert operations for every column using a single select and insert query as an optimization.

- **Computing Attribute Value Mass:** For every distinct value in each column, Algorithm 2 requires the sum of count (mass) of number of entries. In our original implementation, we select distinct values in each dimension; for every distinct value and dimension, we perform a *GROUP BY* operation to identify the mass for a particular attribute value. We combined the select, group by and update operations into a single operation. We *UPDATE* the table in *bulk* by performing *SET* for an attribute using (*SELECT* on attribute value and *sum(count)* with a *GROUP BY* on the main relation) where the attribute value matches. This resulted in a major performance boost as it helped in avoiding many group by operations on large tables.

- **Bulk Deletes:** We also observed that deleting attribute values one at a time iteratively leads to multiple disc accesses. Algorithm 2 has two operations - $B_i = B_i - \{a\}$ and $B = \{t \in B : t[A_i] \notin D_i\}$, which involve removing specific attributes selected by the algorithm from 'distinct attribute value' table and the block. We optimize this operation by performing a *bulk DELETE*.

5.3.1 Results and Analysis

Figure 3 shows the average run-time along with the deviation (error bars) for our custom optimization of using bulk operations. We achieve significant speedup using bulk insertions, updates and deletes. The conclusions are explained below -

- Iterating through rows in SQL tables using a cursor is very expensive as it reads entries from the disk iteratively.

- Bulk inserts, updates and deletes significantly speedup the implementation as they reduce the number of disk accesses.

- Combining multiple SQL statements using *IN* and *WHERE* facilitates bulk operations.

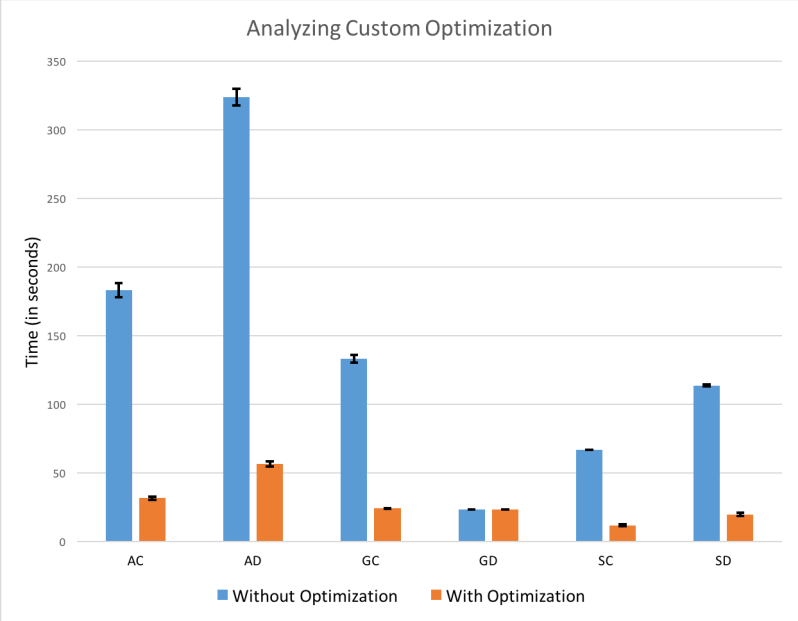

Figure 3: Run-time (in seconds) comparison of the custom optimization strategy with no optimization for different combinations of density measure and dimension selection policy, along with the deviation (error bars). We achieve significant speed-up due to bulk insertions, updates and deletes.

6 Preliminary Experiments (Phase 2)

The different unit tests we performed are described below:

- We first tested our code using a small test dataset with four tuples. We manually computed the expected output at each step and checked the output provided by our implementation. Further, we performed unit tests on different kinds of input data we generated using separate python scripts.

- Using two-way input data, we created various matrices with different configurations of dense blocks (overlapping and non-overlapping with different number of dense blocks in the matrix). We converted these matrices into tensors and associated counts, and provided the same as input to our main module. We ran experiments for all possible configurations of policies and density measures and visualized the dense blocks obtained by the module.

- We also performed experiments on three-way tensors by creating a three dimensional matrix with two different configurations: i) two distinct (non-overlapping) dense blocks, and ii) two overlapping dense blocks having different densities. We identified the dense blocks obtained by all possible configurations of policies and density measures.

We exhaustively tested our implementation on the DARPA dataset. The dataset provided contains three-way tensors - source IP, destination IP, date-time. Date values range from

06/01/1998 to 08/01/1998, with 55 distinct dates. The data contains a total of 4,554,344 tensors.

- For the first set of tests, we aggregated the records based on date values. We obtained the following tensor representation - Source IP, Destination IP, Date, and Count, where count indicates the number of connections between the source IP and destination IP on a specific date. This aggregation results in a total of 140,069 tensors.
- For the next set of tests, we aggregated the records based on time (hourly) values. We obtained the following tensor representation - Source IP, Destination IP, Date, Hour, Count, where count indicates the number of connections between the source IP and destination IP for a specific date and hour. This resulted in a total of 522,236 tensors.

The results and visualizations for few different combinations of policy and density measure on all these datasets are described below -

6.1 Testing with Two-way Tensors

Figures 1-5 provide a visual representation of the top 5 blocks returned by our implementation for different configurations of blocks. Each image represents a 2D input matrix, where i^{th} row and j^{th} column represents a tensor (i, j) with the value in the matrix indicative of the count for that tensor. In each figure, block 1 represents the first dense block found, block 2 represents the second dense block found, and so on. Table 2 provides the densities of the top 5 dense blocks detected for different settings. Our implementation successfully obtains highly dense blocks as the top few blocks for majority of the settings.

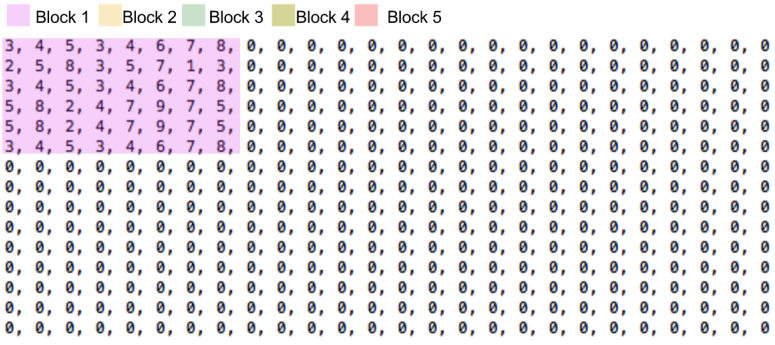

Figure 4: Visualization of Unit-Test 1 results on a 2-way tensor. (Density measure: arithmetic average mass, Policy: dimension selection by density)

Unit-Test 1:
Density measure: arithmetic average mass
Policy: dimension selection by density

Result: Figure 1 shows the algorithm was successfully able to identify the only dense block in this data.

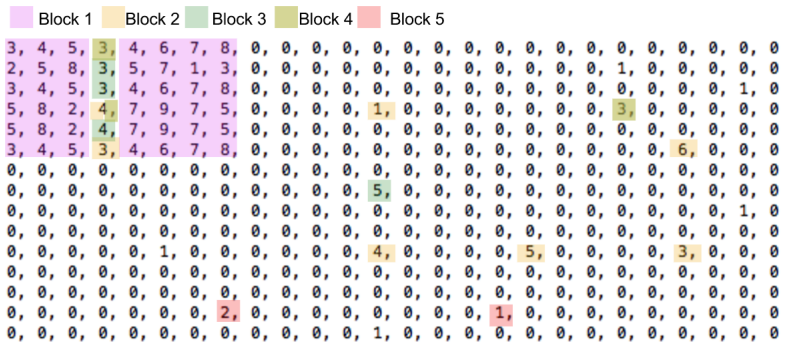

Figure 5: Visualization of Unit-Test 2 results on a 2-way tensor. (Density measure: geometric average mass, Policy: dimension selection by cardinality)

Unit-Test 2:
Density measure: geometric average mass
Policy: dimension selection by cardinality
The algorithm was successfully able to identify the primary dense block in this data. We observe that it did not capture it as one whole dense block. Smaller count values in the block are captured as a part of other smaller (less dense) blocks.

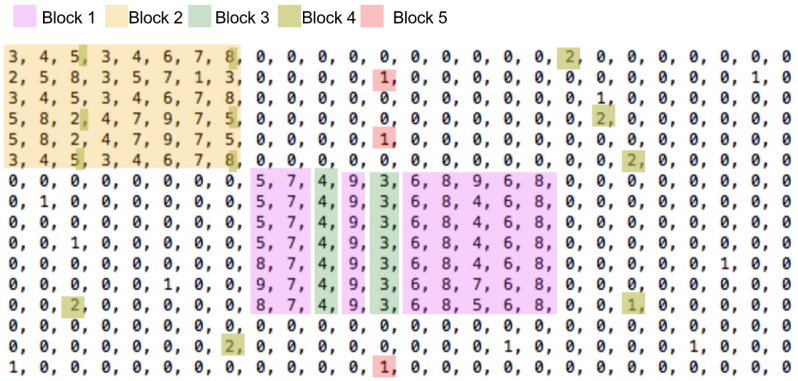

Figure 6: Visualization of Unit-Test 3 results on a 2-way tensor. (Density measure: arithmetic average mass, Policy: dimension selection by cardinality.)

Unit-Test 3:
Density measure: arithmetic average mass
Policy: dimension selection by cardinality

The algorithm was successfully able to identify the two primary dense blocks in this data. The dense block in the middle in split into two blocks - first block (shaded by purple) and third block (shaded by green). Also, we can observe that overlapping blocks are found in this case.

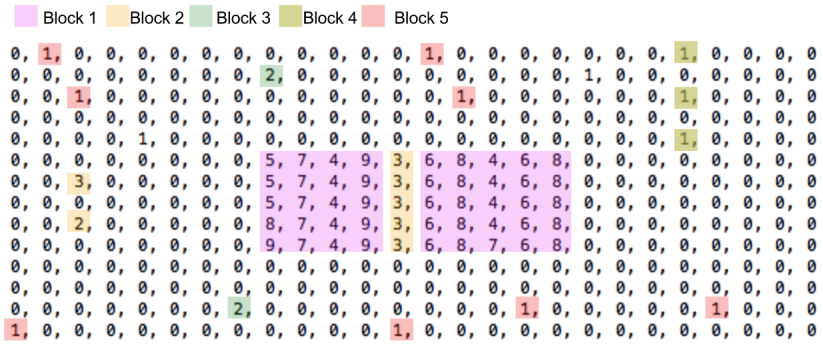

Figure 7: Visualization of Unit-Test 4 results on a 2-way tensor. (Density measure: suspiciousness, Policy: dimension selection by density.)

Unit-Test 4:

Density measure: suspiciousness

Policy: dimension selection by density

The algorithm was successfully able to identify the primary dense block in this data. The dense block in the middle is split into two blocks - first block (shaded by purple) and second block (shaded by yellow).

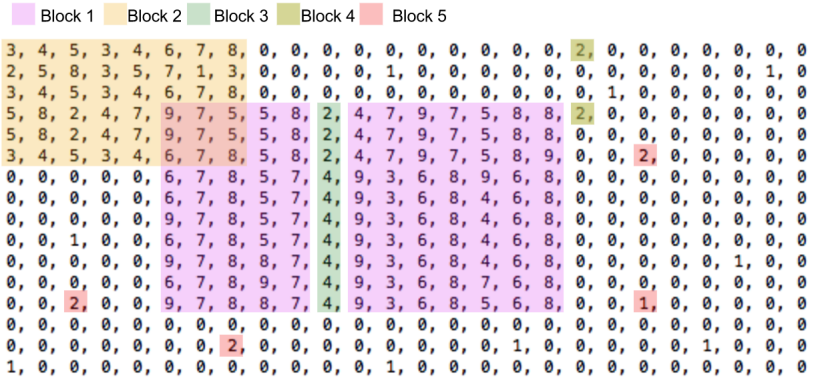

Figure 8: Visualization of Unit-Test 5 results on a 2-way tensor. (Density measure: geometric average mass, Policy: dimension selection by density.)

Unit-Test 5:

Density measure: geometric average mass
Policy: dimension selection by density
The algorithm was successfully able to identify the overlapping dense blocks in this data. Again, the primary block in the middle is split into two blocks - first block (shaded by purple) and third block (shaded by green).

Unit-Test	AC	GC	SC	AD	GD	SD
1	35.4	35.8	3.9	35.4	35.8	4.4
			2.6			10.7
			11.2			18
			23.9			14.6
			33.9			11
2	35.1	35.8	3.9	35.4	35.8	4.4
	7.4		2.6			10.7
	5.0		11.2			18.1
	5.0		23.9			14.6
	2.0		33.9			11
3	52	52.1	212.5	52.0	52.1	212.5
	35.4	35.8	231.1	35.4	35.8	231.1
	10.9	13.1	94	10.9	13.1	94
	8.8	8.8	103.5	8.8	8.8	103.5
	1.5	1.7	7.3	2.0	2.1	7.3
4	42.3	43.9	182.7	42.3	43.9	182.7
	10.0	5.5	22.7	10.0	6.7	29.3
	2.0	3.7	6.6	2.0	3.5	6.2
	1.5	2.0	3.9	1.7	2.1	5.4
		1.7	3.4	1.3	1.4	6.4
5	74.0	74.3	225.4	74.0	74.3	225.4
	35.4	35.8	232.1	35.4	35.8	273.6
	7.0	10.7	210.5	7.0	10.7	92.8
	7.3	16.6	71.7	7.3	2.8	31.5
	2.0	3.1	6.4	2.0	9.3	76.9

Table 2: Densities of blocks detected for Unit-Tests 1-5 on two-way tensors. Block densities are in the order in which the blocks are discovered. Column labels represent the density measure used (A for arithmetic average mass, G for geometric average mass, S for suspiciousness) and the policy for dimension selection (C for dimension selection by cardinality and D for dimension selection by density)

6.2 Testing with Three-way Tensors

Unit-Test 6:
A 100 x 150 x 170 matrix was initialized with all 0s. Two non-overlapping blocks were created - a 24 x 11 x 86 block 80% full (corresponding positions in the matrix filled with 1s), and a 3 x 6 x 10 block 90% full (corresponding positions in the matrix filled with 1s).

Unit-Test 7:
A 100 x 150 x 170 matrix was initialized with all 0s. Two overlapping blocks were created - a 24 x 11 x 86 block 80% full (corresponding positions in the matrix filled with 1s), and a 3 x 6 x 10 block 90% full (corresponding positions in the matrix filled with 1s)

Results and Analysis:
Table 3 provides the densities of top-5 blocks detected for different settings of density measure and policy for selecting dimension. Here, we have presented only the density values (as this data is harder to visualize). We observe that in most of the settings, the algorithm detects the two dense blocks as expected from the designed input. We also observed that the overlapping blocks are detected as two distinct blocks by the algorithm.

Unit-Test	AC	GC	SC	AD	GD	SD
6	419.1	533.5	177.9	450.2	641.3	3161.5
	25.4	28.5	410.9	25.4	28.5	0.26
			356			10.6
			450.2			0.08
			604.8			0.14
7	419.1	533.3	177.9	450.2	641.3	3161.5
	25.4	28.5	410.9	25.4	28.5	0.26
			356			10.6
			450.2			0.08
			604.8			0.14

Table 3: Densities of blocks detected for Unit-Tests 6-7 on three-way tensors. Block densities are in the order the blocks are discovered. Column labels represent the density measure used (A for arithmetic average mass, G for geometric average mass, S for suspiciousness) and the policy for dimension selection (C for dimension selection by cardinality and D for dimension selection by density)

6.3 Analysis - DARPA Dataset

Table 4-8 show the analysis and results of our implementation on the DARPA dataset. We present the top few blocks, along with the detected network attacks for various settings of density measures and dimension selection policies. The aggregation is done date-wise or

time-wise (hourly) for different experiments. Block number 0 indicates the first dense block detected by the algorithm, block number 1 the second dense block and so on. We also observe from the table that the first few blocks detected have very high density and mass values.

Block Number	Density	Mass	Attack Ratio	Attack Type
0	444K	704K	100%	Neptune
1	377K	1M	35.71% 7.14% 7.14% 7.14%	Neptune Anomaly-time anomaly-commands anomaly
2	257K	409K	100%	Neptune
3	88K	527K	23.53% 3.92%	snmpgetattack snmpguess
4	34K	1.5M	99.34%	-
5	5187	5187	100%	satan

Table 4: Network intrusion detection using D-CUBE - densest blocks detected and the corresponding attacks captured in the DARPA dataset aggregated by date. Policy used: dimension selection by cardinality, Density measure: geometric average mass.

Block Number	Density	Mass	Attack Ratio	Attack Type
0	508K	1.5M	100%	Neptune
1	263K	614K	60.0% 20.0%	Neptune Loadmodule
2	71K	330K	35.71% 7.14%	snmpgetattack snmpguess

Table 5: Network intrusion detection using D-CUBE - densest blocks detected and the corresponding attacks captured on DARPA dataset aggregated by date. Policy used: dimension selection by density, Density measure: arithmetic average mass.

Block Number	Density	Mass	Attack Ratio	Attack Type
0	37M	2.4M	11.34%	Neptune
			8.25%	Warez
			5.15%	snmpgetattack
			2.06%	Smurf
			2.06%	Anomaly
			1.03%	Satan
2	4.2M	623K	4.02%	Smurf

Table 6: Network intrusion detection using D-CUBE - densest blocks detected and the corresponding attacks captured on DARPA dataset aggregated by date. Policy: dimension selection by cardinality, Density measure: suspiciousness.

Block Number	Density	Mass	Attack Ratio	Attack Type
0	309K	2.7K	7.84%	Neptune
			1.96%	Warez
1	49K	47	100%	Neptune
2	55K	8.6K	26.41%	snmpgetattack
			2.29%	snmpguess
7	3.9K	805K	9.21%	ipsweep
			8.57%	smurf
8	1.8K	1.7M	29.58%	Smurf
			6.30%	ipsweep

Table 7: Network intrusion detection using D-CUBE - densest blocks detected and the corresponding attacks captured on DARPA dataset aggregated by time (hourly). Policy used: dimension selection by cardinality, Density measure: geometric average mass.

Block Number	Density	Mass	Attack Ratio	Attack Type
0	39.1M	17K	7.8% 2.06% 1.63%	Snmpgetattack Warez Neptune
2	2.1M	1.9K	100%	Smurf
3	4.49M	982K	3.49%	Ipsweep
4	1.3M	11K	100%	Smurf
5	551K	137.8K	100%	Smurf
7	390K	384.8K	99.98%	Smurf
8	196K	150.9K	94.06% 4.49%	Smurf Smurfttl

Table 8: Network intrusion detection using D-CUBE - densest blocks detected and the corresponding attacks captured on DARPA dataset aggregated by time (hourly). Policy used: dimension selection by density, Density measure: suspiciousness.

7 Evaluation (Phase 3)

We perform further analysis on the two datasets - DARPA TCP dump and AIRFORCE TCP dump. This section elaborates on results and analysis on these two datasets.

7.1 DARPA TCP dump

The entire DARPA TCP dump contains 4,554,344 tensors with 4 attributes in each tensor - Source IP, Destination IP, Date and Time. We analyse the top 5 dense blocks retrieved for this dataset using dimension selection by 'Density policy' and 'arithmetic' density as the density measure.

Block Number	Density	Mass	Size	Number of Tensors
0	32,617.33	440,334	13.5	440,334
1	21,791.2	81,717	3.75	81,717
2	20,902.43	836,097	40.0	836,097
3	18,534.26	106,572	5.75	106,572
4	16,124.83	193,498	12.0	193,498

Table 9: Density, Mass, Size and number of Tensors for the top 5 dense blocks obtained from the DARPA TCP dump using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: arithmetic.

Table 9 shows the statistics for the top 5 dense blocks detected on this dataset by the algorithm. The observations and conclusions from the table are below -

- We observe that the density of the detected blocks is high. Density of the blocks decreases as the algorithm progresses.
- For a small volume, many tensors with high mass are detected indicating that the algorithm produces high-quality dense blocks.
- Large number of tensors in a small volume indicate suspicious behaviour.

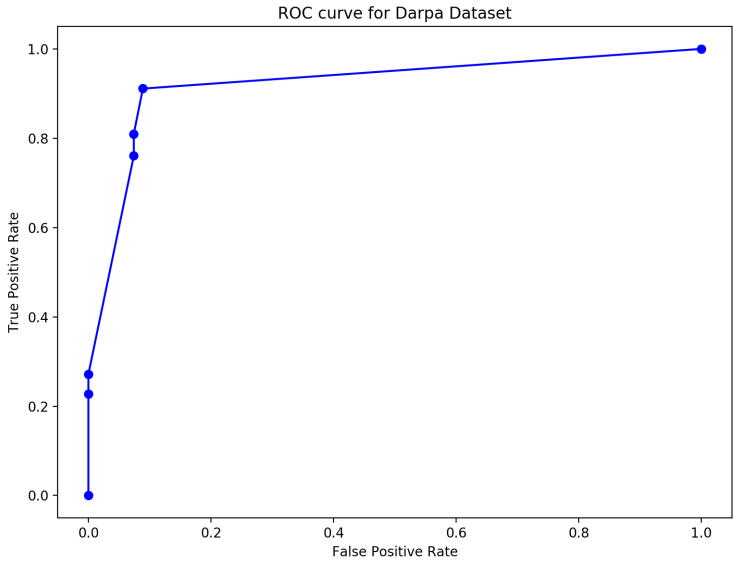

Figure 9: ROC curve for DARPA dataset, indicating True Positive Rate versus False Positive Rate as top 5 dense blocks are detected. Policy used: dimension selection by density, Density measure: Arithmetic.

Figure 9 shows the ROC Curve for the DARPA TCP dump for dimension selection by density and arithmetic density as the density measure.

Area Under ROC Curve: **0.875**

A high value of 0.875 indicates that the dense blocks captured by the algorithm are of high quality and most of the dense blocks capture network attacks (as we have also seen in the earlier section).

Block Number	Attack	Attack Percentage for the Block
0	Neptune	100%
1	Neptune	100%
2	Neptune	99.99%
3	Neptune	100.0%
4	Neptune	99.99%

Table 10: Percentage and type of attack for top 5 dense blocks detected on DARPA TCP dump using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: arithmetic.

Table 10 provides the type of attacks and their percentage in the detected dense blocks for the top 5 blocks obtained by the algorithm. We observe that all the detected blocks represent network attacks!

7.2 Airforce TCP dump

Airforce TCP dump contains 4,898,431 tensors with 7 attributes in each tensor - protocol, network service, bytes sent from the source, Bytes sent from the destination, flag on error status, number of connections made to the host in past two seconds and number of connections made to the service in past two seconds. We analysed for top 5 dense blocks obtained for this dataset using dimension selection by density policy and arithmetic density as the density measure.

Block Number	Density	Mass	Size	Number of Tensors
0	1,930,307.0	1,930,307	1.0	1,930,307
1	1,772,991.5	2,532,845	1.43	2,532,845
2	41,703.97	554,067	13.29	554,067
3	26,799.31	493,873	18.43	493,873
4	18,769.89	168,929	9.0	168,929

Table 11: Density, Mass, Size and number of Tensors for the top 5 dense blocks obtained from the Airforce TCP dump using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: arithmetic.

Table 11 shows the statistics for top 5 dense blocks detected on this dataset by the algorithm. The observations and conclusions from the table are below. We observe similar characteristics in this dataset when compared to results from the Darpa TCP dump. We observe that the density of the blocks detected is high and the density of the blocks decreases as the algorithm progresses.In a small volume, many tensors with high mass are detected indicating that the algorithm produces dense blocks. Large number of tensors in a small volume indicate suspicious behaviour.

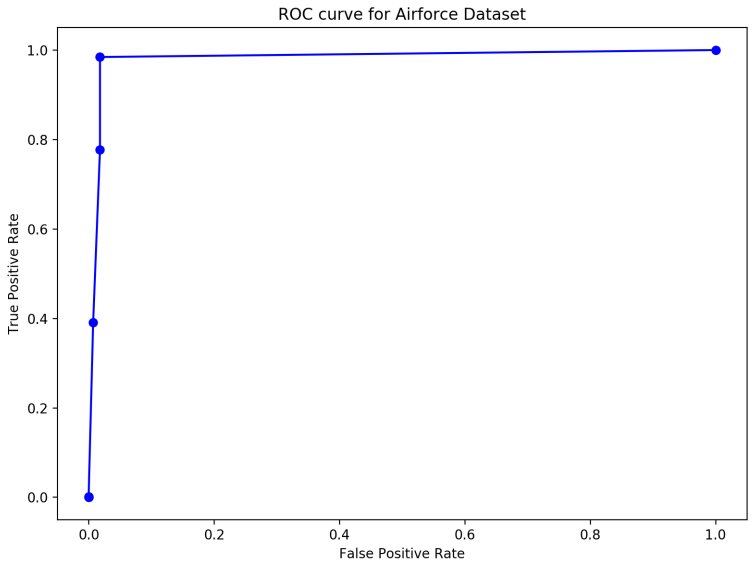

Figure 10: ROC curve for the Airforce dataset - True Positive Rate vs. False Positive Rate as top 5 dense blocks are detected. Policy used: dimension selection by density, Density measure: arithmetic.

Figure 10 shows the ROC Curve for the Airforce TCP dump for dimension selection by density and arithmetic density as the density measure.

Area Under ROC Curve: **0.982**

Table 12 provides the type of attacks and their percentage in the detected dense blocks for the top 5 blocks obtained by the algorithm. We observe that all the detected blocks represent network attacks!

Block Number	Attack	Attack Percentage for the Block
0	Smurf	100%
1	Smurf	100%
2	Neptune	75.87%
	Smurf	9.40%
	Isweep	7.47%
	Portsweep	3.57%
	Satan	3.36%
3	Neptune	55.04%
	Smurf	21.91%
4	Neptune	99.94%

Table 12: Percentage and type of attack for top 5 dense blocks detected on Airforce TCP dump using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: arithmetic.

8 Anomaly Detection in Real World Data

We run our implementation of the D-CUBE algorithm on three real-world datasets - Amazon, Yelp and English Wikipedia Revision History. The results and analysis are presented in subsequent sections -

8.1 Amazon Dataset

The Amazon dataset contains 2,654,297 tensors, each containing 4 attributes - unique identifier for user, unique identifier for apps, review time and number of stars provided in the rating. We use the arithmetic density measure and dimension selection by density policy.

Block Number	Density	Mass	Size	Number of Tensors
0	118.03	3,600	30.5	3,600
1	99.02	7,550	76.25	7,550
2	78.05	1,600	20.50	1,600
3	68.06	1,225	18.0	1,225
4	50.34	77,697	1543.44	77,697

Table 13: Density, Mass, Size and number of Tensors for the top 5 dense blocks obtained from the Amazon dataset dump using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: arithmetic.

8.1.1 Analysis of Results

- The first block is highly suspicious. It contains 3600 entries where the rating 4 is given in the exact same time slot. Another interesting observation is that there are exactly 60 users who have rated 60 apps, each user rating all apps once. This behavior is unnatural and we suspect malicious activity.
- The second block also depicts unnatural user behavior. It contains 7550 entries where the rating 0 or 3 is given in exactly 3 time slots. Moreover, there are exactly 150 users who have rated 150 apps. The counts for 0 rating and rating of 3 are 4525 and 3025 respectively. Also, the number of entries pertaining to each time slot are 2500, 2025 and 3025. This behavior (all values being multiples of 5 and fixed time slots) is clearly suspicious and merits more analysis.
- The third block has exactly similar characteristics as the first block with 1600 entries from 40 users for 40 apps with rating 2 in exactly one time slot.
- The fourth block too has exactly similar characteristics as the first and the third blocks with 1225 entries from 35 users for 35 apps with rating 0 in exactly one time slot.
- The fifth block on the first look doesn't seem suspicious. There are 77697 entries where 3412 users have rated 1745 apps in 1011 time slots. It shows all rating values from 0-4.

Most of the blocks captured are highly suspicious and prove the effectiveness of the D-CUBE algorithm.

8.2 Yelp Dataset

The Yelp dataset contains 2,241,338 tensors with 4 attributes - unique identifier of user, unique identifier for app, review time and number of starts provided as the rating. We use arithmetic density measure and dimension selection by density policy.

Block Number	Density	Mass	Size	Number of Tensors
0	118.03	3,600	30.5	3,600
1	108.04	3,025	30.0	3,025
2	98.04	2,500	25.5	2,500
3	88.04	2,025	23.0	2,025
4	84.74	268,578	3169.44	268,578

Table 14: Density, Mass, Size and number of Tensors for the top 5 dense blocks obtained from the Yelp dataset using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: arithmetic.

8.2.1 Analysis of Results

- The first block is highly suspicious (similar to the first block in the amazon dataset)! It contains 3600 entries where the rating 4 is given in the exact same time slot. Another interesting observation is that there are exactly 60 users who have rated 60 businesses, each user rating all businesses once. This behavior is clearly unnatural and in this case we have detected malicious activity.
- The second block has exactly similar characteristics as the first block with 3025 entries from 55 users for 55 businesses with rating 0 in exactly one time slot; the block clearly depicting suspicious activity.
- The third block too has exactly similar characteristics as the first and the second blocks with 2500 entries from 50 users for 50 businesses with rating 1 in exactly one time slot.
- The fourth block again has 2025 entries from 45 users for 45 businesses with rating 4 in exactly one time slot.
- The fifth block doesn't seem suspicious though. There are 268578 entries where 5404 users have rated 4464 businesses in 2792 time slots. It shows all rating values from 0-4.

Most of the blocks captured merit deeper analysis and clearly prove the utility of the D-CUBE algorithm on real-world data.

8.3 English Wikipedia Revision History Dataset

The Wiki dataset contains 48,97,968 tensors with 3 attributes - unique identifier of user, title of the revised page, and time. We use geometric density measure and dimension selection by density policy.

Block Number	Density	Mass	Size	Number of Tensors
0	2496.11	7,756	3.11	7,756
1	875.84	16,508	18.85	16,508
2	542.46	1,520	2.80	1,520
3	484.74	22,824	47.08	22,824
4	320.14	40,017	124.99	40,017

Table 15: Density, Mass, Size and number of Tensors for the top 5 dense blocks obtained from the Wiki dataset using the D-CUBE algorithm. Policy used: dimension selection by density, Density measure: geometric.

8.3.1 Analysis of Results

- The first block depicts activity from only one user on the same page at different times marking 7,756 entries. On further analysis the user seems to be a bot.
- The second block too is dense with activity on only three pages by three users with 16,508 entries in all in different time slots.
- The third block has 1,520 entries involving only 1 user and 1 single page but many time-slots.
- The fourth block has 22,824 entries involving only 13 users and 11 pages spread over many time-slots. Like the previous blocks many users in this block too are bots.
- The fifth block seems less suspicious though. It has 40,017 entries involving 75 users and 35 pages spread over many time-slots.

Most of the top 5 blocks detected capture unnatural behaviour (mainly bot activity) and further strengthen the effectiveness claim of the D-CUBE algorithm in a real-world setting.

9 Conclusions

Over the course of this project (including the literature survey, implementing D-CUBE and the experimentation phase), we have learnt many new things. Some of the major conclusions were -

- Dense block detection is a highly relevant problem and the research community has constantly been updating the state-of-the-art approaches to find dense blocks in a real-world setting.
- The D-CUBE algorithm is achieves dense block detection with $O(1)$ memory footprint and can thus be used to find dense blocks in very large datasets which would not fit in memory otherwise!
- In this project, we implemented the complete D-CUBE algorithm which takes as input a density measure (arithmetic or geometric), a dimension selection policy (density or cardinality), and the value k to return the top k dense blocks. Our implementation has an $O(1)$ memory footprint.
- In an attempt to optimize our implementation, we tried indexing the tables we create in SQL. Results show that indexing using the right strategy on the right tables leads to better performance.
- As an attempt to optimize the implementation further, we replace iterative operations by bulk inserts, updates and deletes by coming up with slightly more complicated SQL queries. This leads to a significant increase in performance!

- To see the algorithm in action, we tested on smaller (manually constructed datasets). D-CUBE successfully found dense blocks in our two-way and three-way tensored dataset, both overlappping and non-overlapping blocks.
- The algorithm successfully detects dense blocks on real-world datasets. We detected networks attacks in the Airforce and DARPA TCP dump. The blocks returned by the algorithm clearly depict suspicious user behaviour that merits deeper analysis.
- The algorithm also efficiently detects suspicious activity in tensors from other real-world settings like reviews on Yelp and Amazon, and revisions on English Wiki.
- The algorithm scales well as dataset sizes increase and is extremely efficient even on restricted memory machines when dealing with large datasets.

References

[1] M. Castellanos M. Hsu A. Jindal, S. Madden. Graph analytics using the vertica relational database. 2014.

[2] M. Jiang, A. Beutel, P. Cui, B. Hooi, S. Yang, and C. Faloutsos. A general suspiciousness metric for dense blocks in multimodal data. 2015.

[3] B. Hooi K. Shin and C. Faloutsos. M-zoom: Fast dense-block detection in tensors with quality guarantees. 2016.

[4] J. Kim K. Shin, B. Hooi and C. Faloutsos. D-cube: Dense-block detection in terabyte-scale tensors. 2017.

[5] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos. Gbase: an efficient analysis platform for large graphs. 2012.

[6] U. Kang, C.E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. 2009.