Design and Development of Data Indexing Techniques for Mining Large and Streaming Data

THESIS

Submitted in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

JAGAT SESH CHALLA
2006PH120559P

Under the Supervision of

Prof. Poonam Goyal
Associate Professor, Department of Computer Science & Information Systems
BITS Pilani, Pilani Campus, India

&

Prof. Anil Maheshwari
Professor, School of Computer Science
Carleton University, Ottawa, Canada

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
PILANI CAMPUS, RAJASTHAN, INDIA

November, 2019

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
PILANI CAMPUS, RAJASTHAN, INDIA

CERTIFICATE

This is to certify that the thesis titled "**Design and Development of Data Indexing Techniques for Mining Large and Streaming Data**", submitted by **Jagat Sesh Challa** ID No **2006PH120559P** for award of Ph.D. of the Institute embodies original work done by him under my supervision.

Signature of the Supervisor	:	Poonam Goyal
Name	:	**Prof. Poonam Goyal**
Designation	:	Associate Professor & Head Department of Computer Science and Information Systems, BITS Pilani, Pilani Campus Pilani, India
Date	:	Nov 30, 2019

Signature of the co-supervisor:		[illegible]
Name	:	**Prof. Anil Maheshwari**
Designation	:	Professor Department of Computer Science Carleton University, Ottawa Canada
Date	:	November 27, 2019

Dedicated To

My Teachers & The Almighty ....

Acknowledgements

I would like to express my sincere gratitude to several personalities for their constant guidance, help, support and well-wishes. Firstly, I would like to thank the Vice-Chancellor of the BITS-Pilani University, Prof. Souvik Bhattacharya, and the Director of Pilani Campus, Prof. Ashoke Kumar Sarkar for giving a conducive research environment to the Institute, without which my Ph.D. would not have been possible. I would also like to thank the Head of the Department, Dept. of Computer Science & Information Systems, Prof. Poonam Goyal, for providing the required research facilities to our department. I am greatly indebted to my supervisor - Prof. Poonam Goyal, and my co-supervisor - Prof. Anil Maheshwari, who gave me the vision of research, moral support and constant guidance throughout my Ph.D.. I sincerely thank my Doctoral Advisory Committe members - Prof. Navneet Goyal and Dr. K. Haribabu for taking out time for reading my thesis and giving valuable comments and suggestions. I am also thankful to Prof. Sundar Balasubramaniam for his active guidance and support throughout my Ph.D.. I am greatly indebted to all my teachers who have been a great support throughout my life. I also acknowledge the support of my colleagues working in ADAPT Lab - Mr. Saiyedul Islam, Dr. Sonal Kumari, Mrs. Prerna Kaushik, Mr. Nikhil S, Mr. Dhruv Kumar, Mr. Shivin Srivastava, Mr. Amogh Sharma, Mr. Vijay M. Giri, Mr. Dhananjay Mantri and Mr. Anand Wani. I strongly acknowledge the patience and support given by my wife Mrs. Nandini Challa, my parents - Mr. Kesava Rao Challa and Mrs. Usha Sri Challa, and my brother Mr. Manish Kalyan Challa. Last but not the least, I thank the Supreme Almighty for giving me the inspiration to carry out this work.

Abstract

Today, data mining has become a very important technology that is directly/indirectly impacting all aspects of life. Data in this world is growing at an unprecedented rate and today's decision making has become highly complex and data-centric than ever before. There is a lot of data being collected and warehoused from various domains like web, e-commerce, sensor networks, satellites, etc. And the speed of collection of data has also significantly increased to a magnitude of GigaBytes/hour. Traditional data mining algorithms are neither capable of handling such large volumes of data nor can process the data with respect to their velocity.

To deal with such large data, researchers have proposed innovative algorithms. Large volumes of data are handled by data mining algorithms specifically designed for parallel architectures such as shared memory, distributed memory or hybrid architectures. And large velocity of data is handled by stream mining algorithms. The specific area of study that processes such large amounts of data is known as "Big Data Analytics" or "Big Data".

This thesis deals with development of various data indexing techniques to handle "volume", "velocity" and variability" of Big data. This thesis presents the following solutions to handle large volume of data:-

- Presents a data structure known as *Grid-R-tree*, which supports efficient execution of spatial queries such as neighborhood & nearest neighbor queries used by spatial data mining algorithms. Grid-R-tree also enables efficient execution of density-based clustering algorithms like DBSCAN & OPTICS, as well as the *k*-nearest neighbor classifiers.
- Presents a dynamic distributed data structure known as DD-RTREE, which supports effective distribution of incremental datasets over a cluster of

computing nodes. The distribution is based on R-tree spatial containment principles and is shown to preserve spatial locality in the distribution. This aids in efficient execution of parallel spatial data mining algorithms like DBSCAN, OPTICS, etc.

- Presents a few data distribution strategies (P-based Split, PD-Split and CD-Split) for distributing large static datasets over a cluster of computing nodes. These methods are specifically designed for attaining performance gain in parallel density based and hierarchical clustering algorithms. Appropriate recommendations for their usage have also been presented.

To handle large velocity and variability of data, this thesis presents the following:-

- Presents the first anytime set-wise classification algorithm for data streams known as AnySC. It uses a proposed data structure known as *CProf-forest* for handling varying inter-arrival rate of objects in the stream. *CProf-forest* supports anytime incremental update and enables anytime classification of test entities as well. The utlity of the proposed algorithm has been shown with respect to two problems: community detection in twitter and website fingerprinting attack.
- Presents AnyClus and Any-MP-Clus, which are frameworks for anytime clustering of single-port and multi-port data streams, respectively. They use a proposed data structure known as *AnyRtree* to handle variable inter-arrival rate of data objects. The proposed frameworks handle noise and concept drift more effectively than the existing frameworks. The spatial locality preservation of AnyRtree aids them to produce micro-clusters of higher quality and compactness.
- Presents AnyFI and MPAnyFI, which are the first algorithms for anytime frequent itemset mining of single-port and multi-port data streams, respectively. They use a proposed data structure known as *BFI-Forest* to handle variable inter-arrival rate of transactions. The proposed frame-

works are shown to handle high and variable speed streams and identify Frequent Itemsets with high value of recall.

Among the above problems, Grid-R-tree, anytime set-wise classification, anytime FI mining and DD-Rtree, are the first proposed approaches of their kind. All the algorithms have been implemented, experimented and benchmarked with the existing solutions (if any). The overall work done for this thesis shows a significant contribution to research in the area of "Big Data".

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations/Symbols

List of symbols used in Chapters 2 & 3 (Grid-R-tree)	
Term	**Definition**
Grid-R-tree	Grid based R-tree
global-R-tree	first level R-tree of Grid-R-tree
cell-R-tree	second level R-tree of Grid-R-tree, present with each cell
Gm & GM	Fanout parameters of global-R-tree
Rm & RM	Fanout parameters of cell-R-trees
MBR	Minimum Bounding Rectangle
N	Data Size
d	Dimensionality of the dataset
c	Cell size
τ	Threshold on number of points per cell
DL	Data list
CL	Cells list
T	No. of cells
T'	Average no. of cells returned by cell-window query in Grid-R-tree
ϵ	Parameter for neighborhood query
PointWiseNBH	Point-wise neighborhood query in Grid-R-tree
CellWiseNBH	Cell-wise neighborhood query in Grid-R-tree
BF-kNN	Best-First KNN search over and R-tree
RKNN	KNN query over an R-tree
GRKNN	KNN query over Grid-R-tree
CellWiseNBH$_{DO}$	CellWiseNBH over Grid-R-tree with adaptive gridding
PointWiseNBH$_{DO}$	PointWiseNBH over Grid-R-tree with adaptive gridding
GRKNN$_{DO}$	GRKNN over Grid-R-tree with adaptive gridding

List of symbols used in Chapter 4 (AnySC)	
Term	**Definition**
AnySC	Anytime set-wise classification of data streams
CProf-forest	Class Profile Forest
CProf-tree	Class Profile Tree
SC	Set-wise Classification Model
d	Dimensionality of the dataset
N	No. of traning entities
$\mathcal{E}_i$	i^{th} training entity
e_i	No. of data objects in $\mathcal{E}_i$
c	No. of classes
n	No. of test entities
$\mathcal{T}_i$	i^th test entity
Y_r	r^th object in the stream
p	No. of class profiles
q	No. of anchor points
W_i	i^th anchor point
C_i	i^th cluster

f_i	i^{th} relative cluster frequency
$\|U\|$	No. of objects updated in $\mathcal{E}_j$ before adding a point Y_i
AG_S	d-dimensional vector containing sum of the fingerprints of a set of entities
min_stat	Min no. of objects to be accumulated in a test entity before it can be classified
m & M	Fanout parameters of a CProf-tree
PT_e	Pointer to the sub-trees under an node entry e of a CProf-tree
CP_e	Class Profile at node entry e
BF_e	Buffer containing two q-dimensional vectors V_o and V_n
λ	Parameter that controls stream speed in Possion streams
SC_A	SC converted into an anytime model

List of symbols used in Chapter 5 (AnyClus) & Any-MP-Clus	
Term	**Definition**
AnyClus	Anytime Clustering of Data Streams
Any-MP-Clus	Anytime Clustering of Multi-Port Data Streams
N	Data size
d	Dimensionality of the dataset
DS	Data Stream
mc (mc_j)	Micro-Cluster
n_j	No. of points indexed in mc_j
S_j	Vector storing linear sum of points indexed in mc_j for each dimension
SS_j	Vector storing squared sum of points indexed in mc_j for each dimension
μ_j	Mean of mc_j
ρ_j	Radius of mc_j
MBR	Minimum bounding rectangle
pt	Pointer to child node in AnyRTree
b	Buffer in AnyRTree nodes
nb	Noise Buffer in AnyRTree nodes
m & M	Fanout values of AnyRTree
δ	Percentage limit on expansion area to determine noise in AnyRTree
γ	Time interval for noise to concept transition
β	Min no. of points to be accumulated in noise beffire before it can become a concept
$\mathcal{H}$	Hitchiker
n_{mc}	No. of leaf level mcs in AnyRTree
τ	Threshold on radius for merging
m_{mc}	No. of mcs in a given machine
f	Parameter to control radius of mcs
ϵ	DBSCAN parameter
min_pts	DBSCAN paramter
AR	AnyRTree
CT	ClusTree
LT	LiarTree
λ	Parameter that controls stream speed in Possion streams

List of symbols used in Chapter 6 (AnyFI) & MPAnyFI	
Term	**Definition**

AnyFI	Anytime Frequent Itemset Mining of Data Streams
MPAnyFI	Anytime Frequent Itemset Mining of Multi-port Data Streams
I	Dictionary of unique items
tr	A transaction containing a set of items
DS	Data Stream
sDS	A finite contiguous subsequence of transactions from DS
$freq_{sDS}(S)$	frequency count of an itemset S with respect to sDS
FI	Frequent Itemset
SP	Suffix Projections
σ	Support Threshold
ϵ	Error Threshold
f	Decay factor to decay the frequency counts of itemsets
TTWF	Tilted-Time Window Framework
$Head$	First item of a Transaction
θ	An optimization parameter used for θ-deferring
γ	An optimization parameter used for Buffer Pruning
Max_Height	Maximum height of BFI-tree. It is the height at which FP-tree resides.
$hash\_size$	Size of the hash table in a buffer
$buffCapacity$	The maximum number of suffix projections a buffer can store
$batch\_size$	The number of transactions after which to conduct intermittent pruning.
PI	Pruning Interval for pruning a buffer's node
λ_x	Probability that an item i_x appears in an incoming transaction
c	Probability of refining a node of BFI-tree
η	Critical support threshold used to derive counts of k-length FIs
t_{in}	Units of time used to capture a batch in MPAnyFI
f_{tt}	A decay factor used to decay itemsets in TTWF
λ	Parameter used to simulate Poisson stream. λ is the expected inter-arrival rate.
FAN	False Negative Mode
FAP	False Positive Mode
ATL	Average Transaction Length
AFL	Average Frequent Itemset Length

List of symbols used in Chapters 7 & 8 (Data Distribution)

Term	Definition
DD-RTree	Dynamic Distributed R-tree
PD-Split	Paremeterized Dimensional Split
CD-Split	Cell-based Dimensional Split
P-based Split	Projection-based Split
MPI	Message Passing Interface
ϵ	Epsilon for neighborhood query
$MinPts$	DBSCAN parameter
k	Parameter for nearest neighbor query
N	No of data points in a dataset
n	No. of computing nodes in the cluster
τ	Threshold on number of points in a cell
DDS	Distributed Data Structures

IR-Tree	Index-R-tree
MR-Tree	Machine-R-tree
lm & lM	Fanout values of IR-Tree
bc	Buffer Capacity
mc	Machine Capacity
τ'	Threshold on number of points being shifted
mmd	Min-Max Distance
md	Min-Distance

Chapter 1

Introduction

Data Mining has become a pervasive technology that is poised to touch all aspects of our lives. This can be mainly attributed to the fact that data in the world is growing at an unprecedented rate. Along with this, decision making is becoming more and more data-centric and complex than ever before. There is a lot of data being collected and warehoused from various domains like web, e-commerce, transactional systems, sensor networks, satellites, gene expressions, scientific simulations, etc [6]. And moreover the speed of collection of data has also significantly increased to a magnitude of Giga-Bytes/hour. So, the interest of researchers in the field of data mining has increased by leaps and bounds. Innovative algorithms are being designed to solve varieties of new problems that analyze varieties of data and extract useful knowledge. Data mining has applications in almost all disciplines and often requires an interdisciplinary/multidisciplinary approach to solve a particular problem. This has led to significant increase in the scope and strength of data mining tasks.

Tan et. al [6] define data mining as:

- Non-trivial extraction of implicit, previously unknown and potentially useful information from data.
- Exploration & analysis, by automatic or semi-automatic means, of large quantities of data in order to discover meaningful patterns.

Data mining can be broadly classified into four main tasks- *Classification, Clustering, Association Rule Mining* and *Anomaly Detection* [6]. Fig. 1.1 [1] shows a detailed classifica-

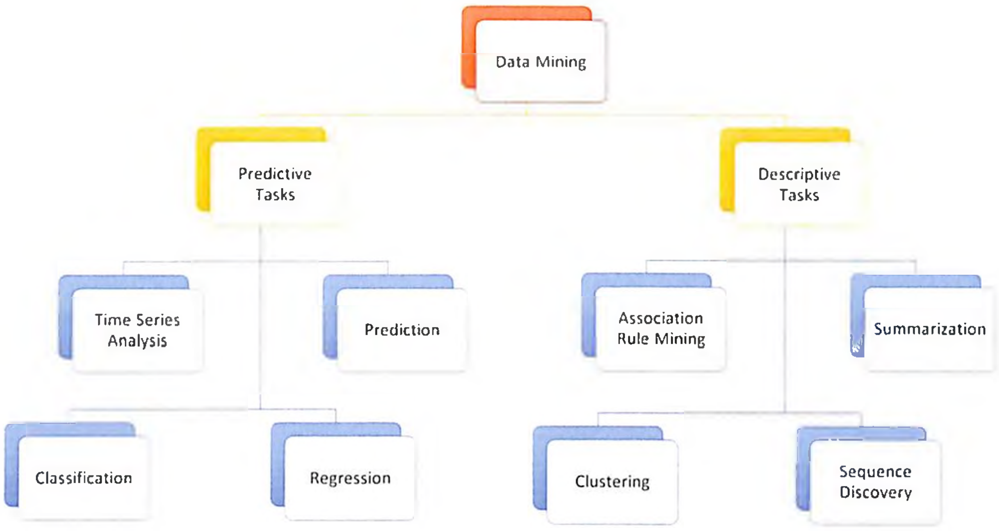


Figure 1.1: Broad Classification of Data Mining Tasks [1]

tion of data mining tasks. Each of these tasks are described below in brief.

Classification Classification is one of the important supervised learning tasks of data mining [6]. Classification algorithms typically use a set of labelled records for training a classification model, that allows us to predict the class label of a previously unseen record. A few popular classification algorithms include: Decision-tree [7, 8, 9], Artificial Neural Networks [10, 11], Rule based Classifier [12, 13], Naive-Bayes Classifier [14, 15], Nearest Neighbor Classifier [16], Support Vector Machines [17, 18], Ensemble Classifiers [19], etc. The applications of classification task include: Document Classification [20], Network Fraud Detection (in cases like credit card transactions or terrorist identification) [21, 22], Email Spam Detection [23], Galaxy Categorization [24], Malignant Cells Detection [25], etc.

Clustering Clustering [6] is an important unsupervised data mining task that have been extensively used and studied in various data mining problems. It is all about finding groups of objects such that the objects in a group will be similar (or related) to one another and different from (or unrelated to) the objects in other groups [6]. Clustering algorithms can be broadly classified into: (1) Representative/partitioning-based clustering [26, 27, 28, 29], (2) Hierarchical clustering [30, 31, 32, 33] (3) Density-based clustering [34, 35, 36], and (4) Subspace Clustering [37, 38, 39]. Various application domains where clustering is used include - social networks [40, 41, 42], image segmentation [43], object identification [44], information retrieval [45, 46], etc.

Association Rule Mining Association Rule Mining has been introduced in [47] for market basket analysis. It involves mining of frequent itemsets (or frequent patterns), which are subsets of items that appear frequently in a transactional dataset [48, 49]. Frequent itemset (or FI) mining algorithms can be broadly classified into two types: *Apriori based* [47, 50, 51] and *FP-growth based* [52, 53]. Apriori based algorithms use apriori principle, where as FP-growth based algorithms use the indexing structure - FP-tree, or similar structures. Various applications of frequent itemset mining include - retail chain analysis, stock market analysis, software bug analysis, fault and fraud prediction, etc. Frequent itemset mining also facilitates other data mining tasks such as classification, clustering and outlier detection [54, 55, 51].

Anomaly Detection Anomaly detection is a data mining task that finds data objects or patterns in a dataset that do not follow an expected behavior [56]. Various classification and clustering algorithms have been used for detecting anomalies [57, 58, 59, 59, 60, 61]. Its applications include network intrusion detection [61] transactional fraud detection [62], image processing [63], health insurance applications [57], etc.

1.1 Data Deluge!

In recent past, data has been growing exponentially, resulting in a data deluge! This is primarily attributed to exponentially increasing utilities of data generating systems such as web, e-commerce, internet of things, transactional systems, embedded systems, digital cameras, particle accelerators, DNA sequencers etc. The primary source of data is web and related systems. According to Statista [2], the number of web users had been growing year by year as shown in Fig. 1.2. As a result, the data being generated is increasing at an exponential rate. The 2018 Study of DOMO Data Never Sleeps 6.0 [3] states that, every minute, there are 3.8M Google searches, Amazon ships 1,111 packages, Uber takes 1389 rides, users watch 4.3M videos on Youtube, 49.3k users post photos on Instagram, and so on. These figures are shown in Fig. 1.3 and are expected to keep increasing year by

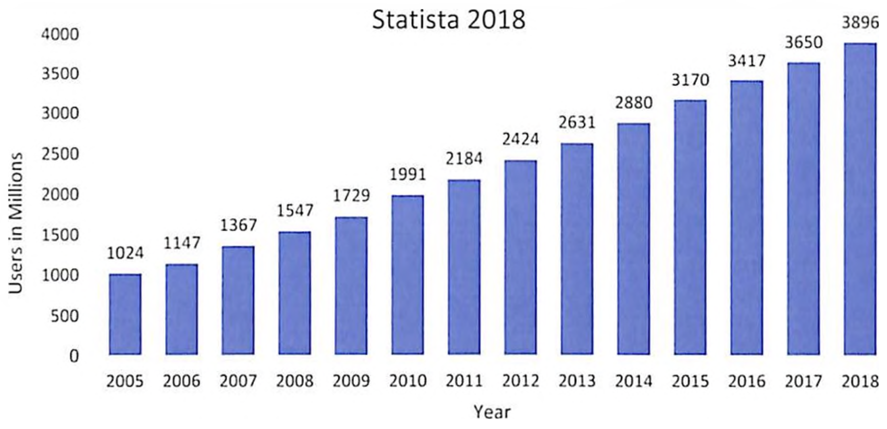


Figure 1.2: Growing number of internet users (Source: [2])

year. Ion Stoica, a director at the University of California, Berkley, made an observation that "Data is growing faster than the Moore's law". The 2018 iteration of the IDC's study states that data is being generated by more than 40 types of devices, from RFID tags and sensors to supercomputers and supercolliders, from PCs and servers to cars and planes [4]. IDC also estimates that from 2013 to 2020, the digital universe will grow by a factor of 10 - from 4.4 trillion GB to 44 trillion GB (which is equal to 5,200 GB of data for every person on earth) (illustrated in Fig. 1.4). Data is increasing more than two times every two years. This indicates that data is expected to grow by a factor of 50 in a span of 10 years (2010 to 2020). By the year 2020, it is expected that, per day data created by a smart home will be over 3,000 GB [64], by self-driving cars will be over 2 Peta Bytes [65], by a connected factory will be 1,000,000 GB [64], and so on.

Although, there is so much of useful data that is being generated, the latest IDC study [4] reveals that only a small fraction (3%) of the world's data is being properly utilized. Also, the world's amount of available storage capacity (i.e., unused bytes) across all media types is growing slower than the growing digital universe. In 2013, the available storage capacity could hold just 33% of the digital universe. By 2020, it will be able to store less than 15%. Because of this, large quantities of useful data is getting lost.

By mining or analyzing such large datasets, the decision making becomes more rational and effective, and is capable of changing the face of the society. This is because of its vast variety of applications in businesses, healthcare, society, economies, education, commerce, IT, etc. However, when data becomes so large, the traditional methods of data

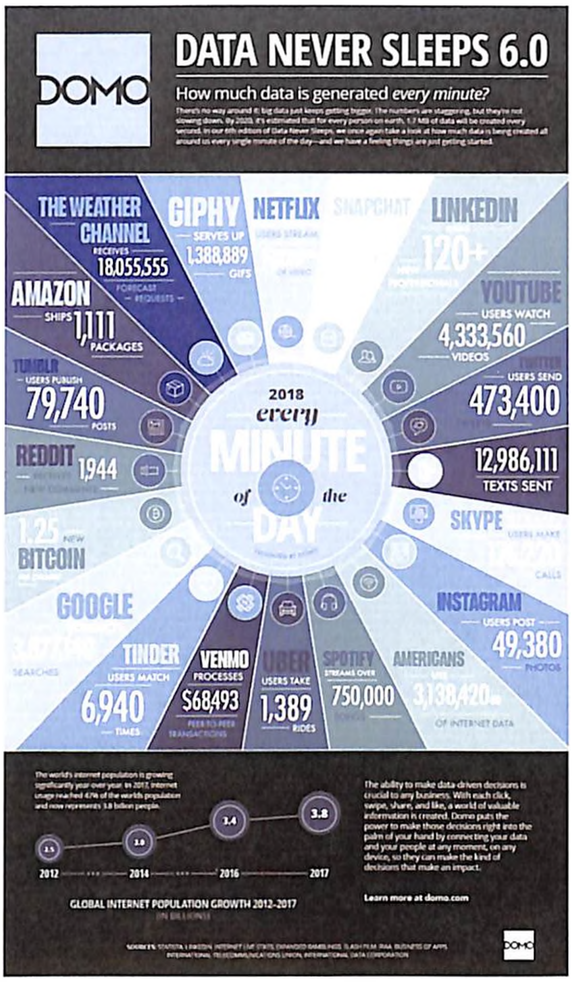


Figure 1.3: Statistics of growing data by DOMO | Data Never Sleeps 6.0 [3]

mining and analytics described in the previous section become unfit for extracting knowledge out of it. This is because they either need too much of time (unreasonable) to process or they are incapable of scaling on bigger hardware setups, or both. This problem has attracted several business analysts, data scientists, and IT professionals to develop efficient and scalable - analytical, computational and storage solutions to mine and analyze such ever increasing data. This lead to the advent of the era of "Big Data".

1.2 Big Data

Big Data or Big Data Analytics refers to solutions to the problems of mining or analyzing very large sized datasets, which are almost impossible to store or process using traditional techniques and hardware. The term "Big Data" was coined by Roger Magoulas [66] from

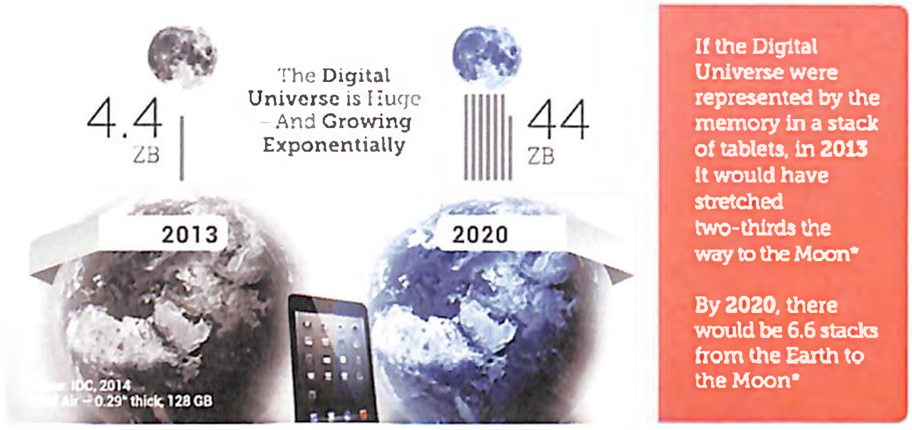

4.4 ZB
The Digital Universe is Huge – And Growing Exponentially
44 ZB
2013
2020
Source: IDC, 2014
iPad Air – 0.29" thick, 128 GB
If the Digital Universe were represented by the memory in a stack of tablets, in 2013 it would have stretched two-thirds the way to the Moon*
By 2020, there would be 6.6 stacks from the Earth to the Moon*

Figure 1.4: IDC prediction of Data Growth (Source: IDC Study 2018: EMC [4])

O'Reilly media. The term explains different characteristics of the digital age data, which are typically 7Vs as illustrated in Fig. 1.5 [5].

Big datasets typically manifest in two forms: *Large Static Data* and *Dynamic Streaming Data*. Large Static datasets have large volumes and are collected over a period of time. They are like snapshot data and don't evolve with time. They are used for a specific study or knowledge extraction. For example, astronomical data, gene expression data, etc. are static datasets that have huge volumes (typically tera bytes) and are analyzed on a whole for a specific purpose. Dynamic stream datasets are those datasets whose data objects are being continuously generated at a fast and variable rate, ordered by time. These streams have data patterns that evolve with time. These patterns are to be captured within the constraints of limited time and storage. Also, the outdated patterns/data that are previously captured are to be continuously discarded as new data keeps arriving. Data from transactional systems, sensor networks, smart digital systems, network traffic, stock markets, etc. are a few examples of streaming data. We shall now describe how each of these types of data are handled by a Big Data Scientist!

1.2.1 Handling Large Static Data

To process large volumes of data, traditional mining algorithms and hardware become inadequate. They will either take very large amount of time to process and analyze or the traditional hardware becomes insufficient to even load data into memory, without which it is difficult to analyze the data. So, data scientists have come up with *High Performance*

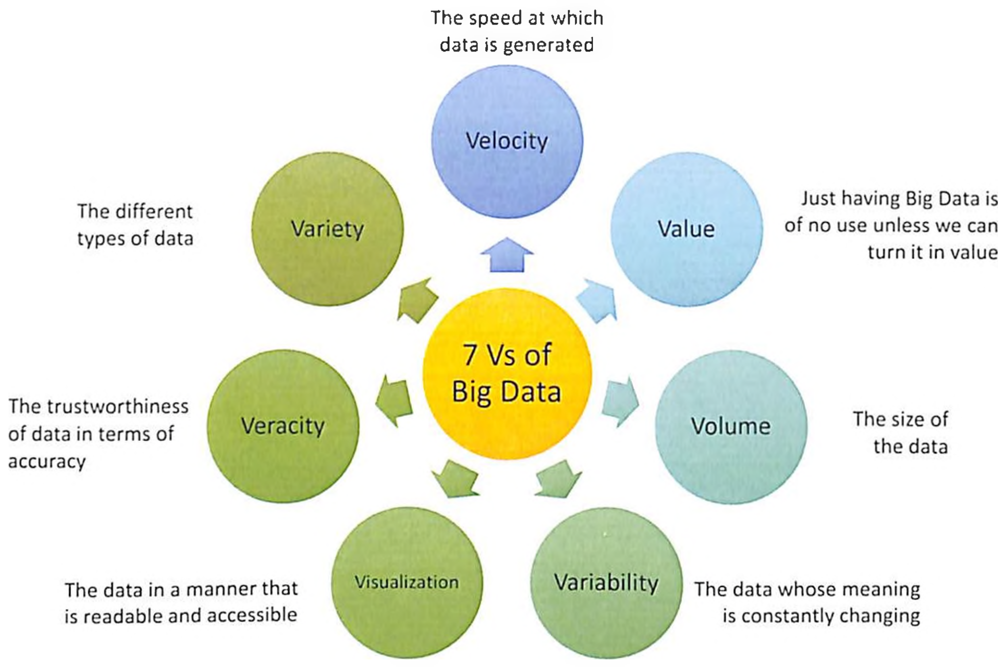

The speed at which data is generated
Velocity
Variety
The different types of data
Value
Just having Big Data is of no use unless we can turn it in value
7 Vs of Big Data
The trustworthiness of data in terms of accuracy
Veracity
Volume
The size of the data
The data in a manner that is readable and accessible
Visualization
Variability
The data whose meaning is constantly changing

Figure 1.5: The 7V's of Big Data (Source: [5])

Computing technologies which can process and analyze such large datasets. They use advanced hardware architectures such as *distributed memory, shared memory, hybrid of these two, distributed shared memory* and *accelerator based parallel architecture*. Distributed memory architecture typically consists of a cluster of computing nodes communicating with each other over a high band width network. HPC frameworks such as Hadoop [67, 68], NIMBLE [69], Spark [70], MPI [71], etc. work over distributed memory. Shared memory architectures comprise of a single workstation that has multiple processors (or processor cores) executing multiple threads in parallel, while sharing a global memory. Technologies such as Posix Threads, OpenMP [72], Intel-TBB [73], etc. support shared memory. Computational Scientists often use hybrid of distributed memory and shared memory architectures, giving rise to hybrid architecture. This comprises of a cluster of computing nodes in which each node supports multi-threading. Typically MPI + OpenMP or MPI + Pthreads combinations are used over such architectures. Distributed shared memory (or Partitioned Global Address Space Model) is again realized with a cluster of computing nodes but this time with a software layer that mimics the behaviour of shared memory. Technologies such as UPC (C) [74], CAF (Fortran) [75], Titanium (Java) [76], Global Arrays (Library) [77], X10 [78], etc. are used over distributed shared memory architectures.

Accelerator based parallel architectures are those that execute a large number of light weight threads in parallel while sharing a global memory. They are typically realized by GP-GPUs, FPGAs, Intel Xeon Phi co-processor [79], etc. Technologies such as CUDA [80], OpenCL [81], OpenACC [82] work over GP-GPUs; OpenCL works over FPGAs, and both OpenCL and OpenMP [72] work over Intel Xeon Phi co-processor.

Data scientists have come up with various data mining algorithms and libraries that leverage one or more of the above architectures to process and analyze large datasets. Few of them include - parallel classification algorithms [83, 84, 85], parallel clustering algorithms [86, 87, 88, 36, 89], parallel frequent itemset mining algorithms [90, 91, 92], etc. The performance of these algorithms scale with increase in hardware resources and thus enable data scientists to process large datasets. Apart from these there are openly available ready to use libraries that support parallel data mining algorithms. They include Apache Mahout [93] for Hadoop, MLlib [94] for Spark, etc.

1.2.2 Handling Streaming Data

The scenario of a data stream is different from that of large static data. Streams are typically characterized by continuously arriving data objects at a fast and variable rate. In streams we do not store data, rather detect patterns in the arriving data and store them. The patterns keep evolving, and these evolving patterns are captured by the stream mining algorithms within the constraints of limited time to process each object and limited memory to store incoming data. Moreover, the outdated patterns are also discarded time to time. This is typically achieved by using various mathematical models such as exponential decay [95], landmark window [96], sliding window [97], tilted-time window [98], etc.

Stream mining has got vast variety of applications. A few of them include - retail chain analysis, stock market analysis, web log analysis, network traffic analysis, mining data feeds from sensor networks, etc. Various stream mining algorithms that have been proposed in literature include - stream classification algorithms [99, 100, 101], stream clustering algorithms [102, 103, 95, 104], stream frequent itemset mining algorithms [98, 96, 97, 105], stream anomaly detection algorithms [106, 107, 108] etc. All these algorithms

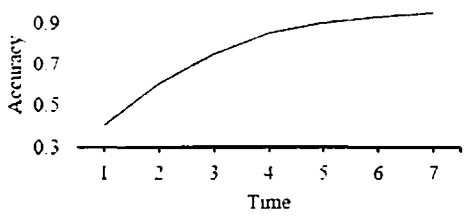


Figure 1.6: Characteristic of an anytime algorithm

usually handle single port stream on a single workstation. For handling multiple streams, a few multi-port mining algorithms have also been proposed. Few of them include - [109, 110, 111, 112, 113, 114, 115]. These algorithms typically leverage distributed memory frameworks for their efficient execution.

1.2.2.1 Anytime Stream Mining

A class of stream mining algorithms that handle varying inter-arrival rate of data streams is known as anytime stream mining algorithms. The stream mining algorithms proposed in literature (described earlier) are budget algorithms, i.e. they are designed for a fixed maximum stream speed (known as *budget*). When the stream speed is higher than the budget, they would have to either process sampled data or buffer unlimited data and eventually fail [104]. And when the stream speed is lower, they sit idle after processing the current data object, until the next one arrives. An ideal algorithm, however, should be able to process any stream speed. Higher speeds should be handled using deferred insertions and spare time available while processing lower speed streams should be utilized for refining the information received. Anytime algorithms are such algorithms that handle variable and high stream speeds and produce a mining result for any given processing time allowance for the incoming objects. They produce a quick approximate result and improve its accuracy with increase in time allowance as shown in Figure 1.6. A few such algorithms proposed in literature include - anytime classification [100, 116], anytime clustering [104, 117, 118], and anytime anomaly detection [119].

1.3 Research Gap and Motivation

Amidst many algorithms and techniques available in literature to process big data, there are quite a few gaps that are identified in terms of availability of specific data indexing mechanisms for mining large static and dynamic stream datasets. We present the identified gaps as follows:

- *Efficient data indexing structures tailor-made for data mining algorithms.* Data mining algorithms such as DBSCAN, OPTICS, KNN Classifier, etc, rely heavily on usage of region (neighborhood) and nearest neighbor queries. They essentially use data indexing structures such as R-tree (and its variants) or kd-tree for indexing data in them such that the above queries are efficiently executed. However, these data structures have certain drawbacks that limits their performance. R-tree and its variants have the problem of downward propagation of overlap in its internal nodes. kd-tree's binary nature leads to large height especially while indexing large datasets. Moreover, these data structures are borrowed from database systems and do not capture any specific querying requirements or patterns of any specific data mining algorithms.
- *Efficient data indexing structures and algorithms for anytime mining of data streams.* Data streams are characterized by arriving data objects at fast and variable rate. Typical stream mining algorithms can not handle streams that have variable or very high inter-arrival rates. Anytime algorithms handle the above streams without any stalls. However, no anytime stream mining algorithm exists for many mining tasks such as set-wise classification and frequent itemset mining. Also, the available anytime mining algorithms are very basic and have drawbacks. There is a lot of scope for designing better and efficient algorithms for anytime mining of variable speed data streams.
- *Efficient data distribution strategies for executing parallel spatial clustering algorithms over distributed memory architectures.* When parallel spatial clustering algorithms are executed over a cluster of computing nodes, the first step is to distribute data over the computing nodes. This distribution is usually done so as to achieve load balancing and preserve spatial locality. This improves the performance of the algorithms,

as these algorithms require spatial locality for execution of various spatial queries (neighborhood and nearest neighbor queries). To achieve this, researchers typically use random or kd-partitioning to distribute data over the nodes. However again, these are not specifically made for parallel spatial clustering algorithms, nor do they are designed to optimize the performance of any specific class of parallel clustering algorithms. There is a lot of scope for designing efficient partitioning strategies both for large static data as well as dynamic incremental data, which can capture specific spatial patterns of parallel density based and hierarchical clustering algorithms, maximize spatial locality preservation and achieve their efficient execution.

1.4 Thesis Contributions

A summary of the main contributions of the thesis are listed below.

- We developed a new data indexing structure known as *Grid-R-tree* which is a simple, yet effective adaptation of R-tree using the concept of Grid. It is a two-level R-tree in which the first level R-tree (known as *global-R-tree*) is for indexing "non-empty" cells resultant of virtual gridding of the search space, and the second level comprises of multiple R-trees (*cell-R-trees*) one each for every cell to index points lying in it. Grid-R-tree supports efficient execution of *region* and *nearest neighbor* queries. It addresses the drawbacks of the conventional data structures such as R-tree, kd-tree, etc. and is experimentally found to handle the above queries more efficiently. Grid-R-tree supports a novel query called *cell-wise ϵ-neighborhood query* (CellWiseNBH), which performs locality aware execution of neighborhood queries observed in density-based clustering algorithms (DBSCAN and OPTICS), and thus makes them efficient. The experimental analysis also demonstrates that using Grid-R-tree for k-NN classifier and DBSCAN clustering algorithm improves their efficiency.

- New anytime mining algorithms for data streams
  - We proposed an Anytime Frequent Itemset mining algorithm for data streams, AnyFI. To the best of our knowledge, this is first such attempt. It uses novel data structure known as *Buffered Frequent Itemset Forest* (BFI-forest), which aids

AnyFI to handle variable inter-arrival rate of incoming transactions in a stream. Its design also enables a user to obtain immediate mining results with best possible accuracy for the available time allowance and improve its quality with increase in time allowance. We also propose MPAnyFI for *anytime FI mining of multi-port data streams* over commodity clusters. It uses AnyFI at each computing node and stores the aggregate FIs in a *tilted-time window framework.* The experimental analysis shows that AnyFI can handle greater speed streams upto 60,000 transactions per second (*tps*), and produce mining results with recall close to 100%. The comparative analysis shows that AnyFI handles higher stream speeds and mines for FIs efficiently, when compared to the existing algorithms. The experiments conducted over MPAnyFI also show its efficiency.

- We propose AnySC, which is an *Anytime Set-wise Classification algorithm* for data streams. To the best of our knowledge, this is the first such approach. It uses a proposed data structure known as *CProf-forest*, for set-wise classification of variable speed data streams. AnySC incrementally updates the CProf-forest using the objects arriving in the stream within any given time allowance dictated by the stream speed. AnySC leverages the hierarchical structure of CProf-trees to classify the test entities within any given processing time allowance. The experimental results show that AnySC can: (*i*) handle variable stream speeds and produce accurate classification results; (*ii*) handle very high speed streams with reasonable performance, unlike the baseline approaches that fail to execute when speed exceeds its budget; (*iii*) give very high classification accuracy when stream speed is low, since it makes use of greater time available to refine the results to the greatest possible degree.

- We propose a framework known as AnyClus which performs online anytime maintenance of micro-clusters for handling data streams that have variable inter-arrival rate of data objects. It uses a proposed variant of R-tree known as *AnyRTree*, that stores summary statistics of the arriving objects in the form of micro-clusters at hierarchical granularities. The design of AnyRtree enables AnyClus to effectively handle noise and capture concept drift. The experimental analysis establish that AnyRTree maintains micro-clusters more compactly and with greater quality when compared to the existing methods. We also extend AnyClus into a parallel framework known as Any-MP-Clus for anytime clustering of multi-port data streams over distributed memory architectures (a cluster of computing nodes). The experiments performed over Any-MP-Clus establish that it can handle multi-port data streams of billions scale efficiently and produce clustering of high quality.

- Data Distribution Strategies
  - We proposed three distribution strategies for distribution of large static data across computing nodes of a cluster for execution of parallel clustering algorithms. They include - *PD-Split*, *Pbased-Split*, and *CD-Split*. Each of the distribution strategies preserve spatial locality and ensure perfect or near perfect load balancing. We experimentally demonstrate the applicability of the proposed methods for various classes of parallel clustering algorithms and benchmark them against the known KD-Split and random partitioning schemes, in terms of parallel data mining algorithms such as DBSCAN, SLINK and SNN.
  - We proposed DD-Rtree, a novel dynamic distributed data structure, based on R-tree [18]. It is useful for distribution of dynamic and incrementally generated data across computing nodes of a cluster for executing parallel spatial data mining algorithms efficiently. The distribution preserves spatial locality and ensures proper load balancing. DD-Rtree structure consists of R-trees at two levels. The first level R-tree is the $index - R - tree$ (IR-TREE), which serves as the index during construction/insertion. The second level comprises of multiple R-trees for each compute node ($MR - TREE$), which indexes data belonging to that. DD-Rtree was experimentally found to outperform best known existing methods in terms of locality in distribution, communication cost, construction & querying time, and performance of parallel data mining algorithms.

All algorithms, including those used for bench-marking, have been implemented in

C/C++. All parallel programs were implemented using MPI and OpenMP. The codes and datasets used for various problems shall be made publicly available on github after all of them get published.

This thesis deals with 3 V's of Big Data - *Volume, Velocity,* and & *Variability*. Grid-R-tree and data distribution strategies deal with the volume, whereas anytime stream mining algorithms deal with velocity and variability.

1.5 Thesis Organization

This thesis has been divided into three parts. Part I on page 16 presents the proposed data indexing structure - *Grid-R-tree*. This part has two chapters - Chapter 2 on page 19 presents the *Grid-R-tree* structure and demonstrates the efficiency in execution of neighborhood and nearest neighbor queries supported by it; Chapter 3 on page 50 presents the usefulness of Grid-R-tree in spatial data mining algorithms. Part II on page 57 presents three algorithms proposed for anytime mining of data streams. It is divided into three chapters - Chapter 5 on page 103 presents AnySC, which is the first anytime set-wise classification algorithm for data streams; Chapter 6 on page 126 presents AnyClus and Any-MP-Clus, which are anytime micro-cluster maintenance frameworks for single port and multi-port data streams respectively; Chapter 4 on page 60 presents AnyFI and MP-Any-FI which are frameworks for anytime Frequent Itemset Mining of single-port and multi-port data streams respectively. Part III on page 153 presents data distribution strategies used for distributing data during execution of parallel data mining algorithms. This part comprises of two chapters - Chapter 8 on page 185 presents a dynamic distributed data structure known as DD-Rtree, which is useful in distributing dynamic incremental data into computing nodes of a cluster while preserving spatial locality in its distribution. Chapter 7 on page 158 presents a few data distribution strategies for large static datasets while maintaining spatial locality and adopting to specific requirements of parallel spatial data mining algorithms. Chapter 9 concludes the thesis and throws an insight on future directions.

Part I

Data Indexing Structures for Efficient Spatial Queries in Data Mining

Multi-dimensional indexing structures are extensively used in different data mining algorithms such as density based clustering algorithms (DBSCAN [34] & OPTICS [35, 120]), hierarchical clustering algorithms (S-LINK [30], C-LINK [6], Average LINK [6]), k-nearest neighbors (k-NN) classifiers [121], etc. For example, *R-tree* [122] and its variants (like *R^+ tree, R^* tree, Hilbert-R-tree,* etc. [123]), *k-d-tree* [124], *quad-tree* [125], *Grid file* [126], etc. are used for efficient execution of *region queries* (*point, window & neighborhood queries*) and *nearest neighbor queries.* These indexing structures were originally designed to index spatial data in database applications. Queries supported by these structures are extensively used in data mining algorithms mentioned above.

These above mentioned data structures are not specifically designed for use in data mining algorithms, i.e.they do not capture any specific access patterns of data mining algorithms. Moreover, these data structures have certain drawbacks associated with them. In R-tree and its variants, the query performance suffers with increase in size and dimensionality of the dataset [123] because, overlap amongst their nodes leads to a very large search space. This problem afflicts most variants of R-trees except *R^+-tree* [123]. But, R^+-Tree is not commonly used because it exhibits higher fan-out and allows duplication of data points in the leaf nodes of the tree, leading to higher memory requirement, and thus rendering it inefficient for indexing large data sets. Moreover, R-tree and its variants do not perform good as the size and dimensionality of data increases. This is because the overlap between MBRs of its nodes increases with increase in data size and dimensionality [123].

Query performance suffers also in k-d-tree and quad-tree. The smaller branching factor in case of k-d-tree results in increased height while indexing large datasets, thus becoming inefficient for neighborhood queries. Quad-tree, on the other hand, is space inefficient for high dimensional data because of presence of a large number of nil pointers [126]. Further, quad-tree is not height balanced and thus does not provide asymptotic guarantees.

Grid file [126] is a hash-based structure that is developed specifically for multi-key accesses. It is commonly used in data indexing and space partitioning. It consists of a *grid file directory* (typically a d-dimensional array) to index cells resultant of gridding, and one *bucket* (a linked list) per cell for indexing data points. Although, the usage of array for the grid file directory guarantees $O(1)$ asymptotic complexity for query retrieval, it leads to inefficient space utilization while indexing datasets of high dimensionality. This is because, as the number of dimensions grow, the distribution of points in the space becomes sparse, leading to large number of null pointers. In such cases, linked lists are usually employed to store the grid directory, which saves space but increases look up time to $O(n)$. Similar problem is observed when the data distribution is skewed. The other limitation of grid file is the chaining of points in buckets, which also results in linear access time. Thus, grid files are efficient only when the number of dimensions are not large and data distribution is uniform [127].

Density-based clustering algorithms like DBSCAN [34] and OPTICS [35] exhibit a phenomena where the neighborhood queries of points lying within the locality of a given point are executed closely rather than randomly. None of the existing data structures exploit this locality to improve their query performance. When R-tree and its variants, k-d-tree or quad-tree are used for such neighborhood queries, the same search subspace is searched repeatedly, (i.e., the same path in the tree is traversed repeatedly) for neighborhood computations of all points lying in one region. Thus, it will be beneficial if a single traversal can compute the neighborhoods for all the points lying in that region.

A data structure called GR-tree [128] has been reported in literature which was designed using R-tree and Grid coordinate division. GR-tree is also two level tree in which first level is an R-tree with leaves as intermediate MBRs. The second level is a coordinate tree which is constructed based on grid coordinate division for each of the leaf level MBR of the first level tree. A coordinate tree represents gridding in the form of a tree, where each leaf represents a cell that stores a circular linked list of points. But, in this GR-tree, the leaves of the first level R-tree are overlapping, which results in searching multiple coordinate trees to locate a data point. This deteriorates the efficiency of a neighborhood query, especially for datasets of high dimensionality and large sizes. Other disadvantage of GR-tree is that the points indexed in each cell are stored in a circular linked list which yields a linear search time. Also, the authors of GR-tree do not validate their results for datasets of large size and high dimensionality. Moreover, GR-tree does not explicitly capture any specific pattern of any of the data mining algorithms.

From the above discussion, we can clearly observe that none of the above described data structures were designed specifically for use in data mining and do not capture any typical requirements of data mining tasks. So, there is a need for efficient data structures for "indexing and mining" large, high-dimensional datasets, and at the same time capture the querying requirements of data mining algorithms. This part has two chapters. In the first chapter (Chapter 2 on the following page), we present a proposed data structure known as Grid-R-tree, which addresses the above limitations and facilitates efficient execution of neighborhood and nearest neighbor queries, specifically for spatial data mining algorithms. The second chapter Chapter 3 on page 50 presents the applicability of Grid-R-tree to spatial data mining algorithms.

Chapter 2

Grid-R-tree: A data structure for efficient neighborhood and nearest neighbor queries

In this chapter, we present a proposed multi-dimensional indexing structure known as *Grid-R-tree*, which is a two-level generalization of R-tree using the concept of grid. The aim of this data structure is to facilitate efficient execution of neighborhood and nearest neighbor queries used in spatial data mining algorithms and also to facilitate efficient execution of clustering algorithms.

2.1 Grid-R-tree

Following the discussion on R-tree presented in Appendix A on page 206, we now present our proposed data structure - *Grid-R-tree*. Grid-R-tree is adaptation of R-tree using Grid. We first propose its design and structure, followed by its construction and complexity analysis.

- Poonam Goyal, Jagat Sesh Challa, Dhruv Kumar, Navneet Goyal, Sundar Balasubramaniam. *Grid-R-tree: A data structure for efficient neighborhood and nearest neighbor queries in data mining*. Submitted for review in Journal of Data Science and Analytics (JDSA), Springer.

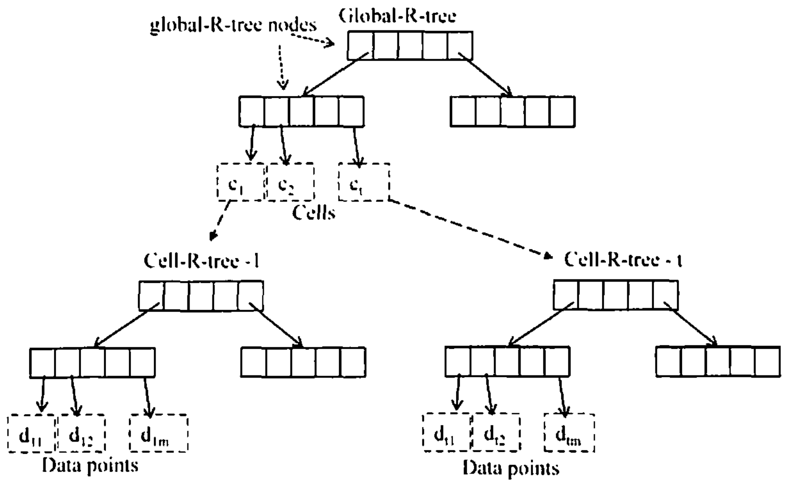


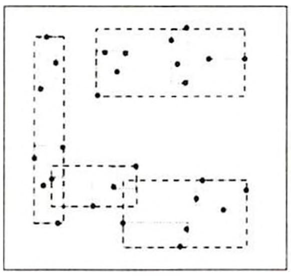
Figure 2.1: Structure of Grid-R-tree

2.1.1 Grid-R-tree: Design & Structure

The structure of Grid-R-tree has been illustrated in Figure 2.1. Grid-R-tree comprises of R-trees at two levels. First, for a given d-dimensional dataset, a d-dimensional uniform virtual grid is first superimposed over the entire search space of the dataset, creating cells which are hyper-cubes (Figure 2.3 on page 22). The "non-empty" cells resultant of this gridding are stored in the first level R-tree, referred to as *global-R-tree* (see Figure 2.1). Then a separate R-tree, referred to as *cell-R-tree*, is constructed for every cell to index data points belonging to that cell. The global-R-tree together with cell-R-trees constitute the Grid-R-tree data structure.

Each node (both internal and external) of the global-R-tree store between Gm and GM entries (where $Gm < GM/2$), except the root which can have less than Gm entries. The entries in the internal nodes store d - dimensional minimum bounding rectangles (referred to as *Gmbrs*). A Gmbr stores a bounding rectangle that contains all the regions indexed by its children. External nodes consist of d-dimensional cells.

Similarly, each node (both internal and external) of a cell-R-tree store between Rm and RM entries (where $Rm \leq RM/2$), except the root which can have less than Rm entries. The entries in the internal nodes store minimum bounding rectangles (Rmbrs) and entries in the external nodes store pointers to data points.


(a)

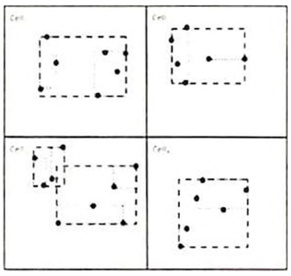

(b)

Figure 2.2: Illustrating MBRs formed over a two dimensional synthetic dataset using (a) R-tree (b) Grid-R-tree

Benefits of Grid-R-tree design. The design of Grid-R-tree into two levels reduces the overall overlap amongst the tree nodes and subsequently leads to reduction of search space. Figure 2.2 shows the MBRs of R-tree and Grid-R-tree when constructed over a small synthetic 2-dimensional dataset comprising of 27 data points. The fanout values for all kinds of trees were chosen to be 2 and 4. All the trees are of two levels. It is very clear from the figure that the MBRs formed by Grid-R-tree (formed by dividing the search space into 4 disjoint cells and then constructing separate R-tree for each cell) exhibit lesser overlap than those formed by R-tree. This reduction in overlap reduces the search space (number of nodes traversed) and computational cost of region and nearest neighbor queries (see Section 2.3.2 on page 38 for experimental results). Note that the quantum of reduction in cost depends on the size of the cells.

2.1.2 Virtual Gridding

Figure 2.3 on the following page shows a virtual grid constructed over the search space of the dataset. We call it virtual because we are not storing any grid index in physical memory. Initially, we compute the data ranging across each dimension at the time of reading the dataset and store that information. Then, we do uniform gridding, i.e. we divide the range (computed above) equally in each dimension, using the length of the cell c (cell size). The cell size c is a very important parameter to be chosen. We follow a heuristic presented in [129, 130], where c is estimated using the average density of the cell, i.e.:

$$c = \sqrt[d]{\frac{V}{N}} \times \tau \quad \text{where} \quad V = \prod_{i=1}^{d}(A_i - B_i)$$

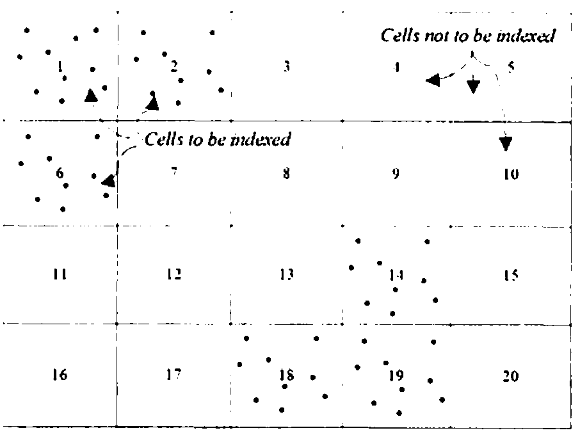


Figure 2.3: Virtual Grid in 2D

where V is the d-dimensional volume of the data ranging of the entire dataset, A_i and B_i correspond to maximum and minimum possible values, respectively, of any data point in the dataset across i^{th} dimension, and N is the data size. τ is a given threshold on the number of points per cell. If the distribution of data points is uniform, the above formula would result in exactly τ points in each cell and it would result in optimal load distribution and query performance. However, for skewed datasets, this would result in certain cells having more number of points than the others, resulting in deterioration of query performance. To overcome this problem, we apply adaptive gridding over the space as explained below:

2.1.2.1 Adaptive Gridding

We perform adaptive gridding on the dense cells. Cells with number of points more than τ, are treated as dense. For every dense cell, we divide the cell size by two in each dimension, resulting in division of the cell into 2^d cells. The non-empty cells that result from this gridding over each dense cell, will be stored in a separate Grid-R-tree referred to as the *cell-Grid-R-tree*. This cell-Grid-R-tree is local to a cell. If any of the divided cells are found to be dense again, the same procedure is applied recursively until no cell is dense. And, finally cell-R-trees are constructed over all the non-dense cells. In practice, not more than two levels of division was required for the data sets used in experimentation. Thus, adaptive gridding handles datasets with variable densities without degrading the query performance. Empirically, we observed that the value of τ in the range of 2000 to 3000 would result in optimal performance of region and nearest neighbor queries in most of the cases (see Section 2.3 on page 36 for experimental results). The results clearly show that the queries using Grid-R-tree with adaptive gridding referred to as *dense optimized* (DO), perform better than their respective non-optimized versions over skewed datasets.

2.1.3 Grid-R-tree: Construction & Insertion

Grid-R-tree is constructed for a given dataset (or a data list) DL, in three steps as described by the pseudo code presented in Algorithm 2.1 on the following page. In the first step, the data ranging across all dimensions is computed and stored in MINGRIDSIZE and MAXGRIDSIZE arrays (lines 4-5 of Algorithm 2.1 on the next page). In the second step, the global-R-tree is constructed over DL (lines 6-14 of Algorithm 2.1 on the following page). Here, for every point p in DL, we first calculate the coordinate boundaries of the cell to which p would belong to (see Algorithm 2.2 on the next page). We then check if a cell with these boundaries exists in the global-R-tree constructed until now in a top-down recursive search similar to that of search in an R-tree (using Algorithm 2.3 on page 25). If the cell exists in the global-R-tree we simply insert p into its PointsList. Else, we create a new cell with the coordinate boundaries calculated above, insert p to its PointsList, and then insert the new cell into the global-R-tree and the cells list CL. Insertion of a cell into global-R-tree (see Algorithm 2.4 on page 25) is also a top-down recursive function similar to that of R-tree insertion. In the third step, after all the points are inserted into their respective cells in global-R-tree, we construct cell-R-trees (which are conventional R-trees) separately for each cell over the data points indexed in their respective PointsLists (lines 15-16 of Algorithm 2.1 on the following page).

The average time complexity of construction of global-R-tree is the complexity of inserting all T cells in it, which is $\mathcal{O}(T \log_{G_m} T)$. If the data size is N, the average number of points in a cell will be N/T, assuming uniform gridding. Then the average case time complexity of construction of all T cell-R-trees is

$$\mathcal{O}\left(T\frac{N}{T}\log_{R_m}\left(\frac{N}{T}\right)\right) = \mathcal{O}\left(N\log_{R_m}\left(\frac{N}{T}\right)\right)$$

Thus, the total average time complexity of constructing a Grid-R-tree is the sum of complexities of global-R-tree construction and cell-R-trees construction, which is:

$$\mathcal{O}\left(T\log_{G_m} T + N\log_{R_m}\left(\frac{N}{T}\right)\right)$$

Algorithm 2.1: Construction of Grid-R-tree

```
procedure CONSTRUCT-GRID-R-TREE ()
    Input  : List of data points DL.
    Output: Grid-R-tree G constructed

    // Step1
    Initialize an empty Grid-R-tree G;
    Initialize an empty cells list CL;
    int $MINGRIDSIZE[d]$, $MAXGRIDSIZE[d]$;
    Find range of values across all d dimensions for all data points and store in MINGRIDSIZE and
      MAXGRIDSIZE arrays;

    // Step2
    foreach Data point p in DL do
        Region r = DETERMINE-CELL(p);
        if (cell C=CHECK-IF-CELL-EXISTS(G.global-R-tree,r)) $\neq$ NULL then
            Insert p into C.pointsList;
        else
            Initialize a new cell C2 with boundaries as r;
            Insert p into C2.pointsList;
            ADD-CELL-TO-GLOBALRTREE(C2, G.global-R-tree);
            ADD-CELL-TO-CELLSLIST(C2, CL);
        end
    end

    // Step3
    foreach cell C3 in CL do
        C3.cell-R-tree = CONSTRUCT-R-TREE(C3.pointsList);
    end
```

Algorithm 2.2: Determining the Cell to which p belongs to

```
procedure DETERMINE-CELL()
    Input  : Data point p, MINGRIDSIZE array, cell size c
    Output: Region r containing boundaries of the cell to which p would belongs to
    int temp; region r;
    foreach dimension i do
        $temp = \lfloor \frac{p[i] - MINGRIDSIZE[i]}{c} \rfloor$;
        $r.bottomLeft[i] = MINGRIDSIZE[i] + temp \times c$;
        $r.topRight[i] = MINGRIDSIZE[i] + (temp + 1) \times c$;
    end
    return r;
```

This complexity is of similar order as that of R-tree. The functions G-SPLIT-NODE(), G-PICK-CHILD(), etc. used in Algorithm 2.4 on the next page are similar to the corresponding functions for R-tree (see Appendix A on page 206).

Grid-R-tree also supports dynamic incremental insertion of a random point into it. To insert a given point p into an existing Grid-R-tree we can follow the same top-down recursive approach explained in construction. We first determine the coordinates of the cell to which p belongs to. If a cell with these coordinates exists in global-R-tree, we simply add p into its PointsList and corresponding cell-R-tree. Else, we create a new cell, insert p into it, and insert the new cell into global-R-tree and CL. The average time complexity of inserting a point into Grid-R-tree equals the sum of average case complexities of inserting

Algorithm 2.3: Checking if cell exists in global-R-tree

```
procedure CHECK-IF-CELL-EXISTS()
    Input  : node node of G.global-R-tree, region r of the cell
    Output: returns the cell if it exists, returns NULL otherwise
    if node.type == EXTERNAL then
        if node indexes a cell C with region r then return C ;
        else return NULL ;
    else if node.type == INTERNAL then
        foreach entry e of node do
            if e contains region r then cell C2 = CHECK-IF-CELL-EXISTS(e.child) ;
        end
    end
    return C2
```

Algorithm 2.4: Adding a Cell to Global-R-tree

```
procedure ADD-CELL-TO-GLOBAL-R-TREE()
    Input  : cell C, node node of global-R-tree/sub-tree
    Output: C inserted into global-R-tree
    if node.type == EXTERNAL then
        Insert C into node as a new entry;
        if node overflows then
            G-SPLIT-NODE(node); // splits the node into two and inserts both the nodes into global-R-tree and
              does necessary adjustments
        end
        G-UPDATE-MBRS-BOTTOM-UP(node);
    else if node.type == INTERNAL then
        bestChild = G-PICK-CHILD(node); // returns most appropriate child to insert
        ADD-CELL-TO-GLOBAL-R-TREE(bestChild); //recursive call
    end
```

a cell into the global-R-tree and inserting a point into cell-R-tree, which is

$$\mathcal{O}\left(\log_{Gm} T + \log_{Rm} \frac{N}{T}\right)$$

This is of the same order as that of insertion in R-tree which is $\mathcal{O}(\log_m N)$.

2.1.4 Deletion in Grid-R-tree

Deletion of a point p from Grid-R-tree is performed in a manner similar to insertion. First the cell to which the point p belongs to is identified. Then the point p is removed from that cell's cell-R-tree. If the cell becomes empty, it has to be removed from global-R-tree. If removal of the cell from global-R-tree or removal of point from cell-R-tree causes node underflow, it has to be handled in a manner similar as that of R-tree (Appendix A on page 206).

2.2 Queries over Grid-R-tree

In this section, we discuss various queries such as *point* query, *window* query, *neighborhood* query and *nearest neighbor* query supported by Grid-R-tree. The details of these queries are discussed in Appendix A on page 206.

2.2.1 Point Query

Point query checks the existence of a given data point p in a dataset. The algorithm executing Point query over Grid-R-tree is illustrated in Algorithm 2.5 on the next page. The algorithm first checks the existence of the cell, to which the data point would belong, in the global-R-tree. If the cell exists, then the query becomes a point query over its cell-R-tree, which is illustrated in Algo Algorithm A.6 on page 209. The average time complexity of point query over Grid-R-tree is the sum of the complexities of the query over global-R-tree and point query over cell-R-tree, which is $\mathcal{O}\left(\log_{G_m} T + \log_{R_m} \frac{N}{T}\right)$.

2.2.2 Cell Window Query

The *cell-window query* or *cell query* is a query which executes over the global-R-tree of Grid-R-tree. This query is illustrated in Figure 2.4 for d=2, where it returns all the cells that are overlapping with a given region or a window r (cells highlighted in grey). Cell-window query is used by window and neighborhood queries. The algorithm executes in a top-down recursive fashion similar to that of a point query (see Algorithm 2.6 on the following page). It simply accumulates all the overlapping cells (present in leaves of the

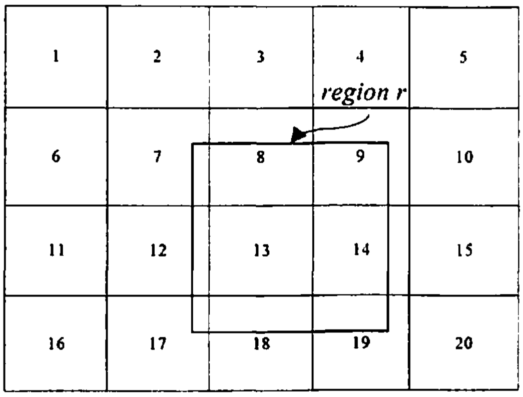


Figure 2.4: Illustrating Cell-Window Query

Algorithm 2.5: Point Query over Grid-R-tree

```
procedure POINT-QUERY-GRID-R-TREE ()
    Input  : Data point $p$, Grid-R-tree $G$
    Output: TRUE if $p$ exists in $G$, FALSE otherwise
    region $r$ = DETERMINE-CELL ($G$, $r$);
    cell $C$ = CHECK-IF-CELL-EXISTS ($G$, $r$);
    if $C \neq$ NULL then return R-POINT-QUERY ($p$, $C.cell\text{-}R\text{-}tree$) ;
    return FALSE;
```

Algorithm 2.6: Cell Window Query

```
procedure CELL-WINDOW-QUERY ()
    Input  : root $node$ of $G.global\text{-}R\text{-}tree$, region $r$, cells list $CL$
    Output: cells list $CL$ containing cells in $G$ overlapping with $r$
    if $node.type$ == EXTERNAL then Add all cells indexed at $node$, that overlap with $r$, to CL ;
    else if $node.type$ == INTERNAL then
        foreach entry $e$ of $node$ do
            if $e$ overlaps with $r$ then CELL-WINDOW-QUERY ($e.child$, $r$, $CL$) ;
        end
    end
```

global-R-tree) it encounters in its traversal to a temporary cells list CL and returns it. The average time complexity this query is $\mathcal{O}(\log_{G_m} T)$.

2.2.3 Window Query

Window query is a query which returns all the data points that lie in a d-dimensional window or a region r, from the entire data space. Figure A.4 on page 210 illustrates the window query for d=2. Algorithm 2.7 on the following page illustrates this for Grid-R-tree. This algorithm first calls the cell-window query which returns all the cells overlapping with the window r. Then, for each cell returned, window query is called over its cell-R-tree, which returns points belonging to that cell lying in r. The results of window queries from all these cell-R-trees are merged into a data list and returned. The average time complexity of this query is the sum of the complexities of cell-window query over global-R-tree and all the window queries over cell-R-trees, which is: $\mathcal{O}(\log_{G_m} T + T' \log_{R_m} (N/T))$ where, T' is the average number of cells returned by cell-window query.

2.2.4 Neighborhood Queries

Neighborhood query (or ϵ-neighborhood query) is a query that returns all the data points that are lying withing a distance of ϵ from a given point p. Figure A.5 on page 210 illustrates this query for d=2. ϵ-neighborhood queries are extensively used in density based clustering algorithms. Algorithm A.7 on page 210 explains how ϵ-neighborhood query is

Algorithm 2.7: Window Query over Grid-R-tree

```
procedure WINDOW-QUERY-GRID-R-TREE ()
    Input : root node of G.global-R-tree, region r
    Output: list of data points lying in region r
    Initialize empty data points lists plist and t_list;
    Initialize an empty cells list CL1;
    CELL-WINDOW-QUERY (node, r, CL1);
    foreach cell C in CL1 do
        R-WINDOW-QUERY (C.cell-R-tree.root, r, t_list);
        Append t_list to plist;
    end
    return plist;
```

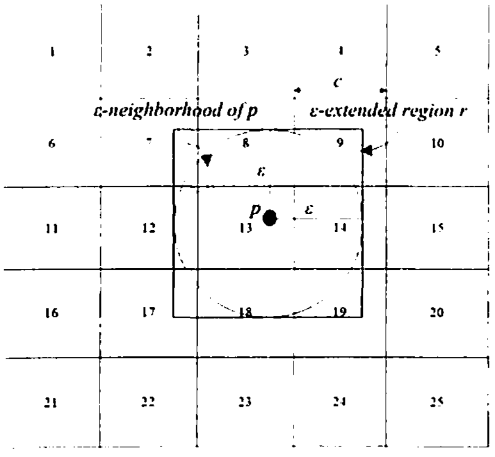


Figure 2.5: Illustrating Proposition 1 & Result 1 for $\lceil\frac{\epsilon}{c}\rceil = 1$

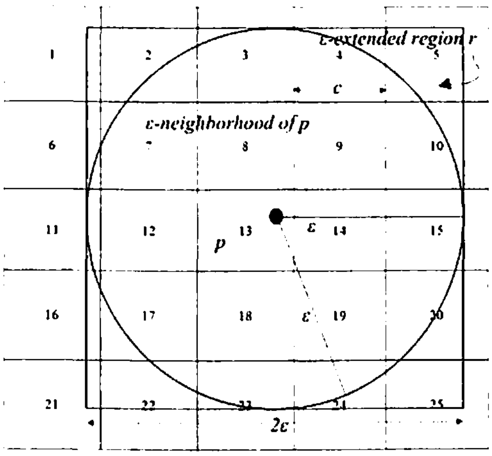


Figure 2.6: Illustrating Result 1 for $\lceil\frac{\epsilon}{c}\rceil = 2$

executed over an R-tree. Before explaining how ϵ-neighborhood query is executed over Grid-R-tree, we first present a few propositions and results.

Proposition 1. In Grid-R-tree, to find the points lying in ϵ-neighborhood of p, it is sufficient to check only those points that lie in the cells that are geometrically overlapping with the window r, which is the ϵ-extended region of p.

Justification. For two dimensions, we prove this claim geometrically. To compute ϵ- neighborhood of p, it is sufficient to examine only those points that lie in cells 7, 8, 9, 12, 13, 14, 17, 18 and 19 in Figure 2.5. The ϵ-neighborhood of p is fully contained in ϵ-extended region (r) of p. So, all the points that lie in the ϵ - neighborhood of p must be contained in the cells overlapping with r. So, any point that lies outside these cells will not contribute to ϵ-neighborhood of p. □

Result 1. The maximum number of cells to be considered for ϵ-neighborhood query of any point p over Grid-R-tree is $\left((2 \times \lceil\frac{\epsilon}{c}\rceil) + 1\right)^d$, where c is the cell size and d is the dimensionality of the dataset.

Proof. We prove this for two dimensions for simplicity. The proof is by induction on $\lceil \frac{\epsilon}{c} \rceil$.

Base Case: When $\lceil \frac{\epsilon}{c} \rceil = 1$, i.e., when $0 < \epsilon < c$, the maximum number of cells to be searched $=(2 \times 1 + 1)^2 = 9$ as shown in Figure 2.5 on page 28. For finding ϵ-neighborhood of any point $p \in$ cell 13, it is sufficient to search a maximum of 9 cells shaded in the figure. Similar illustration is given in Figure 2.6 on page 28 when $\lceil \frac{\epsilon}{c} \rceil = 2$.

Induction Hypothesis: Assume this result to be true for $\lceil \frac{\epsilon}{c} \rceil = l$, i.e. the maximum number of cells to be searched for this case is $(2l + 1)^2$.

Inductive Step: We have to prove this result for $\lceil \frac{\epsilon}{c} \rceil = l + 1$, i.e. the maximum number of cells to be searched for the case $j = l + 1$ is $(2(l + 1) + 1)^2 = (2l + 3)^2$. We need to find the number of additional cells to be added to that of the case for $\lceil \frac{\epsilon}{c} \rceil = l$, in order to get the maximum number. Since for $\lceil \frac{\epsilon}{c} \rceil = l$, ϵ will lie between $(l - 1)c < \epsilon < lc$ and for $\lceil \frac{\epsilon}{c} \rceil = l + 1$, ϵ will lie between $lc < \epsilon < (l + 1)c$, it is sufficient to add one more level of cells to our search region that appears shaded in Figure 2.7 on page 31. Now the size of the side of this new square becomes $2(l + 1) + 1 = 2l + 3$ and the maximum number of cells to be searched then becomes $(2l + 3)^2$. □

This result gives an upper bound of the number of cells to be searched while performing an ϵ-neighborhood query over Grid-R-tree. Grid-R-tree supports two kinds of such queries: *point-wise ϵ-neighborhood query* and *cell-wise ϵ-neighborhood query*. They are explained as follows:

Point-Wise ϵ-Neighborhood Query (PointWiseNBH) This query is the conventional neighborhood query which executes a window query for ϵ-extended region of p and selects those points that lie within ϵ-distance from p (see Algorithm 2.8 on the next page). The ϵ-extended region of p is constructed by extending the coordinates of p across all dimensions by ϵ, in both positive and negative directions. Figure 2.6 on page 28 shows this region for d=2. The average time complexity of point-wise ϵ-neighborhood query over Grid-R-tree is same as that of a window query, which is $O(\log_{G_m} T + T' \log_{R_m} \frac{N}{T})$.

Algorithm 2.8: Point-Wise NBH Query over Grid-R-tree

```
1 procedure POINT-WISE-NBH ()
      Input  : Data point $p$, Epsilon $\epsilon$, root node of G.global-R-tree
      Output: Points lying in $\epsilon$-neighborhood of $p$
2     Construct an $\epsilon$-extended region $r$ of $p$;
3     Initialize an empty list pList;
4     data list t_list = WINDOW-QUERY-GRID-R-TREE (node, r);
5     Insert all points of t_list that lie within $\epsilon$ distance from $p$ into plist;
6     return pList;
```

Cell-Wise ϵ-Neighborhood Query (CellWiseNBH) This query computes ϵ-neighborhoods of all the points in a given cell C in one go. This is an optimized way of computing ϵ-neighborhood queries for all the points lying in C against their computations for each point separately. In density-based clustering algorithms such as DBSCAN and OPTICS, the neighborhood computation of any given point p is followed by neighborhood computations of the points lying in the ϵ-neighborhood of p. CellWiseNBH optimizes these computations by reducing multiple traversals in the same search space. Before presenting its details, we first state the following proposition:

Proposition 2. To find ϵ-neighborhoods of all points lying in cell C, it is sufficient to check only those points that lie in the cells that are geometrically overlapping with the window r, constructed by extending the coordinates of C by ϵ on both sides across all dimensions (ϵ-extended region of C).

Justification. Figure 2.8 on the next page shows the ϵ-extended region r for cell 13. Consider the corner point p of cell 13. p's ϵ -neighborhood is contained within the cells overlapping with r. This is true for all boundaries and corner points of the cell. Therefore we can find ϵ-neighborhoods for all the points lying in this cell by examining only these highlighted cells. □

In CellWiseNbh query (see Algorithm 2.9 on the following page), first a list of cells containing all the cells overlapping with the ϵ-extended region of a given cell C is computed using the cell-window query and stored in an *auxiliary* Grid-R-tree. Then a window query is executed for every point p of the cell over this auxiliary Grid-R-tree. Those points that lie within the ϵ-distance from p are stored as ϵ-neighborhood of p. Since, this is repeated for every point in the cell, it saves multiple top-down traversals over the global-R-tree (multiple scans of same search space) for executing neighborhood queries for all points $\in$ C, thereby improving the overall query performance.

Algorithm 2.9: Cell-Wise NBH Query over Grid-R-tree

```
1 procedure CELL-WISE-NBH ()
      Input : Cell C, Epsilon $\epsilon$, root node of G.global-R-tree
      Output: $\epsilon$-neighborhoods of all points of C computed
2     Construct an $\epsilon$-extended region r for cell C;
3     Initialize an empty cells list CL1; CELL-WINDOW-QUERY(G.global-R-tree.root, r, CL1);
4     Grid-R-tree aux-G = CONSTRUCT-AUX-GRID-R-TREE (CL1);
5     foreach point p indexed by C do
6         Construct an $\epsilon$-extended region r2 for p;
7         t_list = WINDOW-QUERY-GRID-R-TREE (aux-G, r2);
8         Insert all points of t_list that lie within $\epsilon$ distance from p into p.neighborslist;
9     end
```

The average complexity of the executing cell-wise ϵ-neighborhood query over Grid-R-tree is the sum of the complexities of cell-window query, construction of auxiliary Grid-R-tree, search in auxiliary Grid-R-tree, and search in cell-R-trees:

$$\mathcal{O}\left(\log_{Cm} T + T' \log_{GAm} T' + \frac{N}{T} \log_{GAm} T' + \frac{N}{T} T' \log_{Rm} \frac{N}{T}\right)$$

where, the average number of cells returned by cell-window query is T', and GAm represents min-entries of auxiliary Grid-R-tree. The complexities for construction of auxiliary Grid-R-tree and search in auxiliary Grid-R-tree are very small when compared to that of cell-window query and search in cell-R-trees. So the first and last term in the above complexity dominate the middle two terms, making it equal to the complexity of Point-WiseNBH query.

The CellWiseNBH query can be used to make density-based clustering algorithms such as DBSCAN & OPTICS efficient. The implementation of DBSCAN using the Cell-WiseNBH query is presented in Section 3.1 on page 50.

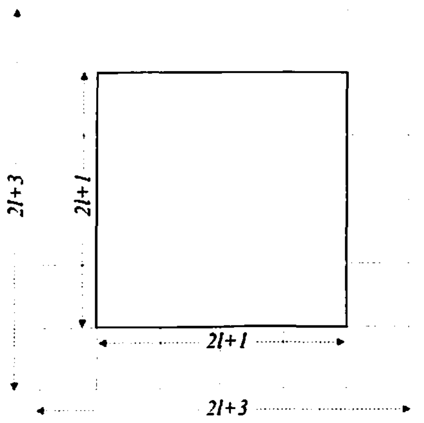


Figure 2.7: Illustrating Inductive Step of Result 1 on page 28

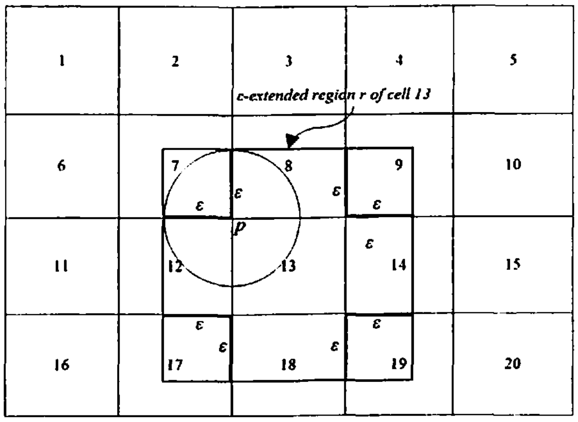


Figure 2.8: Illustrating Proposition 2 on page 30

Algorithm 2.10: Construct Auxiliary Grid-R-tree

```
procedure CONSTRUCT-AUX-GRID-R-TREE ()
    Input  : Cells List $CL$
    Output: Auxiliary Grid-R-tree $aux$-$G$
    Initalize an empty Grid-R-tree $aux$-$G$;
    foreach cell $C$ in $CL$ do
        ADD-CELL-TO-GRID-R-TREE ($aux$-$G$, $C$);
    end
    return $aux$-$G$;
```

Algorithm 2.11: k-NN Query over Grid-R-tree

```
procedure KNN-GRID-R-TREE ()
    Input  : Data point $q$, Grid-R-tree $R$, $k$
    Output: $k$ nearest neighbors of $q$
    Initialize Empty Priority Queue $PQ$;  int $i$ = 1;
    Add root node of the global-R-tree into $PQ$ along with its minDistance from $p$ as key;
    while $PQ$ not empty do
        element $ele$ = REMOVE-MIN ($PQ$);
        if $ele$ is an internal node of global-R-tree or cell-R-tree then
            Add all its entries to $PQ$ with their respective minDist (from $q$) as keys;
        else if $ele$ is an external node of the global-R-tree then
            Add the root nodes of cell-R-trees indexed at it, with their minDistances from $q$ as keys;
        else if $ele$ is an external node of a cell-R-tree then
            Add all its entries to $PQ$ with their respective euclideanDist (from $q$) as keys;
        else if $ele$ is a data point then
            report $ele$ as $i^{th}$ nearest neighbor;  $i$++;
            if $i > k$ then return;
        end
    end
```

2.2.5 Nearest Neighbor Query

k-nearest neighbor (*k*-NN) query is a query that returns the *k* closest data points to a given query point *p* [121]. *k*-NN query for *k*=6 is illustrated in Figure A.6 on page 210, where all the points within the circle form the *k* nearest neighbors of *p*. The best known algorithm for nearest neighbor search over R-tree is the *BF-kNN* [131] explained in Appendix A on page 206. It uses a min-priority queue (*PQ*) that stores nodes of an Grid-R-tree as well as data points indexed in it. The key for insertion into priority queue is the *euclidean distance* for data points and *minDist* for nodes (or MBRs of nodes). BF-kNN is a greedy algorithm with minimum distance as the greedy choice.

k-NN query algorithm for Grid-R-tree is similar to the *BF-kNN* algorithm except that the elements we store in the priority queue (*PQ*) are of three kinds:- *global-R-tree node, cell-R-tree node* and *data point*, in place of only the latter two in case of R-tree. *minDist(p,Z)* (*Z* is the MBR of cell-R-tree or global-R-tree) for both global-R-tree nodes and cell-R-tree nodes are computed in the same way as that for R-tree nodes. Algorithm 2.11 presents the pseudo code. The algorithm first adds the root node of a global-R-tree *G* into *PQ*.

Then in a loop it executes the following steps until k nearest neighbors are found. A REMOVE-MIN() operation is performed over PQ. If the min-object is an internal node of global-R-tree, we add all its indexed entries into PQ; else if it is an external node of global-R-tree, we add the root nodes of the cell-R-trees indexed in the cells stored at the present node. In both cases the entries are added with their respective minDist from q as key values. If the min-object is an internal node of a cell-R-tree, we add all its indexed entries into PQ with their respective minDist from q as key values; else if it is an external node of a cell-R-tree, we simply add all its indexed entries into PQ with their respective euclideanDist from q as the key. If the min object is a data point, it is marked as the i^{th} nearest neighbor. This step is repeated until $i > k$, with i initially set to 1.

Since, the overall flow of the algorithm for R-tree and Grid-R-tree is the same, the average case time complexity of BF-kNN algorithm over Grid-R-tree would be equal to that for an R-tree, which is $O(k \log k)$ [131]. However, the worst case complexity is $O(N)$, wherein all the nodes in the R-tree are added to the priority queue.

Please note that the pseudo codes presented above for both region and nearest neighbor queries are for Grid-R-tree without adaptive grid optimization. The queries over adaptive grid optimized Grid-R-tree would require a minor modifications to the above pseudo codes, and hence are not explicitly presented.

2.2.6 Theoretical Analysis

In this section, we give theoretical evidence that Grid-R-tree exhibits lesser overlap when compared to R-tree, resulting in a smaller search space and consequently in better query performance. A comparative analysis of R-tree and R+-tree has been presented in [132]. They compare the overlap exhibited amongst the tree nodes and number of nodes searched for a neighborhood query, for both kind of trees. We use this analysis to compare the overlap and number of nodes searched for Grid-R-tree and R-tree. The analysis originally presented assumes line segments stored in the leaf nodes. This can be very easily extended for storing d-dimensional data points. Also, the analysis assumes that all the nodes of trees store number of entries equal to the maximum fanout value. So, the analysis states that, the total number of nodes, N_n, that are searched for any neighborhood query over R-tree is:

$$N\_n = h + 1 + \left(\frac{O_v - 1}{C}\right) \cdot \left(\frac{f}{f-1}\right) \cdot \left(1 - \frac{1}{f^{N-1}}\right) \tag{2.1}$$

where h is the height of the R-tree and is given by $h = \log_f \frac{N}{C}$; O_v represents the overlap which is measured by the number of objects overlapping with the queried object and is given by:

$$O_v = \frac{\sigma}{1+\sigma}(N+1) \tag{2.2}$$

where C is the capacity of the page, N is the total number of data objects present in the tree, f represents the fanout and σ represents the size of each data object. When N is large, $(1 - \frac{1}{f^{N-1}}) \approx 1$. So, ignoring this term, Equation (2.1) reduces to:

$$N\_n = h + 1 + \left(\frac{O_v - 1}{C}\right) \cdot \left(\frac{f}{f-1}\right) \tag{2.3}$$

We use this result to prove the following claims for Grid-R-tree. In the analysis presented below, we chose the values of min-entries and max-entries for all the trees appropriately for number of elements indexed in them.

Theorem 1. The overlap in Grid-R-tree is less than that of R-tree, resulting in a smaller search space for neighborhood queries.

Proof. We compare the overlap and number of nodes searched in R-tree vs Grid-R-tree.

R-tree: We rewrite Equation (2.2) and Equation (2.3) here with the corresponding parameters for R-tree substituted:

$$O_{vR} = \frac{\sigma_p}{1+\sigma_p}(N+1) \tag{2.4}$$

$$N\_n_R = h + 1 + \left(\frac{O_{vR} - 1}{C}\right) \cdot \left(\frac{f_R}{f_R - 1}\right) \tag{2.5}$$

The page size, C, shall remain same throughout this analysis.

Grid-R-tree: In case of Grid-R-tree (G), we query on one global-R-tree and multiple cell-R-trees. Thus we consider the number of nodes visited and overlap separately for global-R-tree and cell-R-trees. Let N_G be the number of cells indexed in global-R-tree of G. Then, the average number of points in a single cell-R-tree will be $\frac{N}{N_G}$. Thus, we rewrite Equation (2.2) and Equation (2.3) for global-R-tree and cell-R-tree as Equation (2.6) & Equation (2.7) on the next page and Equation (2.8) on the following page & Equation (2.9) on the next page respectively:

$$O_{vG} = \frac{\sigma_c}{1+\sigma_c}(N_G+1) \tag{2.6}$$

$$N\_n_G = h_G + 1 + \left(\frac{O_{vG}-1}{C}\right) \cdot \left(\frac{f_G}{f_G - 1}\right) \tag{2.7}$$

$$O_{vC} = \frac{\sigma_p}{1+\sigma_p}\left(\frac{N}{N_G}+1\right) \tag{2.8}$$

$$N\_n_C = h_C + 1 + \left(\frac{O_{vC}-1}{C}\right) \cdot \left(\frac{f_C}{f_C - 1}\right) \tag{2.9}$$

Dividing Equation (2.4) on page 34 by Equation (2.6) on page 34, we get:

$$\frac{O_{vR}}{O_{vG}} = \frac{\frac{\sigma_p}{1+\sigma_p}(N+1)}{\frac{\sigma_c}{1+\sigma_c}(N_G+1)} \quad \text{where} \quad \frac{\frac{\sigma_p}{1+\sigma_p}}{\frac{\sigma_c}{1+\sigma_c}} = \mathcal{O}(1)$$

So, we get

$$\frac{O_{vR}}{O_{vG}} \approx \frac{N+1}{N_G+1} \approx \frac{N}{N_G} \implies O_{vG} = O_{vR} \cdot \frac{N_G}{N} \tag{2.10}$$

Similarly, by dividing equation Equation (2.4) on page 34 with equation Equation (2.8), we get:

$$\frac{O_{vR}}{O_{vC}} \approx N_G \implies O_{vC} = \frac{O_{vR}}{N_G} \tag{2.11}$$

So, the total number of nodes to be accessed for a neighborhood query over G will be the sum total of number of nodes to be accessed for one scan of global-R-tree and number of nodes to be accessed for searching in N_G' cell-R-trees, where N_G' is the average number of cells returned by the query over global-R-tree of G. Thus,

$$N\_n_{GR} = N\_n_G + N_G' \cdot N\_n_C \tag{2.12}$$

Substituting the values of N_n_G and N_n_C in Equation (2.12), we get:

$$N\_n_{GR} = h_G + 1 + \left(\frac{O_{vG}-1}{C}\right)\left(\frac{f_G}{f_G-1}\right) + \left(h_C + 1 + \left(\frac{O_{vC}-1}{C}\right)\left(\frac{f_C}{f_C-1}\right)\right) N_G'$$

Now, $\left(\frac{f}{f-1}\right) = \mathcal{O}(1)$ for $f = f_G, f_C$ or f_R. So, we get:

$$N\_n_{GR} = h_G + 1 + (h_C+1)\cdot N_G' + \frac{1}{C}\left(O_{vG} + O_{vC}.N_G' - (N_G' + 1)\right)$$

Substituting the values of O_{vG} and O_{vC} from Equation (2.10) and Equation (2.11) respectively, we get:

$$N\_n_{GR} = h_G + 1 + (h_C+1)\cdot N_G' + \frac{1}{C}\left(O_{vR}\cdot\left(\frac{N_G}{N} + \frac{N_G'}{N_G}\right) - (N_G'+1)\right) \tag{2.13}$$

Now we compare the number of nodes searched in case of R-tree (Equation (2.5) on page 34) with that of Grid-R-tree (Equation (2.13)). The ratio of second terms in these two equations is:

$$\frac{O_{vR}-1}{\left(O_{vR}\cdot\left(\frac{N_G}{N}+\frac{N_G'}{N_G}\right) - (N_G'+1)\right)} \approx \frac{1}{\frac{N_G}{N}+\frac{N_G'}{N_G}}$$

This ratio is large, since $N_G' \ll N_G \ll N$, in practice. This indicated that the overlap in R-tree is higher than that in Grid-R-tree and thus making the number of nodes searched in R-tree on the higher side. Similarly, the ratio of first terms in these two equations is:

$$\frac{h+1}{h_G + 1 + (h_C + 1) \cdot N_G'} \approx \frac{h}{h_G + h_C \cdot N_G'}$$

Since $h_C < h_G < h$, and N_G' is small, this ratio, is likely to be very small (< 1). This poses negative impact on Grid-R-tree by increasing the number of nodes searched. But the positive impact of the reduced overlap is much higher than the negative impact of the increased height. Thus, the number of objects/nodes searched for a neighborhood query in Grid-R-tree is expected to be lesser than that in R-tree. This has been substantiated by an experiment which measures the actual number of nodes traversed by the neighborhood and nearest neighbor queries over Grid-R-tree as well as R-tree, presented in Table 2.2 on page 39. $\square$

2.3 Experimental Results and Analysis

2.3.1 Experimental Setup

All the experiments are conducted on Server that has - Intel Xeon 3.3GHz processor and 32 GB RAM. All the algorithms are implemented in C and the running time is measured using Vampir Trace [133]. The details of the datasets and their respective parameters used for experimentation are given in Table 2.1 on the following page, along with their references. First 13 datasets are real and the remaining are synthetically generated. 3DSRN data set contains geographical information (latitude, longitude and altitude) of road networks in Denmark. MPAGD*, SFONT1M and MPAHALO2.8M datasets are taken from Millennium data repository that contains astronomical data of galaxies. SBUS* datasets contain GPS traces of buses in Shanghai. KDDBIO dataset consists of 74 features of protein hematology. SHUTTLE data set contains data on the features of space shuttle. SKIN data set contains features related to texture of face images. The synthetic random datasets (SR1M2D, 3D & 5D) consists of randomly generated points. Synthetic uniform datasets (SU1M2D, 3D & 5D) comprises of points that are uniformly distributed over the search space. Synthetic normal datasets (SN1M2D, 3D & 5D) consists of data points generated

Table 2.1: Description of Datasets used for Experimentation

S No.	Name of the Dataset	Data Size	Dimensions	Value of Epsilon (ϵ)	Cell Size (c)	Reference
1.	3DSRN	0.34M	3	0.05	0.05	[134]
2.	MPAGD3.2M (Delucia)	3.2M	3	2	2.25	[135]
3.	MPAGD5M (Delucia)	5M	3	1.72	6	[135]
4.	MPAGD56M (Delucia)	56M	3	0.006	3	[135]
5.	MPAGD100M (Delucia)	100M	3	0.006	3	[135]
6.	SBUS2.7M	2.7 M	2	0.00625	0.002	[136]
7.	SBUS4M	4M	2	0.000419	0.0012	[136]
8.	SBUS6M	6M	2	0.000291	0.001	[136]
9.	KDDBIO	0.145M	74	0.005	0.005	[137]
10.	SFONT1M	1M	11	2	6	[135]
11.	MPAHALO2.8M	2.8M	9	30	45	[135]
12.	SHUTTLE	0.058M	9	4	6	[138]
13.	SKIN	0.24M	4	4	6	[139]
14.	SR1M2D	1M	2	10	30	-
15.	SR1M3D	1M	3	40	60	-
16.	SR1M5D	1M	5	120	180	-
17.	SU1M2D	1M	2	5	5	-
18.	SU1M3D	1M	3	4	4	-
19.	SU1M5D	1M	5	2	2	-
20.	SN1M2D	1M	2	0.1	0.15	-
21.	SN1M3D	1M	3	0.62	0.9	-
22.	SN1M5D	1M	5	2.3	3.2	-

using normal distribution. All synthetic datasets contain 1M data points with varying dimensions âĂŞ 2, 3 & 5.

The experimental results presented in the following subsections compare the performance of neighborhood and nearest neighbor queries over Grid-R-tree and R-tree. The results of point and window queries are not presented since they follow the same pattern as that of neighborhood queries. The following notations have been used in this section: RTreeNBH represents neighborhood query and RKNN represents k-NN query for R-tree. CellWiseNBH, PointWiseNBH and GRKNN represent the Cell-wise neighborhood query, point-wise neighborhood query and nearest neighbor query for Grid-R-tree respectively (without adaptive grid optimization). CellWiseNBH_DO, PointWiseNBH _DO and GRKNN_DO represent the respective queries for adaptive grid optimized Grid-R-tree. The value of ϵ and the cell size chosen for experiments are given in Table 2.1 for each dataset. In all our experiments, the length of the side of a cell (c) is kept uniform across all dimensions. However, one can also choose different lengths across different dimensions. The fanout values for all the trees have been appropriately chosen for number of elements indexed in them. In the experiments for k-NN queries, we choose $k = 4$, unless explicitly stated. Also, the execution times shown in the subsequent experiments are measured for executing queries for all the data points of the given dataset.

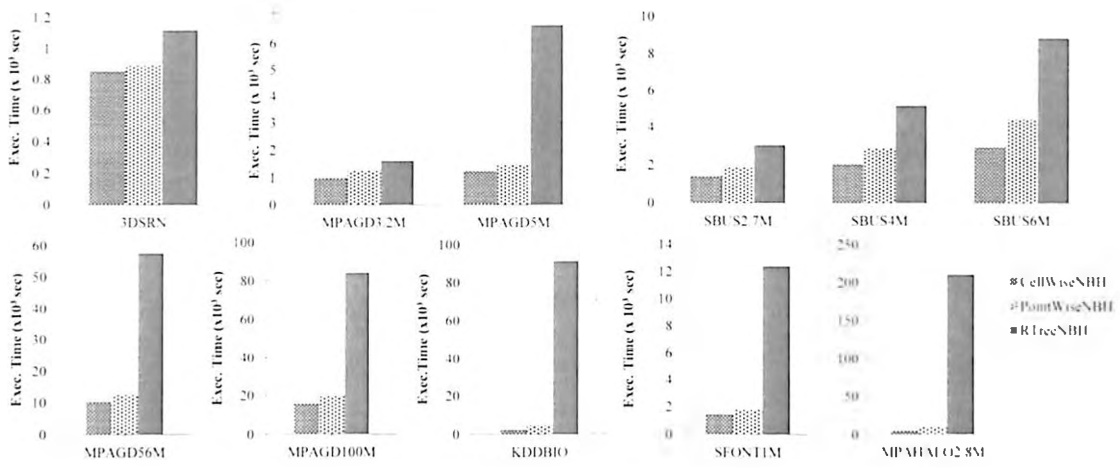


Figure 2.9: Execution time for neighborhood queries for Grid-R-tree and R-tree over various datasets

2.3.2 Performance Analysis of Neighborhood and k-NN Queries

In the first experiment, we measure the execution time of neighborhood and nearest neighbor queries when executed on various real datasets for both R-tree and Grid-R-tree, and the results are presented in Figure 2.9 & Figure 2.10 on the next page. The maximum size of a dataset considered in this experiment is 100M and maximum dimensionality is 74. We can clearly observe from the graphs that, in all cases, the execution time of queries over Grid-R-tree is much lesser than that of R-tree. The maximum speed up achieved is 42 for CellWiseNBH query, 21 for PointWiseNBH query and 25 for KNN query. This clearly indicates that Grid-R-tree outperforms R-tree for all the queries considered. The improvement achieved is attributed to reduction in search space when compared to that of R-tree (as explained in Section 2.1.1 on page 20). It can also be observed from these figures that greater reduction is achieved for the datasets of high dimensionality (MPAHALO2.8M with 9 dimensions, SFONT1M with 11 dimensions and KDDBIO for 74 dimensions). This is because, R-tree exhibits very large overlap while indexing high dimensional datasets and thus leads to increase in search space [123]. Whereas, the overlap exhibited by Grid-R-tree is very less in spite of high dimensionality, and thus gives very good query performance.

In order to support the above results, we measured the average number of nodes visited and average per query time in R-tree and Grid-R-tree, while executing both kinds of queries for 3DSRN dataset. The results presented in Table 2.2 on the next page clearly

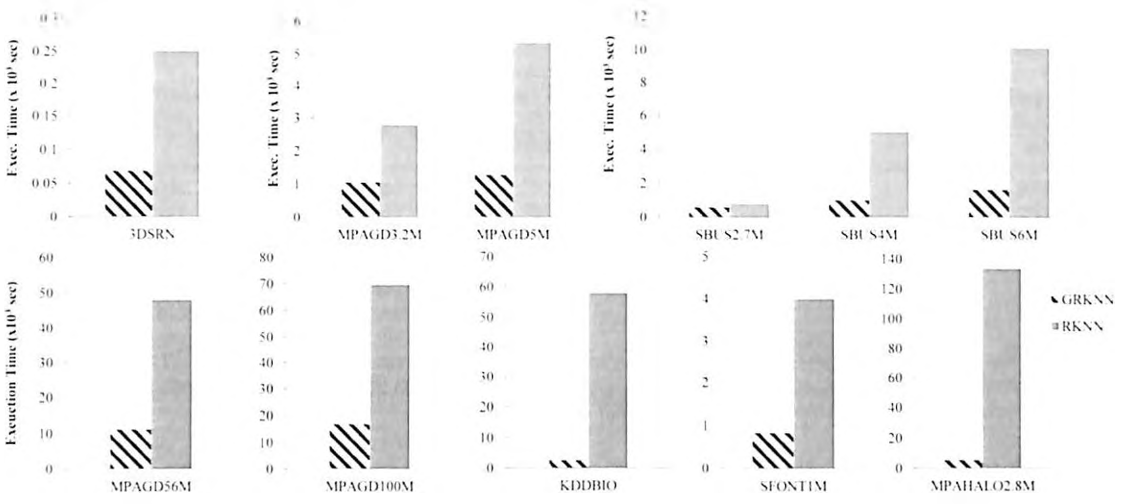


Figure 2.10: Execution time of nearest neighbor queries for Grid-R-tree and R-tree over various datasets

Table 2.2: Average number of nodes visited in neighborhood and nearest neighbor queries for 3DSRN dataset

Factors	RTreeNBH	CellWiseNBH	PointWiseNBH
Average Time Per NBH Query (in milli seconds)	2.575	1.955	2.046
Average number of internal nodes per NBH Query	9048	6966	7376
	R-KNN	GR-KNN	
Average Time Per k-NN Query (in milli seconds)	0.574	0.156	
Average number of internal nodes per k-NN Query	2164	595	

show that the avg. number of nodes visited is lesser for Grid-R-tree. This is in sync with the theoretical analysis presented in Section 2.2.6 on page 33.

We evaluate the above queries over synthetic datasets to examine the robustness of Grid-R-tree for different characteristics of data. The results presented in Table 2.3 on the next page clearly show that the performance of Grid-R-tree is very encouraging for synthetic datasets also. The reduction in query execution time over Grid-R-tree observed in case of synthetic normal and synthetic random datasets is higher because the dense regions present in these datasets tend to make R-tree perform poorly. Grid-R-tree performs better because of reduced search space, owing to its two-level design.

In the next experiment, we evaluate the performance of Grid-R-tree for varying dimensionality of the dataset. Two datasets have been chosen and sampled for different number of dimensions. SFONT1M dataset (originally 11 dimensions) has been projected for 3, 5, 7 & 9 dimensions and MPAHALO2.8M dataset (originally 9 dimensions) has been projected for 3, 5 & 7 dimensions. The value of ϵ in all these experiments has been

Table 2.3: Execution time (in seconds) for various queries on synthetic datasets

Dataset	RTree-NBH	Cell-WiseNBH	Point-WiseNBH	% improvement in Cell-WiseNBH	% improvement in Point-WiseNBH	RKNN	GRKNN	% improvement in KNN
SR1M2D	2254	240	463	89.35%	79.45%	2995	153	94.89%
SR1M3D	6476	324	637	94.99%	90.16%	7659	316	95.87%
SR1M5D	37373	847	1750	97.73%	95.31%	52814	1245	97.64%
SU1M2D	362	75	101	79.06%	71.89%	543	113	79.13%
SU1M3D	1577	389	432	75.32%	72.60%	1104	179	83.76%
SU1M5D	2619	1425	1335	45.57%	48.99%	1187	327	72.47%
SN1M2D	2451	174	376	92.89%	84.64%	4142	339	91.8%
SN1M3D	9316	386	904	95.86%	90.29%	11782	667	94.3%
SN1M5D	63770	3084	7082	95.12%	90.89%	53425	5887	88.9%

chosen such that the average number of points coming in ϵ- neighborhood of any point for all the projected datasets remains approximately the same. The results presented in Figure 2.11 on the following page clearly show that the performance of neighborhood and nearest neighbor queries for Grid-R-tree greatly improves with increase in number of dimensions, when compared to that of R-tree. The improvement is higher for datasets of high dimensionality because of reasons explained in the first experiment.

The next experiment evaluates the query performance of Grid-R-tree with datasets of varying size. Two sets of datasets - Shanghai Bus (SBUS 2.7M, 4M & 6M) and Delucia5M (MPAGD5M) (Sampled for 1M, 2M, 3M, 4M & 5M data points) are used for this analysis. In all experiments, the average number of points in the ϵ-neighborhood is maintained approximately the same for uniformity, similar to the previous experiment. Results presented in Figure 2.12 on page 42 clearly indicates that as the size of the data grows, the query performance of Grid-R-tree becomes better when compared to that of R-tree. Again, this is attributed to the reduction in the search space. The performance of k-NN queries is also maintained with increase in data size.

We also study the robustness of Grid-R-tree over datasets with variable/high densities. The experiment has been conducted on samples of MPAGD5M (delucia5M) data set (1M to 5M). These samples are generated by increasing the number of data points coming in ϵ-neighborhood proportionately with growth in data size (unlike random sampling in the previous experiment). The results presented in Figure 2.13 on page 42 show that improvement in execution time is maintained for both kinds of queries with increase in density of the dataset.

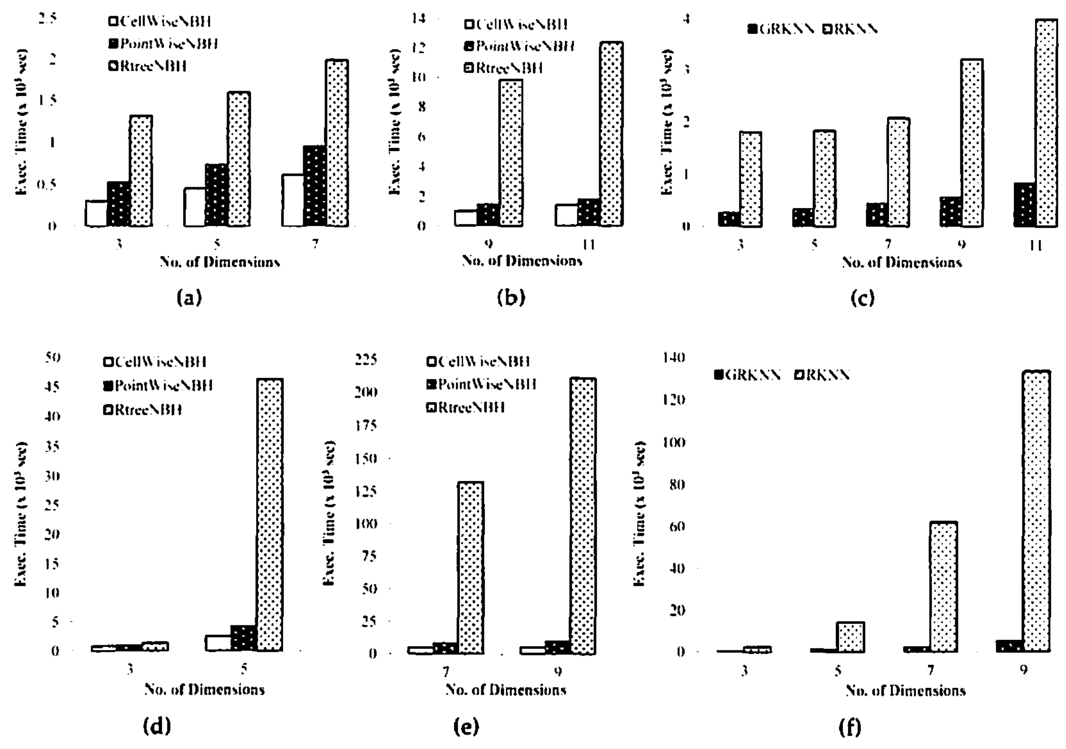


Figure 2.11: Execution time of (a) & (b) neighborhood queries and (c) nearest neighbor queries with varying dimensions on SFONT1M dataset. Execution time of (d) & (e) neighborhood queries and (f) nearest neighbor queries with varying dimensions on MPAHALO2.8M dataset

The next experiment studies the effect of varying cell size on neighborhood and nearest neighbor queries on Grid-R-tree over 3DSRN dataset and results are presented in Figure 2.14 on page 43. For the case of k-NN queries, there has been no specific pattern observed in performance with variation in cell size. However, the query performance for Grid-R-tree has always been better than that of R-tree. In case of neighborhood queries, it can be seen that a dip is observed in the curves for both kinds of neighborhood queries at a particular cell size. The same has been observed for other datasets as well. The explanation for this dip is as follows: when the cell size of Grid-R-tree is too small the total number of cells obtained from gridding is very high and the average number of points per cell is very low. In this case the query time of global-R-tree is higher and dominates the query time of cell-R-trees. When the cell size is too large, the total number of cells resultant of gridding is very less and the average number of points per cell is very high. In this case the query time of cell-R-trees dominates. So, the cell sizes around which the dip is observed in the curve is the range of optimum cell sizes where there is a balance

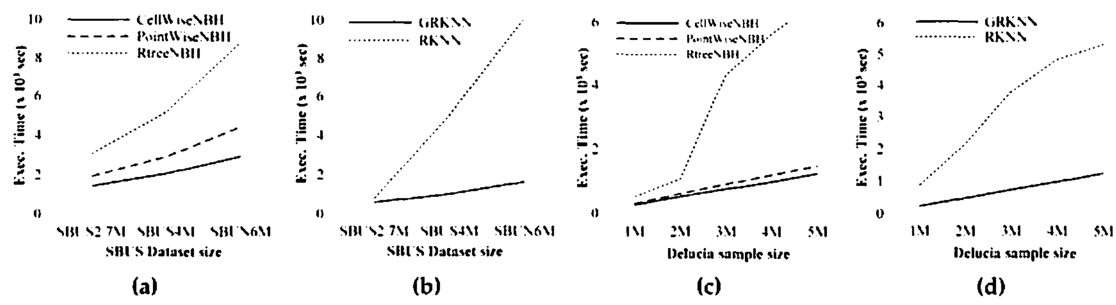


Figure 2.12: Execution time of neighborhood queries and nearest neighbor queries with increase in data sizes on (a) & (b) SBUS datasets; (c) & (d) samples of delucia (MPAGD5M)

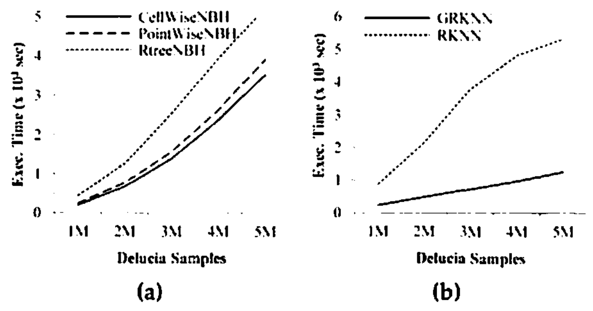
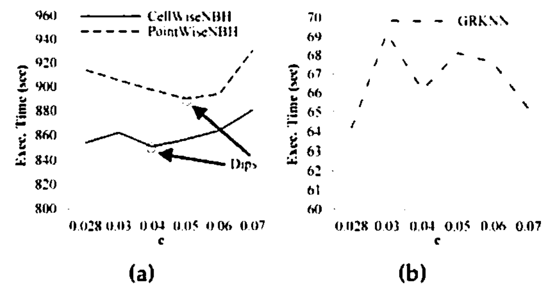


Figure 2.13: Execution times of (a) neighborhood (b) k-NN queries; with increase in density of the dataset for Delucia (MPAGD5M) data samples

established between the query time of global-R-tree and cell-R-trees. Any cell size in this range is expected to give maximum speed up. However, it is not necessary to conduct this experiment every time to measure the optimal cell size. The analysis presented in Section 4.2 on Grid-R-tree with adaptive grid optimization, shows that it eliminates the need for a separate experiment to determine the optimal cell size. This has been further substantiated with experimentation in Section 2.3.3 on the following page.

In the next experiment, we study the effect of varying ϵ and k on the performance of neighborhood and nearest neighbor queries respectively, on Grid-R-tree over MPAGD3.2M dataset keeping cell size (c) constant ($c = 2$). The results presented in Figure 2.15 on the next page clearly indicate that the performance improvement of neighborhood queries for Grid-R-tree is consistent with increase in ϵ. Also increase in ϵ led to increase in the execution time of all kinds of queries because, it will bring more points inside the ϵ-neighborhood of any point. Similarly, the results also show that the performance improvement of k-NN query over Grid-R-tree is consistent with increase in value of k. Similar results were obtained for other datasets as well.



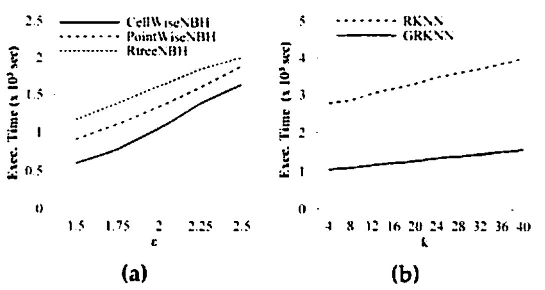


Figure 2.14: Execution time of (a) Neighborhood (b) k-NN queries with increase in cell size (c) for 3DSRN dataset

Figure 2.15: Exec. time of (a) NBH queries with variation in ϵ; (b) k-NN queries with variation in k; for MPAGD3.2M

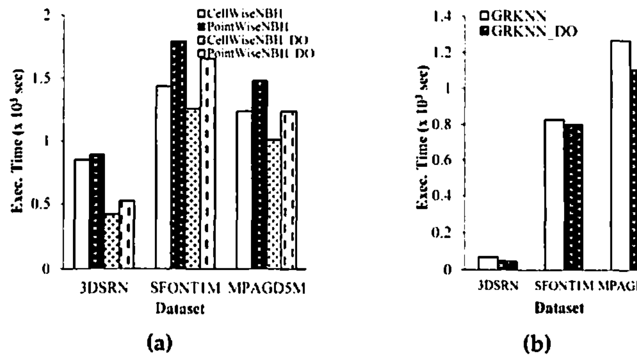
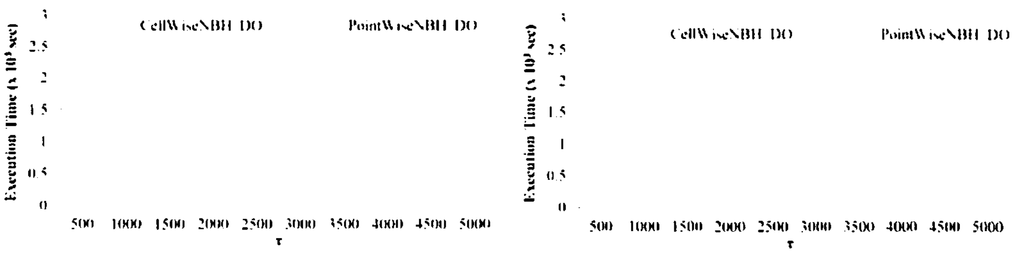


Figure 2.16: Execution Time of queries over adaptive grid optimized Grid-R-tree vs original Grid-R-tree

2.3.3 Performance of queries over Grid-R-tree with adaptive grid optimization

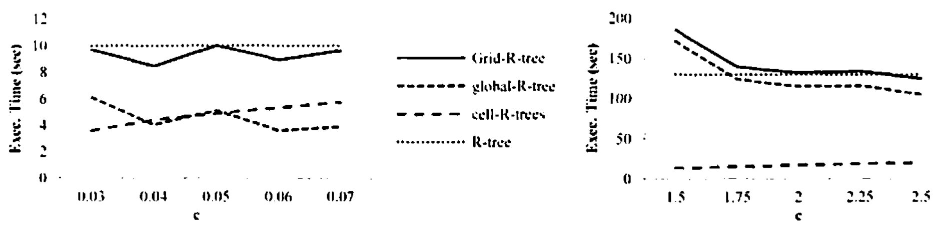
In this section, we present the performance results of neighborhood and nearest neighbor queries over datasets with skewed distribution using adaptive grid optimized Grid-R-tree. The experiments were conducted over following datasets- 3DSRN, SFONT1M and MPAGD5M, which are skewed in their distribution. The value of τ had been chosen to be 3000. The results presented in Figure 2.16 clearly show that queries over adaptive grid optimized Grid-R-tree perform better than their corresponding queries over non-optimized Grid-R-tree. This is because adaptive grid optimization achieves better load distribution and aids in executing the queries more efficiently.

We also conducted an experiment to measure the neighborhood query performance of Grid-R-tree with adaptive grid optimization for varying τ. The results presented in Figure 2.17 on the next page clearly show that the value of τ between 2000 and 3000 gives optimal query performance for neighborhood queries over both the datasets. This



(a) (b)

Figure 2.17: Execution time of neighborhood query over Grid-R-tree with adaptive grid optimization with varying τ for (a) SBUS2.7M & (b) MPAGD5M datasets



(a) (b)

Figure 2.18: Construction time of Grid-R-tree & R-tree for (a) 3DSRN and (b) MPAGD3.2M datasets with increase in cell size (c)

is because, choosing a very small τ results in a large number of cells leading to a large search time in global-R-tree of Grid-R-tree and global-R-trees of cell-Grid-R-trees. On the other hand, choosing a large τ leads to very less number of cells, resulting in large search time in cell-R-trees. The same behavior has been observed for other datasets as well. The value of τ in the range - [2000-3000], gives optimal query performance as better load distribution between global-R-tree and cell-R-trees is achieved in this range.

2.3.4 Construction and Query Execution Time: Grid-R-tree vs R-tree

A comparative analysis on construction and neighborhood query execution times for Grid-R-tree and R-tree has been conducted for datasets: 3DSRN and MPAGD3.2M. The results presented in Figure 2.18 clearly show that in Grid-R-tree, an increase in cell size leads to reduction in construction time of global-R-tree, but increases the construction time of cell-R-trees. The results also indicate that the total construction time of Grid-R-tree (global-R-tree + cell-R-trees) is comparable to that of conventional R-tree for most of the cell sizes. In certain cases the construction time of Grid-R-tree is even lesser ((a) of

Figure 2.18 on page 44). This is attributed to the fact that search time during insertion is also reduced in Grid-R-tree. Also, the increase in time of Grid-R-tree construction in case of (b) part of Figure 2.18 on page 44 (c=1.5), is much lesser than the reduction in time achieved for neighborhood and k-NN queries.

Table 2.4: Construction and querying times for R-tree and Grid-R-tree over MPAGD3.2M dataset with c=2.25

Operation on R-tree	**Time (sec.)**	**Operation on Grid-R-tree**	**Time (sec.)**
Construction	130	Construction	134
R-tree-NBH	1613	PointWiseNBH	1289
		CellWiseNBH	975

Table 2.4 presents the execution time of neighborhood queries for MPAGD3.2M dataset along with the construction times for R-tree and Grid-R-tree (global-R-tree + cell-R-trees). The cell size c was chosen to be 2.25 units. The results show that the reduction in execution time for CellWiseNBH and PointWiseNBH queries were 638.484 sec. (39.5%) and 323.894 sec. (20.1%) respectively when compared to the neighborhood queries over R-tree. Whereas, the increase in Grid-R-tree construction time was only 4.03 sec. (3.04%). This establishes the efficiency of Grid-R-tree over R-tree.

2.3.5 Tradeoff in choice of R-tree vs k-d-tree for analysis

In this section we present a tradeoff analysis conducted between R-tree and k-d-tree. The neighborhood query execution time for R-tree and k-d-tree over 3DSRN and MPAGD3.2M datasets are presented in Table 2.5 on the following page. The results clearly show that the time required to execute neighborhood queries for all the points of the dataset is very less for R-tree when compared to that of k-d-tree. This is primarily attributed to the fact that the height of the k-d-tree becomes large as the size of the data and the number of dimensions grow. This was the reason why k-d-tree was dropped from further experimentation. Part I on page 16 also presents the problems of quad-tree and shows that k-d-tree is better than quad-tree.

Table 2.5: Neighborhood query exec. time of R-tree and k-d-tree over 3DSRN & MPAGD3.2M

Dataset	Exec. Time for R-tree (sec)	Exec. Time for k-d-tree (sec)
3DSRN	1119	2708
MPAGD3.2M	1613	17172

2.4 Discussion

In this section, how Grid-R-tree has addressed the drawbacks associated with conventional data structures described in preamble of Part I on page 16.

- Grid-R-tree, by its design into two levels, reduces the downward propagation of overlap amongst MBRs of its nodes, as the cells are completely free from overlap. This solves the problem of overlap in R-tree and its variants and thus giving very good query performance as illustrated by experiments in Section 2.3 on page 36. Also, reduction in overlap helped in maintaining the query performance with increase in size and dimensionality of the dataset, unlike R-tree which had shown deterioration. Also, note that all these advantages are achieved without any increase in construction time of the indexing structure.
- Query performance for Grid-R-tree has also been better than kd-tree and quad-tree as illustrated by an experiment in Section 2.3.5 on page 45. Grid-R-tree manifests small height and larger fanout due to which deterioiration of query performance as in the case of kd-tree doesn't occur here. Also, kd-trees are found to be better than quad trees [124, 126] as it completely avoids nil pointers and is height balanced. Thus we can claim without hesitation that that Grid-R-tree outperforms both quad-tree and kd-tree.
- Grid-R-tree also addresses the drawbacks of grid file structure. As explained in preamble of Part I on page 16, usage of array for the grid file directory guarantees $\mathcal{O}(1)$ asymptotic complexity for query retrieval, but leads to inefficient space utilization while indexing datasets of high dimensionality. And usage of linked lists instead saves space, but leads to $\mathcal{O}(n)$ look-up time. Similar problem is observed when the data distribution is skewed. So, in order to maintain both space and query efficiency, Grid-R-tree uses R-tree (global-R-tree) to store non-empty cells resulting from virtual gridding instead of the grid directory. R-tree guarantees $\mathcal{O}(\log T)$ aver-

age case performance, for T number of cells indexed in it, irrespective of sparsity or dimensionality of the dataset. Grid-R-tree also uses R-trees (cell-R-trees) for indexing points in a cell thereby avoiding chaining exhibited by buckets in the grid file. Additionally, we have also demonstrated in Section 2.3 on page 36 that Grid-R-tree can guarantee efficient query execution for data sets of any kind of distributions including uniform, skewed, normal distribution, etc. This is in contrast with grid files which give optimal query performance only when when the number of dimensions are not large and data distribution is uniform [127].

- Grid-R-tree also addresses the drawbacks of the proposed GR-tree structure [128]. As explained in preamble of Part I on page 16, in GR-tree, the leaves of the first level R-tree are overlapping, which results in searching multiple coordinate trees to locate a data point. This deteriorates the efficiency of a neighborhood query, especially for datasets of high dimensionality and large sizes. Whereas, the design of Grid-R-tree has the leaves of global-R-tree fully disjoint, as a result of which only one cell-R-tree will be required to be searched to locate a data point, leading to reduction in search space. Other disadvantage of GR-tree is that the points indexed in each cell are stored in a circular linked list which yields a linear search time. Whereas, in Grid-R-tree, the points in a cell are indexed in cell-R-tree leading to logarithmic search time. Moreover, GR-tree does not explicitly capture any specific pattern of any of the data mining algorithms as we do for density based clustering algorithms (see Chapter 3 on page 50).

2.5 Main Contributions

- In this chapter, we propose a data indexing structure known as *Grid-R-tree* which is a simple, yet effective adaptation of *R-tree* using Grid. It is a two-level R-tree in which the first level R-tree (known as *global-R-tree)* is for indexing "non-empty" cells resultant of virtual gridding of the search space, and the second level comprises of multiple R-trees (*cell-R-trees*) one each for every cell to index points lying in it.
- Grid-R-tree supports efficient execution of *region* and *nearest neighbor* queries. Grid-R-tree handles the above queries more efficiently than the other conventional index-

ing structures such as R-tree, k-d-tree, etc, as illustrated by experimental results in Section 2.3.2 on page 38.

- The structure of Grid-R-tree's helps in overcoming the drawbacks of the above conventional data indexing structures, arising due to large size and high dimensionality in datasets (such as increase in overlap and height in R-tree & k-d-tree, respectively). It also guarantees efficient query execution over data sets of any kind of distribution including uniform, normal, skewed, etc., as substantiated by results in Section 2.3.2 on page 38.
- Grid-R-tree supports a novel query called *cell-wise ϵ-neighborhood query* (CellWiseNBH), which performs locality aware execution of neighborhood queries observed in density based clustering algorithms (DBSCAN and OPTICS). CellWiseNBH computes ϵ-neighborhoods for all the points of a cell in a single traversal of global-R-tree, thus saving repeated traversals of the same search space (path in the tree). This query enables us to re-design the above algorithms making them efficient than their native versions (see Section 3.1 on page 50).
- An adaptive grid optimization has been applied to Grid-R-tree to handle *dense cells* (cells having number of points greater than a threshold τ) which are resultant of indexing datasets of variable density (skewed datasets). This optimization achieves better load distribution in the cells and thus improves the performance of above queries, as evident from results presented in Section 2.3.3 on page 43.
- We also present a supporting theoretical analysis, which theoretically proves that Grid-R-tree exhibits lesser overlap amongst the MBRs of its nodes and results in reduction of search space for the above queries, when compared to the conventional R-tree (see Section 2.2.6 on page 33). In particular, it can be seen that absence of overlap across multiple cells (or cell-R-trees) leads to reduction in search space and better query performance.

2.6 Conclusions & Future Work

2.6.1 Conclusions

This chapter proposes Grid-R-tree, which is a simple, yet effective adaptation of R-tree using the concept of Grid. Grid-R-tree supports efficient execution of conventional queries such as region queries and nearest neighbor queries. The design of Grid-R-tree into two levels breaks down the downward propagation of overlap and thus reduces the search space, leading to improvement in query performance. The construction time of Grid-R-tree is comparable to that of conventional R-tree whereas its performance over various queries is much better than that of R-tree. The experimental results presented clearly suggest that Grid-R-tree outperforms R-tree for the queries listed above. They also demonstrate that the query performance doesn't deteriorate with increase in dimensionality, size and density of the dataset. Grid-R-tree also supports a new type of query called CellWiseNBH query, which helps in executing the neighborhood queries for all points in a given cell efficiently. CellWiseNBH query has been used to re-design DBSCAN clustering algorithm whose details are presented in Chapter 3 on the following page and the results clearly indicate that the re-designed version performs better than the native DBSCAN. Grid-R-tree with adaptive grid optimization has also been proposed to deal with dense cells formed due to skewness in data, which further improves the query performance.

2.6.2 Future Directions

The gridding concept used in Grid-R-tree can also be applied to other variants of R-trees such as *R*-tree, Greene-R-tree, Hilbert-R-tree,* etc. [123] and other multidimensional indexing structures. This is expected to give even better performance as it will result in further minimization of overlap. Grid-R-tree can be directly used in any other domain like Geographical Information Systems and Multimedia, that require region and nearest neighbor queries. A distributed/ concurrent version of Grid-R-tree can also be designed to adapt itself to high performance computing paradigms.

Chapter 3

Applicability of Grid-R-tree to Data Mining

3.1 Data Mining Algorithms using Grid-R-tree

Any data mining algorithm that requires usage of neighborhood and nearest neighbor queries can be made to execute faster using Grid-R-tree. We discuss two of them here: *k-NN classifier* and *DBSCAN clustering*. We also give a theoretical bound on the speed up that can be achieved for any algorithm that uses Grid-R-tree.

3.1.1 k-NN Classifier using Grid-R-tree

k-NN classifier [121] classifies test data using k-nearest neighbor query over the training dataset. Initially, Grid-R-tree (or R-tree) can be built over the training data points, and k-NN query can be executed over it for test data points to find their respective class labels. The performance of k-NN classifier for Grid-R-tree vs R-tree is illustrated in Figure 3.1 on the next page for Shuttle [138] and Skin [139] datasets. Each of these datasets have training and test data, with the ratio of 3:1 approximately. The results presented in the figure clearly indicate that k-NN classifier over Grid-R-tree is more efficient than that over R-tree.

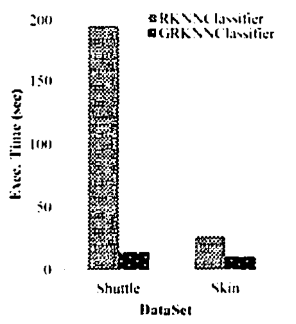


Figure 3.1: Execution time of k-NN Classifier over R-tree and Grid-R-tree for Shuttle and Skin datasets

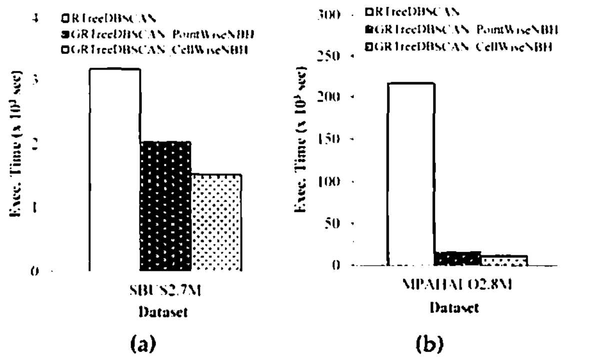


Figure 3.2: Execution time for DBSCAN over R-tree and Grid-R-tree for (a) SBUS2.7M (b) MPA-HALO2.8M datasets

3.1.2 DBSCAN clustering using Grid-R-tree

It is a well known fact that DBSCAN algorithm devotes a major portion of execution time to neighborhood queries. These neighborhood queries are made to execute faster using Grid-R-tree. We present two ways of executing DBSCAN algorithm. First we use PointWiseNBH query to execute DBSCAN normally and then we use CellWiseNBH to execute DBSCAN in an optimized way.

In the first case the neighborhood queries that are required for execution of DBSCAN, are executed simply by using PointWiseNBH query over Grid-R-tree. The pseudo code of the DBSCAN algorithm using PointWiseNBH query is presented in Algorithm 3.1 on the following page & Algorithm 3.3 on the next page. The change is that R-tree neighborhood query is replaced by PointWiseNBH query over Grid-R-tree. Rest of the program is same as that of classical DBSCAN using R-tree. To use CellWiseNBH query for DBSCAN, we make a minor modification to the algorithm without changing its actual flow. We take advantage of CellWiseNBH query to pre-compute the neighborhoods of all the points of a given cell and store them temporarily as neighbors of those points. Algorithm 3.2 on the following page & Algorithm 3.4 on the next page illustrate the changes to be made to the classical DBSCAN code. Instead of calling neighborhood query of a given point p directly, we first find out to which cell it belongs to and then execute CellWiseNBH query over that cell. This query computes neighborhoods of all the points belonging to that cell and stores them temporarily. Next time when we need the neighborhood for another point p', we first check if its neighborhood has already been computed by any of the previous calls to CellWiseNBH query. If it is so, we simply use it, else we identify the cell to which it belongs to and execute CellWiseNBH query over that cell and then use neighborhood of p' for further clustering. Rest of the algorithm remains the same as that of native DBSCAN. This algorithm is more efficient than the previous one as the neighborhood query time is greatly optimized.

Algorithm 3.1: DBSCAN Clustering using PointWiseNBH

```
procedure DBSCAN-PointWiseNBH()
    Input  : Data points list DL, Grid-R-tree G,
             Epsilon $\epsilon$, Min points MinPts
    Output: List of clusters clusList
    foreach point p in DL do
        if p is visited then continue;
        Mark p as visited;
        Data list nbhlist = Point-Wise-NBH(p, $\epsilon$,
          G.root);
        if NbhList.count < MinPts then
            Mark p as NOISE;
        else
            Initialize a new cluster Clus1;
            Expand-Cluster (p, G, NbhList,
              Clus1, $\epsilon$, MinPts);
            Insert Clus1 into clusList;
        end
    end
```

Algorithm 3.2: DBSCAN Clustering using CellWiseNBH

```
procedure DBSCAN-CellWiseNBH()
    Input  : Data points list DL, Grid-R-tree G,
             Epsilon $\epsilon$, Min points MinPts
    Output: List of clusters clusList
    ..........
    ..........
    ..........
    region r; cell c1;
    r=Determine-Cell(p,MINGRIDSIZE,c);
    C1=Check-If-Cell-Exists(G.global-R-tree,r);
    Cell-Wise-Nbh (C1, $\epsilon$, G.root);
    if p.neighborslist.count < MinPts then
        Mark p as NOISE;
    else
    ..........
    ..........
    ..........
    ..........
```

Algorithm 3.3: Expand Cluster using PointWiseNBH

```
procedure Expand-Cluster-PointWiseNBH()
    Input  : Data point p, Grid-R-tree G, Data
             points list NbhList, cluster Clus1,
             Epsilon $\epsilon$, Min points MinPts
    Output: Complete Formation of Cluster Clus1
    Initialize a temporary points list tempNbhList;
    Add p to Clus1;
    foreach point p' in NbhList do
        if p' not visited then
            Mark p as visited;
            tempNbhList = Point-Wise-NBH (p,
              $\epsilon$, G.root);
            if tempNbhList.count $\geq$ MinPts then
                NbhList = NbhList $\cup$
                  tempNbhList;
            end
        end
        if p' is not a member of any cluster then
            Add p' to Clus1;
        end
    end
```

Algorithm 3.4: Expand Cluster using CellWiseNBH

```
procedure Expand-Cluster-CellWiseNBH()
    Input  : Data point p, Grid-R-tree G, Data
             points list NbhList, Cluster Clus1,
             Epsilon $\epsilon$, Min points MinPts
    Output: Complete Formation of Cluster Clus1
    ..........
    ..........
    ..........
    ..........
    ..........
    if neighborhood of p' is not computed then
        region r; cell C1;
        r=Determine-Cell(p,MINGRIDSIZE,c);
        C1=Check-If-Cell-Exists
        (G.global-R-tree,r);
        Cell-Wise-Nbh (C1, $\epsilon$, G.root);
    tempNbhList = p'.neighborsList;
    ..........
    ..........
    ..........
    ..........
```

All the three variants of DBSCAN (using RtreeNBH, PointWiseNBH and CellWiseNBH) have been executed over SBUS2.7M and MPAHALO2.8M data sets and the results are presented in Figure 3.2 on page 51. The ϵ values chosen for experimentation are presented in Table 2.1 on page 37. The value of *Minpts* has been set to 5. The results indicate that DBSCAN execution time over Grid-R-tree is much lesser than that of R-tree over both the datasets. More specifically the execution with CellWiseNBH query is most efficient as this captures the query access pattern of DBSCAN. Note that the memory requirement of DB-SCAN using CellWiseNBH query is little higher since we are storing neighborhoods. But since the number of points lying within a cell is very less, the total memory required to store their neighborhoods is also very less. Also, as soon as the point is processed by DB-SCAN, the memory allocated to its neighbors list is cleared. Thus, the space complexity doesn't increase.

Similar experimentation can be performed for OPTICS [35] clustering as well. Grid-R-tree can also be used for other variants of DBSCAN such as IDBSCAN [140] and KIDB-SCAN [141] for efficient neighborhood queries in the same fashion.

3.1.3 Upper bound on speed up achieved by using Grid-R-tree

In this section, we present a theoretical upper bound on the speed up that can be achieved for any algorithm that uses queries supported by Grid-R-tree. We give the analysis with the help of DBSCAN clustering. As mentioned before, the execution time of DBSCAN has a major portion of time spent on performing neighborhood queries. Let p be the fraction of time spent on neighborhood queries. The speed up that can be achieved for executing DBSCAN is governed by speed up achieved in this portion of the program. Let z be that speed up achieved by using Grid-R-tree for neighborhood computations, instead of R-tree. By Amdahl's Law [142] the maximum speed up achieved for executing DBSCAN by replacing R-tree with Grid-R-tree will be:

$$\frac{1}{\frac{p}{z} + 1 - p}$$

This is the theoretical upper bound for speed up that can be achieved. This analysis can be generalized for any algorithm that uses Grid-R-tree instead of R-tree for its neighborhood computations. This result is verified for DBSCAN for SBUS2.7M and MPAHALO2.8M datasets. Table 3.1 on the following page separately presents the time of execution of entire DBSCAN and the portion comprising of neighborhood queries for the above datasets, along with the actual speed and the expected maximum speed up calculated using the above equation. It is very clear from the table that the actual speed up is very close to the

Table 3.1: Theoretical Upper Bound vs Obtained Speed Up for DBSCAN Clustering

Dataset	Query Type	Exec. Time for NBH queries only (sec.)	Exec. Time for DBSCAN (sec.)	Improvement factor in NBH queries (z)	Actual Speedup	Expected Max Speedup
	R-tree-NBH	3071.241	3174.043	-	-	-
SBUS2.7M	PointWiseNBH	1924.425	2028.425	1.595	1.564	1.565
	CellWiseNBH	1420.79	1525.79	2.161	2.080	2.083
	R-tree-NBH	210331.325	210465.903	-	-	-
MPAHALO2.8M	PointWiseNBH	9624.115	9769.988	21.854	21.542	21.567
	CellWiseNBH	4896.365	5042.693	42.956	41.736	41.834

estimate for both the datasets.

3.1.4 Other Uses of Grid-R-tree

Grid-R-tree have been used to re-engineer clustering algorithms like DBSCAN and SLINK from their basics [32, 33, 36]. These re-engineered versions work at grid level with operations over cells as well as over points, rather than only points as in the native algorithms, and give exact clustering results as that of their native versions. Grid-R-tree can also be employed in Grid based clustering algorithms like BANG [129], AMR [130], etc. to index non-empty cells in place of other indexing structures. Grid-R-tree can also be used in any other application domains that require extensive use of neighborhood and nearest neighbor queries.

3.2 Discussion

- Gridding has been used in several clustering algorithms: GRIDCLUS [129], BANG [143], STING [144], AMR [130], etc. These algorithms work at a coarser granularity performing operations on partitioned cells rather than on points. Our contribution on Grid-R-tree is in demonstrating that gridding can be used effectively to perform queries on points as required by many data mining algorithms such as DBSCAN, OPTICS, SLINK, k-NN classifier, etc, rather than cell level operations.
- We used Grid-R-tree to efficiently implement data mining algorithms - DBSCAN and k-NN classifier. A few variants of the DBSCAN algorithm - IDBSCAN [140], KIDBSCAN [141], DBSCALE [145] - have been proposed in literature which are efficient than the original DBSCAN. However, these variants use sampling techniques

to reduce the number of neighborhood queries resulting in "approximate" clustering. On the contrary, we have focused on reducing the computation time for each neighborhood query (rather than number of queries) without compromising on the correctness of DBSCAN clustering as shown by above experimental results.

3.3 Main Contributions

- In this chapter, we demonstrated the utility of Grid-R-tree in spatial data mining algorithms like k-NN classifier and DBSCAN clustering algorithm and have shown that usage of Grid-R-tree improves their performance (See Section 3.1 on page 50).
- We have also derived a theoretical upper bound on the speed-up that can be achieved by using Grid-R-tree in place of any other indexing structure using Amdahl's law.

3.4 Conclusions

In this chapter, we have demonstrated how usage of Grid-R-tree can optimize the performance of data mining algorithms such as k-NN classifier and DBSCAN clustering. The experimental results clearly show that CellWiseNBH query clearly optimizes the search space during its execution making the neighborhood queries efficient, and thus making DBSCAN efficient.

Part II

Anytime Mining of Data Streams

A data stream is characterized by continuously arriving data objects at a fast and variable rate, ordered by time. Mining data streams is typically constrained by limited available time to process and limited memory to store the incoming data objects. The time available to process each arriving object depends upon the stream speed and usually allows only for a single pass. And, within these constraints, evolving patterns have to be captured. Mining of data streams has enormous applications of various domains that include - retail chain analysis, network traffic analysis, web log analysis, mining data feeds from sensor networks, surveillance systems, disease surveillance systems, etc.

Researchers have proposed various algorithms for mining data streams. These stream mining algorithms can be broadly categorized into four tasks - Classification, Clustering, Frequent Itemset Mining and Anomaly Detection. Various stream classification algorithms have been proposed in - [146, 147, 148, 14, 99]. A survey of these algorithms can be found in [149]. Similarly, various stream clustering algorithms include - CluStream [102], HP-Stream [103], DenStream [95], Optics Stream [150], D-Stream [151], MR-Stream [152], etc. A survey of these can be found in [153]. Various algorithms proposed for frequent itemset mining in data streams include - Sticky Sampling & Lossy Counting [154], FP-Stream [98], CPS-Tree [53], DSM-FI [96], SWP-Tree [97], VSW [155], etc. A survey of these can be found in [156, 157]. And, various algorithms for anomaly detection in data streams are proposed in [109, 158, 159, 160, 108]. A survey of these can be found in [106].

With increasing complexity of the modern data generating systems, data is being generated in the from of multiple sources resulting in multi-port streams. Researchers have proposed various algorithms for mining multi-port data streams as well. Few such algorithms are - Classification - [161, 162, 163]; Clustering - [164, 165, 166, 167, 168]; FI Mining - [169, 170, 115, 171, 172]; and Anomaly Detection - [110, 173, 174].

The existing algorithms (both sequential and multi-port algorithms), however, miss out on two important characteristics of real-time data streams. Firstly, very often *streams do not have constant speed* or *inter-arrival rate* of transactions that can greatly vary depending upon the application domain. For example, in retail chain analysis, the rate of arrival of transactions is higher during rush hours and lower during other times. Similar scenario can occur in domains like sensor networks, web server logs, etc. The algorithms proposed in literature are typically budget algorithms, i.e. they are designed for a fixed maximum stream speed (known as *budget*). When the stream speed is higher than the budget, they would have to either process sampled data or buffer unlimited data and eventually fail [104]. And when the stream speed is lower, they sit idle after processing the current data object, until the next one arrives. An ideal algorithm, however, should be able to process any stream speed. Higher speeds should be handled using deferred insertions and spare time available while processing lower speed streams should be utilized for refining the information received.

The second characteristic missed out is that they lack the ability to produce immediate mining results with compromised accuracy, if required. There are applications such as stock market analysis, where the mining results are sometimes instantaneously required. For example, a short term stock investor for whom time is money, would require an instantaneous result. On the other hand, a long term investor would wait for some additional time until a more accurate result is computed. So, an ideal algorithm should give a mining result almost immediately as soon as a request comes from the user, and improve its quality/accuracy with increase in time allowance. Figure 1.6 on page 9 shows the behaviour of an anytime algorithm, where accuracy of the result improves with increase in time allowance for processing. The existing stream mining algorithms lack such capability. They typically execute an offline phase to mine for the final result, which is computationally expensive and hence cannot provide immediate mining results.

The above two properties - 1. *handling varying inter-arrival rate of transactions*, and 2. *giving the best possible result according to the available time allowance*; are the characteristics of an anytime mining algorithm for data streams. Only a few such anytime algorithms have been proposed in literature. They include - clustering [104, 117, 118], classification [100, 101, 175, 116], and amomaly detection - [119]. These algorithms address either one or both the properties of an anytime mining algorithm.

A few anytime mining algorithms have also been proposed for static datasets (non-stream environment). They typically have only the second characteristic of the anytime algorithm. They include - Classification [176, 177, 178, 179], Clustering [180, 181] and FI Mining [182]. Since, the focus of anytime algorithms in this thesis is pertaining to data streams, we don't discuss them further.

In this part, we develop three anytime mining algorithms for data streams - one each for *frequent itemset mining* (Chapter 4 on the next page), *set-wise classification* (Chapter 5 on page 103) and *clustering* (Chapter 6 on page 126). They are presented as follows:

Chapter 4

Anytime Frequent Itemset Mining of Data Streams

4.1 Frequent Itemset Mining

Frequent Itemset (or Frequent Pattern) Mining is the task to discover frequently occurring items in a transactional database. For example, consider the sales database of a bookstore, where each transaction consists of books purchased by a customer. The task of finding sets of books most frequently purchased together by customers is a frequent itemset mining problem. The bookstore can use the knowledge acquired from frequent itemset mining for advertising, shelf placement, etc. Similarly, there are many other application areas for frequent itemset mining that include - catalog design, store layout, customer segmentation, telecommunication alarm diagnosis, and so on.

Frequent itemset (FI) mining has been well studied for static datasets in two broad categories: *Apriori like* methods [47, 50, 51, 183] and *FP-tree like* methods [184, 53]. The

- Poonam Goyal, Jagat Sesh Challa, Shivin Srivastava, Navneet Goyal. *AnyFI: An Anytime Frequent Itemset Mining Algorithm for Data Streams.* In Proceedings of 2017 IEEE International Conference on Big Data (IEEE Big Data 2017), pp. 942-947, 11-14 December 2017, Boston, MA, USA
- Poonam Goyal, Jagat Sesh Challa, Shivin Srivastava, Navneet Goyal. *Anytime Frequent Itemset Mining of Transactional Data Streams.* Submitted for review in Big Data Research, Elsevier.

former find FIs of length k from a set of pre-generated candidate itemsets of length $k-1$. They scan the dataset multiple times and generate many candidate itemsets of which many could be infrequent. To address this issue FP-tree was proposed which uses FP-growth algorithm [184]. This reduces the number of dataset scans to two and doesn't enumerate candidate itemsets like in Apriori.

4.2 Frequent Itemset Mining in Data Streams

Mining for *frequent itemsets* (FIs) in transactional data streams is commonly used in various applications such as *retail chain analysis, stock market analysis, web log analysis, network traffic analysis, mining data feeds from sensor networks,* etc. Researchers have proposed a few algorithms for mining FIs from data streams [154, 98, 53, 96, 97, 155]. These algorithms have two phases - *online* & *offline*. In the online phase, they insert the incoming transactions into a summary structure, either batch by batch [98, 53, 96] or transaction by transaction [97]. And, whenever a request for mining result comes from the user, they execute the offline phase to extract FIs from their summary structures.

For modeling data streams, researchers have typically used *landmark window* [154, 98, 96] and *sliding window* [53, 183, 97, 155, 185] models. Landmark window summarizes the entire stream from its beginning, whereas sliding window only keeps a fixed number of last arrived transactions. Sticky-Sampling and Lossy Counting [154] are the first algorithms proposed for FI mining of data streams. They are based on Apriori and follow the landmark window model. They produce less accurate results with a theoretical error bound. FP-Stream [98] is the next approach that uses tilted-time window model (variation of landmark window). It takes in transactions batch-wise and for each batch it builds an FP-tree, mines for FIs, and inserts them into a pattern tree that has tilted-time windows stored at its nodes. Pattern trees are mined similar to FP-Trees to output the mining result. FIDS [186] is another approach proposed that processes transactions batch-wise. They propose a new representation for the items and a data structure known as $Lattice_{reg}$ to maintain FIs coupled with a fast pruning strategy. They also use the tilted-time window model. DSM-FI [96] is another approach which also uses landmark window. It keeps a forest of prefix trees which are similar to FP-trees. Every arriving transaction of size k

is converted into $k - 1$ smaller transactions which are suffixes of the original transaction, and each suffix is inserted into its respective tree in the forest. This algorithm is a budget algorithm and has to insert all the incoming transactions completely into the forest irrespective of the stream speed. Mining for FIs in DSM-FI is similar to the Apriori approach wherein the infrequent itemsets of size k are broken down into itemsets of size $k - 1$ and are checked if they are frequent.

CPS-Tree [53] is a FP-tree based algorithm that uses sliding window model. It takes in the transactions batch-wise, and undergoes re-structuring and pruning after processing every batch, which is time consuming. It uses FP-growth for mining FIs on request. MFI-TransSW [183] is another sliding window based approach that represents the items in the form of a bit vector and uses Apriori for mining. SWP-Tree [97] is also a sliding window based algorithm that uses a prefix tree similar to FP-tree. It additionally uses decay on support count of items stored in it to give higher weightage to recently arrived ones. Mining for FIs from SWP-tree is similar to that of FP-Tree. Another algorithm which adjusts the size of sliding window on demand, is the VSW [155]. It uses ECLAT [51] which is a variant of Apriori and is slow in its insertion and mining. VSW also uses expensive computations to compute the new window size after processing every batch of transactions and thus is not capable of handling high speed streams. Another recent algorithm is WIS [185], which mines for FIs within a given time horizon. It uses a test window which is a minimal window to discard the infrequent itemsets. However, in this approach, the support counting is limited to the test window. Another approach MSWTP [105], which mines for top k frequent itemset from the stream over a sliding window without using any threshold on support. They propose SWTP-tree to store the itemsets in the sliding window and provide a lower bound on the support of k^{th} frequent itemset using Chernoff bound theory.

With increase in utilities of systems that generate data from multiple sources, mining for FIs from multiple streams is increasingly becoming popular. Researchers have proposed a few algorithms for FI mining over multi-port data streams. H-Stream [169] is the first proposed approach, which extends the FP-Stream [98] and uses tilted-time window framework for storing FIs from multiple streams. PAMS [170] is another algorithm that mines for progressive sequential patterns from multiple data streams. It uses a structure known as PSM-tree for storing FIs. DIMine & CooMine [115] are another set of algorithms proposed for mining frequent co-occurrence patterns from multiple streams. A co-occurrence pattern is an itemset that occurs in at least a pre-defined number of streams out of many incoming streams. They use a structure known as *Seg-tree* to index itemsets. Similarly, DFCP & MDFCP [171] are another set of algorithms that mine for co-occurence patterns from multiple streams. More recently proposed algorithm is the CP-Graph [172] that mines for top k co-occurrence patterns across multiple streams in real-time. They use sliding window model and a structure known as CP-Graph, which is a hybrid index of a graph and an inverted file structure that is used for efficient support counting and pruning of infrequent itemsets.

4.2.1 Research Gap and Motivation

The existing algorithms (both sequential and multi-port algorithms), however, miss out on both the characteristics of anytime mining algorithms for data streams explained in Part II on page 57. All of them are budget algorithms, i.e., they are designed for fixed maximum speeds beyond which they would fail to process the stream. Also, these algorithms either use Apriori like methods or FP-tree like methods for mining FIs. Apriori like methods enumerate a large number of candidate itemsets and thus are very slow in mining for FIs, especially for lower support thresholds. Similarly, FP-tree like methods enumerate conditional pattern trees to compute the FIs and thus take time to output the result. It can be seen that both kind of methods are not capable of delivering an immediate mining result (even with compromised accuracy) if required. Hence, all of the above algorithms do not fit the bill for requirements of an anytime FI mining algorithm for data streams.

There is one anytime algorithm for FI mining in multi-user applications over a large static database [182]. It uses sampling and addresses the second aspect of an anytime algorithm, i.e., it gives an immediate approximate mining result and improves it with increase in time allowance. However, it is static in nature and doesn't fit for streams. Thus, there doesn't exist any anytime algorithm for FI mining of data streams.

In this chapter, we present AnyFI which is the first Anytime Frequent Itemset mining algorithm for data streams, It is characterized by both the aspects of an anytime algo-

Table 4.1: Notations and Definitions

Notation	Definitions
$I = \{i_1, i_2, ..., i_n\}$	I is a **Dictionary** of unique items. Each literal i_x denotes a unique item. Literals are considered as integers.
$S = \{i_x, ..., i_y\}$	S is an **Itemset**, if $S \subseteq I$; $x \leq y$ and $x, y \in [1...n]$.
$tr = (tid, arrTime, S')$	tr is a **Transaction** consisting of the tuple - *trans ID, arrival time* of the transaction and an *itemset* S'. If for an itemset $S, S \subseteq S'$, we say that tr contains S.
DS	A **Data Stream** is a continuous unbounded sequence of transactions, $DS = tr_1, ..., tr_j, ...$, where tr_j is the j^{th} arrived transaction.
sDS	A finite contiguous subsequence of transactions from DS.
$freq_{sDS}(S)$	It is the **frequency count/support count** of an itemset S with respect to sDS (no. of transactions in sDS where S has occurred).
Frequent Itemset	Given a minimum support threshold $\sigma(0 < \sigma < 1)$, S is said to ***frequent*** (or ***σ-frequent***) in sDS, if $freq_{sDS}(S) > \sigma|sDS|$. Similarly, given an error threshold ϵ $(0 < \epsilon \leq \sigma)$, S is said to be **ϵ-frequent** or **sub-frequent**, if $\epsilon|sDS| \leq freq_{sDS}(S) < \sigma|sDS|$. And if $freq_{sDS}(S) < \epsilon|sDS|$, S is said to be **infrequent**.
SP	Given a transaction tr containing an itemset S. Let the items in S be ordered according to some pre-defined total order (lexicographical ordering). Then for the ordered itemset $< abcd >$, the **suffix projections** will be: $< abcd >, < bcd >, < cd > and < d >$. The total number of suffix projections will be $|S|$.

rithm. The rest of the chapter is organized as follows: Section 4.3 gives brief background on related concepts; Section 4.4 on page 66 explains the proposed data structure - BFI-forest; Section 4.5 on page 68 presents the proposed algorithm - AnyFI; Section 4.6 on page 88 presents the parallel framework MPAnyFI; Section 4.7 on page 90 presents the experimental results; Section 4.8 on page 101 highlights the main contributions of this chapter; followed by conclusions and future work in Section 4.9 on page 102.

4.3 Preliminaries

4.3.1 Definitions & Problem Statements

Table 4.1 gives a few notations and definitions. Following these definitions, we can define the problem of *FI mining from a data stream* as: *given DS, sDS and σ, find itemsets in sDS that have $freq_{sDS} \geq \sigma|sDS|$.* When we process transactions arriving with variable inter-arrival rate, the problem is referred to as ***Anytime FI Mining of Data Streams***.

Researchers use ϵ to prune away infrequent itemsets in order to save space and time [154, 96]. AnyFI also uses ϵ to prune away infrequent itemsets. This introduces approximation in results, and hence the quality of results is measured using - *precision* and *recall*.

4.3.2 Stream Model

The set sDS is modelled as a window of different forms - landmark window [98], sliding window [97, 183], damped window[97], etc., for different algorithms explained in Section 4.2 on page 61. In our work, we had used two stream models -*Damped Window* model for modeling stream in AnyFI, which is the sequential algorithm; and *Tilted Time Window Framework* for modeling multi-port streams in the parallel framework - MPAnyFI. We explain each of these as follows:

4.3.2.1 Damped Window

Damped window lowers the effect of older transactions with time and thus lets us control the contribution of a transaction with respect to its arrival time. Recent transactions get greater weightage and hence recent trends can be captured more effectively. We use a decay factor f, $0 < f < 1$, to decay the frequency counts of itemsets with time. We denote frequency count of an itemset S at time T_1 as $freq(S,T_1)$. At time T_2 ($T_2 > T_1$), the decayed frequency count of S will be:

$$freq(S,T_2) = freq(S,T_1) \times f^{(T_2-T_1)} \tag{4.1}$$

If we do not decay frequency counts with time, the frequency counts would be the true frequency counts as in a pure landmark window model with window starting from beginning of the stream. In AnyFI, whenever we update/increment the frequency count of an itemset due to arrival of a new transaction, we first decay the existing frequency count using Equation (4.1) and then increment it, i.e., if itemset S has occurred in a transaction arrived at time T_2, then the updated frequency count of S will be:

$$freq(S,T_2) = freq(S,T_1) \times f^{(T_2-T_1)} + 1 \tag{4.2}$$

Whenever we compute frequency count of an itemset S with respect to a support threshold (ϵ or σ), sDS is the window containing transactions occurring between the timestamps - current tid ($curr_{tid}$) and tid of the first transaction ($ftid$) where S has occurred. $ftid$ is stored in tree nodes of BFI-forest (see Section Section 4.4 on the next page).

4.4 BFI-Forest

BFI-forest is a summary structure used for anytime processing of incoming transactions by AnyFI and MPAnyFI. It stores the enumerated *suffix projections* (SPs) (see Table 4.1 on page 64 for definition of SP) of the incoming transactions in a compressed way. They are so arranged in the forest that it becomes easy and efficient to mine for FIs. Figure 4.1 on the next page illustrates its structure, which is elucidated as follows:

- BFI-forest consists of a set of *BFI-trees* (Buffered Frequent Itemset trees), whose total number $\leq |I|$. The root node of each tree represents a unique item from I. Figure 4.1 on the following page shows a BFI-forest constructed over $I = \{a, b, c, d, e\}$, and thus contains five BFI-trees. Each node of a BFI-tree represents an itemset that is formed by collecting all the items in the traversal path from the root to the node. For example, *node7* in *Tree1* of Figure 4.1 on the next page represents the itemset $\{a, c, d\}$, which is formed by the path: *node1* $\rightarrow$ *node6* $\rightarrow$ *node7*.
- An internal node of a BFI-forest stores the following fields:
  - ◇ $item$: It is an integer identifier representing a literal in set I. It is the *item indexed* at the current node.
  - ◇ $efreq$: It is the *frequency count* of the itemset represented at this node.
  - ◇ $ftid$: It is the transaction id (*tid*) at which this node was created, i.e., the first occurrence of the itemset represented by the current node. It is used for pruning infrequent itemsets (see Section 4.5.1 on page 68).
  - ◇ $ltime$: It is the *timestamp* at which the node was last accessed or updated. It is used to decay the the node's frequency count with time under the damped window model.
  - ◇ $buff$: This field represents the buffer at the node. Buffers store SPs that are incompletely processed and are waiting to be inserted into the tree. Their processing has been deferred either due to *insufficient time allowance* or *out of deliberation* to achieve some optimization goals (see Section 4.5.1 on page 68). Thus, buffers become a *key requirement* for AnyFI. Buffers in BFI-trees are implemented as *hash tables* with linear chaining as shown in Figure 4.1 on the following page. Using a hash table for the buffers makes the complexity of

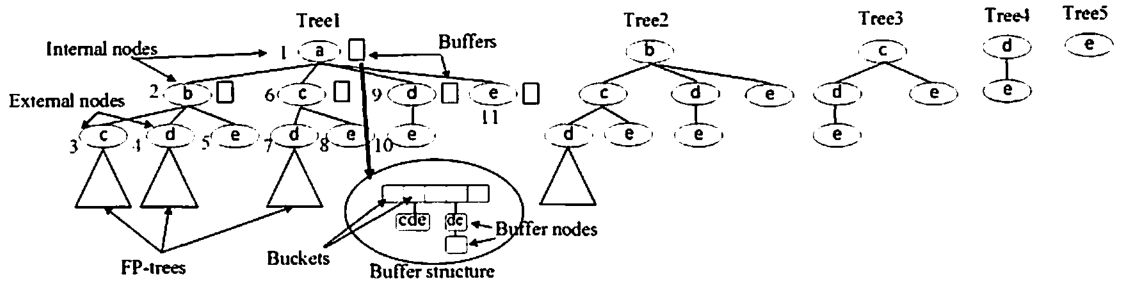


Figure 4.1: Structure of BFI-forest and BFI-trees

insertion and removal of a projection from any of its bucket to be $O(1)$. Each bucket in the hash table is a linked list of *buffer-nodes*. Each buffer-node stores *(i.)* $partial\_trans$: a SP; *(ii.)* $ftid$: Timestamp at which this buffer-node was created; *(iii.)* $ltime$: The last accessed/updated timestamp of the buffer-node; and *(iv.)* $efreq$: The frequency count of the SP stored at the buffer-node. The buffer has a limit on the number of buffer-nodes it can hash in total, which we denote as $buffCapacity$. The size of the hash table array (*hash_size*) is typically chosen as 10% of $|I|$. We use simple mod function as the hash function for the buffer, i.e. for a SP $p\_tr = < abc >$, the value of the bucket to which it should get indexed is computed by - a % $|hash\_size|$, where a is the first item in p_tr. Note that a (even b or c) is an integer literal $\in I$.

◇ *childArr*: It is an array of max size - $|I|$, used to store pointers to the sub-trees indexed at the current node. The size of this array varies among the tree node as we insert lexicographically ordered transactions (see Section 4.5 on the next page). Consider *Tree1* in Figure 4.1 which was constructed over $I = \{a, b, c, d, e\}$. The node at level 0 (node 1) will have a child array of size 4 as it can have sub-trees with their roots containing items - b, c, d & e. Similarly, node 2 would have a child array of size 3 as it can have sub-trees with their roots containing items - c, d & e only. Node 2 can never have item a in any of the sub-trees due to insertion of lexicographically ordering transactions. Thus array of size 3 is sufficient. Similarly, other nodes will contain child arrays of sizes as per the number of children they can have. Node 11 will not have any child array as there is no item in I that can occur after e. Similar scheme applies to other trees in the forest.

- All the nodes of BFI-trees that are present at a height, *Max_Height*, are called external nodes. The value of *Max_Height* is user defined. In Figure 4.1 on page 67, *Max_Height* is 2. So, the nodes 3, 4, 5, 7, 8 & 10 are external nodes. *node*11 is not external since it is not at *Max_height*. External nodes differ from internal nodes only in one field - instead of *childArr*, they store *fpRoot* which is a pointer to an FP-tree [184]. The suffix projections reaching *Max_Height* during insertion and refinement step are inserted into FP-trees. This reduces memory consumed by BFI-forest and eliminates the need to store so many infrequent projections (see Section 4.5). If there are no projections to be stored in the FP-tree beneath a given external node, *fpRoot* remains NULL (nodes 5, 8 & 10). Also note that buffers are present in both internal and external nodes.

4.5 AnyFI: Anytime Frequent Itemset Mining of Data Streams

The AnyFI algorithm in principle consists of the following steps that are elaborated in subsequent subsections.

- Read the incoming transactions one by one and order them lexicographically.
- Insert each transaction into BFI-forest and refine the trees of the forest as per the available time allowance.
- Intermittently prune the infrequent itemsets from BFI-forest after arrival of a certain batch of transactions.
- Whenever user requests for a mining result, BFI-forest is mined for FIs within the given time allowance.

4.5.1 Inserting a transaction into the BFI-forest and Refinement of BFI-forest

The insertion of an incoming transaction and its refinement is an anytime operation, i.e., it is interruptible whenever the time allowance expires, which is typically triggered by arrival of a new transaction. It is explained as follows:

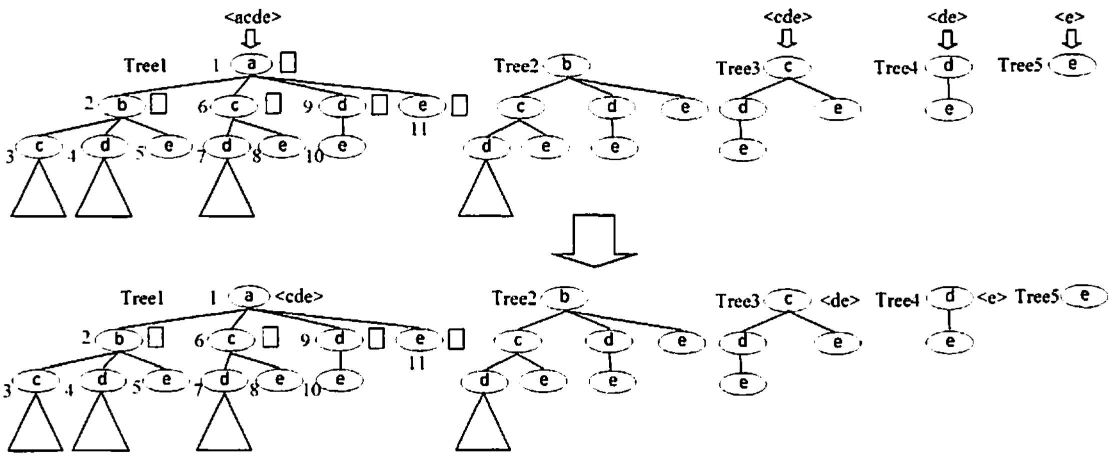


Figure 4.2: Inserting a transaction $< acde >$ into BFI-forest

Algorithm 4.1: Any-FI

```
1 procedure Any-FI(stream)
      Input  : Stream of continuously arriving transactions at a variable rate
      Output: Each transaction being inserted into the BFI-forest
2     count ← 0;
3     forest ← Init-BFI-Forest();
4     while !EOF(stream) do
5         trans ← Get-Next-Trans(stream);
6         Order-Lexicographically(trans);
7         Insert-In-BFI-Forest(forest, trans);
8         count ← count + 1;
9     end
```

4.5.1.1 Inserting a transaction into a BFI-forest

Each incoming transaction (*trans*) from the stream is processed one after the other (Algorithm 4.1). It is first lexicographically ordered and then inserted into the BFI-forest (lines 5-7 of Algorithm 4.1). To insert *trans* into the BFI-forest, we take its suffix projections and insert each projection into the root of the corresponding BFI-tree (lines 2-6 of Algorithm 4.2 on the next page). This is illustrated in Figure 4.2. Suppose $trans =< acde >$, we would insert the SPs: $< acde >$ in *Tree*1, $< cde >$ in *Tree*3, $< de >$ in *Tree*4 and $< e >$ in *Tree*5 as shown in the figure. To insert $< acde >$ in *Tree*1, we first increment the $efreq$ of the root node of Tree1 (which contains item a), after decaying it using Equation (4.2) on page 65 (Algorithm 4.3 on the following page). Please note that T_1 and T_2 will be *ltime* and *current_time* respectively. We then place the rest of the projection $< cde >$ (formed after trimming the Head, which is a) into its buffer. Similarly, we process the remaining projections by first incrementing the $efreq$ of the root nodes of trees and inserting the trimmed suffix projections into the buffers of the respective tree roots as shown in the

Algorithm 4.2: INSERT-IN-BFI-FOREST

```
1 procedure INSERT-IN-BFI-FOREST($forest, trans$)
      Input  : A BFI-forest and an incoming transaction
      Output: Transaction inserted into the BFI-forest
2     foreach $suffix\_proj$ of $trans$ do
3         $rootNode \leftarrow forest[\text{HEAD}(suffix\_proj)].root$;
4         $rootNode.efreq \leftarrow \phi(rootNode).efreq - 1$;
5         $trimmed\_proj \leftarrow suff\_proj$ after trimming its HEAD;
6         INSERT-IN-BUFFER($rootNode.buff, trimmed\_proj$);
7     end
8     if $NEW\_TRANS\_ARRIVED$ then return;
9     REFINE-BFI-FOREST($forest$);
```

Algorithm 4.3: ϕ

```
1 procedure $\phi(node)$
      Input  : A BFI-tree node or a buffer node
      Output: BFI-tree node with decayed frequency count
2     $node.efreq \leftarrow node.efreq \times f^{CURR\_TIME - node.ltd}$;
3     return $node$;
```

figure. Since $< e >$ is singleton, its insertion into *Tree5* will just be increment of $efreq$ of its root.

Since, buffers are implemented as hash tables with linear chaining, to insert $< cde >$ into the buffer of root of Tree1, we find the hash value of $< cde >$ to identify the bucket into which it has to be inserted ($bucketid = c \bmod hash\_size$). All the projections starting with c will be hashed to this bucket. Then we check if a buffer node indexing $< de >$ already exists in this bucket. If yes, we increment its $efreq$ by 1, after decaying it. Otherwise we create a new buffer for it and append it to the end of the bucket (see Figure 4.1 on page 67). All other projections are also inserted in the same way into their corresponding buffers.

This operation - *taking suffix projections and inserting into the root nodes of trees,* is an atomic operation (non-interruptible) (lines 2-6 of Algorithm 4.2). We then check if a new transaction has arrived, if yes we stop here and process the newly arrived transaction (lines 7-8 of Algorithm 4.2). However, if a new transaction has not arrived, we refine the forest until the time allows (line 9 of Algorithm 4.2).

4.5.1.2 Refinement of BFI-Forest

Each tree of BFI-forest undergoes refinement in a random biased way (Algorithm 4.4 on the next page). We select a tree for refinement from the forest with non-uniform probability, where we give higher weight to the trees whose root nodes have items of

Algorithm 4.4: Refine-BFI-Forest

```
1 procedure Refine-BFI-Forest(forest)
      Input : A BFI-forest
      Output: Trees of BFI-forest refined until time allows
2     while exitFlag != TRUE do
3         tree ← a biased random tree from forest;
4         if tree is not being pruned then
5             exitFlag ← Refine-BFI-Tree (tree.root);
6         end
7     end
```

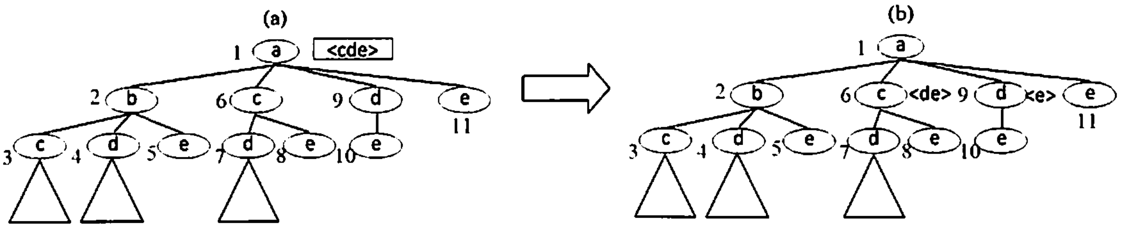


Figure 4.3: Refinement of BFI-tree

lower rank in I (when I is lexicographically ordered). If the probability of selecting a tree with root representing item i_j (j is the rank in lexicographically ordered I) is w_j, then $w_j = \frac{|d_j|}{\sum_j |d_j|}$, where $d_j = |I| - j + 1$ and $w_1 > w_2 > ... > w_{|I|}$. One can observe from Figure 4.1 on page 67 that the trees with roots representing items from lower ranks in lexicographically arranged I will have larger size than those with higher ranks. So, by doing a biased selection this way, larger trees get higher chance of getting selected for refinement. This makes sure that all the trees in the forest is refined uniformly. Also, we must note here that if we select a tree for refinement that is undergoing intermittent pruning (see Section 4.5.2 on page 75), we don't refine it and select another tree. After finishing refinement of a tree, if time allowance is remaining, we will pickup another tree and refine it. If during the refinement of a tree, a new transaction arrives, *exitFlag* becomes TRUE (lines 5 & 24 of Algorithm 4.5 on the following page), and we quit from there to process the newly arrived transaction.

To refine a BFI-tree, we take out the SPs stored in the buffers of its nodes and expand them into sub-trees until time allows. The refinement of each tree is carried out in depth first order (Algorithm 4.5 on the next page). Consider *Tree*1 in Figure 4.3, where $< cde >$ is inserted in its root's buffer. We start refining *Tree*1 beginning with its root (*curr_node* = *root*). First, we take out the first SP from a randomly chosen bucket of its buffer (lines 7-9 of Algorithm 4.5 on the next page). Consider $< cde >$ was taken out. We then update

Algorithm 4.5: REFINE-BFI-TREE

```
1  procedure REFINE-BFI-TREE(tree)
       Input  : A BFI-tree to refine
       Output: BFI-tree refined until time allows
2      stack ← INIT-NEW-STACK();
3      PUSH(stack, tree.root);
4      while IS-NOT-EMPTY(stack) do
5          if NEW_TRAN_ARRIVED then return TRUE ;
6          curr_node ← POP(stack);
7          randBuffNode ← GET-RAND-NODE(curr_node.buff);
8          randBuffNode.efreq ← $\phi$(randBuffNode);
9          ptrans ← randBuffNode.partial_trans;
10         if GET-HEIGHT(node) $\geq$ LEAF_LEVEL then
11             FP-INSERT(curr_node.fpRoot, ptrans)
12         else
13             foreach suffix_proj of ptrans do
14                 child ← curr_node.childArr[HEAD(suffix_proj)];
15                 if randBuffNode.efreq is $\epsilon$ - infrequent and child is NULL then
16                     continue;
17                 end
18                 if child is NULL then
19                     curr_node.childArr[HEAD(suffix_proj)] ← NEW-BFI-NODE();
20                 end
21                 INSERT-IN-BUFFER(child.buff, suffix_proj);
22                 child.efreq ← $\phi$(child.efreq) + randBuffNode.efreq;
23             end
24             if NEW_TRANS_ARRIVED then return TRUE ;
25             foreach suffix_proj of ptrans do
26                 affected_child = curr_node.childArr[HEAD(suffix_proj)];
27                 if buffer pruning condition for affected_child is met then
28                     PRUNE-BUFFER(affected_child)
29                 end
30                 if affected_child is $\epsilon$-frequent then
31                     if affected_child has a child into which suffix_proj can be inserted or affected_child is
                         $\theta$-frequent then
32                         PUSH(stack,affected_child);
33                     end
34                 end
35             end
36         end
37     end
38     return FALSE;
```

its frequency count using Equation (4.1) on page 65. If *curr_node* is an external node, we would insert this projection into the FP-tree beneath it (lines 10-11 of Algorithm 4.5). Otherwise, we take SPs of $< cde >$ and insert them into the buffers of the corresponding children of *curr_node* as shown in Figure 4.3 on page 71 (nodes 6 & 9 get projections into their buffers). The insertion into buffer is same as explained before. We also increment the frequency counts of nodes 6 & 9 with the frequency count of the corresponding SP being inserted into their buffers, after decaying them (line 20 of Algorithm 4.5). If any child into which a projection has to be inserted, doesn't exist, we create that child, insert the SP into its buffer and assign it a frequency count (lines 17-18 of Algorithm 4.5). Also, while inserting into the buffer, we take care that the buffers do not overflow. If they exceed a pre-defined capacity- *buffCapacity*, we remove a random SP from the bucket in which

Algorithm 4.6: PRUNE-BUFFER

```
procedure PRUNE-BUFFER($node$)
    Input  : A node of BFI-tree
    Output: Buffer of the node pruned
    foreach $buff\_node$ in $node.buff$ do
        if $\phi(buff\_node) < \epsilon.(curr_tid - buff_node.ftid)$ then
            if $node.childArr[\text{HEAD}(buff\_node.partialTrans)]$ is NULL then
                delete $buff\_node$ from $node.buff$;
            end
        end
    end
```

we are trying to append the new projection. After this step, we check if a new transaction has arrived. If yes, we quit the refinement and move on to process the next transaction. Note that at the beginning of refinement of every node, we do check if a new transaction has arrived (line 5 of Algorithm 4.5 on page 72), if yes, we simply quit this function and proceed to process the newly arrived transaction; otherwise we proceed to further steps.

Buffer Pruning After insertion of projections into the sub-trees, we now conduct buffer pruning, which prunes infrequent projections lying in the buffers. So, the buffers of affected children of *curr_node* (children into which suffix projections were inserted in the previous step - nodes 6 & 9 in Fig.Figure 4.3 on page 71) are pruned before we proceed with further refinement. We do not conduct buffer pruning each time we visit a given node in the traversal. This is because, each time we visit a node's buffer, we may not have infrequent projections. So, we let it accumulate a few infrequent projections so that all of them can be removed in one go. We conduct buffer pruning in intervals of some minimum time decided by a parameter γ and the height of the node. It can be observed that closer the node to the root, more filled will be its buffer. Thus, buffers at lesser depth must be pruned more often than the buffers at greater depth. The pruning interval (PI) for each node is computed using the following formula-

$$PI = \lceil (batch\_size)/(10 \times \gamma \times height(node) \rceil) \tag{4.3}$$

where *batch_size* is the number of transactions after which we perform intermittent pruning (see Section 4.5.2 on page 75). So, whenever we are visiting a node, we prune its buffer only when it was last pruned at least PI transactions earlier (lines 24-25 of Algorithm 4.5 on page 72). In buffer pruning (Algorithm 4.6), we visit every buffer-node in a given buffer (nodes from all the buckets) and prune them. If *partial_trans* in a buffer node being visited, is not ϵ-frequent (after decaying its frequency count), then we check if the current *affected_child* (node for which buffer pruning is being conducted) has a child in its *childArr* that corresponds to the head of the *partial_trans*. If it does, then we don't delete this projection, as we might lose a potential FI by removing it. Otherwise we safely delete it.

θ-deferring After buffer pruning, we now select which nodes in the tree are to be refined further. We do not refine all the nodes in the tree, rather we deliberately defer the refinement of certain nodes to save space and time. We let the nodes accumulate more itemsets in their buffers before they are refined or expanded for insertion into their sub-trees. This step is critical in making the insertion and refinement step faster as it avoids repeated insertions/removals of same infrequent itemsets into/from the forest. This is because the frequency counts of the itemsets represented by the nodes keeps deceasing as we go down the tree. Refining the lower level nodes each time will lead to insertion of infrequent itemsets in the form of infrequent sub-trees, which will be pruned again in the intermittent pruning step. This leads to repeated insertions and removal of infrequent itemsets. This can be avoided, if we are selective in refining nodes and let the buffers in the nodes accumulate more itemsets in the them before they can be refined. Moreover, many infrequent itemsets will be pruned from buffers itself, rather than getting expanded into a large number of infrequent sub-trees. This deferred refinement is achieved by a tuning parameter θ and this process is referred to as θ-deferring. In this step, for every affected child (children in which suffix projections were inserted earlier), we first check whether it is ϵ-frequent or not. If yes, then we check whether the sub-tree corresponding to the head of the projection to be inserted into it, is present in its *childArr* or not. If this is so, we push this node (*selected_child*) into the stack, so that it can be refined in subsequent iteration of the DFS order (lines 26-28 of Algorithm 4.5 on page 72). For example, consider node 6 in Figure 4.3 on page 71. If it is ϵ-frequent and there exists a sub-tree with root *d* present in its *childArr* (node 7), we would want this node to be refined further and thus, push it into the stack. However, if the sub-tree doesn't exist, then we check if the *selected_child* is θ-frequent or not. If it is so, only then we would want this node to be refined further and we push it into the stack. Else, we don't refine this node and let it accumulate more transactions in its buffer before it gets refined further, thus saving space which would otherwise be occupied by an infrequent sub-tree. After this, we proceed

Algorithm 4.7: PRUNE-FOREST

```
procedure PRUNE-FOREST(forest)
    Input  : BFI-forest
    Output: BFI-forest post pruning
    foreach tree in forest do
        stack ← INIT-NEW-STACK();
        PRUNE-BUFFER(tree.root);
        if $\phi(tree.root) < \epsilon.(curr\_tid - tree.root.ftid)$ then
            DELETE-SUB-TREE(tree.root);
            continue;
        else
            PUSH(stack);
        end
        while IS-NOT-EMPTY(stack) do
            node ← POP(stack);
            PRUNE-BUFFER(node);
            foreach child in node.childArr do
                if IS-EXTERNAL-NODE(child) then
                    FP-PRUNE(child.fpRoot);
                else if $(\phi(node.buff[child.item]) - \phi(child)) < \epsilon.(curr\_tid - child.ftid)$ then
                    DELETE-SUB-TREE(child);
                else
                    PUSH(stack, child);
                end
            end
        end
    end
```

for the next iteration in the DFS traversal, where the nodes accumulated in the stack are refined (line 6 of Algorithm 4.5 on page 72).

4.5.2 Intermittent Pruning of BFI-forest

Since we are operating in a stream environment, it is required to regularly prune away itemsets which become infrequent from our summary structure, to manage the space constraint as time progresses. This is achieved by an intermittent pruning mechanism (Algorithm 4.7). This step also helps in making insertion efficient as it avoids visiting unnecessary branches in the trees, by pruning them periodically after a certain *batch_size* of arriving transactions. We iterate over all the trees in the forest and prune each tree separately. To prune a given tree, we use depth first traversal. In the traversal, when we encounter a node that is not ϵ-frequent (or is infrequent), we delete it along with all its child sub-trees. Removing the sub-trees beneath infrequent nodes doesn't affect the accuracy of the algorithm because all itemsets in those sub-trees can never be frequent. If however, the node is ϵ-frequent, then we insert it into the stack so that its children can be pruned in subsequent iterations. If the node we visited was a leaf, we prune its FP-tree as explained in [97], i.e., if an infrequent item is inside the FP-tree, we delete this item from all branches of the FP-tree by traversing the similar node links present in it. And after deleting the nodes with that item, we merge their child branches with those of their parents recursively and update the frequency counts. For more details, please refer to [97].

Proposition 3. Deletion of an infrequent node (or a sub-tree) from a given BFI-tree doesn't affect the correctness of the results, i.e. frequency counts of the itemsets in the same tree or the other trees are not affected.

Justification. AnyFI, enumerates all possible suffix projections of incoming transactions, and inserts them in different trees of the forest. And during the refinement step while taking a projection down the tree, SPs are enumerated at every level and get stored in independent branches. The frequency counts of itemsets represented by the other branches (or trees) have no connection with the node being deleted. There is exactly one node for representing any given itemset in the entire forest. So, unlike other algorithms like DSM-FI [96], deletion of a sub-tree in BFI-tree does not affect the other branches of the tree (and other trees as well), and thus there is no need to update or remove items from other branches of the trees. Hence, it doesn't interfere with frequency counts of other itemsets in the forest. Also, during deletion of a node (or a sub-tree) from the tree, we are not required to update the frequency counts of nodes in the path starting from the node being deleted to the root (like in FP-tree). This is because, we do not store prefixes in BFI-trees. So the itemset represented until the parent of the node being deleted would still remain frequent with the frequency count as it is. This saves a lot of tree traversals during the intermittent pruning step, making it very efficient.

We conduct intermittent pruning of the forest without actually halting or disturbing the incoming stream. To achieve this, we use multi-threading. We dedicate a separate thread to perform this task. While a given tree is being pruned, we continue inserting SPs of incoming transactions into its root node. However, we do not refine this tree (line 4 of Algorithm 4.4 on page 71) until pruning finishes. Also, since pruning of a tree is efficient (as substantiated by proposition Proposition 3), the intermittent pruning step has negligible effect on the accuracy of the algorithm.

Algorithm 4.8: ANY-MINE

```
procedure ANY-MINE($forest, min\_sup$)
    Input  : BFI-forest and support threshold
    Output: Set of frequent itemsets whose frequency counts are greater than $min\_sup$
    EMPTY-BUFFERS(forest);
    MINE-FOREST ($forest, min\_sup$);
```

Algorithm 4.9: EMPTY-BUFFERS

```
procedure EMPTY-BUFFERS($forest$)
    Input  : A BFI-forest
    Output: BFI-forest with buffers emptied
    $queue \leftarrow$ INIT-NEW-QUEUE();
    foreach $tree$ in forest do
        ENQUEUE($queue, tree.root$);
    end
    while IS-NOT-EMPTY($queue$) do
        if $TIME\_OVER$ == TRUE then return;
        $node \leftarrow$ DEQUEUE($queue$);
        if $|node.buff| > 0$ then
            FLUSH-BUFFER($node.buff$);
        end
        foreach $child$ in $node$ do
            Enqueue($queue, child$);
        end
    end
```

4.5.3 Mining BFI-forest for Frequent Itemsets

FIs are extracted from the BFI-forest in an offline manner whenever user requests for them. The mining is very simple and straight forward. During insertion, we have inserted suffix projections of incoming transactions in all the trees. And within each tree we have enumerated all possible suffix projections and inserted them either into the tree or in the buffers of the tree nodes. Thus, we don't have to enumerate any candidate itemsets like in Apriori or generate conditional trees like in FP-growth. This makes mining BFI-forest faster than that of existing algorithms (see Section 4.7 on page 90 for results). All we need to do is to empty all the buffers in the trees and simply traverse the tree in depth first order accumulating the itemsets whose frequency counts are greater than a threshold - min_sup or σ (Algorithm 4.8). When we empty the buffers, we traverse every node in each tree in DFS order (Algorithm 4.9), and flush out the projections in the buffer and push them to the next level of the tree as was done in the refinement step (ignoring infrequent projections).

After the buffers are emptied, we traverse each tree in DFS order again (Algorithm 4.10 on the next page), where we check at every node if the itemset represented by it is σ-frequent or not. If yes, we store the itemset in $FIset$, which is a set to accumulate FIs (lines

Algorithm 4.10: MINE-FOREST

```
procedure MINE-FOREST($forest, min\_sup$)
    Input  : BFI-forest and minimum support threshold $min\_sup$
    Output: Set containing frequent itemsets
    $itemStack \leftarrow$ INIT-NEW-STACK();
    $FISet \leftarrow$ INIT-NEW-SET();
    foreach $tree$ in $forest$ do
        $nodeStack \leftarrow$ INIT-NEW-STACK();
        PUSH($nodeStack, tree.root$);
        while IS-NOT-EMPTY($nodeStack$) do
            $curr\_node$ = POP($nodeStack$);
            if $curr\_node.efreq > min\_sup$ then
                PUSH($itemStack, curr\_node.item$);
                INSERT-IN-SET($FISet$,COLLECT($itemStack$));
                if IS-EXTERNAL-NODE($curr\_node$) then
                    $fpItemsList \leftarrow$ FP-GROWTH($curr\_node, fpRoot$);
                    foreach $fpItem$ in $fpItemsList$ do
                        INSERT-IN-SET($FISet$, CONCATENATE(COLLECT($itemStack$)), $fpItem$);
                    end
                else
                    foreach $child$ of $curr\_node$ do
                        PUSH($nodeStack, child$);
                    end
                end
            end
            POP($itemStack$);
        end
    end
```

9-11 of Algorithm 4.10). Whenever an external node is encountered, we simply mine the FP-tree beneath and concatenate all the FIs that come from FP-tree with the itemset represented by the current node, and add all of them to the *FISet* (lines 12-15 of Algorithm 4.10). For example, in Fig. Figure 4.3 on page 71, if *acd* is frequent, the FIs mined from the FP-tree beneath node 7, will be appended to *acd* and added to *FISet*. Finally *FISet* would consists of all the FIs. In the pseudo code (Algorithm 4.10) we have used two explicit stacks- *nodeStack* for the DFS traversal (storing tree nodes); and *itemStack* to store all the items in the traversal path from *root* to the current node being traversed. The COLLECT() procedure returns an itemset made out of all the items in the *itemstack*, which is the itemset represented by the current node.

Proposition 4. To mine for an itemset starting with an item i_j, it is necessary and sufficient to mine the tree with root representing i_j.

Justification. Consider mining for an itemset $\{b,c,d\}$, i.e. $i_j = b$. Consider insertion of a transaction (say $< abcd >$) into the forest, where we inserted SPs - $< abcd >$, $< bcd >$, $< cd >$ and $< d >$ into the trees with roots having a, b, c and d respectively. So, the contribution of transaction $< abcd >$ to the frequency count of itemset $\{b,c,d\}$ is taken care by the insertion of SP $< bcd >$ into the tree with root as b. Similarly its contribution to the frequency count of $\{cd\}$ is taken care by insertion $< cd >$ into the tree with root having c and so on. Thus, for mining $\{b, c, d\}$ it sufficient to mine the tree with root having item b. □

Note that the proposition Proposition 4 on page 78 is also applicable within each tree as well. This also gives another justification for the correctness of intermittent pruning step, where we are able to prune a sub-tree of a given BFI-tree without affecting other BFI-trees or other branches in the same tree (see Section 4.5.2 on page 75).

The ANY-MINE algorithm, consists of two steps - 1) EMPTY-BUFFERS, and 2) MINE-FOREST. We observed that EMPTY-BUFFERS is the step that takes majority of the time of ANY-MINE. Therefore, we have made EMPTY-BUFFERS step anytime (line 6 of Algorithm 4.9 on page 77), i.e. it is interruptible and when the quantum of time allotted by the user expires, the algorithm exits from this step and quickly mines the forest accumulating FIs to output them. At this point, the residual projections in the buffer, if any, are ignored, and a very quick mining result with compromised accuracy is obtained. Note that given more time allowance for the empty-buffers step, the accuracy of the mining results improves (see Fig. Figure 4.16 on page 94).

4.5.4 Why is AnyFI Efficient? A Summary:

ANYFI enumerates all possible SPs of the incoming transactions and inserts them into the BFI-Forest. This makes the *mining of FIs* from the forest *very efficient*, making it merely a traversal of its trees without enumerating candidate itemsets like in apriori like methods [96, 47, 50] or generating conditional trees like in FP-growth like methods [98, 97, 53, 184, 105]. Also, the insertion and refinement step is carried out efficiently by using techniques like θ-deferring, buffer pruning, intermittent pruning and usage of FP-tree beyond *Max_height*. They are summarized as follows:

- *Theta-deferring*: θ-deferring deliberately delays the refinement of certain nodes in the tree and processes only those SPs that have the potential to become frequent in future. This prevents repeated creation and deletion of infrequent sub-trees, leading to reduction in memory and process time.

- *Buffer Pruning and Intermittent Pruning*: Buffer pruning, prunes the infrequent itemsets from the buffers attached to the tree nodes alongside refinement, and contributes to memory reduction. Intermittent pruning, prunes infrequent sub-trees in the forest, also leading to memory reduction.
- *Usage of FP-tree*: We know that, as the length of the itemsets increase, their frequency counts reduce [6] and many of the greater length itemsets would be infrequent. In BFI-forest, these greater length itemsets are stored at the bottom levels of BFI-trees, of which most of them would be infrequent. Also, at such height, branching in the BFI-tree shall be very high due to storage of large number of SPs, which means we shall end up storing many infrequent subtrees. And, these sub-trees would be repeatedly created and deleted in the process of insertion and intermittent pruning respectively. However, we know that FP-trees are efficient in storage, especially when their sizes are small. So, keeping FP-trees beyond depth = *Max_height* in a BFI-tree, avoids storing those infrequent subtrees, and thus reduces memory consumption and processing time. Also, as mining small sized FP-trees is efficient, the mining of BFI-forest remains efficient.

4.5.5 Theoretical Analysis

In this section, we present theoretical analysis for: *(a)* deriving the cost of insertion and refinement step of AnyFI; *(b)* deriving the complexity of space occupied by BFI-forest. We make the following assumptions for this analysis:

We model the stream of transactions by assuming a probability distribution on the dictionary elements. Given set $I = \{i_1, i_2, ...\}$ containing all the items in the dictionary with size = $|I|$. Let λ_x be probability that i_x appears in an incoming transaction (λ_x estimates the support count of singleton itemset $\{i_x\}$). We assume that the items occur independent of each other in the incoming transactions. Also given is the decay factor f $(0 \leq f \leq 1)$, which is used to decay the support counts of itemsets arrived in older transactions with time. We estimate the behaviour of θ-deferring using a probability, which we refer to as c. So, c denotes the probability of refining any given node in the tree, i.e. removing a SP from the node's buffer and further enumerating and inserting its SPs

- *Buffer Pruning and Intermittent Pruning*: Buffer pruning, prunes the infrequent itemsets from the buffers attached to the tree nodes alongside refinement, and contributes to memory reduction. Intermittent pruning, prunes infrequent sub-trees in the forest, also leading to memory reduction.
- *Usage of FP-tree*: We know that, as the length of the itemsets increase, their frequency counts reduce [6] and many of the greater length itemsets would be infrequent. In BFI-forest, these greater length itemsets are stored at the bottom levels of BFI-trees, of which most of them would be infrequent. Also, at such height, branching in the BFI-tree shall be very high due to storage of large number of SPs, which means we shall end up storing many infrequent subtrees. And, these sub-trees would be repeatedly created and deleted in the process of insertion and intermittent pruning respectively. However, we know that FP-trees are efficient in storage, especially when their sizes are small. So, keeping FP-trees beyond depth = *Max_height* in a BFI-tree, avoids storing those infrequent subtrees, and thus reduces memory consumption and processing time. Also, as mining small sized FP-trees is efficient, the mining of BFI-forest remains efficient.

4.5.5 Theoretical Analysis

In this section, we present theoretical analysis for: *(a)* deriving the cost of insertion and refinement step of AnyFI; *(b)* deriving the complexity of space occupied by BFI-forest. We make the following assumptions for this analysis:

We model the stream of transactions by assuming a probability distribution on the dictionary elements. Given set $I = \{i_1, i_2, ...\}$ containing all the items in the dictionary with size = $|I|$. Let λ_x be probability that i_x appears in an incoming transaction (λ_x estimates the support count of singleton itemset $\{i_x\}$). We assume that the items occur independent of each other in the incoming transactions. Also given is the decay factor f $(0 \leq f \leq 1)$, which is used to decay the support counts of itemsets arrived in older transactions with time. We estimate the behaviour of θ-deferring using a probability, which we refer to as c. So, c denotes the probability of refining any given node in the tree, i.e. removing a SP from the node's buffer and further enumerating and inserting its SPs into the sub-trees in the insertion and refinement step. Now, we derive the expected time complexity of insertion and refinement a transaction.

Theorem 2. The expected cost of insertion and refinement of a m-length transaction into BFI-forest is $\Theta\left(\frac{m\ (1+c)^m}{c}\right)$.

Proof. First, we compute the expected cost of refinement of a SP into a BFI-Tree beginning from the root. Let X be a SP ($X =< i_1, i_2, ..., i_n >; i_l \in I$) of length n and let $T(n)$ denote this cost. So, the total time for refinement of X is the total time for refining all the tree nodes traversed in the path of refinement of X. Note that the nodes in the refinement path are refined with probability c to accommodate θ-deferring. Then, for *length* $= 1$ (base case): $T(1) = \Theta(1)$, which is the cost of incrementing the support count of the tree node. For *length* $= n$:

$$T(n) = \underbrace{\Theta(n)}_{\text{cost of refining the root}} + \underbrace{c \times [T(n-1) + T(n-2) + ... + T(1)]}_{\text{cost of refining the sub-trees underneath root}} \tag{4.4}$$

which is the sum of costs of *(i.) refining the root,* i.e. enumerating the SPs of X, inserting them into the buffers of their respective sub-trees and then incrementing the frequency counts, and *(ii.) refining the sub-trees* underneath the root. Now, Equation (4.4) implies that there exists constants a_1 and a_2 such that the following inequality holds for some $n > n_0$:

$$a_1\ n + c\ [T(n-1) + ... + T(1)] < T(n) < a_2\ n + c\ [T(n-1) + ... + T(1)]$$

Using this we can express $T(n)$ and $T(n-1)$ respectively as:

$$T(n) < a_2\ n + c\ [T(n-1) + ... + T(1)] \tag{4.5}$$

$$T(n-1) > a_1\ (n-1) + c\ [T(n-2) + ... + T(1)] \tag{4.6}$$

Equation (4.5) can be re-written as

$$T(n) - a_2\ n - c\ T(n-1) < c\ [T(n-2) + ... + T(1)] \tag{4.7}$$

Now, substituting the RHS part of Equation (4.7) in Equation (4.6), we get an upper bound on $T(n)$ which is:

$$T(n) < (a_2 - a_1)\ n + a_1 + (1+c)\ T(n-1) \tag{4.8}$$

Similarly, considering the alternate arrangement of inequalities for $T(n)$ and $T(n-1)$, we can get a lower bound on $T(n)$, which will be similar to Equation (4.8) with inequality sign reversed. We derive the complexity using the upper bound (Equation (4.8) on page 81) herein, and a similar derivation can be shown with the lower bound as well. Considering $a_2 - a_1 = a_3$, we get:

$$T(n) < a_3\, n + a_1 + (1+c)\; T(n-1)$$

Substituting $T(n-1)$ in terms of $T(n-2)$ and considering $a_4 = a_1 - a_3$, we get:

$$T(n) < a_3\, n + a_1 + (1+c)\; [a_3\,(n-1) + a_1 + (1+c)^2\; T(n-2)]$$

$$T(n) < a_3\, n + a_1 + (1+c)\; a_3\, n - (1+c)\; a_3 + (1+c)\; a_1 + (1+c)^2\; T(n-2)]$$

$$T(n) < a_3\, n + a_1 + (1+c)\; a_3\, n + (1+c)\; (a_1 - a_3) + (1+c)^2\; T(n-2)]$$

$$T(n) < a_3\, n\; [1 + (1+c)] + a_1 + (1+c)\; a_4 + (1+c)^2\; T(n-2)]$$

Similarly, substituting $T(n-2)$ in terms of $T(n-3)$, $T(n-3)$ in terms of $T(n-4)$ and so on we get:

$$T(n) < a_3\, n\; [1 + (1+c) + ... + (1+c)^{n-2}] + a_1 + (1+c)\; a_4 + (1+c)^2 a_5 + ... + (1+c)^{n-2}\; a_{n+1} + (1+c)^{n-1}\; T(1)$$

$$T(n) < a_3\, n\; \frac{(1+c)^{n-1} - 1}{c} + SOME\_CONSTANT + (1+c)^{n-1}\; \Theta(1)$$

This can be simplified to:

$$T(n) < (a_3\, n + c)\; \frac{(1+c)^{n-1}}{c} + SOME\_CONSTANT$$

Using this, we compute the cost of insertion and refinement of a transaction *tr* of length *m* into the BFI-forest. This is equal to sum of *(i.)* cost of enumerating SPs of *tr*; *(ii.)* inserting SPs into the buffers of the root nodes of respective BFI-Trees; *(iii.)* cost of incrementing the frequency counts; and *(iv.)* cost of refining all BFI-Trees. This is equal to:

$$Total\ time = \underbrace{\Theta(m)}_{\text{(i), (ii) \& (iii)}} + \underbrace{T(m) + T(m-1) + T(m-2) + ... + T(1)}_{\text{(iv)}}$$

by solving which we get:

$$Total\ time < (a_m\, m + c)\frac{(1+c)^m}{c} + SOME\_CONSTANT$$

Similarly, if we derive using the lower bound on $T(n)$, as explained earlier, we get a similar equation with inequality sign reversed. Thus we can safely claim that the total time is: $\Theta\left(\frac{m\ (1+c)^m}{c}\right)$. □

Note that actual expected time will be much lesser than the derived complexity because - 1) Usage of use FP-trees at *Max_height* reduces the size of each tree and also the expected insertion and refinement time, as insertion in FP-tree is a $O(m)$ operation; 2) While computing the expected time, we ignored the anytime property (interruption in between) and gave worst case time when there is sufficient time allowance available to complete insertion and refinement of a transaction. Also note that the value of c has been experimentally found to be in the range - [0.05 - 0.2] for the values of θ chosen based on recommendations given in Section 4.7.0.4 on page 96. Also, an analysis has been presented in Section 4.7.0.5 on page 97, where the curve obtained by the above complexity is plotted and compared with the curve that plots the actual time taken for insertion and refinement of varying length transactions.

We now derive the space occupied by the BFI-forest. Since we enumerate all possible SPs and expand them into sub-trees, the total space occupied by BFI-forest will be of the order of total number of nodes in it, along with their buffers. Since every itemset is represented by a unique node in BFI-forest, the total number of nodes in the forest will be $\leq$ total number of ϵ-frequent itemsets. It will be "less than" because the decay factor f decays the frequency counts with time. We provide an estimate on maximum number of nodes present in the forest at the steady state, while accounting the role of the decay factor.

Consider a k-length itemset $\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}$ which is a subset of I and the items are arranged in non-increasing order of their frequency counts, i.e. $\lambda_{l_1} \geq \lambda_{l_2} \geq \lambda_{l_k}$, with their indices following the order $0 \leq l_1 < l_2 < ... < l_k < |I|$. We first derive a support threshold known as critical support threshold (η), which is the minimum support required for an itemset to be present in the BFI-forest at steady state.

Theorem 3. All those k-length itemsets that have $\lambda_{\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}} > \eta$, where $\eta = (1-f)\epsilon$, will be present in the forest at steady state. $\left(\lambda_{\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}} \text{ denotes } \prod_{j=1}^{k} \lambda_{l_j}\right)$

Proof. Let us consider that we have processed a good number of transactions in the stream, and the forest is at steady state. We now examine the itemsets which can exist in the tree at this state. Let $freq(\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}, t)$ be the decayed frequency count of the itemset $\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}$ at time t. For brevity, we denote $\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}$ as $i_{\{l_1, l_2, ..., l_k\}}$. We get:

$$freq(i_{\{l_1,l_2,...,l_k\}}, t) = f \times freq(i_{\{l_1,l_2,...,l_k\}}, t-1) + X_t \quad (0 < f < 1)$$

where $X_t = 1$ if $i_{\{l_1,l_2,...,l_k\}}$ occurs in the latest transaction tr arriving at time t; 0 otherwise

Taking expectations, we get

$$E[freq(i_{\{l_1,l_2,...,l_k\}}, t)] = f \times E[freq(i_{\{l_1,l_2,...,l_k\}}, t-1)] + E[X_t]$$

$\because E[X_t] = 1 \times Pr[X_t = 1] + 0 \times Pr[X_t = 0]$ by definition, we get

$$Pr[X_t = 1] = Pr[i_{l_1} \in tr \wedge i_{l_2} \in tr \wedge ... \wedge i_{l_k} \in tr] = \prod_{j=1}^{k} Pr[i_{l_j} \in tr] = \prod_{j=1}^{k} \lambda_{l_j} = \lambda_{\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}}$$

On writing $E[freq(i_{l_1,l_2,...,l_k}, t-1)]$ in terms of $E[freq(i_{l_1,l_2,...,l_k}, t-2)]$ and so on, we get

$$E[freq(i_{l_1,l_2,...,l_k}), t] = f^{t-1} + (\lambda_{\{l_1,l_2,...,l_k\}}) \frac{1 - f^{t-1}}{1 - f}$$

After long time, at steady state we have: $\lim_{t\to\infty} E[freq(i_{l_1,l_2,...,l_k}, t)] = \frac{\lambda_{\{l_1,l_2,...,l_k\}}}{1-f}$. So, an itemset $\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}$ would persist in the forest if is at least ϵ-frequent, i.e., $\frac{\lambda_{\{l_1,l_2,...,l_k\}}}{1-f} > \epsilon$. This means that all the itemset that have $\prod_{i_{l_j} \in I} \lambda_{l_j} > (1-f)\epsilon \ (= \eta)$ would persist in the forest. □

Now, we shall find an estimate on total number of itemsets that qualify the above critical support threshold criteria. This shall estimate the total number of nodes in the BFI-forest, giving us an estimate of space occupied. Let λ_x now be modelled using the *zipf* distribution [187], such that $\lambda_x = \lambda(x) = \frac{c_1}{c_2 + x}$, where c_1 & c_2 are constants with $c_1, c_2 > 0$ & $c_1 \leq 1$ and x is the rank of the item i_x in non-increasingly sorted I (according to the frequency counts). Using this assumption, we now estimate the count of k-length FIs in two steps: *(1)* in Lemma 1 on the following page we estimate the total number of integral points lying inside a high dimensional hyperbola [188] centered at the origin and bounded by the positive coordinate axes (see Figure 4.4a on the next page); *(2)* we use Lemma 1 on the following page to estimate the count of k-length FIs.

A k-length frequent itemset of the form $\{i_{l_1}, i_{l_2}, ..., i_{l_k}\}$ (items are arranged in non-

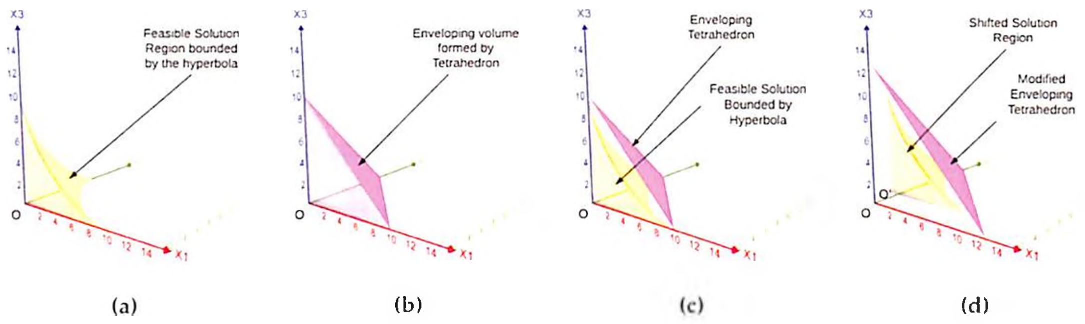

X3
Feasible Solution Region bounded by the hyperbola
O
X1
(a)
Enveloping volume formed by Tetrahedron
(b)
Enveloping Tetrahedron
Feasible Solution Bounded by Hyperbola
(c)
Shifted Solution Region
Modified Enveloping Tetrahedron
O'
(d)

Figure 4.4: (a) The solution region bounded by a high dimensional hyperbola and the co-ordinate planes; (b) The enveloping tetrahedron which approximates the hyperbola; (c) The tetrahedron fully covering the feasible solution region; (d) The solution region and enveloping tetrahedron after shifting the origin to $(1, 1, 1)$.

increasing order of their frequency counts) can be represented by a point on a k-dimensional hyper-plane with its coordinates $= (l_1, l_2, ..., l_k)$. After modelling the sorted support counts of items by the zipf function, we will find that the FIs are restricted to lie within a k-dimensional hyperbola - $(x_1 + a)(x_2 + a) \ldots (x_k + a) = D$ (see Theorem 4 on the next page). Note that $x_j (1 \leq j \leq k)$ denotes a dimension on k-dimensional hyper-plane. By estimating the total number of integer lattice points that lie inside the volume formed by the above hyperbola (Figure 4.4a), we can estimate the count of k-length FIs. For this, we find an enveloping solid (a k-dimensional tetrahedron) that envelopes the volume of the solid formed by the k-dimensional hyperbola as shown in Figure 4.4b & Figure 4.4c. The volume of this enveloping tetrahedron shall now give us an estimate of the volume of the hyperbola, which will give an estimate of the total number of integer lattice points lying inside the solid formed by the hyperbola. This is achieved by the following lemma:

Lemma 1. The number of solutions of the equation $(x_1 + a)(x_2 + a) \ldots (x_k + a) \leq D$ such that $0 \leq x_1 < x_2 < \ldots < x_k$, $x_i \in \mathbb{Z}$, $i \in \{1, 2, ..., k\}$ and $p > 0$, is less than $\dfrac{(I_{nt} + k + 1)^k}{k!}$, where $I_{nt} = \dfrac{D}{a^{k-1}} - a$ is the intercept over any coord. axis.

Proof. We begin by identifying the intercepts cut by this equation on each of the k axes. By symmetry all of them are at an equal distance from the origin. From Figure 4.4a, it is easy to see that the intercepts are at a distance of $\frac{D}{a^{k-1}} - a$ from the origin. Let $I_{nt} = \frac{D}{a^{k-1}} - a$ be this value.

Consider a k-dimensional tetrahedron (simplex) with $(k + 1)$ vertices - $(v_0, v_1, ..., v_k)$, such that v_0 lies on the origin, while v_i lies on x_i (the i^{th} coordinate axis) at a distance I_{nt} from the origin, i.e. $v_i = I_{nt} \times e_i$ (see Figure 4.4c on page 85) with e_i as the basis vector. The volume of the solid bounded by this tetrahedron is easier to find as compared to the original curve. The equation of this tetrahedron is $x_1 + x_2 + \ldots + x_k = I_{nt}$. In its current form, the enveloping tetrahedron is not covering the points which are lying on the surface of the original solid since the volume will only consider interior points. So we shift the origin to the point $(1, 1, ..., 1)$ such that its equation now becomes $(x_1 - 1) + (x_2 - 1) + ...(x_k - 1) = I_{nt}$, i.e., $x_1 + x_2 + ... + x_k = I_{nt} + k$ (Figure 4.4d on page 85). Finding solution to this equation will now give us an estimate of the number of points within, as well as on the boundary of the solid formed by the hyperbola. The new tetrahedron must have intercept $> I_{nt} + k$ in all dimensions. To keep a margin, we take it to be $I_{nt} + k + 1$. Hence, its vertices are at a distance of $I_{int} + k + 1$ from origin.

For a tetrahedron with vertices - $(v_0, v_1, \ldots, v_k)$, its volume [189] is: $\left|\frac{1}{k!}\det\left(v_1 - v_0, v_2 - v_0, \ldots, v_n - v_0\right)\right|$, where $\det\left(v_1 - v_0,\quad v_2 - v_0,\quad \ldots,\quad v_n - v_0\right)$ is the determinant of the diagonal matrix with the given diagonal elements. Note that $v_r - v_0$ represents an edge of the tetrahedron. For all such values of r, the length of an edge is equal to I_{nt}. Since the determinant of a diagonal matrix is simply the product of its diagonal elements, the volume of the new enveloping tetrahedron, will be $= \frac{(I_{nt} + k + 1)^k}{k!}$. □

Note that in Figure 4.4 on page 85, the hyperbola is shown only for the first quadrant. We now estimate the total number of k-length frequent itemsets in BFI-forest.

Theorem 4. Number of itemsets of length k in the BFI-forest is estimated by $\frac{(I_{nt_k} + k + 1)^k}{k!}$, where $I_{nt_k} = \frac{c_1^k}{c_2^{k-1}}\eta - c_2$.

Proof. A singleton itemset $\{i_l\}$ $(0 \leq l_1 \leq |I|)$ is frequent if $sup(\{i_{l_1}\}) \geq \eta$. Replacing the true support of $\{i_{l_1}\}$ with its expected support, we get:

$$\frac{c_1}{c_2 + l_1} \geq \eta \implies \frac{1}{c_2 + l_1} \geq \frac{\eta}{c_1} \implies c_2 + l_1 \leq \frac{c_1}{\eta} \implies l_1 \leq \frac{c_1}{\eta} - c_2$$

So, for all values of $l_1 \geq 0$ satisfying the above inequality, we get a corresponding item $i_{l_1} \in I$ which is 1-FI with respect to the support η. So, estimated total number of 1-FIs is $\left\lceil \frac{c_1}{\eta} - c_2 \right\rceil + 1$.

Count of 2-FIs: Let us consider an arbitrary 2-length FI $\{i_{l_1}, i_{l_2}\}$ such that $0 \leq l_1 < l_2 < |I|$. For $\{i_{l_1}, i_{l_2}\}$ to be frequent, $sup(\{i_{l_1}, i_{l_2}\}) \geq \eta$. Again replacing true support with the expected support we get:

$$\frac{c_1}{l_1 + c_2} \times \frac{c_1}{l_2 + c_2} \geq \eta \implies \frac{c_1^2}{(l_1 + c_2)(l_2 + c_2)} \geq \eta \implies (l_1 + c_2)(l_2 + c_2) \leq \frac{c_1^2}{\eta} \tag{1.9}$$

So now our problem reduces to find l_1 and l_2 such that they satisfy inequality (1.9). According to Lemma 1, the number of such (l_1, l_2) pairs is estimated by $\frac{I_{nt_2}^2 + 3}{2!}$ where $I_{nt_2} = \frac{c_1^2}{c_2 \eta} - c_2$.

Count of k-FIs: The general case of estimating the number of k-FIs is a simple extension of the above. For $< i_{l_1}, i_{l_2}, ..., i_{l_k} >$ to be frequent, it's support must exceed η, i.e., $\prod_{r=1..k} \frac{c_1}{l_r + c_2} \geq \eta$. We will finally get the constraint similar to inequality mentioned in lemma Lemma 1 on page 85, i.e. $(l_1 + c_2)(l_2 + c_2) \ldots (l_k + c_2) \leq \frac{c_1^k}{\eta}$. So, the number of $(l_1, l_2, ..., l_k)$ tuples that satisfy the above inequality are: $\frac{(I_{nt_k} + k + 1)^k}{k!}$ where $I_{nt_k} = \frac{c_1^k}{c_2^{k-1}\eta} - c_2$. □

In the formula derived above, for lower values of k, the numerator term dominates the denominator. And subsequently after a maxima is reached, the denominator term starts dominating. This makes the count of FIs increase with increase in k initially up to a certain value and then later decline. This can be observed by results presented in Section 4.7.0.5 on page 97 (Table 4.5 on page 99) that compare our estimates with the actual counts. Note that the total number of nodes in the BFI-forest will be much lesser than the total number of itemsets computed using Theorem 4 on page 86 (for all values of k). This is because, we use FP-trees in the BFI-forest beyond the depth = *Max_Height*.

Also, since we know that the total number of SPs stored in the buffers is limited by *Buff_Capacity*, the total space occupied by the buffers in the worst case is proportional to the number of nodes in the forest. However, in practice the nodes at the upper levels of the trees have buffers filled closed to its capacity, and buffers at the lower levels of the trees are less filled. This is substantiated by an experiment presented in Section 4.7.0.5 on page 97 (Table 4.6 on page 99) where the average size of the buffers is shown for each level when the forest is at the steady state. So, the contribution of buffers to space is also

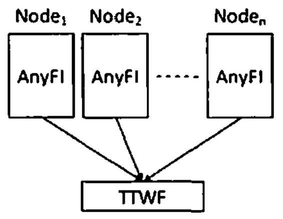


Figure 4.5: The MPAnyFI Framework

considerably lesser than the worst case.

4.6 MPAnyFI: Anytime FI Mining of Multi-Port Data Streams

We extend AnyFI into a parallel framework known as MPAnyFI, for anytime FI mining over multi-port data streams. MPAnyFI works over distributed memory architecture, which is typically a cluster of computing nodes. Each computing node (or a port) receives a stream where transactions are arriving at variable inter-arrival rate. We mine for FIs from all these streams and produce the mining result. Figure 4.5 illustrates the MPAnyFI framework. Every computing node in the cluster runs AnyFI independently for the stream it receives, and captures the incoming transactions in their respective BFI-forests in batches of t_{in} units of time. At the end of each batch, every computing node captures the stream for the next batch in a fresh BFI-forest. Also, at the end of each batch, every computing node executes the following steps:

- Flush all the buffers in the BFI-forest (Algorithm 4.9 on page 77).
- Prune the forest (Algorithm 4.7 on page 75) to eliminate all infrequent sub-trees. This leaves only ϵ-frequent itemsets.
- Encode all the trees along with the frequency counts stored at each node and send the forest to a master machine. Encoding can be done using any of the tree traversal algorithms like BFS or DFS. We used BFS encoding.

The master machine receives and decodes all the forests received from different computing nodes. Then we do a pair-wise merging of forests received from all the machines resulting into a single BFI-forest for this batch. This merged BFI-forest contains only the ϵ-frequent itemsets received from all the computing nodes in last t_{in} units of time. We now insert this forest into the tilted-time window framework [98] and update it.

Consider Figure B.1 on page 212 showing the TTWF. Let us start with an empty TTWF. Say a new batch of transactions has arrived and we have built the merged BFI-forest (say F_1). We store F_1 in w_1. Now lets say another batch of transactions arrive (resulted in forest F_2). At this time, we move F_1 to w_2 and put F_2 in w_1. Now after another batch of transactions arrives (forest F_3 arrives), we shift F_1 to the temporary window present in level 2 (tw_2), shift F_2 to w_2 and insert F_3 in w_1. This state is depicted in Figure B.2 on page 213. When the next batch arrives (F_4), we do the following steps:

- Merge forests in w_2 and tw_2 (forests F_1 and F_2).
- Decay the frequency counts of itemsets in the merged forest using the decay factor - f_{tt} ($0 < f_{tt} \leq 1$) similar to using Equation (4.1) on page 65. (Note that f_{tt} is different from f used by AnyFI.)
- Prune to the forest using DFS similar to Algorithm 4.7 on page 75 to eliminate ϵ-infrequent itemsets.
- Move the merged forest into w_3 as shown in the figure.
- Move F_3 to w_2 and insert F_4 into w_1.

Going this way, after receiving another 4 batches of transactions, the forest in w_3 ($F_1 + F_2$) will be placed in tw_3. And after another 4 batches, the forests in w_3 and tw_3 will be merged and stored in w_4 as was done previously. And in this way the TTWF grows logarithmically.

A user can request for mining results (containing FIs) from a specific window or for a duration covered by multiple windows. For this, we traverse the trees of BFI-forests present in these windows, accumulate the FIs (as was done in Algorithm 4.10 on page 78) and return them as the mining result.

Please note that since we use f_{tt} to decay the frequency counts of itemsets stored in BFI-forests in TTWF, it eliminates the need to separately decay them in individual computing nodes using the parameter f. Capturing transactions batch-wise in fresh BFI-forests also eliminates the need for intermittent pruning in each computing node.

It can be observed that MPAnyFI addresses only the first characteristic of an anytime mining algorithm, i.e. it handles varying inter-arrival rate of transactions. Since, MPAnyFI involves communication between the computing nodes of the cluster, and also the mining of multiple forests in the TTWF, it is not feasible to generate an immediate mining result in shorter time allowances. Hence, it doesn't cater to the second characteristic of an anytime algorithm.

4.7 Experimental Results & Analysis

All experiments were performed on a Linux workstation with an i7 processor & 32 GM RAM. All programs are implemented in C. We used both synthetic and real datasets for experimentation. The synthetic datasets are generated using IBM Synthetic Data Generator [190]. The nomenclature of the synthetic datasets is as follows: *1MD1000T10I4* represents a dataset that has *1M* transactions, drawn from a dictionary of 1000 unique items (*D1000*), with average transaction length of 10 (*T10*) and average FI length of 4 (*I4*). The details of the real datasets are given in Table 4.2. The *Retail* dataset [191] contains *market basket data* from a Belgian retail store. MSNBC [192] is a *click stream dataset* describing page visits on *msnbc.com*. We evaluate the quality of results produced using *precision* and *recall*.

Table 4.2: Details of Real Datasets used for experimentation

Dataset	#Transactions	# Unique Items	Average Transaction Length
Retail	88162	16470	10.3
MSNBC	989818	17	1.71

We evaluate the quality of results produced using *precision* and *recall*. For details on these quality measures, please refer to appendix C on page 214.

The ANY-MINE algorithm can mine FIs from the BFI-Forest in two modes: *False Negative* mode (FAN) and *False Positive* mode (FAP). In FAN mode, we mine with support σ, where we get precision = 1, i.e., the result will not have any itemset which is not σ-frequent. However, we may miss some of the σ-FIs leading to reduction in recall. In FAP mode, we mine with support = ϵ, where we should ideally get results with recall = 1 and compromised precision. In our anytime algorithm, at very high speeds we may lose some suffix projections from the buffers, whenever they overflow. As a result, we may not always get recall of 1. In all our experiments, we choose FAN mode, where precision is 1

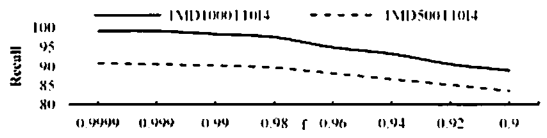


Figure 4.6: Effect of decrease in f on recall

and study the effect on recall with variation in various parameters.

To simulate a stream with varying inter-arrival rates, we use Poisson streams, which is a stochastic model used to model random arrivals [193]. It takes in a parameter λ which controls the speed of the stream. For, $\lambda = \frac{1}{x}$, the model generates an expected number of λ transactions (or objects) per second , with expected inter-arrival rate of x sec. between any two consecutive transactions (or objects).

The values of the other parameters chosen for experimentation are: Max_Height = 5, θ=0.05, γ=2, $batch\_size$ = 10000, $buffCapacity$ = 100 and $hash\_size$ is 10% of $|I|$. These are default, unless explicitly stated.

Since, AnyFI uses a damping factor f over the frequency counts of the itemsets to differentiate the contribution of old and newly arrived transactions, the recall of the output computed by AnyFI, with respect to non-decayed ground truth will always be lesser and will show a decreasing pattern with decrease in f. Figure 4.6 illustrates this effect for *1MD1000T10I4* and *1MD500T10I4* datasets with θ=0.001, λ=20k, ϵ=0.005 and σ=0.01. We can clearly observe that recall has reduced for both the datasets with decrease in f. So, for fairness, we generate the ground truth using the transactions whose frequency counts are decayed, and then compare our results with it.

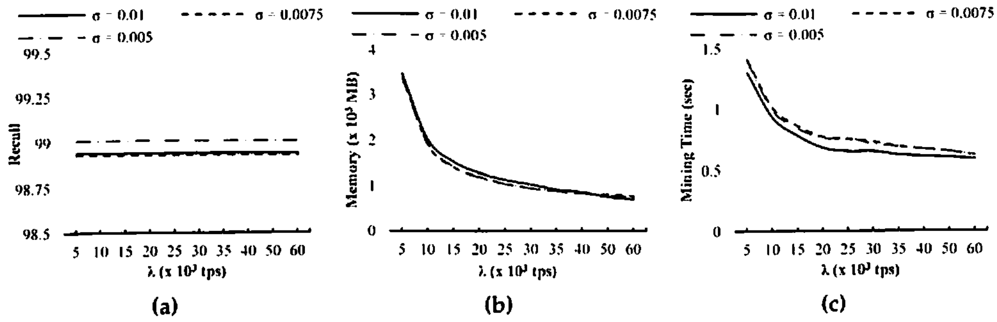


Figure 4.7: Effect of varying stream speed (λ) and σ on (a) Recall (b) Peak Memory Consumption (c) Mining time for 1MD1000T10I4 dataset

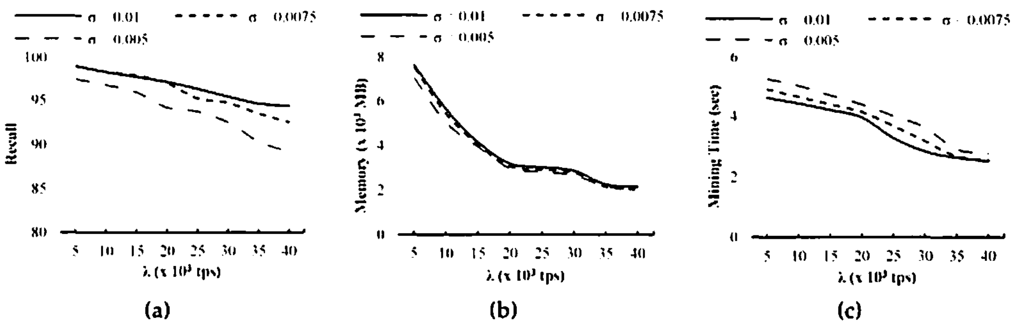


Figure 4.8: Effect of varying stream speed (λ) and σ on (a) Recall (b) Peak Memory Consumption (c) Mining time for 1MD1000T15I8 dataset

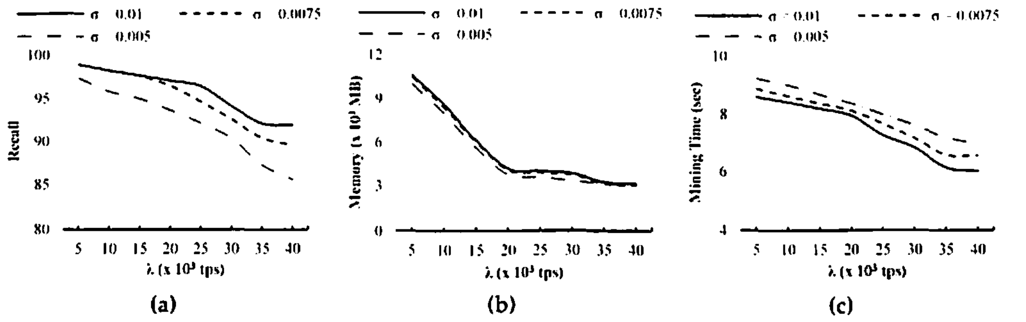


Figure 4.9: Effect of varying stream speed (λ) and σ on (a) Recall (b) Peak Memory Consumption (c) Mining time for 1MD1000T20I15 dataset

4.7.0.1 Performance of AnyFI

In this subsection, we evaluate the *quality of mining results* produced by AnyFI along with the behavior of *peak memory consumption* and *mining time*, with variation in various parameters. In all experiments, the Any-Mine algorithm has been run without any interruption (i.e., without exercising the anytime property), unless explicitly stated, to evaluate the quality of mining results with respect to various parameters.

In the first experiment, we analyze the effect of varying stream speed (λ) on recall(%), peak memory consumption & mining time, with different support thresholds (σ), on the datasets - 1MD1000T10I4 (Figure 4.7 on page 91) , 1MD1000T15I8 (Figure 4.8) and 1MD1000T20I15 (Figure 4.9), with ϵ=0.005 and f=0.99. The results observed for 1MD1000T10I4 dataset show that recall (close to 100%) is not affected by variation in stream speed. However for the other two datasets, recall has shown slight decline with increase in stream speed and decrease in σ. This is because at higher stream speeds, pro-

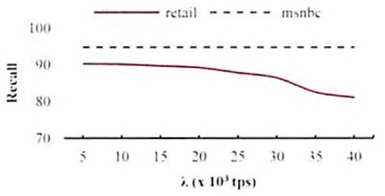


Figure 4.10: Effect of increasing stream speed on recall for real datasets

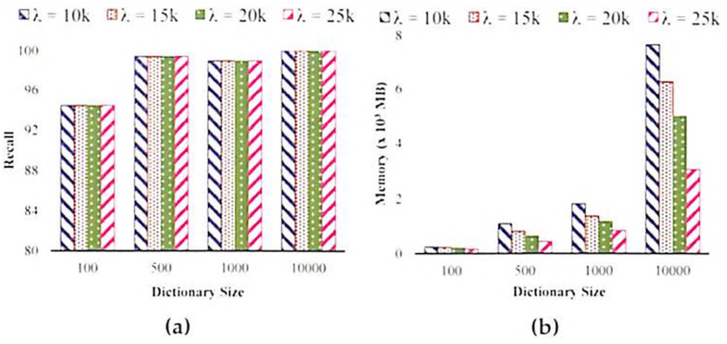


Figure 4.11: Effect of varying stream speed (λ) and varying dictionary size on (a) Recall (b) Memory for 1MD*T10I4 datasets

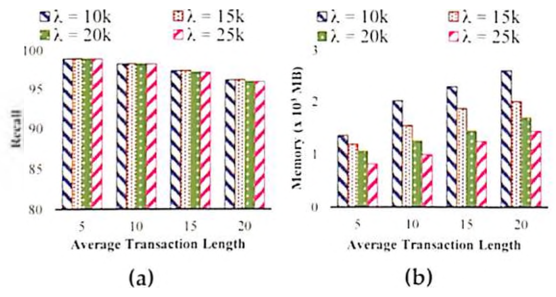


Figure 4.12: Effect of varying stream speed (λ) and average transaction length on (a) Recall (b) Memory for 1MD1000T*I4 datasets

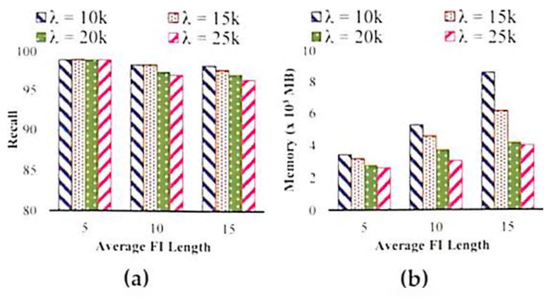


Figure 4.13: Effect of varying stream speed (λ) and average Frequent Itemset length on (a) Recall (b) Memory for 1MD1000T20I* datasets

jections are removed from the buffers when they become full, which leads to reduction in recall. The peak memory consumption for all three datasets, has reduced with increase in stream speed, at all values of σ. This is because, the algorithm is not able to refine the tree nodes frequently and thus more transactions get buffered at higher speeds which stops the forest to grow large. Mining time has also reduced with increase in stream speed due to reduction in number of nodes visited in the DFS traversal, at high speeds.

Next, we study the behavior of AnyFI on real datasets with increase in stream speeds. Figure 4.10 shows the affect on recall for retail and msnbc datasets, with f=0.99, ϵ=0.002 and σ=0.01. The reduction in recall observed for retail dataset with increase in stream speed, is because of longer transactional length possessed by transactions in it, due to which the percentage of transactions that get sufficient time to get completely processed is reduced.

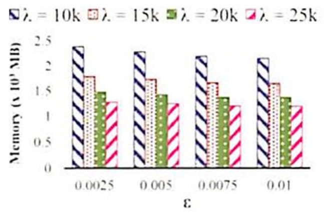


Figure 4.14: Effect of varying ϵ on peak memory consumption

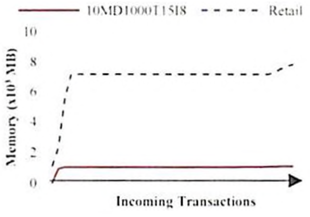


Figure 4.15: Pattern of Memory consumption in stream processing

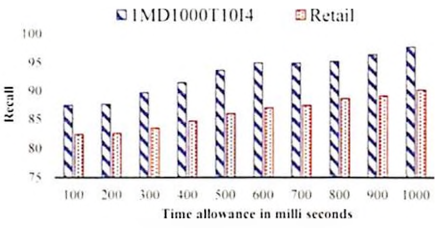


Figure 4.16: Effect on recall with increase in mining time allowance for 1MD1000T10I4 & retail

Next, we study the effect of increase in dimensionality (dictionary size) of the dataset on recall and peak memory consumption (Figure 4.11 on page 93) for different stream speeds on *1MD*T10I4* synthetic datasets, with ϵ=0.005, σ=0.01 and f=0.99. The results show improvement in recall and increase in memory consumption with increase in dimensionality, at all stream speeds. Increase in recall is because, the dataset becomes sparse with growing dimensions and thus the number of FIs in the dataset reduces. Peak memory consumption increases because of increase in number of trees in the forest for higher dimensionality.

Next, we study the effect of variation in average transaction length (ATL) (Figure 4.12 on page 93) and average FI length (AFL) (Figure 4.13 on page 93) on recall & memory, for 1MD1000T*I4 and 1MD1000T20I* datasets respectively, at different stream speeds with f=0.99, ϵ=0.005 and σ=0.01. The results show slight reduction in recall with increase in ATL or AFL at all stream speeds. This is because of the increase in density of dataset in both cases, which leads to increase in processing time for each transaction, which further leads to increase in chance for buffer overflows. Increase in peak memory is attributed to higher number of projections getting enumerated and stored in the forest at larger ATL or AFL.

Next, we study the effect of change in epsilon on peak memory consumption for 1MD1000T10I4 dataset with σ=0.01 (Figure 4.14). The results show that increase in epsilon leads to reduction in peak memory consumption, because at greater epsilon values, larger number of itemsets become infrequent and get pruned.

Next, we study the pattern of memory consumption of AnyFI while processing the stream for 10MD1000T15I8 and retail datasets (Figure 4.15). For synthetic dataset we choose f=0.99, ϵ=0.005 and σ=0.01. For retail dataset we choose f=0.99, ϵ=0.0025 and

Table 4.3: Comparing Speed and Memory of AnyFI with budget algorithms

Algorithm	Category	1MD1000T10I4		retail	
		Speed (tps)	Memory (MB)	Speed (tps)	Memory (MB)
FPStream	Tilted-Time W	27000	99	15000	220
SWP-tree	Sliding Window	12000	530	8200	1020
DSM-FI	Landmark Window	9200	1200	6300	2500
VSW	Sliding Window	900	700	650	1600
AnyFI	Damped Window	upto 60000	600-3400	upto 40000	1900-7000

Table 4.4: Comparision of Mining Time (in seconds)

Algorithm	100KD100T10I4	100KD100T15I8
SWP-Tree/FP-Stream	5.8 sec	20.6 sec
DSM-FI	3.3 sec	18.5 sec
Any-FI	1.2 sec	6.5 sec

σ=0.003. For both datasets, λ=20000 tps. The results show that memory consumption initially increases and later on remains almost the same with slight fluctuations throughout the run until it finishes insertion of all transactions. The increase in memory observed towards the end of the curves is because of emptying of the buffers which happens before the mining step.

4.7.0.2 Anytime Mining of Frequent Itemsets

We study the quality of FI mining with variation in time allowance to mine. Figure 4.16 on page 94 presents the effect on recall with increase in time allowance for mining, which is the second anytime mining feature of our algorithm. We conduct this experiment on 1MD1000T10I4 and retail datasets with λ=20000, f=0.99, ϵ=0.0025 and σ=0.005. The result clearly shows that AnyFI is able to output mining results with compromised accuracy, within a few milli seconds. And then it is able to improve its recall with increase in time allowance to mine, for both the datasets.

4.7.0.3 Comparing AnyFI with existing approaches

In this subsection, we do a comparative study of AnyFI with existing algorithms. First, we compare the speed handling capacity and peak memory consumption of AnyFI with the existing budget algorithms for 1MD1000T10I4 (ϵ=0.005 & σ=0.01) and retail datasets (ϵ=0.002 & σ=0.01). The batch size in all of them is chosen to be 10,000. The results given in Table 4.3 clearly show that these algorithms have limited budget and cannot process higher stream speeds. Whereas our algorithm is able to work for speeds upto 60,000

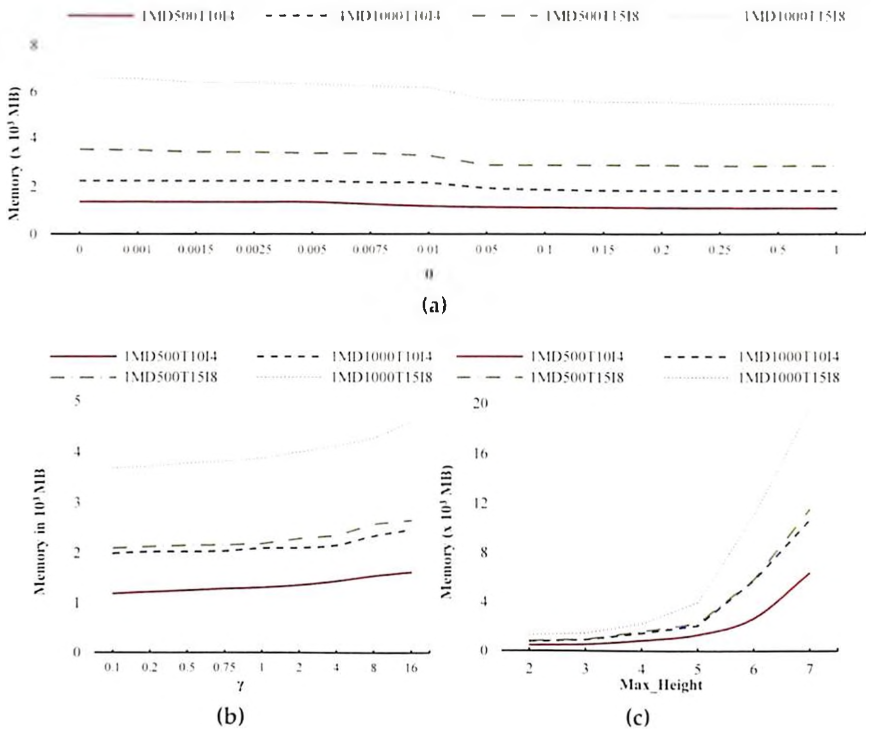


Figure 4.17: Effect on Memory consumption with increase in (a) θ (b) γ (c) *Max_height*

tps for the same datasets. We can also observe from the table that the peak memory requirement for AnyFI is at little higher than the existing approaches. This is because of storing a large number of SPs.

Next, we compare the mining time of FP-growth, DSM-FI and AnyFI for 100KD100T10I4 and 100KD100T15I8 datasets (see Table 4.4 on page 95). For fair comparison, we insert the complete dataset into the summary structures of all the three algorithms without conducting any pruning and then mine for FIs. Mining in AnyFI is running in complete mode, i.e. without anytime interruptible feature. The values of parameters chosen are as stated in the previous experiment. The results show that the mining time in AnyFI is lesser when compared to the existing approaches. It is worth noting that FP-growth is used in SWP-tree and FP-Stream.

4.7.0.4 Parameter Tuning and Recommendations

In this section, we study the effect of various parameters used by AnyFI and give recommendations for choosing their appropriate values. First, we study the affect of varying θ (Figure 4.17a), γ (Figure 4.17b) and *Max_height* (Figure 4.17c) on peak memory consumption, over 1MD500T10I4, 1MD- 1000T10I4, 1MD500T15I8 & 1MD1000T15I8 datasets, with λ=10k, f=0.99, ϵ=0.005 & σ=0.01. The results for θ show that peak memory reduces with increase in θ, for all datasets. This is because, as we delay the refinement of a transaction down the tree, the memory consumption reduces. Similar pattern have been observed for other datasets as well. From the observations made, we recommend to choose θ in [0.05 - 0.2]. The results for γ show that peak memory increases with increase in γ. This is because, as the pruning interval increases (γ dictates the size of pruning interval), the number of obsolete subsets getting accumulated in the BFI-forest also increases. From the observation made, we recommend to choose γ in [0.5 - 2]. The results for Max_Height (level for FP-tree) show that the peak memory grows with increase in Max_Height, because of creation of a large number of branches for storing enumerated SPs at greater depths. We recommend use of $Max\_Height \leq 5$ for datasets of high dictionary sizes and larger average transaction lengths.

Next, we study the affect of varying $hash\_size$ (size of the hash table in a buffer) (Figure 4.18 on the following page) and $buffCapacity$ (max no. of SPs in a buffer) (Figure 4.19 on the next page) on recall and peak memory consumption for 1MD1000T15I8 and retail datasets, with the above parameter values. The results show that increase in $hash\_size$ has no effect on recall. This is because of the algorithm design, which ensures constant time buffer accesses irrespective of its size. However, increase in $buffCapacity$ has led to increase in recall until a point beyond which the increase in not significant. This is because, increase in $buffCapacity$ reduces the chance of a loss of SPs from the buffer. Memory consumption, on the other hand, has increased with increase in both $hash\_size$ and $buffCapacity$ for obvious reasons. So, we can restrict the value of $hash\_size$ to 5% or 10% of I, and $buffCapacity$ to 100, as beyond this there is no significant change observed in recall. Note that similar observations have been made for other speeds and other datasets as well.

4.7.0.5 Experiments establishing Theoretical Analysis

In this section, we present a few experimental results that verify the theoretical analysis presented in Section 4.5.5 on page 80. We first validate the time complexity of insertion and refinement of a m-length transaction derived in Theorem 2 on page 81, with that of

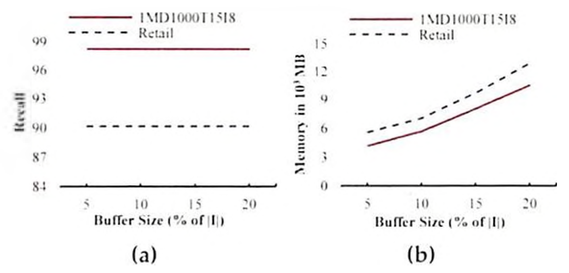


Figure 4.18: Effect on (a) Recall and (b) Memory consumption with increase *hash_size*

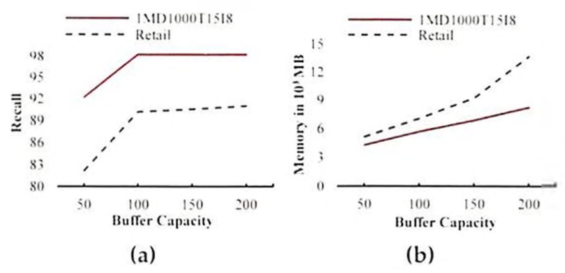


Figure 4.19: Effect on (a) Recall and (b) Memory consumption with increase *buffCapacity*

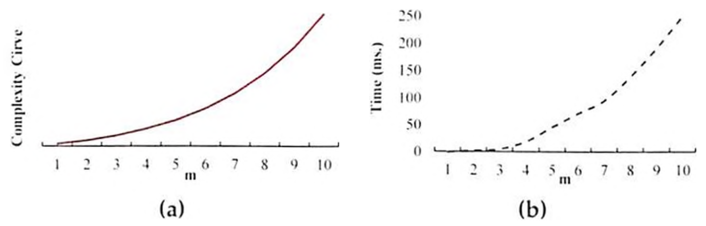


Figure 4.20: Comparing (a) complexity curve vs (b) actual time taken for processing a transaction with increase in transaction length - m

real time values of insertion and refinement, with varying m. AnyFI has been run in non-anytime mode in this experiment. The curve in Figure 4.20a plots the complexity curve derived with c=0.2, for varying m. The curve in Figure 4.20b plots the actual time taken for insertion and refinement of m-length transaction for 1MD1000T10I4 dataset. We can clearly see that both the curves show similar behaviour and hence can justify the correctness of our analysis.

We also verify the counts of k-length FIs estimated by Theorem 4 on page 86, with that of true FI counts produced by FP-growth. c_1 and c_2 of zipf function were chosen by performing curve fitting [194] with that of the original values for each dataset separately using a stream sample. The results were computed for 1MD1000T10I4 & 1MD1000T15I8 datasets at three values of σ - 0.05, 0.01, 0.005. k is varied from 1 to 7. The results presented in Table 4.5 on the next page show that the estimated count of FIs is very close to the actual counts for most values of k.

We also empirically measure the total number of SPs stored in the buffers of the nodes in the forest level by level for 1MD1000T10I4 & retail datasets, with θ=0.001, λ=20, ϵ=0.005,

Table 4.5: FI counts - Estimated vs Actual: with varying *k* for 1MD1000T10I4 and 1MD1000T15I8 datasets

		1MD1000T10I4							1MD1000T15I8						
	k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
s=0.05	**Predicted**	58	82	36	10	5	1	1	86	126	58	29	11	6	1
	Actual	62	98	28	9	4	0	0	92	143	46	25	9	5	0
s=0.01	**Predicted**	124	1765	1265	291	19	3	1	157	2622	5349	2014	243	56	12
	Actual	90	1396	994	269	17	2	0	104	1873	3782	635	72	16	4
s=0.005	**Predicted**	136	3048	9204	941	48	3	1	201	4274	14045	9473	3013	1026	56
	Actual	94	2237	5096	721	52	3	0	158	3256	9305	7284	2487	856	42

Table 4.6: Size of the buffers in the nodes (level wise)

	1MD1000T10I4		Retail	
Level	**Max(%)**	**Avg(%)**	**Max(%)**	**Avg(%)**
1 (root)	100	10.75	100	20.85
2	58	7.03	69	10.58
3	37	1.31	42	5.92
4	10	1.00	33	3.48
5	5	1.00	27	2.16

σ=0.01 and *Max_Height* = 5. The results presented in Table 4.6 show the maximum and average occupancy of buffers of nodes at each level in the forest. They clearly show that the buffers at the top level of the trees are more filled than the buffers at the lower level. However the avg. occupancy remains quite less due to the optimizations made like θ-deferring, buffer pruning & intermittent pruning. Similar behaviour has been observed for other datasets too. This proves the claim made in Section 4.5.5 on page 80 that the contribution of buffers to the space occupied by BFI-forest is considerably less.

4.7.1 Experiments on MPAnyFI

All experiments are conducted on a cluster of 32 computing nodes which are IBM x3250 m4 Servers. Each server has Intel Xeon (64-bit) processor and 32 GB RAM. All implementations are in C with MPI. In all experiments, *f* is chosen to be 1, i.e., the frequency counts are not decayed within the local BFI-forests present in the computing nodes. Rather *f_tt* is used to decay the frequency counts in TTWF as explained in Section 4.6 on page 88. *f_tt* is chosen to be 0.99.

4.7.1.1 Experimental Results

First, we measure the peak memory consumption across any slave computing node as the stream progresses for 960MD100T10I4 dataset. This dataset has been equally divided amongst the computing nodes, where each node gets 30M transactions for 32 computing nodes. The parameters are: ϵ=0.005, σ=0.01, t_{in}=10 sec., *theta*=0.05 and λ=10k and 20k. The peak memory consumption across any slave node for first 20 batches of the stream has been presented in Figure 4.21a. The results show that peak memory consumption is stable and doesn't fluctuate with stream progression. The average deviation in from the mean is 79.97 KB for λ=10k and 100.24 KB for λ=20k.

Next we measure the number of nodes present in the forest of each window in TTWF (stored at the master node), along with the mining time required for mining FIs from each window. We use the same dataset along with same parameter values as above. We conduct this experiment for 32 nodes and all measurements were taken after the entire dataset has been processed. The results presented in Figure 4.21b show that the rate of increase in number of nodes in the forest declines with increase in age of the windows at both the speeds. This is because of use of decay factor f_{tt} and logarithmic nature of TTWF. The mining time of each window also shows similar behavior (Figure 4.21c). This establishes the memory efficiency of MPAnyFI.

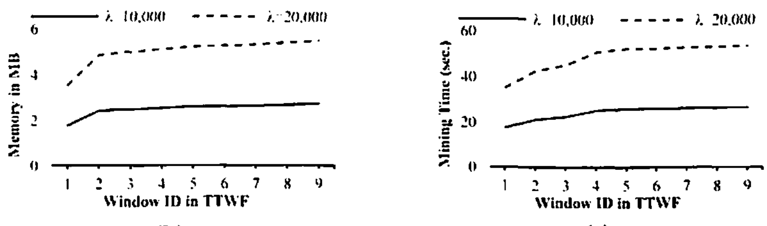


Figure 4.21: (a) Max memory at a computing node with stream progression (b) Memory consumed by windows of TTWF after processing the complete dataset (c) Mining time of each window in TTWF

4.8 Main Contributions

- We present an Anytime Frequent Itemset mining algorithm for data streams, AnyFI, characterized by both the properties of an anytime algorithm. To the best of our knowledge, this is first such attempt.
- We propose a novel data structure known as *Buffered Frequent Itemset Forest* (BFI-forest), which stores buffers at its tree nodes and aids AnyFI to handle variable inter-arrival rate of transactions. Its design also enables a user to obtain immediate mining results.
- We also propose MPAnyFI for *anytime FI mining of multi-port data streams* over commodity clusters. It uses AnyFI at each computing node and stores the aggregate FIs in a *tilted-time window framework*.

Salient Features of proposed work

1. *Fast Mining*. AnyFI inserts all suffix projections of incoming transactions (defined in Section 4.3.1 on page 64) into the BFI-forest, depending upon the available time allowance. As a result, mining the forest for FIs becomes a simple traversal of its trees accumulating FIs without generating any candidate itemsets as in apriori like methods [47, 50, 51], or conditional trees as in FP-growth like methods [53, 184, 98, 97], thus making mining very efficient.
2. *Quick approximate result*. AnyFI can give an immediate approximate mining result with best possible accuracy for the available time allowance, and can improve its quality with increase in time allowance.
3. *Key Concepts Used*. AnyFI uses techniques such as *θ-deferring, buffer-pruning, intermittent pruning* and *usage of small sized FP-trees* within the BFI-Forest, to efficiently manage memory consumption, which otherwise would have been high as we store a large number of enumerated suffix projections. θ-deferring deliberately delays the insertion of infrequent & semi-frequent projections into BFI-forest and thus saves memory and time. Buffer-pruning prunes infrequent projections from buffers whenever time allows. Intermittent pruning prunes infrequent sub-trees from the forest at regular intervals without disturbing the stream. Using FP-trees within the BFI-forest saves unnecessary creation and deletion of sub-trees that store infrequent itemsets.

4. *Efficiency & Performance.* The extensive experimental analysis shows that AnyFI brings in the features of an anytime algorithm. It also establishes that AnyFI can handle greater speed streams upto 60,000 transactions per second (*tps*), with recall close to 100%. The comparative analysis shows that AnyFI handles higher stream speeds and mines for FIs efficiently, when compared to the existing algorithms. Experiments have also been conducted to tune the parameters used by AnyFI (θ, γ) and recommendations are given for choosing their values appropriately for maximizing efficiency. The experiments conducted over MPAnyFI also show its efficiency.

4.9 Conclusions and Future Work

4.9.1 Conclusions

We presented AnyFI which is the first anytime FI mining algorithm for data streams. AnyFI incorporates both the functionalities of an anytime algorithm - *ability to handle variable stream speeds, & ability to give an immediate mining result with compromised accuracy if required and improve its accuracy with increase in time allowance.* AnyFI uses a novel data structure known as BFI-forest, which handles stream of transactions arriving with varying inter-arrival rate. Also, unlike other methods, mining BFI-forest requires a simple traversal of its trees accumulating FIs, making it very efficient. The experimental analysis presented shows that AnyFI can handle variable and high stream speeds while maintaining high recall. We have also extended AnyFI into a parallel framework known as MPAnyFI for anytime FI mining of multi-port streams. This framework uses *Tilted-Time Window Framework* to summarize the entire stream in logarithmic space. The experimental results also establish its efficiency.

4.9.2 Future Directions

In future, we shall extend MPAnyFI to mine for frequent (or top k) co-occurrence patterns from multiple streams.

Chapter 5

Anytime Set-wise Classification of Data Streams

5.1 The Set-wise Classification Problem in Data Streams

Due to increasing utilities of data generating systems such as web, sensor networks, retail chains, etc., classification in data streams has become very popular [146, 99, 148, 147, 14, 149]. These algorithms build a classification model on the initial training corpus, which is used to classify the test objects arriving in the stream. These algorithms allow only a single pass for classifying each test object.

In many stream applications such as community detection from text feeds, website fingerprinting attack, retail chain analysis, etc., the classification labels are not associated with individual data objects, but with groups of objects. Each group is treated as an indivisible *entity* with an associated class label. And, a class label can be meaningfully assigned to an entity only by studying the overall distribution pattern of objects in it, rather than studying a single object. Consider Figure 5.1 on the next page[2]. It has two kinds

- J. S. Challa, P. Goyal, V. M. Giri, D. Mantri and N. Goyal. *AnySC: Anytime Set-wise Classification of Variable Speed Data Streams.* In Proceedings of 2018 IEEE International Conference on Big Data (Big Data 2018), pages 967-974, IEEE Press, 2018

[2]borrowed from [195]

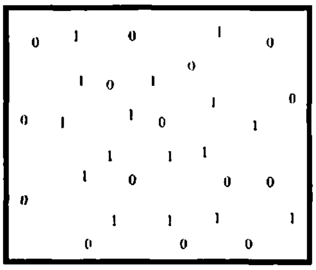


Figure 5.1: Illustrating Set-wise Classification Problem

of data objects - *0*s and *1*s. At a first look, there doesn't seem to be any difference in the distribution of *0*s and *1*s as they are interleaved and no proper boundary exists. However, if we carefully observe, we can see that *0*s are more concentrated near the boundaries and *1*s are more concentrated in the interior region. By observing such distribution patterns, we can meaningfully assign a class label to the set of all *0*s or set of all *1*s. This kind of problem is known as the *Set-wise Classification problem* [195] and can be found in the applications described below:

5.1.1 Applications

Community Detection using text feeds. Community detection allows us to find groups of users who have common interests [196]. These groups can be used for targeted advertising on social networks or viral advertising campaigns. There can be multiple users belonging to various communities creating text feeds on a social networking website such as twitter. Using these feeds (or tweets), we can predict the community to which each user belongs to. This problem can be viewed as a set-wise classification problem. Each tweet can be considered as a data object, each user can be considered as an entity (associated with tweets tweeted by him), and a community class label can be associated with each user, with each class having multiple users associated. We can clearly observe that based on a single tweet, we may not be able to meaningfully assign a class label to a user. However, by studying the pattern formed by a set of tweets by a user, we can study his behavior and appropriately assign a class label. In this problem, we can first construct an initial classification model over a given sample of tweets from various users (entities) with known community class labels. Then in the stream we will receive and process real-time tweets that are being posted by various users. The stream can receive tweets from both labeled and unlabeled users. The tweets from labelled users (training entities) can be used to incrementally update the classification model. The tweets from unlabelled users can be used to construct the test entities (unlabelled users), whose class labels can be predicted using the above model. As more tweets belonging to a user arrive, the prediction becomes more accurate.

Website Fingerprinting attack. It is a Trac Analysis Attack, where network attackers try to breach web navigation security and privacy [197, 198]. Certain web users accessing web pages use anonymous communication mechanisms to hide the content and meta data exchanged between the browser and the server hosting the web page using methods like Tor network [199]. An Attacker can use ML techniques to identify the web page accessed by a user by capturing the network packets secretly (even encrypted ones). Such attackers can target individuals, businesses and governments. In order to prevent them, researchers study different attack schemes and provide counter measures. This attack can be modeled as set-wise classification problem over a stream of network packets captured when various users are accessing the web. Given a set of web pages that are being accessed by the users, each web page shall have different network traces associated where each trace consists of uplink and downlink packets generated when a user loads the web page. Each packet contains information like time, direction and length in bytes. We can consider a group of consecutive packets (known as a *burst*) going in a specific direction as a data object. Each burst is characterized by *burst length* and *direction*, which are the features used by the attackers [197]. The set of packets (bursts) exchanged (uplink and downlink) between the user and the server for loading a complete web page forms a *trace*, which can be considered as an entity. Each trace can be associated with a class label (web page label). So, in this problem, an individual packet (or a burst) may not be associated with a class label since similar packets can occur over multiple web pages. Instead, a class label is associated with a set of packets (or set of bursts), which forms a trace (entity). We can build an initial classification model over a given sample of packets from various traces with known class labels. In the stream, the packets received from labeled traces can be used to incrementally update the training model, and packets from unlabeled traces can be used to construct test entities whose class label can be predicted using the classification model.

To the best of our knowledge, only a couple of approaches have been proposed for set-wise classification for data streams. The approach in [195] formally defines this problem and presents a classification model & method for classifying test entities. Its classification model consists of sets of class profiles (one set for each class). A class profile is an object that characterizes the average distribution pattern of a set of entities (see Section 5.2). A test entity is matched with all class profiles (from all classes) to find the closest, whose class label is assigned to the test entity. The second approach [198] extends the previous by using an ensemble of classifiers such as nearest neighbors, bayesian classifier, etc.

The rest of the chapter is organized as follows: Section 5.2 presents the set-wise classification model for data streams proposed in [195]; Section 5.3 on page 111 presents the proposed approach - AnySC; Section 5.4 on page 117 presents the experimental results and analysis; Section 5.5 on page 122 discusses how AnySC addresses the limitations of existing models; Section 5.5.1 on page 124 highlights the main contributions of this chapter; Section 5.6 on page 124 concludes this chapter and gives recommendations for future work.

5.2 Background: The Set-wise Classification Algorithm for Data Streams (SC)

In this section, we describe the set-wise classification algorithm for data streams (SC) proposed in [195].

Let there be N training entities (labeled) in the entire corpus denoted by $\mathcal{E}_1 ... \mathcal{E}_N$, where each entity $\mathcal{E}_i$ has e_i data objects in it. Let there be a total of c different classes, with associated class labels $[1...c]$. Let d be the dimensionality of the dataset. Each object in an entity is a d-dimensional vector. Similarly, let there be a set of n test (unlabeled) entities $(\mathcal{T}_1 ... \mathcal{T}_n)$. Let the data objects in the stream be received in the form of tuples - $< Y_1, entityid_1, label_1 > ... < Y_r, entityid_r, label_r > ...$ and so on. Y_r is a d-dimensional object which could either belong to a training entity or to a test entity; $entityid_r$ is the *id* of the entity to which the object belongs to; and $label_r$ is its class label. If Y_r belongs to a training entity, its class label will be in the range $[1...c]$, and if it belongs to a test

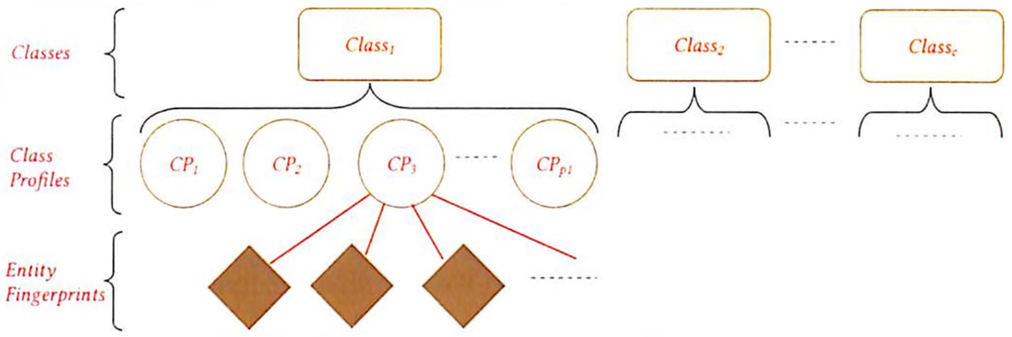


Figure 5.2: The Set-wise Classification model of *SC*

entity, its class label will be -1. So, the stream receives a mixture of objects belonging to training and test entities. Note that training and test entities are disjoint. *So, given a set of training entities $\mathcal{E}_1...\mathcal{E}_N$, the problem of set-wise classification in data streams is: constructing a classification model using a sample of objects from training entities, using which we can classify test entities whose objects arrive in the stream.*

5.2.1 The Set-wise Classification model

The classification model for *SC*, is constructed over an initial sample of data objects from training entities and is then incrementally updated using the training objects received in the stream. The model consists of: *c classes*, *p class profiles* and *N training entities* ($c << p << N$), as shown in Figure 5.2. Each class contains a subset of class profiles and each class profile is an aggregate of a subset of training entities. It is explained as follows:

An entity (either training or test) is represented by a q-dimensional vector known as *fingerprint*, which contains the distribution pattern of objects belonging to it. Initially, a set of q data objects known as *anchor points* (denoted by $W_1...W_q$) are selected from the initial training sample, k-means clustering (with $k = q$) is performed over it and the final centroids are chosen as the anchors. Note that q is an input parameter. The distribution pattern of objects for any given entity is captured as a frequency vector around these q anchors forming a histogram like structure, which is the fingerprint. The value of q dictates the granularity of the fingerprint. It is defined as follows:

Definition 5.1. (FINGERPRINT). Let there be r objects denoted as $Y_1...Y_r$, in a given entity $\mathcal{E}_j$. Let these objects be assigned to their respective closest anchor points (one in $W_1...W_q$), resulting in a partitioning consisting of q clusters $C_1...C_q$. Let the corresponding (relative) cluster frequencies be denoted as $f_1 ... f_q$, where $\sum_{i=1}^{q} f_i = 1$. Then, the *fingerprint* of $\mathcal{E}_j$, defined with respect to the above q anchors, is denoted by the q-dimensional vector $[f_1 ... f_q]$.

Given a set of data objects $Y_1 ... Y_r$ belonging to an entity $\mathcal{E}_j$, the fingerprint of $\mathcal{E}_j$ by constructed by incrementally adding each object to it. For each Y_i, we first identify the closest anchor point using *euclidean distance* as the distance measure. Say the index of selected anchor point is *ind* in the fingerprint vector. Let the number of objects updated in the fingerprint of $\mathcal{E}_j$ before adding Y_i be $|U|$. Then the new fingerprint vector after adding Y_i to $\mathcal{E}_j$ can be computed as:

$$f_i' = \begin{cases} \frac{f_i \cdot |U|}{|U+1|} & i \neq ind \\ \frac{f_i \cdot |U| + 1}{|U+1|} & i = ind \end{cases} \tag{5.1}$$

After constructing the fingerprints for the training entities in the initial data sample, the class profiles are created over them. A *class profile* contains the average fingerprint characterization for a given set of entities. It is defined as:

Definition 5.2. (Class Profile). Given a set of entities $S = \mathcal{E}_1 ... \mathcal{E}_s$ belonging to one class with label l and associated q-dimensional fingerprints $L_1 ... L_s$. A class profile is a tuple of: (i) A q-dimensional vector AG_S containing the sum of the fingerprints of all the entities $\in S$ across all dimensions, i.e. $AG_S[p] = \sum_{i=1}^{s} L_i$ for each dimension $p \in [0 ... q-1]$. (ii) An integer value s storing the number of entities aggregated in this class profile. (iii) A class label l.

Class profiles are used for predicting class labels of test entities. There can be multiple class profiles associated with each class and to construct them, k-means is applied over the entities belonging to a given class, and one class profile is constructed for each cluster by aggregating the entities belonging to it. Let p_i be the number of class profiles for the class with label i. The total number of class profiles in the model will be $p = \sum_{i=1}^{c} p_i$. Thus, the SC's model comprises a set p of class profiles with associated class labels.

5.2.2 Processing Incoming Stream Objects

The stream receives a mixture of objects belonging to labeled and unlabeled entities. The objects from labeled (training) entities are used to incrementally update the classification model. The objects from unlabeled (test) entities are used to construct the test entities, which are assigned appropriate class labels using the classification model.

For incoming object $Y_i \in$ training entity $\mathcal{E}_j$, the following steps are executed to incrementally update the classification model: (*i*) Identify the closest anchor point. (*ii*) Update the fingerprint of $\mathcal{E}_j$ using Equation (5.1) on page 108. (*iii*) Determine the closest class profile within the same class and re-assign $\mathcal{E}_j$ to it if necessary, and update the class profile statistics.

Since, class profiles are simple aggregate vectors, they posses additivity property by which we can add (or subtract) an entity fingerprint vector to (or from) it. This property is used to update the class profile statistics, wherein we subtract the old fingerprint vector from it and add the newly updated one. If the incoming object $Y_i \in$ a test entity $\mathcal{T}_j$, the first two steps are the same as the above. In the third step, a class label is assigned to $\mathcal{T}_j$ if it was not assigned previously, or $\mathcal{T}_j$ is re-assigned to the most appropriate class (either same or a different class). Before a class label is assigned to it, the algorithm lets the test entity accumulate at least *min_stat* objects in it. This allows the test entity to accumulate sufficient number of objects into it before it can be meaningfully assigned to a class. As the number of objects being accumulated to an entity increases, its class label prediction becomes more accurate, since its objects' distribution pattern shall be known more accurately. So, to assign a class label to $\mathcal{T}_j$, all p class profiles (belonging to all c classes) are scanned to identify the closest, whose class label is assigned to $\mathcal{T}_j$. Cosine similarity is used to measure distance between a fingerprint and a class profile, both of which are q-dimensional vectors.

The model described above has certain limitations and cannot be used over variable stream speeds (as explained in Section 5.2.3 on the next page). It's limitations and how they have been addressed by AnySC, is described in Section 5.5 on page 122.

5.2.3 Research Gap & Motivation

Limitations of Budget Approaches

- The *SC* model is a budget algorithm. It takes a fixed duration of time (*budget*) for assigning class label to each test entity. However, real-time streams do not have constant speed and thus cannot guarantee availability of such budget for every object received. For example, in community detection using text feeds, the rate of arrival of tweets is not fixed. It can vary depending upon many factors. Similarly, in website fingerprinting attack, the number of packets exchanged in the network is high during peak hours and low during other times.
- The classification model of *SC* pose a flat linear structure (list of class profiles) that can not handle *variable inter-arrival rate* of objects, which is a key requirement in the above applications. When the stream speed is lesser than their budget, they successfully process the incoming objects, and then sit idle for the remaining time allowance without any effort to improve the accuracy. And when the stream speed is higher than the budget, they fail to execute! An ideal algorithm, however, should be able to process any stream speed, giving the best possible result within any given time allowance, even if it is approximate, and refine its accuracy with increase in time allowance. Such an algorithm is known as an *anytime algorithm*, as explained in Part II on page 57. *SC* is a budget algorithms and lacks such capabilities.

A few algorithms were proposed in literature for anytime classification in data streams based on nearest-neighbors [100] and Bayesian Classifiers [101, 175, 116]. The first proposed approach [100] was based on nearest-neighbors. This approach orders the training data objects into a specific order so as to minimize the error while classifying the objects arriving in the stream with variable rate. Anytime Bayesian Classifier has been proposed in [101], which uses a hierarchical indexing structure known as *Bayes Tree* to insert data objects at hierarchical granularities. MC-tree [175] and BT* [116] are its extensions that are more accurate classifiers than Bayes Tree. All the three trees are based on R-tree structure, storing guassian kernels and gaussian mixture models in external and internal nodes respectively. However, none of them were designed for set-wise classification of data streams.

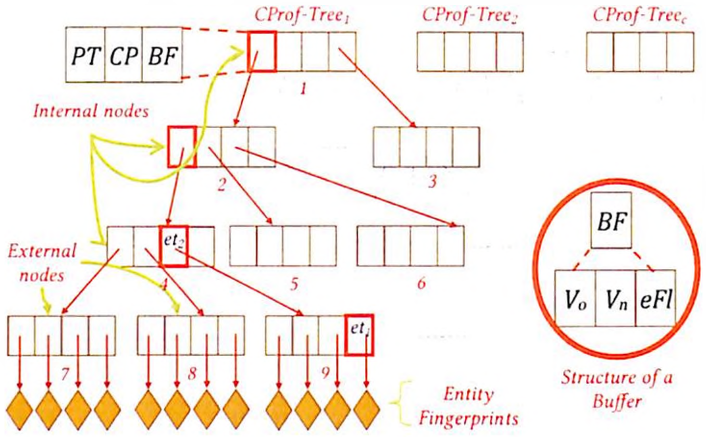


Figure 5.3: CProf-forest

5.3 Anytime Set-wise Classification Model - AnySC: The Proposed Method

We now present our proposed method - AnySC, which is the first anytime set-wise classification algorithm for data streams. We first present the data structure - *CProf-forest* (Figure 5.3), which serves as the classification model for AnySC. It consists of *CProf-trees* whose total number is equal to the number of classes (c). CProf-trees store hierarchy of class profile vectors in their internal nodes. We define CProf-tree as follows:

Definition 5.3. CProf-tree. It is a height balanced multi-dimensional indexing structure (see Figure 5.3) having the following properties:

- All nodes (both internal and external) contain between m and M entries. The root has at least one entry. Figure 5.3 shows a CProf-tree with $m = 2$ and $M = 4$.
- All nodes store a pointer *Parent* which points to the parent entry.
- An entry e in an external node stores the q-dimensional fingerprint vector representing an entity.
- An entry e in an internal node stores: (i) a pointer PT_e pointing to the root of sub-tree beneath e; (ii) a class profile CP_e storing the aggregate of all the entity fingerprints indexed at its child node, which is the root of the subtree indexed at e; (iii) a *buffer* BF_e, which consists of 2 q-dimensional vectors - V_o & V_n, and a flag *eFl* which is set to TRUE if buffer is empty. The two vectors of BF_e is used for

deferred updation of CProf-tree that occurs while processing high speed streams (see Section 5.3.1 on the next page).

Definition 5.4. CPROF-FOREST. It is collection of c CProf-trees, one for each class.

The structure of CProf-tree results in an hierarchy of class profiles from coarser to finer granularity as we descend down the tree from root to the leaf level. This enables us to represent the distribution patterns of the objects belonging to a class in a hierarchical fashion. The root of each tree summarizes the distribution pattern of all the entities belonging to a class at coarsest granularity. And, as we keep going down the levels in the tree, the granularity of the class profiles (stored at internal nodes) becomes finer and finer, until we reach the leaf nodes containing entity fingerprints, which are of finest granularity. Consider the CProf-tree in Figure 5.3 on page 111. We can arrange the nodes in increasing level of granularity as follows: $1 < 2,3 < 4,5,6 < 7,8,9$. This model of hierarchical granularities helps in classification of test entities within any given time allowance dictated by variable inter-arrival rate of objects (see Section 5.3.2 on page 114).

A CProf-tree for a given class is constructed by dynamic insertion (of logarithmic cost) of training entities belonging to it, in a top-down recursive fashion similar to that of an R-tree. The only difference is that the distance metric is computed between the entity fingerprint vector and class profile vectors stored at the internal nodes (using cosine similarity metric), rather than with the Minimum Bounding Rectangles as in R-trees (refer [122] for more details). The node overflows are handled in a similar manner as that of an R-tree and the node splits propagate upwards leading to CProf-tree growing upwards. The vectors in the buffers of CProf-trees are left empty during the creation of the initial CProf-forest in the training phase.

Definition 5.5. ANYTIME SET-WISE CLASSIFICATION IN DATA STREAMS. Given a set of training (labeled) entities $\mathcal{E}_1 ... \mathcal{E}_N$, with associated labels l_1 l_2 ... l_N, we construct a CProf-forest over an initial sample of objects $\in$ training entities, which serves as the classification model for classifying test entities $\mathcal{T}_1$ $\mathcal{T}_2$ ... whose objects are arriving in the stream with variable inter-arrival rate.

5.3.1 Anytime Incremental Model Update

The classification model is incrementally updated using training objects received in the stream, within the variable time allowance dictated by variable inter-arrival rate of objects. It is illustrated with Figure 5.3 on page 111 & Algorithm 5.1 on page 115 and explained as follows:

Let the arriving training object be $< Y_r, entityid_r, label_r >$. Firstly, we need to update the fingerprint of the entity to which Y_r belongs to (Algorithm 5.1 on page 115: line 3), which is the entity with id = $entityid_r$. This entity will be indexed in a leaf node of the CProf-tree with class label - $label_r$. To do this update, we first identify the nearest anchor point among q anchors and update the fingerprint using Equation (5.1) on page 108. This operation can be done in $\mathcal{O}(1)$ time by additionally indexing all the entities in a separate array with their respective ids as the rank. After this, we need to perform a bottom-up update of the tree in which the above entity is present, in order to accommodate the change that has just happened. We need to update the class profile aggregates along the path from the current entity to the root node. For this, we make use of the additivity property of the vector AG (stored in class profiles), where at each class profile in this bottom-up path, we subtract the old aggregate of the entities present in the sub-tree rooted at it and add the new aggregate (lines 13-25). For example, consider Figure 5.3 on page 111. Let the entity which just got updated be stored at entry et_1 of leaf node 9. The entry et_2 at node 4 contains a class profile that stores the aggregate of all the fingerprints indexed at node 9. To accommodate this change, we update the class profile indexed at et_2, by subtracting the old fingerprint vector of entity at et_1 from it (*vect_old*), and adding the newly updated fingerprint to it (*vect_new*). We then proceed for subsequent iteration and in this way the update propagates up to the root along the path consisting of entries highlighted in orange in the figure. However, if during the bottom-up update, the time allowance for processing this object expires (triggered typically due to arrival of a new object in the stream), we *defer* its completion. For this, we use V_o and V_n vectors stored in the buffers for temporarily storing the old and new aggregate vectors, respectively (lines 7-12). Say, the time allowance has expired before the class profile indexed at et_2 got updated, we simply leave the old and new fingerprint vectors of entity indexed at et_1 in the buffer of its parent entry et_2, and exit to process the newly arrived object. Next time when a new update triggered by the insertion of another data object passes through et_2, the vectors in its buffer are taken out and processed along with (lines 13-25). The class profile at et_2 will be updated with respect to the old and new vectors stored in the buffer in a similar manner explained above (lines 17-21), along with the recent old and new vectors (lines 16 & 22). And then the combined update propagates upwards in subsequent iterations. This way the deferred update triggered by an old object can reach the root and get eventually completed. Note: In this algorithm, the update of a class profile is a *unit operation*. Unit operation is an atomic non-interruptible operation, i.e., we will process the next object only after this operation completes, even if the next object arrives before it finishes.

In the above algorithm, it is possible that multiple vectors, resultant of update of different entities, get accumulated in the same buffer. However, owing to the additive nature of these vectors, the algorithm's correctness is not affected and the class profiles in the tree remain consistent. The above bottom-up update is of *logarithmic order* of the number of entities indexed in the tree.

Also, note that the buffer vectors store aggregates of multiple vectors, eliminating the need to keep multiple objects in the buffers like in BFI-Forest explained in the AnyFI chapter. In AnyFI buffers were hash tables storing multiple suffix projections. Here buffers store aggregates of multiple vectors.

5.3.2 Anytime Classification of test entities

Let the arrived test object be $< Y_r, entityid_r, -1 >$. First we update the fingerprint of the entity $\mathcal{T}_{entityid_r}$ with respect to Y_r as was done in case of training objects. If $\mathcal{T}_{entityid_r}$ doesn't exist in this test set, we first create it and then update. Once $\mathcal{T}_{entityid_r}$ contains at least *min_stat* objects updated in its fingerprint, we can meaningfully assign it to a class. The process of classifying a test entity is illustrated in Algorithm 5.2 on page 116. Essentially, we do a *best-first* traversal of the CProf-trees using c number of priority queues, one for each class. These traversals can be interrupted anytime, i.e. whenever a new object arrives in the stream we can assign a class label to the current test entity on the basis of class profiles of different trees visited until now, and then exit to process the newly arrived

Algorithm 5.1: INSERT AND UPDATE IN CPROF-TREE

```
1  procedure INSERT-AND-UPDATE-IN-CPROF-TREE()
       Input  : A CProf-tree $CPT_i$, a training object: $Y_r, entityid_r, label_r$
       Output: $Y_r$ inserted into $CPT_i$ and $CPT_i$ updated bottom-up until time allows
2      $vect\_old = \mathcal{E}_{entityid_r}.fingerprint$;
3      Update $\mathcal{E}_{entityid_r}.fingerprint$ with respect to $Y_r$;
4      $vect\_new = \mathcal{E}_{entityid_r}.fingerprint$;
5      $curr\_entry$ = ENTRY_OF($\mathcal{E}_{entityid_r}$);
6      while NODE_OF($curr\_entry$) != $root$ do
7          if $NEW\_OBJECT\_ARRIVED$ then
8              foreach $dimension$ $p \in [0...q]$ do
9                  NODE_OF($curr\_entry$).$parent.BF.V_n$[p] += $vect\_new$[p];
10                 NODE_OF($curr\_entry$).$parent.BF.V_o$[p] += $vect\_old$[p];
11             end
12             NODE_OF($curr\_entry$).$parent.BF.eFl$ = FALSE;
13             return
14         $curr\_entry$ = NODE_OF($curr\_entry$).$parent$;
15         foreach $dimension$ $p \in [0...q]$ do
16             $temp = curr\_entry.CP$[p];
17             $curr\_entry.CP.AG$[p] -= $vect\_old$[p];
18             if $curr\_entry.BF.eFl == FALSE$ then
19                 $curr\_entry.CP.AG$[p] -= $curr\_entry.BF.V_o$[p];
20                 $curr\_entry.CP.AG$[p] += $curr\_entry.BF.V_n$[p];
21                 $curr\_entry.BF.V_o$[p] = 0;
22                 $curr\_entry.BF.V_n$[p] = 0;
23             $curr\_entry.CP.AG$[p] += $vect\_new$[p];
24             $vect\_old$[p] = $temp$;
25             $vect\_new$[p] = $curr\_entry.CP.AG$[p];
26         end
27         $curr\_entry.BF.eFl$ = TRUE;
28     end
```

object. The class label assigned should be the most accurate for the given processing time allowance. For this, we shall have to compare the current test entity with respect to class profiles/entities belonging to all the classes and then assign the class label of the closest class profile/entity (using cosine similarity as distance measure). However, we may not have sufficient time to scan all the class profiles/entities of all classes. So, we start our comparisons with the class profiles stored in the root node of all the trees. This is because the entries in the root node summarize all the entities of a given class. So, we add all the entries of the roots to their respective priority queues (lines 2-5). And we iteratively refine our search space by removing the closest entry to the test object from each priority queue, and add its child entries into the same priority queue (lines 7-10). This step increases the granularity of the search space, because the class profiles stored in the entries which got just added to the priority queues are of finer granularity than those that were removed. In this way we iteratively refine our search space in every tree in best-first manner, until a point of time when the priority queues are only filled with leaf entries that index entities. However, our processing time allowance may not

Algorithm 5.2: CLASSIFYING A TEST ENTITY

```
1  procedure CLASSIFY-TEST-ENTITY()
       Input  : A CProf-forest $Forest$, a test entity $\mathcal{T}_r$
       Output: Class label assigned to $\mathcal{T}_r$
2      Initialize $c$ Priority Queues $PQ_1$ ... $PQ_c$;
3      foreach $CProf\text{-}tree\ CPT_i\ of\ Forest$ do
4          foreach $entry\ e\ in\ CPT_i.root$ do
5              $PQ_i$.ADD($e$);
6          end
7      end
8      while $TRUE$ do
9          foreach $CProf\text{-}tree\ CPT_i\ of\ Forest$ do
10             $temp$ = REMOVEMIN($CPT_i$);
11             foreach $entry\ e\ in\ temp.PT$ do
12                 $PQ_i$.ADD($e$);
13             end
14         end
15         if ($NEW\_OBJECT\_ARRIVED$ or $all\ leaves\ are\ scanned$) then
16             $\mathcal{T}_r$.label = CLOSEST(REMOVEMIN($PQ_1$), REMOVEMIN($PQ_2$), ... , REMOVEMIN($PQ_c$)).$label$;
17             return;
18     end
```

allow us to reach this point. So, whenever time allowance expires, we assign the current test entity to the class of the most closest entry (could be an entity or a class profile) among all the entries in all the priority queues, and then exit to process the newly arrived object (lines 11-13). So, the label assigned to test entity will be the most accurate one for the given time allowance. If we have sufficient time allowance to reach the leaf levels of the CProf-trees, the classification accuracy will be highest. So, when processing time allowance is high at lower stream speeds, or when we execute without any constraint on processing time allowance for each object (*non-anytime mode*), we reach finer granularities of the CProf-trees and get highly accurate results (see Figure 5.6 on page 121 and Table 5.2 on page 122). Also, as the test entities evolve with more number of objects being added into them, the classification accuracy also improves. The unit operation in the above algorithm is removal of an entry from a PQ and adding its children to it.

We conducted an empirical study, where we tested our approach by refining only a subset of top k nearest classes rather than refining all the classes (see Figure 5.4 on page 119). The results clearly show that it is sufficient to refine top 4 classes to get accurate results for all datasets used in experimentation.

5.4 Experimental Results & Analysis

All experiments were conducted over a workstation with 32 GB RAM, Intel i7 processor and Ubuntu 14.04 installed in it. All programs were implemented in C. Table 5.1 lists the datasets used for experimentation. A brief description about them is as follows:

Table 5.1: Datasets used for Experimentation in Anytime Set-wise Classification

Name	Size	#Dim	#Entities	#Classes	Entities in each class
Synth	1M	10	1K	50	equal (20 entities/class)
ECover	0.58M	10	1.5K	10	equal (150 entities/class)
CDetect	1.68M	-	751	6	skewed(14- 455 entities/class)
WebFing	111M	2	410K	2,000	approximately equal (190-210 entities/class)

5.4.0.1 Synth

It a synthetic dataset generated using guidelines given in [195]. It has 1M 10-dimensional objects generated using *gaussian mixture models*. It consists of 50 classes, with each class having 20 entities (total 1000 entities).

5.4.0.2 ECover

This is generated by converting the Forest Cover dataset (FC) [200] into an applicable form using guidelines given in [195]. We use first 10 dimensions of FC, which are numeric. It has 0.58M objects distributed in 10 different classes. Objects in each class have been clustered into 150 different entities, giving a total of 1500 entities.

5.4.0.3 CDetect

We created this dataset by collecting tweets of various Indian celebrities that include - Politicians, Sportsmen, Journalists, Film stars, Philanthropists and Businessmen (6 classes). A maximum of 3200 historical tweets were collected from each user using *Tweepy API* on 15 Aug 2018. The data set consists of 1,688,365 tweets (data objects), collected from 751 users (entities), belonging to 6 communities (class labels). We extracted keywords from the tweets using Stanford Postagger [201] and represented each tweet as vector around these keywords. We use Jaccard Coefficient as the distance measure to compute distance between two objects (tweets), unlike euclidean distance for other dataset. Also, the distribution of entities per class is skewed in this dataset (14-455 entities/class), unlike the previous two which were uniform.

5.4.0.4 WebFing

The Website Fingerprinting attack dataset has been borrowed from [197]. It consists of approx. 111M bursts (data objects) categorized into 41K traces (entities) from 2000 web pages (classes). This dataset also has uniform distribution of entities/class (190-210).

We set 80% of the entities as training entities and 20% as test entities for all datasets. We build the initial training model using 20% of the objects from training entities. The remaining objects arrive in the stream in a random mixture of training and test objects. The values of fanout parameters for CProf-tree have been set to: $m = 2$ and $M = 4$, Choosing larger fanout values can lead to increase in linear scan within each node of the tree. So, it has to be in accordance with the number of entities indexed in the tree. & *min_stat* is set to 50. The term *SC* indicates the baseline result generated by using the method proposed in [195].

The streams with varying inter-arrival rate are simulated using Poisson streams that takes an input parameter λ and generates objects with rate λ objects/sec., as explained in Section 4.7 on page 90.

We use *accuracy* as the measure of evaluation for all datasets except CDetect, for which we use *F1-score*. This is because CDetect has skewed distribution of entities per class, and hence use of accuracy can give misleading results. The remaining datasets have equal or almost equal distribution of entities per class. So, accuracy is used. For more details on the above measures refer to Appendix C on page 214.

The streams with varying inter-arrival rate are simulated using Poisson streams that takes an input parameter λ and generates objects with rate λ objects/sec., as explained in Section 4.7 on page 90.

We use Jaccard Coefficient as the distance measure to compute distance between two objects (tweets) for CDetect dataset (unlike euclidean distance for other three datasets), since it is based on text data. Also, we use *accuracy* as the measure of evaluation for all datasets except CDetect, for which we use *F1-score*. This is because CDetect has skewed

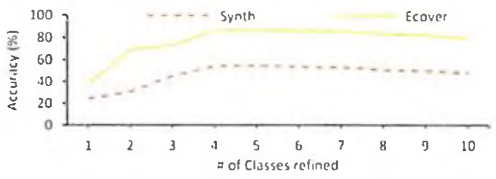


Figure 5.4: Effect on Accuracy with variation in number of classes refined at λ=60,000 ops

distribution of entities per class, and hence use of accuracy can give misleading results. The remaining datasets have equal or almost equal distribution of entities per class. So, accuracy gives good results.

5.4.1 Experimental Results

In the first experiment, we measure the classification accuracy of test entities for AnySC with increase in number of classes refined, at speed (λ) = 40,000 ops & q = 40 for *Synth* and *ECover* datasets. The results presented in Figure 5.4 clearly show that accuracy increases until we reach 4 classes and then remains steady for sometime and later shows a decline. The initial increase is because, when we refine lower number of classes, we may not reach the closest class profile. This problem reduces with increase in number of classes refined. The decline in later part of the curve is because, as the number of classes refined increases, after a point in the curve, the additional classes being added for refinement does not involve any of the closest probable classes that could help in improving classification accuracy. Rather, they only add up to the computation overhead, and does not allow us to reach finer granularity levels of CProf-trees, leading to reduction in classification accuracy. It can be observed from Figure 5.4 that refining only the top 4 classes yields better classification results for both the datasets. Similar pattern was observed for other datasets as well. This is also evident from results presented in the last column of Table 5.2 on page 122, which gives the %age of test entities whose true class value actually lies within top 4 closest classes refined by AnySC when run in non-anytime mode. Therefore, in subsequent experiments, we restrict our search to *top 4* closest classes and utilize the spare time to reach finer granularity levels within the closest classes to obtain more accurate results.

In the next experiment, we study the classification accuracy/ F1-score of AnySC with variation in number of anchor points (q) at two different stream speeds - λ=30,000 (mod-

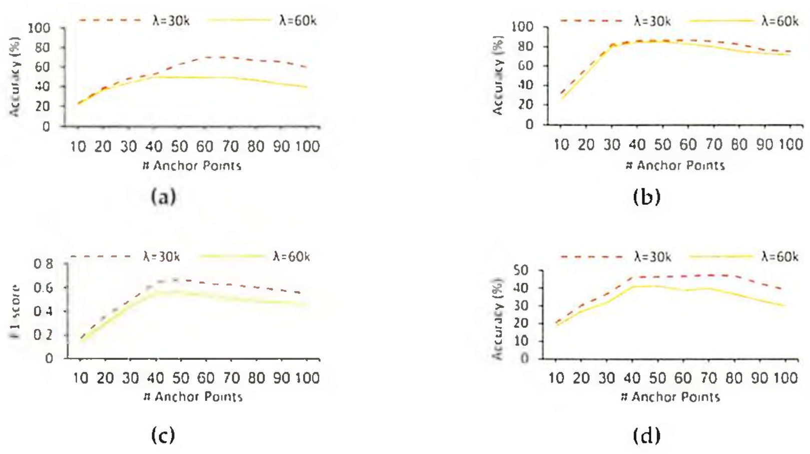


Figure 5.5: Effect of varying q on performance for (a) *Synth* (b) *ECover* (c) *CDetect* (d) *WebFing* datasets

erate speed) and λ=60,000 (high speed). The results presented in Figure 5.5 show that the accuracy increases with increase in anchor points up to a certain point and then declines for all the datasets at both the stream speeds. The decline is because, as the number of anchor points increase, the distance computation cost increases due to which the speed handling capacity reduces and thus with lower processing time allowances at higher speeds, we will not be able to reach finer granularities in CProf-trees.

We also tested the performance of AnySC with variation in stream speed and compared it with that of *SC*. The number of class profiles for *SC* has been set to 100 for Synth & ECover, 150 for CDetect & 20,000 for WebFing. We studied the accuracy/ F1-score at q = 40 & 60. The results presented in Figure 5.6 on the next page clearly show that AnySC is able to give a classification result at all the speeds, unlike *SC* which is limited to giving results up to its budget respective to the dataset (budget speeds are marked with circles and their values are given in Table 5.2 on page 122). We can also observe that the accuracy at lower stream speeds is much higher for AnySC than *SC* whose accuracy is constant for all speeds up to its budget. This is because AnySC refines its search space to finer granular levels of CProf-trees (upto the entities at the leaf level), which is not possible in *SC*. Also note that the rate of reduction in accuracy with increase in stream speed is greater for q=60, which is due to increase in cost of distance computations.

To get a fair baseline comparison, we tried to convert the *SC* model into an anytime model, where we do a linear scan of the class profiles list only to the extent time allows.

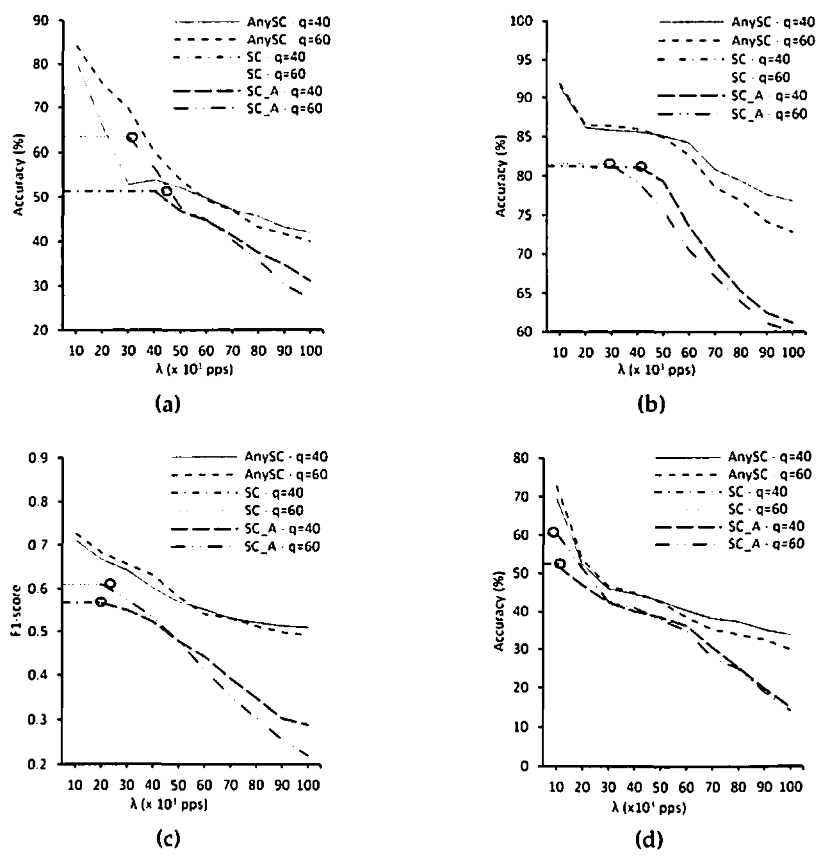


Figure 5.6: Effect of varying λ on performance of AnySC for (a) *Synth* (b) *ECover* (c) *CDetect* (d) *WebFing* datasets

For e.g., if we have a list of 100 class profiles, and time allowance is sufficient to scan only the first 15 class profiles, we assign class label to the test entity only on the basis of these 15 class profiles scanned. We refer to this strategy as *SC_A*. Figure 5.6 also presents results comparing *SC_A* with AnySC. Note that *SC* and *SC_A* shall give the same result until *SC*'s budget speed. The results clearly show that accuracy of AnySC is higher than *SC_A*. This is because, giving a class label to a test entity on the basis of only a few scanned class profiles doesn't guarantee the scanning of its actual closest class profile, which is the case in *SC_A*.

Next, we compared the accuracy/ F1-score of AnySC with that of *SC* at *SC*'s budget. The results presented in Table 5.2 on the following page clearly show that AnySC outperforms in every case. AnySC is giving better accuracy than *SC* even at the budget speed of *SC*. We also present the accuracy of AnySC without applying the anytime feature i.e. each time the refinement will take place up to the finest level in the CProf-trees with-

Table 5.2: Comparison of AnySC with SC at its budget and non-anytime performance of AnySC at $q = 40$

Dataset	Max. Speed of SC (budget) (approx.)	Accuracy /F1 of SC at its budget	Accuracy/ F1 of AnySC at SC's budget	Non-anytime accuracy /F1 of AnySC	Non-anytime speed of AnySC (approx.)	% test entities with actual class within top 4 classes refined
Synth	46,000 ops	51.2	53.4	80.7	14,000 ops	85.1
ECover	41,000 ops	81.2	85.6	91.5	11,000 ops	95.6
CDetect	25,000 ops	0.568	0.651	0.715	9,000 ops	90.3
WebFing	12,000 ops	52.3	65.1	69.8	12,000 ops	80.1

out any constraint on time allowance. AnySC gives remarkable performance in this case. However, this hampers its speed handling capacity, as it takes more time to reach finest granularity levels (entity fingerprints at leaf nodes). Note that the difference observed in non-anytime accuracy (column 5) and percentage of test entities with true class value within top 4 refined classes in non-anytime mode (column 7), is because of anomalies present in the dataset.

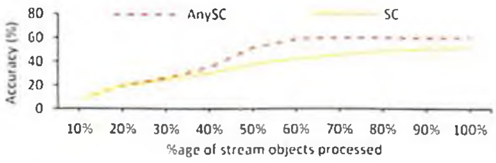


Figure 5.7: Performance of AnySC with stream progression

We also studied the evolution of test classification accuracy with stream progression for both AnySC and *SC*. The results presented in Figure 5.7 show that the accuracy increases with more objects received for both *SC* and *AnySC* for Webfing dataset. This is because: (*i*) model gets incrementally updated and drift in patterns is captured as more training objects are received; (*ii*) the distribution patterns of the test entities are predicted more accurately when more test objects are received. Similar behavior has been observed for other datasets as well.

5.5 Discussion

AnySC addresses the following limitations of *SC*:

Flat Structure Limitation. *SC*'s model is a linear flat structure consisting of a list of p class profiles. Test entities are assigned labels by identifying the closest in this list using a linear scan (of $\mathcal{O}(p)$ cost) that takes a fixed amount of time (budget) to complete, irrespective of the stream speed. AnySC addresses this limitation by using *CProf-forest*, which stores class profiles at hierarchical granularities and thus enables us to classify test entities within any given time allowance. We scan the hierarchy of class profiles in increasing order of granularity until time allows and give best possible class label to the test entity based on the finest granularity reached until now. Also, the model allows us to reach the closer coarser class profiles faster (class profiles stored in the top levels of the trees), without scanning the whole list of class profiles as was the case in *SC*.

Classification Accuracy at low speeds. At low speed streams, when the time available is more than the budget, *SC* doesn't utilize the excess time to improve the classification accuracy. The accuracy remains the same as its budget speed. However, *AnySC* makes best use of such time by reaching finest levels in the trees where entity fingerprints are stored, and thus gives very high classification accuracy.

Efficient incremental update. The cost of incremental update of the classification model in *AnySC* is little higher than the *SC* because of the bottom-up update in CProf-trees. However, due to the support given by buffers, we can always defer & merge multiple such updates (as explained in Section 5.3.1 on page 113) and traverse together leading to reduction in the update cost. This reduction is visible in *AnySC*'s capability of handling high speed streams.

Efficient Class Assignment. Class label assignment cost is logarithmic for AnySC, unlike linear cost for *SC*, thus making it more efficient.

Class profiles construction. The class profiles for each class in *SC* are constructed using k-means algorithm, with *k* as a user given parameter. On the other hand, in *AnySC*, the class profiles are created as natural outcome of CProf-forest construction, eliminating the need of any input parameter.

5.5.1 Our Contributions

- We have proposed AnySC, which is an *Anytime Set-wise Classification algorithm* for data streams. To the best of our knowledge, this is the first such approach.
- It uses a proposed data structure - *CProf-forest*, which is built over the initial training data, and serves as the classification model for processing the incoming test objects. The *CProf-trees* in the CProf-forest store class profiles at hierarchical granularities. AnySC incrementally updates the CProf-forest using the objects arriving in the stream that belong to the labeled entities, within any given processing time allowance dictated by the stream speed. The incremental updates are deferred in case the time allowance expires before the update finishes, and are completed alongside subsequent updates occurring on the same traversal path in the underlying CProf-tree.
- AnySC leverages the hierarchical structure of CProf-trees to classify the test entities within any given processing time allowance (see Section 5.3 on page 111).
- We have also developed the CDetect dataset which consists of tweets of various Indian Celebrities. This can be used as a benchmark for community detection problems.

5.6 Conclusions & Future Work

In this chapter, we proposed AnySC, which is the first *Anytime Set-wise Classification algorithm* for data streams. AnySC uses a proposed data structure known as *CProf-forest*, which serves as the classification model for processing the incoming stream. AnySC is capable of incrementally updating the CProf-forest using the labelled objects arriving in the stream within any given processing time allowance without any budget contraint. Similarly, it can handle unlabelled objects by classify the test entities within any given processing time allowance. The experimental results presented in Section 5.4 on page 117 show that AnySC can: (*i*) handle variable stream speeds and produce accurate classification results; (*ii*) handle very high speed streams with reasonable performance, unlike the baseline approach (*SC*) [195] that fails to execute when speed exceeds its budget; (*iii*) give very high classification accuracy (compared to *SC*) when stream speed is low, since it makes use of greater time available to refine the result to the greatest possible degree. The experiments also demonstrate the applicability of set-wise classification problem over both the applications described above.

5.6.1 Future Directions

AnySC model can be further improved by incorporating exponential decay in its training entities in order to perform intermittent pruning of outdated entities. This enables AnySC to detect concept drift more efficiently. AnySC model can also be adopted to image datasets.

Chapter 6

Anytime Clustering of Data Streams

Clustering of data streams has been increasingly becoming important in the recent past owing to rapid rise in utilities of IoT, transactional systems, real-time systems and other data generating systems. In a typical stream clustering algorithm, data objects continuously arrive into the system where an algorithm processes them and stores their summary statistics. Clustering of data streams have the same constraints of time and memory as described in Part II on page 57. Various stream clustering algorithms proposed in literature include - CluStream [102], HP-Stream [103], DenStream [95], Optics Stream [150], D-Stream [151], MR-Stream [152], SWEM [202], etc. These algorithms store summary statistics of the incoming data. For example, CluStream [102], DenStream [95], HP-Stream [103] and Optics-Stream [150] use micro-clusters to store the summary statistics (micro-clusters were originally conceived in BIRCH [203]). Then a suitable clustering algorithm (like DBSCAN, K-means, SLINK, etc.) is applied over the means of these micro-clusters in an offline manner (see Section 6.1.1 on page 129). The above algorithms also use mathematical models such as pyramidal time frames [102], exponential decay [95], sliding window [202], etc. to represent the age of micro-clusters. These models enable the system to weigh down the influence of older data and capture concept drift.

- Jagat Sesh Challa, Poonam Goyal, Ajinkya Kokandakar, Dhananjay Mantri, Pranet Verma, Sundar Balasubramaniam, Navneet Goyal. *A New Micro-Cluster based Approach for Anytime Clustering of Data Streams that handles Noise and Concept Drift.* Submitted for review in Journal of Experimental & Theoretical Artificial Intelligence (TETA), Taylor & Francis

In various real time applications like sensor networks, network traffic management, security surveillance systems, disease surveillance systems, web log analysis, etc., data is being generated from multiple sources resulting in multiple streams. Researchers have proposed various methods [164, 165, 166, 167, 168] for clustering of data objects arriving from multiple sources. These methods collect summary of arriving data streams in parallel among different computing nodes of a cluster, and then later compute their global aggregate on demand. For example, the methods proposed in [164, 166] repeatedly update the global k-centers using k-mediod clustering over the computing nodes in parallel and store them as the global aggregate. Similarly, the method proposed in [168] stores parameters related to EM-clustering as a global aggregate and updates them with evolving patterns at the computing nodes. In some multi-port algorithms like [204, 111, 205, 113, 112], based upon the characteristics of the incoming objects in each incoming stream, the streams themselves are clustered into groups that evolve with change in characteristics of incoming objects.

6.0.1 Research Gap & Motivation

One major problem associated with any real-time stream (including multi-port streams) is that, the rate of arrival of data objects is not fixed. Many data generating systems generate data at a varying rate as explained in Part II on page 57. The traditional stream clustering algorithms [102, 95, 103, 151, 150, 152, 168, 164, 166] are not capable of handling such varying inter-arrival rates. They run on their own limited stream speed handling capability and are not flexible. Hence, they are unfit for such use.

To address this issue, anytime stream clustering algorithms have been proposed. They perform anytime online maintenance of micro-clusters for streams that have varying inter-arrival rate of objects. A few anytime stream clustering algorithms [104, 117, 118] have already been proposed in literature. However, there are a few drawbacks associated with them. ClusTree [104] is a generic technique for anytime clustering of data streams that uses a summary structure, also known as ClusTree, to capture streams having variable speed in the form of micro-clusters. ClusTree has a drawback of inserting the incoming objects using distance computations performed from each object to the means of the aggregated micro-clusters that are stored in internal nodes of the tree. This method of insertion doesn't gaurantee the insertion of the object at its most appropriate spatial location in the tree while preserving spatial locality. This leads to reduction in overall purity of the micro-clusters being indexed, especially for high dimensional data. Also, ClusTree is not capable of handling noise and detecting concept drift. LiarTree [117] is an extension of ClusTree that captures noise and detects concept drift. However each noise to concept transition in LiarTree, builds a completely new sub-tree, which again distorts spatial locality of the micro-clusters indexed in the tree, and thus suffers drop in their purity. SubClusTree [118] is ClusTree 's adaptation to subspace clustering.

A concurrent version of ClusTree (uses multi-threading) was proposed for handling multi-port data streams in a shared memory environment [206]. However, this work presents only a conceptual model without substantiating with experimental analysis.

In this chapter, we propose AnyClus, which is a framework for anytime micro-cluster maintenance of variable speed data streams that handles noise, captures concept drift, preserves spatial locality, and produces micro-clusters of greater quality. It addresses the drawbacks presented above using a proposed variant of R-tree, *AnyRTree*, which allows insertion of objects arriving in the stream with varying inter-arrival rate. We also propose a parallel framework, Any-MP-Clus, which handles multi-port streams efficiently and produce clustering of high quality.

The rest of the chapter is organized as follows: Section 6.1 gives some preliminary concepts; Section 6.2 on the next page presents the proposed framework AnyClus along with the proposed data structure - *AnyRTree*; Section 6.3 on page 137 presents the proposed parallel framework Any-MP-Clus; Section 6.4 on page 141 presents experimental results & analysis; Section 6.5 on page 150 Section 6.6 on page 151 concludes the chapter while giving directions for future work.

6.1 Preliminaries

Let DS represent the incoming data stream, consisting of a set of objects $X_1, X_2, ... X_k ...$ arriving at time-stamps $T_1, T_2, ... T_k ...$ where X_i $(x_i^1, x_i^2, ..., x_i^d)$ is a d-dimensional object.

6.1.1 Micro-Clusters (mcs)

Micro-clusters (or *mcs*) are a popular technique used in clustering of data streams [102, 103, 95, 150, 104, 117]. It is a very small cluster that stores summary statistics of a few very closely related data objects. It is defined as follows:

Definition 6.1. For representing a set of d-dimensional objects $X_1,...,X_n$, a micro-cluster is a tuple: $mc_j=(n_j,S_j,SS_j)$, where each of these entries is defined as follows:

- n_j is the number of represented objects in micro-cluster j.
- S_j is a vector of size d storing sum of data values of all the represented objects for each dimension, i.e. for a dimension p, $S_j[p] = \sum_{i=1}^{n_j} x_i^p$.
- SS_j is also a vector of size d storing squared sum of data values of all the represented objects for each dimension, i.e. for a dimension p, $SS_j[p] = \sum_{i=1}^{n_j} (x_i^p)^2$.

The above stream clustering algorithms exploit the additive property of mcs to incrementally aggregate incoming data objects. For merging a data object X_i into a micro-cluster mc_j, we do the following operations for each dimension p:

$$S_j[p] = S_j[p] + x_i^p....(1) \qquad SS_j[p] = SS_j[p] + (x_i^p)^2....(2) \qquad n_j = n_j + 1....(3)$$

Also, two mcs (mc_a & mc_b) can be merged into mc_{mer} as follows:

$$S_{mer}[p] = S_a[p] + S_b[p]...(4) \qquad SS_{mer}[p] = SS_a[p] + SS_b[p]...(5) \qquad n_{mer} = n_a + n_b...(6)$$

The mean and radius of a mc can be computed as follows:

$$Mean(\mu_j) = \frac{S_j}{n_j}...(7) \quad Radius(\rho_j) = \sqrt{\left(\frac{SS_j}{n_j}\right) - \left(\frac{S_j}{n_j}\right)^2}...(8)$$

6.2 AnyClus: The Proposed Framework

AnyClus is the proposed framework for online maintenance of mcs for handling streams that have variable and high inter-arrival rates. In its online phase, AnyClus receives data objects from the stream, and inserts them into *AnyRTree*, which stores and maintains mcs for the arriving data. AnyRTree is described as follows:

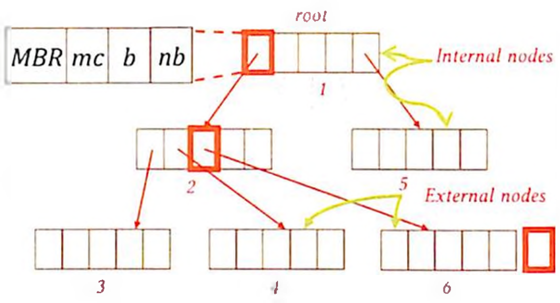


Figure 6.1: Structure of AnyRTree

6.2.1 AnyRTree

AnyRTree is the proposed hierarchical indexing structure, illustrated in Figure 6.1. It is a variant of R-tree, which stores mcs at hierarchical granularities in its nodes, alongside the *minimum bounding rectangles* (MBRs) [122]. There are two types of nodes in AnyRTree: *internal* and *leaf*. The internal nodes store the following entries: a pointer pt, a minimum bounding rectangle MBR, a micro-cluster mc (S, SS, n), a buffer b and a noise buffer nb (see Figure 6.1). pt points to the root of the child sub-tree underneath the current entry. MBR is a d-dimensional rectangle that bounds all the data objects indexed in the child sub-tree. mc stores the aggregate of all the objects indexed at the leaf level of the underneath child sub-tree. Buffer b is another micro-cluster, which is used for temporary local aggregation of incompletely processed data objects (see Section 6.2.2 on the following page). It is this buffer that enables deferred anytime insertion of data objects into AnyRTree. The noise buffer nb is used for handling noise (see Section 6.2.2 on the next page). Entries of leaf nodes store a micro-cluster mc that summarizes the data objects aggregated in it. They, however, do not contain MBR or buffers. This is because inserts at leaf level are final and any kind of deferred processing or handling of noise is not required. Definition 6.2 defines AnyRTree.

Definition 6.2. *AnyRTree* is a height balanced multi-dimensional indexing structure having the following properties:

- Each node (both internal and leaf) contains between m and M entries. The root has at least one entry.
- An entry e in an internal node stores the following entries: (i) a pointer pt to the child sub-tree; (ii) an minimum bounding rectangle MBR; (iii) a micro-cluster mc;

(iv) a buffer b; and (v) a noise buffer nb.

- An entry e in a leaf node stores a micro-cluster mc.

One can see that AnyRTree stores hierarchy of mcs. A mc in an entry of any internal node stores the aggregate of the mcs stored in its child, which themselves are aggregates of mcs present in their own children. In this way, any mc in an internal node summarizes all the leaf level mcs indexed in the subtree rooted at it.

6.2.2 Anytime Micro-Cluster Maintenance of AnyRTree

AnyClus handles the incoming stream by continuous insertion of each object into AnyRTree. The insertion procedure for AnyRTree is a top-down node by node traversal similar to that of an R-tree, with an additional support of anytime interruption and noise handling. Anytime interruption means that the insertion of a data object progresses as long as the processing time allowance permits, after which it interrupts (even if its insertion is not complete) to process a newly arrived object. This is achieved using "buffer and hitchhiker" concept [104]. While inserting an object, if time allowance expires before the insertion finishes, the object is left inside the buffer of the closest entry (calculated using min-distance) of the node being traversed, and the algorithm proceeds to process the newly arrived object. When insertion of another object passes through the same insertion path, it picks up the object stored in the buffer and takes it along as a hitchhiker, to finish its insertion to the leaf level.

The insertion step also handles noise using noise buffers and enables capture of concept drift. An object is treated to be noise with respect to a node in the traversal, if merging this object to the closest entry in the node, leads to expansion of the entry's MBR by more than δ%. We then insert this noise object into the closest entry's noise buffer. We perform periodic checks of the noise buffers to detect any noise to concept transition. This is achieved by using a time interval γ. For each entry of a node, after every γ units of time, we check the mc in the noise buffer for any possible noise to concept transition. If noise becomes a concept, we carry this mc as another hitchhiker down for insertion. There are two criteria for noise to buffer transition: 1) absorption of a mc into entry's MBR should lead to less than δ% increase in area of entry's MBR; 2) the noise buffer must have accumulated at least β new objects in it. All three parameters - δ, γ and β are user defined. We give appropriate recommendations for choosing their values in Section 6.4.1 on page 143.

Algorithm 6.1 on page 134 gives the pseudo code for anytime insertion of a data object X into AnyRTree, R. It proceeds in the following steps:

- The algorithm takes an object X to the root node of R and recursively descends down the tree to locate the most appropriate leaf node for its insertion.
- At each internal node we encounter in the traversal, we find the closest entry e to X (line 3 of Algorithm 6.1 on page 134) using min of min-distances. If there is a tie, we break it using the distance of X from the means of the mcs stored in the entries.
- We then check if X is noise with respect to e (lines 4-5). If it is so, we insert it into e's noise buffer using Eqs. 1, 2 & 3.
- If we were carrying a hitchhiker $\hat{H}$ object with us, we also check if it is noise. If so, we do the same steps to absorb it into the noise buffer of its nearest entry e_h (line 8).
- At this point in time, we check if a new object has arrived arrives (say Y), we interrupt the insertion of X and process Y (lines 12-15). For doing so, we merge X into the buffer of e ($e.b$) using Eqs. 1, 2 & 3, and then quit its insertion. We also merge the hitchhiker to the buffer of its nearest entry.
- Next time when we descend down the same path for insertion of another object, we carry the mc stored in the buffer down as a hitchhiker to complete its insertion (lines 20-22 of Algorithm 6.1 on page 134). While doing so, we update the statistics in $e.mc$ with respect to the descending object X and the hitchhiker $\hat{H}$, and then recursively descend into the sub tree rooted at e.
- While carrying the hitchiker down, if the subsequent traversal path of descending object X and hitchiker $\hat{H}$ differ, we merge $\hat{H}$ into the nearest entry's buffer and continue insertion of X (lines 6-10).
- If it is now time to do the periodic check for noise to concept transition for this entry, we do the check with the criteria described earlier (lines 17-19). If the transition

happens, the noise buffer is merged with a separate hitchhiker $\hat{h}_n$ and is handled separately. It is made sure to reach the leaf level without being locally merged anywhere in between. The handling of this separate hitchiker is not depicted in Algorithm 6.1 on the next page, for simplicity.

- When we encounter an external node (lines 23-30), we first merge $\hat{H}$ to its nearest entry in the node using Eqs. 4, 5 & 6 and empty it. Then we create a new mc containing X using Eqs. 1, 2 & 3, and store it as a new entry in the node as shown in node 6 of Figure 6.1 on page 130.
  - ◇ If creation of a new entry has created a node overflow, we split the node into two and accommodate the newly created entry. The split operation is similar to split of an R-tree node, which can propagate up to the root leading to creation of a new root. If however, there is a new object that has arrived and there is no time to perform split, we simply merge the two closest entries in this node and exit.

We can also enforce a limit on total number of leaf level mcs in the tree. Whenever the limit is reached, we don't let the tree grow further, i.e., we don't let any further node splits to happen and merge the incoming objects (or mcs) to their nearest entries only instead of creating new entries.

The concept of aggregating incoming objects into buffers and later carrying them down the tree, is known as local aggregation. More than one data object can get aggregated into any of the two kinds of buffers and the aggregated mcs in the buffers are carried down as hitch-hikers in a future descent. Local aggregation helps in retaining the maximum possible granularity of the object at any given stream speed.

Note here that the buffers store aggregated objects like in AnySC and unlike AnyFI. So, only one object is sufficient to be stored in the buffer.

We can clearly observe from the insertion algorithm that it is interruptible and at the same time makes best use of available time. AnyRTree is capable of processing very fast streams and at the same time use greater time allowances during lower stream speeds to refine the clustering model. Whenever, the processing time allowance given to a data object is less, it places it in the most appropriate buffer and exits the insertion, rather than

Algorithm 6.1: Insert-In-AnyRTree

```
procedure Insert-In-AnyRTree ()
    Input : AnyRTree node $node$, Data Object $X$, Hitchiker $\hat{H}$, Hitchiker $\hat{H}_n$
    Output: $X$ inserted into sub-tree rooted at $node$ until time allows
    if $node$ is an internal node then
        $e$ = Get-Closest-Entry($node$, $X$); $flagctr = 0$;
        if IsNoise($X$, $e$) then
            Merge-MC-To-MC($X$, $e.nb$); $flagctr$++;
        end
        if $\hat{H} \neq$ NULL then
            $e_h$ = Get-Closest-Entry-MC($node$, $\hat{H}$);
            if IsNoise($\hat{H}$, $e_h$) then Merge-MC-To-MC($\hat{H}$, $e_h.nb$) ;
            else if $e \neq e_h$ OR $flagctr = 1$ then
                Merge-MC-To-MC($\hat{H}$, $e_h.b$); $\hat{H}$ = NULL;
            end
        end
        if $flagCtr = 1$ then exit ;
        if new object arrived then
            Merge-Object-To-MC($X$, $e.b$);
            if $\hat{H} \neq$ NULL then Merge-MC-To-MC($\hat{H}$, $e.b$) ;
            exit;
        else
            if time to check for noise to concept transition then
                if Check-Noise-To-Concept($e.nb$) == $TRUE$ then
                    Merge-MC-To-MC($e.nb$, $\hat{H}_n$); Merge-MC-To-MC($\hat{H}_n$, $e.mc$); $e.nb$ = NULL;
                end
            end
            if $e.b \neq$ NULL then
                Merge-MC-To-MC($e.b$, $\hat{H}$); Merge-MC-To-MC($\hat{H}$, $e.mc$); $e.b$ = NULL;
            end
            Merge-Object-To-MC($X$, $e.mc$); Insert-In-AnyRTree($e.child$, $X$, $\hat{H}$, $\hat{H}_n$);
        end
    end
    if $node$ is a leaf node then
        if $\hat{H} \neq NULL$ then
            $e_h$ = Get-Closest-Entry($node$, $\hat{H}$); Merge-MC-To-MC($\hat{H}$, $e_h.b$); $\hat{H}$ = NULL;
        end
        $newMC$ = Create-Micro-Cluster($X$); Insert $newMC$ as a new entry in $node$;
        if $node$ $overflows$ then
            if $new$ $object$ $arrived$ then
                Merge-Closest-Two-Entries($node$); exit;
            end
            else Split-Node($node$) ;
    end
```

giving up all together. And then, it makes the best use of future descents down the same path by carrying the object inserted into the buffer as hitchhiker. Similarly, whenever the time allowance is more, insertion reaches the most appropriate leaf in the tree. This shows that AnyRTree handles variable stream speeds effectively. Also, note that a mc closer to root would be of coarser granularity, whereas the one at greater depth would be of finer granularity. The processing time allowance for an incoming object affects the granularity at which it gets absorbed into the tree. Greater the time allowance, greater the granularity at which the object gets absorbed.

6.2.2.1 Time Complexity

The insertion into AnyRTree simply follows a traversal which is logarithmic in height of the tree. In the worst case, every data object will reach a leaf node. So, if n_{mc} is the total number of leaf level mcs in the tree, the worst case insertion complexity will thus become - $O(log_m n_{mc})$, where m is the minimum number of entries in each node of the tree. Note that in worst case $n_{mc} = N$, where N is the data size.

6.2.3 Key Factors of AnyRTree design

We gain the following advantages from the design of AnyRTree:

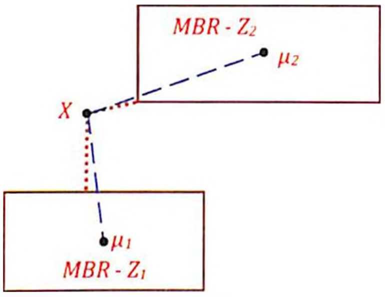


Figure 6.2: Distance Computations: AnyRTree vs ClusTree & LiarTree

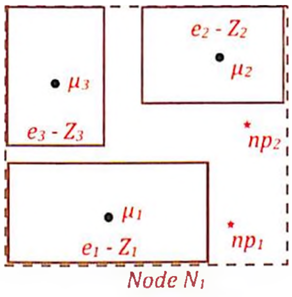


Figure 6.3: Illustrating Granular Noise Buffers of AnyRTree

- *Efficient method of descent in insertion.* The major advantage of AnyRTree when compared to ClusTree and LiarTree is that the AnyRTree is more closer to the R-tree spatial containment principles. AnyRTree additionally keeps MBRs in the entries of its nodes along with mcs. While descending into the tree during the insertion of an object, an entry is selected for descending down that has the minimum of min-distances from the descending object X, in case of AnyRTree. Whereas, in ClusTree & LiarTree, the entry is selected based on minimum distance from the means of the mcs present in the entries. This is illustrated in Figure 6.2. It is clear from the figure that the object X will descend into the entry whose mean is μ_1 in case of ClusTree & LiarTree, and the entry with MBR Z_2 will be selected in case of AnyRTree. So, X will be inserted into spatially closer nodes under Z_2 than that of Z_1 in AnyRTree, and thus, the object gets inserted into more appropriate leaf nodes than in ClusTree & LiarTree. This achieves better, compact and purer mcs (substantiated by experiments in Section 6.4.1 on page 143).

- *Greater Granularity of Noise Buffers for efficient handling of noise and concept drift.* Another advantage of AnyRTree is its greater granularity in noise buffers. LiarTree has only one noise buffer per node, whereas AnyRTree has number of noise buffers equal to number of entries in a node. This helps in capturing noise at greater granularity. Consider Figure 6.3 on page 135, which has a node N_1, having three entries - e_1, e_2 & e_3, whose MBRs are Z_1, Z_2 & Z_3, and means are μ_1, μ_2 & μ_3, respectively. Lets say, we have two noise points detected - np_1 and np_2. Note that np_1 is spatially closer to entry e_1 and np_2 is spatially close to e_2. LiarTree would have simply aggregated both of them into a single noise buffer which would have led to creation of a single aggregate which would have been far away from both e_1 and e_2. However, AnyRTree keeps separate noise buffers for each entries separately, leading to insertion of np_1 into noise buffer of e_1 and np_2 into noise buffer of e_2. This keeps the noise points more spatially distinct, so that in case they become a concept in future, the spatial locality is maintained and the mcs thus formed are purer (see Table 6.3 on page 145).

- *Effective noise to concept transition preserving spatial locality.* When noise to concept transition takes place, LiarTree creates a new subtree for that resultant mc, which is indexed as a new entry in node. This new subtree now grows top-down, unlike the remaining sub-trees that grow bottom-up. The mc resultant of the above transition, could now be actually closer to one of the entries in the node. LiarTree forces even such spatially close mcs to be inserted into the new sub-tree, thus making them spatially far from nearby mcs and thus distorting the spatial locality of the tree. However, AnyRTree inserts the such resultant mcs into the sub-tree underneath closest entry of the node, so that they are absorbed into spatially closest mcs, if required. This results in effectively preserved spatial locality of the tree and as a result of which we get more purer mcs. (See Table 6.4 on page 146).

6.3 Any-MP-Clus

Any-MP-Clus is the proposed parallel framework for anytime clustering of multi-port data streams. Figure 6.4 illustrates its work flow. It has two phases - *online* and *offline*. In the online phase, data objects arriving at variable inter-arrival rate are received at multiple computing nodes in parallel and are stored as mcs in separate AnyRTrees for each computing node. The online phase executes in batches of t_{in} units of time. After receiving a batch, the *leaf level* mcs captured in AnyRTrees from all the computing nodes are processed in the offline phase, while a fresh batch of data objects is captured into fresh AnyRTrees at each computing node. These two things happen in parallel and is achieved by using multi-threading at each computing node. We have described how each computing node captures data objects into AnyRTree in the previous section. We now describe the offline processing that happens for each batch of mcs.

6.3.1 Offline Phase

After receiving the stream for t_{in} units of time across all the computing nodes, we execute the offline phase. In this phase, we process the mcs created in this time window across all the machines, aggregate them and efficiently store them in the tilted-time window framework. The offline phase has four steps as shown in Figure 6.4. We now explain each of these in detail.

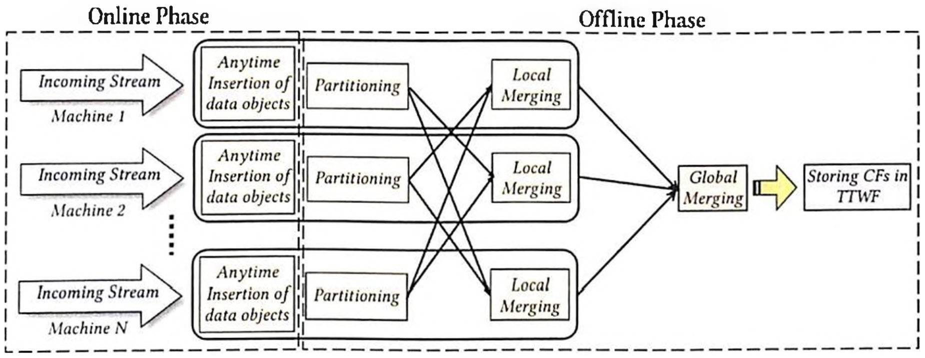


Figure 6.4: Workflow of Any-MP-Clus

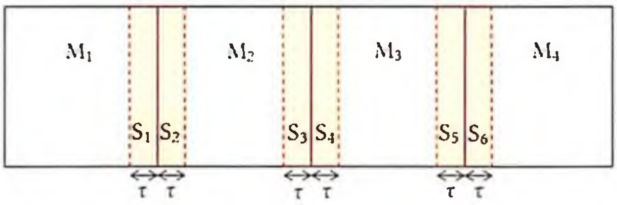


Figure 6.5: Data Partitioning

6.3.1.1 Partitioning

After executing the online phase, we are left with leaf level mcs in AnyRTrees across multiple computing nodes. For efficient aggregation (merging) of these mcs in parallel, we create a spatially disjoint partitioning having equal number of mcs in each computing node (or machine) to ensure load balancing. Figure 6.5 shows a spatially disjoint partitioning for four machines over a two dimensional dataset. To achieve this, we compute the boundaries of the splits (which are medians) such that each partition gets equal number of data objects. Computation of these medians would require entire dataset to be present on a single machine. So, it would lead to a very high communication cost if all the machines were to transfer their mcs to that single machine. To minimize communication cost, we achieve a similar disjoint partitioning with "almost" equal number of mcs in each machine. We take a sample of mcs (say $q\%$) from each machine and then send them to all other machines. Every machine now gets the same set of sampled mcs over which each machine computes medians equal to number of machines minus 1, with respect to one dimension only (as shown in Figure 6.5 for x-dimension). The dimension that has maximum spread is chosen for splitting. Once the medians are computed, the mcs are partitioned according to them and moved to their respective machines to achieve disjoint partitioning as shown in Figure 6.5. Please note that all the medians are computed over the means of the mcs. Also, in case of large number of streams being received, we can use more than one dimension for partitioning.

Please note that we use two MPI broadcast calls in total for this step. Median computations are performed locally and doesn't involve any communication.

Algorithm 6.2: Local-Merging

```
procedure Local-Merging ()
    Input  : Machine $M_1$
    Output: Leaf-Level mcs of $M_1$ aggregated to coarser granularity
    List $ml$ = $M_1.microclustersList$;   RTree $R_1$ = Create-R-Tree ($ml$);
    List $ml_{new}$ = New-List();   $flag$ = TRUE;
    while $flag$ do
        $flag$ = FALSE;
        foreach $mc_x$ in $ml$ do
            if $mc_x.marked$ == $FALSE$ then
                $mc_y$ = Get-One-NN($mc_x$,$R_1$) // gets closest unmarked mc;
                if Merged-Radius($mc_x$, $mc_y$) $< \tau$ then
                    $mc_{merged}$ = Merge-MCs ($mc_x$,$mc_y$);   insert $mc_{merged}$ into $ml_{new}$;
                    mark $mc_x$ and $mc_y$ as processed in $ml$;   $flag$ = TRUE;
                else
                    mark $mc_x$ as processed in $ml$;   append $mc_x$ to $ml_{new}$;
                end
            end
        end
        $ml$ = $ml_{new}$;   Empty($ml_{new}$);
    end
```

6.3.1.2 Local Merging

In this step, every machine aggregates the mcs received from different machines in the partitioning step (which are now locally present), into a single set of mcs to be stored in Tilted-Time Window Framework (TTWF). The mcs are aggregated to a user-defined granularity, using a user defined threshold on maximum radius referred to as τ. We do an iterative pair-wise merging of mcs based on τ, i.e. we merge a pair of mcs only if the merged mc has radius below τ. The smaller or larger choice of τ produces mcs at finer or coarser granularity respectively. Algorithm 6.2 gives the pseudo code for the local merging step. In brief, for every iteration, we iterate through the entire list ml of mcs and merge every closest pair of unprocessed mcs mc_x & mc_y, if the radius of the merged mc mc_{merged} is less than τ. We use R-tree to index mcs as it aids in finding the closest mc in logarithmic time. And then we add mc_{merged} to a new list ml_{new}, and mark mc_x and mc_y as processed in ml. However, if we do not merge them, we mark mc_x as processed in ml and add it ml_{new}, to give it a chance to get merged to some other mc in the next iteration. After processing all the mcs in ml, we proceed to the next iteration where we process ml_{new} in the same way. Finally, when there is no possibility of merging of mcs anymore, we exit the loop and present ml as the list of aggregated mcs.

Let m_{mc} be the number of mcs in a given machine. The worst case complexity of finding a pair of closest mcs shall be $O(m_{mc})$. So, for each iteration, total cost shall be $O({m_{mc}}^2)$.

Since, we do parwise merging, expected number of iterations shall be $O(\log m_{mc})$. Then the worst case complexity of local merging step shall be $O({m_{mc}}^2 \log m_{mc})$.

6.3.1.3 Global Merging

After the local merging step at each machine, global merging is done. In this step, we perform merging of the mcs across the machines. For this, we transfer mcs lying in the strips of width τ from all the machines to a master machine. For the distribution shown in Figure 6.5 on page 138, we do a pairwise merging of strips - S_1 & S_2; S_3 & S_4; and S_5& S_6. This merging is also similar to local merging, where we merge micro-clusers based on the threshold criteria - τ. When the number of streams (or machines in the cluster receiving streams) is high, we can use tree-parallel mode of merging as well [36]. This would be more efficient than merging all pairs of strips in a single master. The above step shall take total number of MPI calls equal to number of machines in the cluster.

Finally, we have merged mcs lying in all the machines along with the globally merged mcs in the master machine. All these sets of aggregated mcs are then stored in a Tilted-Time Window Framework (TTWF) [98], which is explained in Appendix B on page 212. Note that this step would require additional MPI calls equal to number of computing nodes.

We can clearly see that, the mcs have been merged in parallel across all the machines of the cluster using the steps - *local merging* and *global merging*. However, one could take an alternative sequential approach to merging, where we transfer all the mcs to a single machine and aggregate them (would require MPI calls equal to number of computing nodes). However, this leads to a higher merging time. Table 6.5 on page 150 substantiates this argument and also shows that purity of mcs resultant of parallel merging is approximately same as for those resultant of sequential merging.

6.3.1.4 Tilted-Time Window Maintenance of micro-clusters

We store the aggregated mcs generated for each batch in tilted-time window framework. As explained in Appendix B on page 212, TTWF [98] keeps a set of windows (Figure B.1 on page 212) which records the entire stream from its beginning, but in logarithmically decreasing order of granularity. The average radius of the mcs in w_1 and w_2 will be the lowest and it keeps on increasing from w_3 to w_4 and so on. Following the description of maintenenance of TTWF from Appendix B on page 212, we now explain how TTWF is maintained with mcs:

We start with an empty TTWF. After receiving three batches, we shall have w_1, w_2 and tw_2 full. These sets of mcs were aggregated with τ as the radius threshold. When next batch arrives, we merge sets in w_2 and tw_2 and place them in window w_3. The radius threshold used for this merging will be $\tau \cdot f$ where $f > 1$ is a user defined parameter. Essentially, we increase the radius threshold for merging, which is controlled by f. In general, for merging mcs from two windows to place in window w_i, the radius threshold for merging will be equal to $\tau \cdot h \cdot f$, where $h = i - 2$. For window w_3, $h = 1$. The merging of the mcs of two windows happen in a similar way as that of local merging step. In this way, by exponentially increasing the radius threshold of the mcs in the windows, we are progressively achieving coarser granularity of the mcs, and thus giving lesser and greater importance to older and recent data respectively. And when we have to perform clustering over these mcs, we simply pick up a required subset of windows and perform clustering over the mcs present in them. In those clustering results, the recently arrived objects will have the highest importance.

Please note the the offline phase of AnyClus is similar to that of Any-MP-Clus, but without any merging step. The leaf level mcs of the single AnyRTree are aggregated to appropriate granularity and are fed into TTWF as explained above. The whole thing is executed only on a single computing node. Also note that, along with the noise to concept transitions that happen in the online phase, TTWF also supports in capturing the concept drift for both AnyClus and Any-MP-Clus.

6.4 Experimental Results and Analysis

All experiments are conducted on a cluster of 32 computing nodes, where each node is an IBM x3250 m4 Server with Intel Xeon CPU E3-1230 v2 @ 3.30GHz (64-bit) processor and 32 GB (DDR3 âĂŞ 1600 MHz) RAM. The experiments on AnyRTree are conducted over a single node of the cluster. All algorithms were implemented in C with MPI. The

Table 6.1: Details of the datasets

Dataset	No. of Data Objects	No. of Dimensions	No. of Classes/Clusters
FOREST COVER (FC)	0.58 M	10	7
KDDCUP1999 (KDD)	4.8 M	38	24
SYTHETIC CLUSTER (SC)	1M	3	4
MPAGD3.2M (M32)	6 M	2	9464
SFONT1M (SF)	1 M	11	6690
FOF57M (FOF)	57 M	3	1613820
MPAGD1B (M1B)	1 B	10	—

details of the datasets used for experimentation are given in Table 6.1. Forest Cover (*FC*) is a labelled data from UCI repository [200]. It has 0.58M objects with 10 numerical attributes. KDDCUP1999 ([207]. Synthetic Cluster (*SC*) is synthetically generated dataset that consists of 1M objects with 3 dimensions. It consists of 4 well separate clusters (4 x 0.2M objects) as well as noise (0.2M objects). The last four datasets are unlabeled and taken from Millennium repository and contain astronomical data of galaxies in the sky [135].

Ground Truth Generation. The ground truth (class labels) for FC and KDD are already available in the dataset. SC has been synthetically generated and class labels are known. Ground truth for the remaining datasets has been created using DBSCAN clustering, by treating each cluster and noise point as a separate class. The parameters chosen for DBSCAN over these datasets are as follows: ϵ=2 & min_pts=5 for MPAGD3.2M (*M32*); and ϵ=1 & min_pts=5 for SFONT1M (*SF*); & ϵ=1 and min_pts=6 for FOF57M (*FOF*). These values were chosen by an experiment on each of these datasets, where we order the data objects based on the increasing order of their distances from their k^{th} nearest objects, using $k = min\_pts$, and a plot is generated with x-axis as the ID of the data object and y-axis as the distances computed above. In this plot wherever the curve takes a steep curve, the distance at that point is chosen as ϵ [6]. We couldn't run DBSCAN on MPAGD1B (*M1B*) dataset because of limited available computational resources to process such large dataset as a result of which we couldn't generate its ground truth.

We perform two sets of experiments. In the first, experiments are performed on a single-port stream to compare the effectiveness of AnyRTree (AR) with respect to ClusTree (CT) and LiarTree (LT) (Section 6.4.1 on the following page). In the second set, experiments are performed over Multi-Port streams (Section 6.4.2 on page 148), to highlight the proposed parallel framework. We have not explicitly conducted a separate experiment to show the effectiveness of the sequential AnyClus framework, as the results are of similar nature as those for Any-MP-Clus. We evaluate the quality of mcs produced by AnyRTree, ClusTree and LiarTree in terms of two parameters - (i) *granularity*: the number of mcs present at the leaf level of the tree, and (ii) *purity*: the average purity of leaf level mcs (see Appendix C on page 214 for definition of purity) The streams with varying inter-arrival rate are simulated using Poisson streams that takes an input parameter λ and generates objects with rate λ objects/sec., as explained in Section 4.7 on page 90.

The default values of parameters chosen for AnyRTree are δ = 5%, γ = 0.5 sec and β = 10. These parameters have been chosen on the basis of parameter tuning experiments conducted (see Figure 6.9 on page 147, Figure 6.10 on page 147 and Figure 6.11 on page 147) The noise probability for LiarTree has been chosen to be 0.7 as suggested in its chapter. The fanout parameters has been set to m=2 and M=4 for all the trees, as per the recommendations given in [104]. Also note that for fair comparision of ClusTree and LiarTree with AnyRTree, we don't employ exponential decay in them. AnyRTree doesn't have exponential decay as aging is taken care by TTWF. In every experiment, we insert the initial 10% of the data into the indexing structure (any of the above) without anytime features, and then start the stream.

6.4.1 Experiments on AnyRTree

All experimental results shown in this subsection are performed for single-port stream. In each experiment, we insert the entire dataset into each of the summary structures, and compare the quality of mcs produced.

First, we compare the purity and granularity of the leaf-level mcs generated by AnyRTree (AR), ClusTree (CT) and LiarTree (LT) at different stream speeds, for both labelled (FC & KDD) and unlabelled (M32 & SF) datasets. The labels of the curves in all the figures follow the notation: <dataset> - <tree>. For example, "FC-AR" indicates that the experiment was conducted on FC dataset using AnyRTree. The results presented in Figure 6.6 on the next page & Figure 6.7 on the following page show that AnyRTree has generated mcs

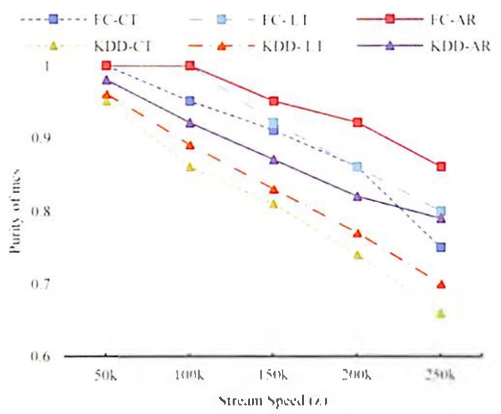


Figure 6.6: Purity of leaf level mcs generated by AnyRTree vs ClusTree & LiarTree for FC & KDD at different stream speeds

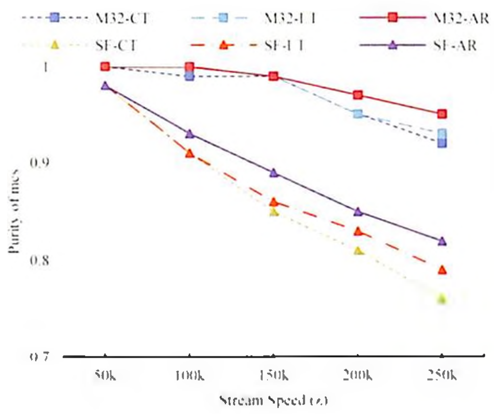


Figure 6.7: Purity of leaf level mcs generated by AnyRTree vs ClusTree & LiarTree for M32 & SF at different stream speeds

Table 6.2: Granularity of mcs generated by AnyRTree vs ClusTree & LiarTree for FC & KDD at different stream speeds

Dataset	50k	100k	150k	200k	250k
FC-CT	450564	352482	281436	220003	182474
FC-LT	449852	349865	274865	214587	176325
FC-AR	423386	340684	266358	202477	163677
KDD-CT	2081826	929505	466450	259595	144552
KDD-LT	2071865	903654	448752	239547	135842
KDD-AR	1717778	856284	404157	225124	123736

with greater purity than ClusTree and LiarTree for both the datasets, especially at higher stream speeds where aggregation is high. Also, the difference in purity for KDD (38 dimensions) is higher than that for FC (10 dimensions). This shows that AnyRTree performs better for high dimensional datasets. Also, the results presented in Table 6.2 show that the number of mcs generated at leaf level (granularity) is slightly less for AnyRTree. This shows that AnyRTree generates less number of micro-clusters with greater purity, which implies that the mcs generated are more compact, i.e. AnyRTree keeps spatially closer points in the same micro-cluster.

Next, we compare the purity of leaf level mcs generated by AnyRTree, ClusTree and LiarTree for a given speed (λ=100k ops), while varying the granularity of mcs generated over FC and KDD datasets. For each run, we fix the maximum number of leaf level mcs to be indexed in the tree and do not let the tree grow beyond it (explained in Section 6.2.2 on page 131). The results presents in Figure 6.8 on the next page show that even at coarser granularity AnyRTree is able to produce high quality mcs.

Next, we compare AnyRTree and LiarTree in terms of their ability to capture noise and

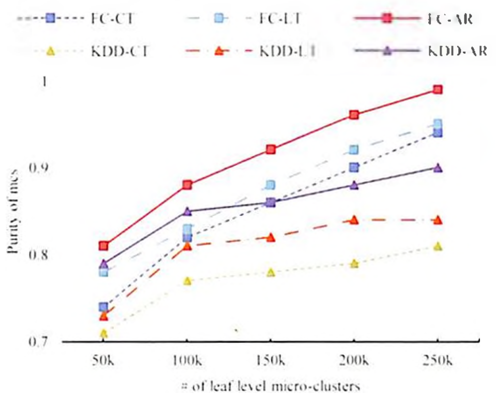


Figure 6.8: Purity of leaf level mcs generated by AnyRTree vs ClusTree & LiarTree for FC & KDD for varying granularity at Λ=100k

concept drift. For this we created a synthetic dataset (SC) of 1M objects and 3 dimensions, which consists of 4 well separated clusters with each cluster having 0.2M objects in it. It also contains 0.2M noise points randomly spread across the dataset. We simulate the stream in such a way that the objects arrive cluster by cluster and noise points come arbitrarily. This simulates evolving concepts in the stream. For fair comparison, we run this experiment in non-anytime mode (without fixing any limit on processing time allowance for any incoming object) and insert the objects into AnyRTree and LiarTree. We measure the purity of mcs produced intermittently at regular interval of 1 second. The results presented in Table 6.3 show that the purity of the leaf level mcs produced is higher for AnyRTree than LiarTree at all the time intervals measured. This shows its effectiveness in capturing noise and concept drift as explained in Section 6.2.3 on page 135.

Table 6.3: Purity of leaf level mcs produced by AnyRtree vs LiarTree for SC dataset

Time Interval (Sec.)	1	2	3	4	5	6	7	8	9	10
Purity of LiarTree	0.79	0.81	0.81	0.78	0.77	0.79	0.80	0.78	0.82	0.79
Purity of AnyRTree	0.88	0.89	0.91	0.89	0.87	0.90	0.90	0.89	0.89	0.88

We also compare the preservation of *spatial locality* for AnyRTree and LiarTree. Both AnyRTree and LiarTree are based on R-Trees and it is a well known fact that R-trees are very efficient in executing ϵ-neighborhood and k-nearest neighbor queries. The greater the preservation of locality in an R-tree, the better the performance of above queries. So, we can check performance of both the queries over AnyRTree and LiarTree to establish which one has better preservation of spatial locality. We use DBSCAN Clustering and k-NN Classifier algorithms to check for the performance of neighborhood and nearest neighbor queries respectively, as these queries are extensively used in them (> 95% of execution time). For this, we index a complete dataset into both AnyRTree and LiarTree and then convert Liartree into an R-tree by bottom-up construction of MBRs with the leaf level mcs as data points. AnyRTree intrinsically has MBRs in its nodes, so there is no need for any such conversion. So, finally we will have two R-trees - one from LiarTree and one from AnyRTree, both of which have MBRs in the internal nodes and data objects (means of mcs) in the leaf nodes. We execute DBSCAN and k-NN Classifier [121, 208] over the objects in the leaf level, using the above R-trees for queries. We do this for M32 dataset with ϵ=2 and *min_pts*=6 for DBSCAN, and k=4 for k-NN Classifier, and compare the run-time performance. While indexing the complete dataset, the speed of the stream was set to λ=50k for both the trees. We run this experiment in two modes. In the first, we fix the number of leaf level mcs to 0.5M. This will fix the total number of queries executed. In the second, we don't enforce any such limit. The results presented in Table 6.4 clearly show that the performance of DBSCAN algorithm has always been better for AnyRTree than LiarTree, in both the modes. Also, one can see that the memory requirement is only a little higher for AnyRTree in the first mode, where the number of mcs indexed is fixed. This is due to increase in granularity of buffers. However, in the second mode, the memory is lesser than LiarTree due to lesser number of mcs indexed in AnyRTree. This establishes the ability of AnyRTree to preserve spatial locality without any substantial increase in memory requirement. Also note that DBSCAN is very commonly used for offline clustering in typical stream algorithms. Thus usage of AnyRTree also makes the offline clustering efficient.

We then conduct experiments to give recommendations for choosing appropriate val-

Table 6.4: DBSCAN and k-NN Classifier Exec. Time: AnyRTree vs LiarTree on M32 dataset

		With limit on no. of mcs	**Without limit on no. of mcs**
	No. of mcs	0.5M	1.53M
	Purity of mcs	0.85	0.99
LiarTree	**Memory Occupied by the tree**	521 MB	1404 MB
	DBSCAN Exec. Time	763 sec.	1816 sec.
	k-NN Classifier Exec. Time	312 sec.	803 sec.
	No. of mcs	0.5M	1.39M
	Purity of mcs	0.92	0.99
AnyRTree	**Memory Occupied by the tree**	559 MB	1256 MB
	DBSCAN Exec. Time	621 sec.	1396 sec.
	k-NN Classifier Exec. Time	264 sec.	617 sec.

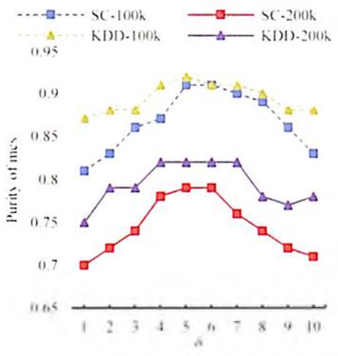


Figure 6.9: Purity of leaf level mcs generated by AnyRTree with variation in δ

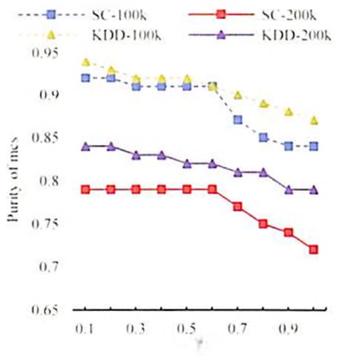


Figure 6.10: Purity of leaf level mcs generated by AnyRTree with variation in γ

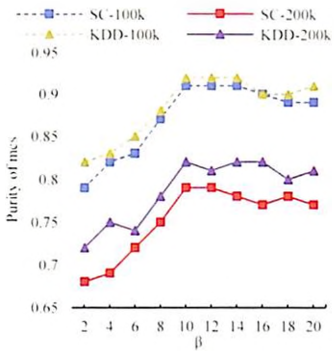


Figure 6.11: Purity of leaf level mcs generated by AnyRTree with variation in β

ues of user defined parameters - δ, γ and β. Please note that δ is the percentage of area expansion permissible to treat an object as concept (not noise) while inserting into a subtree; γ is the time interval after which we check for any possible noise to concept transition for a given entry of a node; β is the minimum number of objects to be aggregated into a noise buffer, before we start treating it to be concept. Figure 6.9, Figure 6.10, Figure 6.11 show the purity of mcs produced for SC and KDD datasets with variation in δ, γ and β values respectively, while keeping the other two to their default values. Two values of stream speed (λ) has been chosen - 100k & 200k. We can observe in Figure 6.9 that when the value of δ is too less, it is considering true concepts also as noise, which is distorting purity of mcs produced. And when it is high, the purity is distorted even then as noise points are considered as true concepts. So, value of δ between 5% to 10% is appropriate as observed for both the datasets at both the stream speeds. Regarding the value of γ, Figure 6.10 shows that smaller the value of γ more purer are the mcs produced. However, until it becomes too high, the purity of mcs doesn 't have much change. So value of 0.1 to 0.5 sec. is appropriate. One can choose this depending upon the stream speed as well. The value of β, when too low or too high, reduces purity (see Figure 6.11). It has to be optimal. This is because at low values, noise is immediately made concept. So, it has to be sufficiently high to get high purity. Typically one can choose β to be equal to the *min_pts* paramater used in offline DBSCAN clustering. Value above 10 is recommended as observed for both the datasets. Please note that the above recommendations have been used and worked well with other datasets as well.

6.4.2 Experiments on Any-MP-Clus

In this section we conduct experiments to evaluate ANY-MP-CLUS. We use ClusTree, LiarTree and AnyRTree in the online phase, for its evaluation. In all the experiments presented in this subsection, we process the entire dataset and then take the results. The value of the parameter f, which controls the granularity of mcs in windows of TTWF, is chosen to be 1.1. The value of t_m, i.e. the duration of window w_1 in TTWF has been chosen to be 2 seconds for FOF dataset and 10 seconds for M1B dataset. The value of τ (maximum radius threshold for w_1) has been chosen to be 0.005 for FOF and 0.05 for M1B. The value of q, which is the percentage of data objects communicated to other machines for median computations, has been chosen to be 10%. We evaluate the quality of mcs produced over FOF dataset using with average purity of the mcs, using the class labels generated by DBSCAN. For M1B dataset, we use Silhouette co-efficient instead, as we couldn't generate its ground truth.

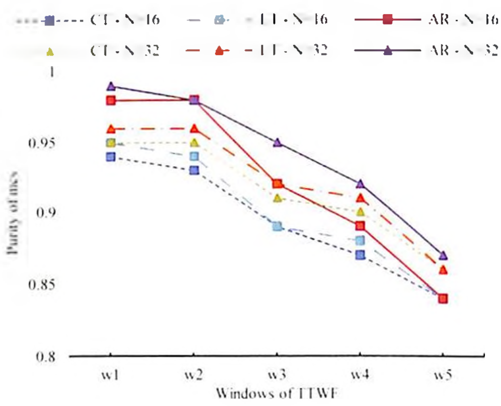


Figure 6.12: Purity of mcs generated by AnyRTree vs ClusTree & LiarTree in various windows of TTWF over FOF dataset

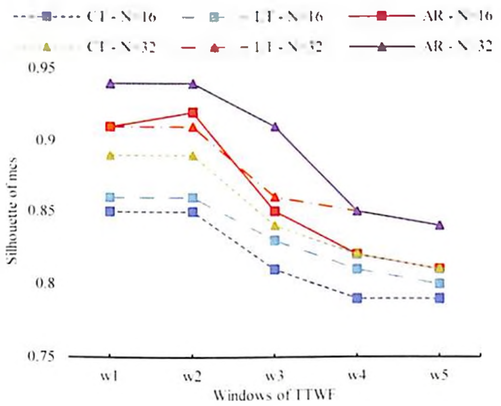


Figure 6.13: Sil. Co-eff. of mcs generated by AnyRTree vs ClusTree & LiarTree in various windows of TTWF over M1B

First, we measure the quality of mcs produced by ClusTree, LiarTree and AnyRTree in different windows of the TTWF for FOF and M1B datasets, while using 16 and 32 machines of the cluster. The number of machines of the cluster indicates the number of arriving streams. The stream speed has been fixed to λ=150k ops across all the machines. The results shown in Figure 6.12 & Figure 6.13 indicate that the quality of mcs produced in windows w_1 and w_2 are the highest and it decreases as we progress to the later windows. This is because of greater aggregation present in the later windows. The experiments also

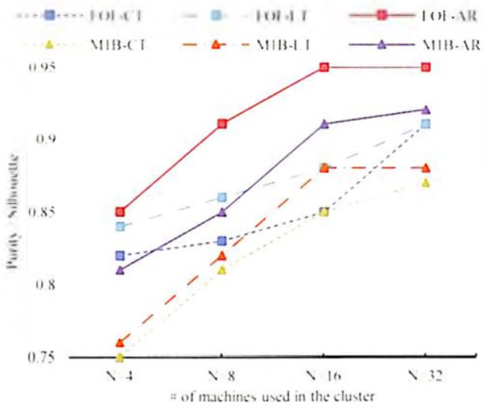


Figure 6.14: Quality of DBSCAN clustering for mcs produced by AnyRTree, ClusTree & LiarTree in windows w1, w2 & w3 over FOF dataset

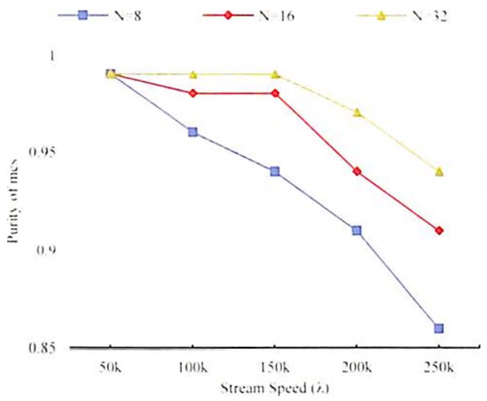


Figure 6.15: Purity of mcs in w_1 generated by AnyRTree at different stream speeds for varying number of machines over FOF dataset

establish the fact that AnyRTree has been consistently performing well in all the windows (especially in the latest windows), when compared to ClusTree and LiarTree.

Next, we measure the quality of clustering result produced by applying DBSCAN clustering over the mcs generated by AnyRTree vs ClusTree & LiarTree. We take all the mcs from windows w1, w2 and w3 (in case of all three trees), while varying the number of machines (streams). While performing DBSCAN over the mcs, we treat their means as stand-alone data objects and apply clustering over them. For DBSCAN clustering over FOF dataset, we choose ϵ=1 and min_pts=6. And for M1B, we choose ϵ=2 and min_pts=6. We measure quality of clustering for FOF using purity and for MPAGD1B using silhouette. The stream speed has been chosen to λ=150k ops across all the machines. The results are presented in Figure 6.14. The results show that the quality of clusters produced is greater for AnyRTree than the remaining two. Also, the quality of clustering increases with increase in number of machines due to increase in granularity of the mcs, which is resultant of increase in number of mcs processed.

Next, we measure the purity of mcs produced in window w_1 at different stream speeds, while varying the number of machines used in the cluster, for AnyRTree over FOF dataset. Please note that the number of machines used is equal to number of streams being captured. The results presented in Figure 6.15 show that with increase in number of streams, the purity of the mcs achieved also increases at all the stream speeds. This is because with increase in number of streams, the overall granularity also increases and

Table 6.5: Parallel Merging Time vs Sequential Merging Time

Strategy	Purity	Merging Time
Parallel Merging	0.9542	31 sec.
Sequential Merging	0.9551	341 sec.

thus achieving mcs of greater purity.

Next, we compare the efficiency of parallel merging with that of sequential merging. In parallel merging, we merge mcs using local merging and global merging steps as described in the offline phase of the proposed framework (See Section 6.3.1 on page 137). This parallel merging leverages a cluster of machines and hence is faster. In sequential merging, we instead transfer all the data objects from multiple machines to a single master machine and merge all of them over there. Table 6.5 shows the purity of mcs produced by both strategies along with the time taken to perform each of them over FOF dataset at stream speed of λ=150k ops, for a single window of duration t_{in}=60 sec. The results clearly show that the purity of mcs is almost the same in both the cases, whereas the total merging time using parallel merging over 32 nodes is very less when compared to that of sequential merging. Similar results were observed for other datasets as well.

6.5 Main Contributions

- We propose AnyClus, which is a framework for anytime micro-cluster maintenance of variable speed data streams that handles noise, captures concept drift, preserves spatial locality, and produces micro-clusters of greater quality. It addresses the drawbacks of ClusTree and LiarTree and hence produces mcs of greater quality.

- AnyClus uses a proposed variant of R-tree, *AnyRTree*, which is an indexing structure that stores micro-clusters at hierarchical granularities using R-tree spatial containment principles. AnyRTree enables the above features of AnyClus.
- We also propose a parallel framework, Any-MP-Clus, for anytime micro-cluster maintenance of multi-port streams over distributed memory architectures (cluster of computing nodes). This is the first such framework proposed. Any-MP-Clus also uses AnyRTree at each computing node to capture incoming streams.

6.6 Conclusions & Future Work

6.6.1 Conclusions

In this chapter, we proposed AnyClus & Any-MP-Clus, which are frameworks for any-time micro-cluster maintenance of single-port and multi-port data streams respectively. They use a proposed variant of R-tree known as *AnyRTree* for indexing the incoming stream objects (arriving with variable rate) in the form of micro-clusters of hierarchical granularity. Both AnyClus & Any-MP-Clus store the micro-clusters in a logarithmic tilted-time window framework. Any-MP-Clus uses a simple parallel merging strategy that leverages the cluster of computing nodes to merge and aggregate micro-clusters.

The experimental results presented in Section 6.4.1 on page 143 show that AnyRTree - 1) *produces micro-clusters of greater quality and compactness,* 2) *captures noise and concept drift more effectively,* and 3) *preserves spatial locality more effectively leading to improvement in offline clustering performance,* when compared to ClusTree and LiarTree. The experiments also establish the efficiency and scalability of Any-MP-Clus.

6.6.2 Future Directions

In future, we can work upon changing the representation of both kinds of buffers to contain multiple micro-clusters, which can improve their quality. One can also design a similar parallel framework for anytime subspace clustering as well.

Part III

Data Distribution Strategies

Data Clustering is one of the most commonly used data mining technique for knowledge discovery. Clustering groups data into meaningful groups, known as clusters, such that the dissimilarity between objects belonging to the same cluster is minimized and dissimilarity between objects from different clusters is maximized [6]. A few commonly used clustering algorithms include - *partitioning based* clustering algorithms (K-means [26], K-mediods[209]), *density based* clustering algorithms (DBSCAN[34], OPTICS [35], SNN [210]), *hierarchical* clustering algorithms (SLINK [30], CLINK [6], ALINK [6], BIRCH[203]), *subspace* clustering algorithms (CLIQUE [211], MAFIA[37], ENCLUS [212], PROCLUS [213], ORCLUS [214], FINDIT [38]), *grid* based clustering algorithms (STING [144], CLIQUE [211], MAFIA [213], WAVECLUSTER [215]) etc. These algorithms are commonly used in many applications such as satellite image segmentation [216], noise filtering and outlier detection [107], clustering of bio-informatics data [217], finding halos in cosmology [218], prediction of stock prices [219], etc.

Due to advent of Big Data, there is a huge data deluge created as generation of data has become faster and cheaper. To discover knowledge from such data, parallel clustering algorithms have been proposed to work over distributed memory architectures. Their design is usually data parallel, where they distributed data among the available computing nodes of a cluster and process each chunk in parallel. A few such solutions include - parallel partitioning based clustering algorithms [29, 220, 27, 220, 28, 221, 222], parallel density based clustering algorithms [36, 223, 86, 224, 87], parallel subspace clustering algorithms [89, 225, 226, 227, 228, 229], parallel hierarchical clustering algorithms [88, 230, 33, 231], parallel grid based clustering algorithms [232, 233, 234], etc. These solutions are typically based on high performance computing frameworks such as MPI, Hadoop, Spark, etc., which run the algorithms on a cluster of computing nodes.

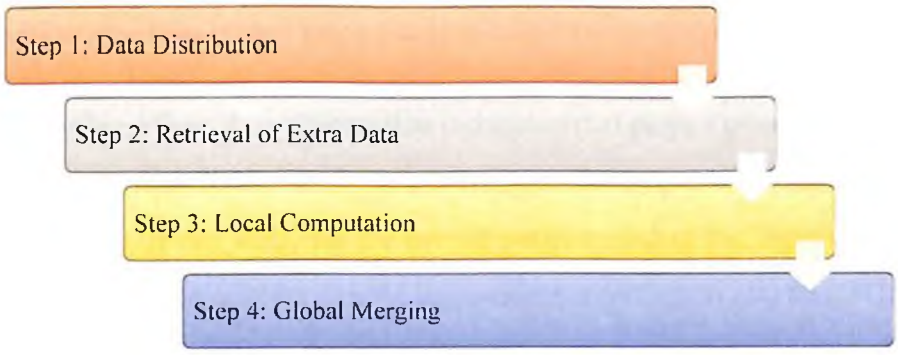

Step 1: Data Distribution
Step 2: Retrieval of Extra Data
Step 3: Local Computation
Step 4: Global Merging

Figure III.1: Workflow of a Parallel Clustering Algorithm

A typical parallel clustering algorithm has its work-flow as shown in Fig. III.1 in the previous page. It has the following steps:

1. *Data Distribution*: In this step, the data is distributed among all the computing nodes of the cluster depending upon the design and requirements of the algorithm. Distribution of data enables each computing node to process the data chunk allocated to it in parallel. Load balancing in terms of memory and computational overhead is an important criterion to be considered while distributing the data to achieve optimal speed up of the parallel algorithm. Commonly used distribution strategies are *random partitioning* and *spatial partitioning* (like kd-tree).
2. *Retrieval of Extra Data*: This is an optional step and is usually used by algorithms using spatial partitioning. In this step, each computing node fetches data, required for local computation, from other computing nodes of the cluster. This require inter-node communication. Some algorithms do not require any data from other computing nodes, for example - parallel K-means. Whereas some algorithms might require data from all other computing nodes, for example - parallel DBSCAN that uses random partitioning.
3. *Local Computations*: In this step, each computing node executes a local step for the portion of data allocated to it and produces a local clustering result. This step may require inter-node communication, depending upon the algorithm.
4. *Global Merging*: In this step, the local clustering result from all the computing nodes are merged either sequentially or in parallel to give a global clustering result. An algorithm may iterate over steps 2-4, if required by its design.

In the above workflow, data distribution is the step that plays a pivotal role in reducing the cost of data communication in the local computations step or in iterations global and local computations. It can influence the overall performance of the algorithm by several factors. Some parallel algorithms simply use random distribution [31, 88, 231] and some parallel algorithms that execute spatial queries (like neighborhood and nearest neighbor queries) use a distribution that preserves spatial locality of some points or majority of points (except the boundary points) [235, 86, 120, 236, 36, 33, 230]. By preservation of spatial locality, we mean that for a given data point p, the data points surrounding p are available locally in the same compute node. Spatial locality also leads to minimum possible overlap in the search space of different computing nodes. Consider Fig. III.2 on previous page, which shows three kinds of distributions over four computing nodes - A, B, C and D. The first distribution (a) has no preservation of spatial locality in distribution of data points to different machines. The search space of all the machines overlap a lot and is almost the same. In the second distribution (b) the data points are somewhat spatially organized. However, there are some overlaps in their search space. In the third distribution (c) the data points of are perfectly spatially organized with no overlap in the search space of the machines.

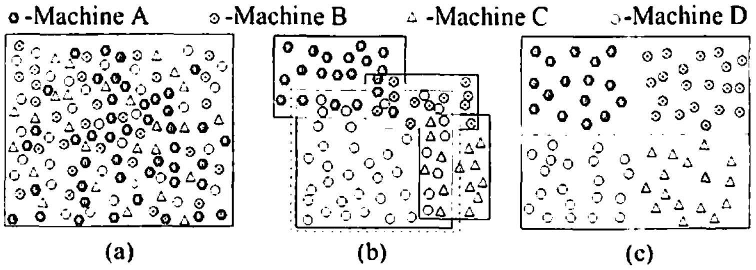


Figure III.2: Spatial locality preservation (a) None (b) Moderate (c) High

Preservation of spatial locality in the distribution helps in reducing the communication cost as well as the computation cost involved in steps 2,3 & 4 of the algorithm. Consider the case of parallel DBSCAN. In the local computations step, this algorithm executes ϵ-neighborhood queries for all the points in every computing node. Consider the computing node A of Figure. III.2. To compute ϵ- neighborhoods correctly for all the points in it, we shall require to fetch extra data points from all other computing nodes in case of Figure III.2 (a), as *epsilon*-boundaries for some points of node A are overlapping with the search spaces of all other computing nodes. We can clearly see that there no spatial locality maintained in the distribution shown in Figure. III.2 a. For distribution shown in Figure III.2 (b), a lesser number points shall be needed to be retrieved, and still fewer points in case of distribution III.e (c). This is because spatial locality is best preserved in distribution (c). Thus, a good spatial distribution helps in reducing inter-node communication of step

2, make local computations in step 3 efficient, and as well make the merging step (step 4) efficient.

Literature reveals only very few distribution strategies used in parallel clustering algorithms. They include - *random distribution* [31, 88, 231], *kd-tree distribution* [235, 86, 120, 236, 36] and *grid/cell-based distribution* [237, 33, 230, 224]. Initial approaches to parallel clustering had used random partitioning [238, 239, 240, 241, 242, 88]. However, researchers soon realized that it become a performance bottleneck for many clustering algorithms like density based and hierarchical clustering algorithms. So recent algorithms use spatial partitioning to get more efficient design of the parallel algorithms [86, 224, 36, 230, 89]. Other algorithms which don't rely on spatial locality principles simply use random distribution.

Research Gap & Motivation

Here are a few drawbacks associated with the existing data distribution schemes:

- Although the above distribution schemes are being used for parallel data mining algorithms, they are not specifically designed for such use. Also, they don't capture any specific data access patterns associated with any typical spatial parallel clustering algorithms. One can make tailor made partitioning scheme for a class of algorithms like density based or hierarchical clustering algorithms, which specifically capture the design requirements of that respective class.

- Although, the above data distribution schemes achieve good spatial locality, the distribution scheme however is static. That is, once the partitioning is done, it can't be incrementally updated. We shall have to redo the entire partitioning to include new data points. This renders them unfit for usage in dynamic incremental datasets.

- Another drawback associated with these schemes is that they require to scan (and store sometimes) the entire data into memory for computing the partition boundaries. It would thus become very expensive to use them for distributing large datasets. This is because, memory in a single machine may not be sufficient to accommodate the entire dataset, and thus leading to a lot of disk I/O. To the best of our knowledge, in only one instance, kd-tree like distribution has been done for distributing large data using sampling [223] for eliminating disk I/O. This partitioning scheme is also static.

To, address the above limitations, we present two kinds of data distribution schemes in this part. Chapter 7 on the following page presents a few data distribution schemes for large static datasets, while addressing the specific data access patterns of density based and hierarchical agglomerative clustering algorithms. It also presents a few distribution schemes for datasets that don't fit into main memory, while eliminating the need for disk I/O. Chapter 8 on page 185 presents a dynamic distributed data structure known as DD-RTREE, which is meant for data distribution for large dynamic incremental datasets.

Chapter 7

Data Distribution for Large Static datasets

In this chapter we present a few data distribution strategies proposed for distributing large static datasets over a cluster of computing nodes. We evaluate each of these in terms of performance of parallel clustering algorithms. Before presenting the methods and experimentation, we give a brief survey of parallel clustering algorithms proposed in literature for distributed memory architectures, with specific discussion on the data distribution strategy they employ.

7.1 Survey of Parallel Clustering Algorithms

We survey the existing MPI based parallel clustering algorithms in four broader categories and are explained below. Table 7.1 on page 168 summarizes the data distribution strategies used by these algorithms.

- Jagat Sesh Challa, Poonam Goyal, Nikhil S, Amogh Sharma, Sundar Balasubramaniam, Navneet Goyal, "Effective Data Distribution Strategies for Parallel Spatial Clustering Algorithms based on MPI", To be submitted to ACM TKDD.

7.1.1 Parallel partitioning-based clustering algorithms

Partitioning based clustering algorithms are those algorithms that create k number of partitions in the data, while reducing the inter-cluster similarity and increasing the intra-cluster similarity of data points in each cluster. The clustering is performed iteratively, until the it converges to the expected thresholds on distances specified above. The most basic partitioning based clustering is the *K-means* algorithm [6]. It starts with k centroids and assigns all the points in the dataset to their nearest centroid. Then the centroids are updated with respect to the membership obtained. The above steps are iteratively repeated, wherein we get new centroids after every iteration. The iterations continue until a threshold is reached on the inter-cluster similarity. Variations of k-means clustering include k-medians [243], k-mediods [29], Fuzzy C-means [244], etc. k-medians computes medians at every iteration instead of mean. k-mediods computes mediods, which are actual points in the dataset, after every iteration. This makes it more robust to noise and outliers than k-means. Fuzzy C-means is a variant of k-means, which performs soft clustering.

Parallelization of k-means algorithm is very straight-forward for an MPI based distributed memory architecture. Initially data is randomly distributed to each computing node and a list of global centroids is created, which is known to each computing node. In every computing node, data points are assigned to the nearest centroid and the centroids are updated. Then an average of all the centroids is taken globally leading to a new set of global centroids. Using these new global centroids the next iteration begins and in this way the algorithm goes on for subsequent iterations until a threshold criteria is met.

A few implementations of MPI based k-means algorithm that have been proposed include [28, 245, 246, 27]. All of them are based on the workflow explained above. [28] presented the basic parallel k-means clustering algorithm with the above workflow. [245] used parallel k-means to cluster large remote sensing data. [246] presented parallel k-means algorithm using coresets. [27] presented an efficient parallel algorithm for selection of initial seeds of k-means clustering.

A few implementations of k-means on MapReduce and Spark frameworks have also been presented in literature [247, 221, 248, 249, 250, 29, 251].

7.1.2 Parallel density-based clustering algorithms

DBSCAN is the most commonly used density-based clustering algorithm [34]. It finds clusters of data points with respect to two parameters $\epsilon(>0)$ and $MinPts(<0)$. It computes ϵ-neighborhood for each point in the dataset and labels the point as core, border, or noise. A core point initiates a cluster and the cluster is expanded by computing neighborhoods for points in the ϵ-neighborhood of the above core point, until no core point is found. This completes the expansion of the cluster. The next random point from the remaining unprocessed points is visited to extract another cluster and this process continues until all the points are processed. The time complexity of the DBSCAN algorithm that uses R-tree for neighborhood queries, is $O(n \log n)$ where n is the number of data points to be clustered.

Early approaches to parallelization of DBSCAN were based on master-slave computation model [238, 239, 252, 253, 254, 255, 256]. All these approaches simply distribute data to the slave computing nodes randomly without maintaining any spatial locality. However, they do maintain load balancing to ensure maximum speed up. The first such parallel approach to DBSCAN, known as PDBSCAN [238], uses *dR*-tree* for region queries. dR*-tree is a variant of R*-tree [257] in which R*-tree is replicated over multiple computing nodes for efficient data access on a distributed system. The next approach [239] gave various optimizations to DBSCAN and proposed a parallel version too. This parallel version distributes data over multiple computing nodes randomly, performs local clustering and then the local results are merged to get global clustering at the master node. The next approach [252] gave a parallel DBSCAN algorithm in which DBSCAN is divided into two major operations: clustering assignment and neighborhood querying. Master node performs clustering assignment while all slaves perform neighborhood queries in parallel for the data partition they have. Random partitioning is used here as well. [253] also presents a similar master-slave model based parallel DBSCAN in which each slave keeps a copy of an R*-tree. It is very much similar to the previous approach. [256] also gave a parallel DBSCAN, named as P-DBSCAN, which distributes the data among several nodes, builds Priority R-tree on each node, runs local DBSCAN, and aggregate the local results to get global clustering results. Priority R-tree is a variant of R-tree which performs efficient region queries. It uses a kind of projection based distribution for data partitioning, which is spatially disjoint.

The major drawback of master-slave model is the sequential data access pattern and serialized computations which affects the scalability of the parallel algorithm. All the approaches described above incur high communication cost between master and slave nodes. The parallelization during the merging phase is also limited. And most importantly they don't exploit the spatial locality exhibited by neighborhood queries used in DBSCAN clustering during their data distribution phase and most of them simply use random distribution. All of the above reasons render the master-slave model for DBSCAN inefficient.

The first approach to DBSCAN that breaks the sequential data access pattern in a solid way is PDSDBSCAN [86]. To do so it uses union-find (UF) data structure, which also makes it amenable to parallelization and achieves better scalability. The authors presented the parallel versions PDSDBSCAN-D and PDSDBSCAN-S for distributed and shared memory systems, respectively. The same authors have given two more hueristic-based approximate DBSCAN clustering algorithms - Pardicle [87] and BD-CATS [223], which are based on PDSDBSCAN-D. These two algorithms are capable of processing massive datasets with some approximation in the results. All the above algorithms use kd-tree based data distribution, perform local clustering at each node, and then the local clusterings are merged into a global clustering output.

Recently, a grid-based parallel implementations of DBSCAN, HPDBSCAN [224], has been presented. In this algorithm, following an initial random distribution, points are redistributed to computing nodes using a cost heuristic, thus achieves a distribution similar to a grid. Then local computations are performed on each computing node and then the results are merged into global clustering. Another grid-based DBSCAN, GridDBSCAN [36], has been proposed recently which reduces the total number of neighborhood queries as well as the search space for each query, while producing exact DBSCAN clustering output. It uses Grid-R-tree (described in Chapter 2 on page 19) for efficient computation of cell-wise neighborhoods. GridDBSCAN is parallelized for distributed memory, shared memory and hybrid architectures. It uses kd-tree based partitioning for data distribution, performs local GridDBSCAN clustering on each node and merges the local clusterings in a tree-parallel way. The experimental results claim better scalablity and run-time than the previously proposed MPI based exact clustering approaches.

Apart from MPI based solutions we have solutions for GPGPU based systems, MapReduce/Hadoop, Spark and hybrid systems. A few GPGPU based parallel DBSCAN implementations are proposed in [237, 258, 259, 260]. A few Map-Reduce based implementations are proposed in [261, 262, 263, 264, 265, 266]. Similarly, a few Spark based implementations are proposed in [267, 268, 269, 270, 271, 272].

7.1.2.1 Parallel OPTICS

OPTICS (Ordering Points To Identify the Clustering Structure) is a hierarchical density-based clustering algorithm [35]. OPTICS addresses major limitation of DBSCAN, i.e., the problem of detecting meaningful clusters in a dataset that has varying density. OPTICS gives an overview of the cluster structure of a given dataset with respect to density and contains information about every cluster level. For this, OPTICS generates a linear ordering of points where spatially closest points are arranged as neighbors. Additionally, for each point, a spatial distance (known as reachability distance) is computed which represents the density. Once the ordering and the reachability distances are computed using ϵ and *MinPts*, we can now generate clusters for a particular value of ϵ' (known as clustering distance), where $\epsilon' \leq \epsilon$.

The first parallel version of OPTICS clustering was proposed in [235]. They re-engineered the algorithm using Prim's MST algorithm and presented a variant called *MST-OPTICS*. MST-OPTICS breaks the sequential data access pattern of OPTICS algorithm and makes it amenable to parallelization. They present POpticsD which is the parallel version that works over distributed memory architectures. POpticsD uses random distribution and doesn't rely on spatial locality in distribution. On each partition, a local MST is constructed in the local computations phase, and all those MSTs are merged into a global MST. Clustering results obtained by this approach are comparable but not exactly the same as that obtained by classical OPTICS algorithm.

The next parallel approach is DOPTICS, which is presented in [236, 120]. DOPTICS is data parallel approach that uses spatial data partitioning using kd-tree. Data is then distributed among the processing elements and stored locally in R-trees [122]. At each processing element, OPTICS is run locally and a hierarchical merging of cluster-orderings is done to get the final cluster-ordering. The clustering results obtained are identical to classical OPTICS.

There are no known parallel implementations of OPTICS for MapReduce and Spark in literature.

7.1.2.2 Parallel Shared Nearest Neighbor Clustering

Shared Nearest Neighbor Clustering (or SNN) is a density-based clustering algorithm that uses a similarity measure known as *SNNSimilarity* [210]. SNN-similarity for two points is defined as the number of shared neighbors if they are in each other's nearest neighbors lists. A DBSCAN like algorithm is applied over the core points (using SNNsimilarity) to identify clusters of arbitrary size and shape and filters out noise/outliers. It is especially suited for high dimensional data.

To the best of our knowledge there is only attempt of parallelization of SNN for MPI based clusters [230]. The authors also present parallel versions for shared memory and hybrid architectures. They first presents R-SNN algorithm which is a modification to the classical SNN algorithm. R-SNN uses uses R-tree for nearest neighbor computations and is more optimized in terms of memory requirement. It is a single pass algorithm and processes data cluster-wise. A SPMD (Single Processor Multipe Data) based parallelization of R-SNN, *Parallel R-SNN*, is then presented. Parallel R-SNN uses kd-tree partitioning for data distribution, then local computations are performed over each partition and then the local results are merged to a global clustering. The spatial partitioning ensures good load balancing and makes the merging step efficient.

A parallel JP-Clustering algorithm for MapReduce framework has also been proposed [273]. More recently a parallel version of SNN has been presented for MapReduce framework [274]. A variant also exists for GPGPUs [275]. There is no known variant for Spark in the literature.

7.1.3 Parallel hierarchical clustering algorithms

Hierarchical clustering is also one of the popular techniques of clustering. There are two kinds of hierarchical clustering proposed in literature:

i. *Top-down*, also known as *Hierarchical Divisive Clustering* (HDC). It starts with considering all the points in a single cluster and then recursively splits the clusters until some criteria is met [6]. The criteria could be a limit on inter-cluster distances or on number of clusters.

ii. *Bottom-up*, also known as *Hierarchical Agglomerative Clustering* (HAC). It starts with considering individual point as a cluster and then repeatedly merges the closest pairs of clusters until one of the above criteria is met.

Figure 7.1: Dendrogram

The result of any of the above clustering techniques is a *dendrogram* (see Fig. 7.1), which is a tree-like structure showing the clusters agglomerated at each level. There are many variants of HAC such as AverageLINK [6], SLINK [30], CLINK [6], etc. But the most popular and widely used among them is the single-linkage or *SLINK* algorithm. HAC's variants differ from each other in terms of the ways the proximity distance between a pair of clusters is defined. SLINK algorithm has both time and space complexity of $O(n^2)$.

The early approaches of parallel hierarchical clustering were based on similarity matrix [240, 241, 242]. The first parallel hierarchical algorithm was presented in [240]. It has a time complexity of $O(n^2)$ and was based on single instruction multiple data model (SIMD) that uses shuffle exchange network to access similarity matrix and input data. The next approach was presented in [241], which uses re-configurable optical buses (AROB) architecture. The limitations of the above two approaches is that they are designed for specialized parallel architectures. A MPI-based approach was presented in [242]. In this approach, similarity matrix along with the data points is distributed across multiple nodes and then synchronized at each merging step. The clustering quality is dependent on the chosen input parameter threshold. The above similarity matrix based approaches incur a high communication cost for iteratively updating the similiarity matrix. This limits their performance & scalability, and renders them unfit to process large amount of data. Note that the above approaches don't employ any spatial partitioning scheme to distribute data among the processors.

In more recent approaches, SLINK algorithm has been viewed as an Minimum Spanning Tree (MST) problem and parallel SLINK as constructing MST in parallel. An MPI based distributed memory parallel clustering algorithm, known as CLUMP was proposed [231]. They have considered whole data as graph which is partitioned randomly into smaller sub-graphs composed of complete bipartite graphs, then computed MST for each sub-graph. These local MSTs are merged to get the final MST. The basic idea is to minimize the communication cost at the expense of redundant computations. Another approach was presented in [276], which gave a parallel hierarchical algorithm using parallel Euclidean Minimum Spanning Tree (EMST) for AROB distributed memory and PRAM shared memory systems. These algorithms assume uniform distribution of data points, which allows partitioning of the data space into uniform grids.

The next parallel MPI based algorithm was PINK [88], which is similar to CLUMP. They also minimize the communication cost by decomposing the problem into sub-problems, which removes redundant computations at the same time. This approach partitions the data into k equal partitions randomly and assign each $\binom{k}{2}$ possible combinations to various nodes for cross-edge and self-edge computations. At each node a local clustering is performed where MSTs are computed locally. Then these MSTs are merged into a global MST resulting in the final clustering. A similar algorithm known as SHRINK [31] was also presented for shared memory systems. Both the above approaches use disjoint set data structure for merging clusters at each iteration.

The Sibson's SLINK algorithm does not take into account *spatial locality* of data. And thus all its parallelizations use random distribution of data. An efficient HAC algorithm known as Partially Overlapping Partitioning (POP) has been proposed which exploits spatial locality [277]. The pPOP algorithm [278] is a parallel implementation of the POP algorithm for shared memory architectures. It uses a partially overlapping partitioning scheme for data distribution.

The most recent parallel version SLINK is *dGrid*SLINK [33, 32]. It is a parallel version of GridSLINK algorithm (proposed by the same authors). *Grid*SLINK exploits spatial locality of data using adaptive gridding, and reduces the number of distance calculations, while producing exactly the same dendrogram as that of classical SLINK. *Grid*SLINK has been parallelized for distributed memory (*dGrid*SLINK), shared memory (*sGrid*SLINK) and hybrid architectures (*hGrid*SLINK). *dGrid*SLINK ensures load-balancing by spatially distributing equal amounts of data to multiple nodes using a spatial distribution (which we call as CD-Split). After data distribution, at each node GridSLINK is executed leading to local MSTs. Local computations in GridSLINK are more optimal than in PINK as this exploits spatially locality attained by grid. Then the local MSTs are merged into a global MST in a tree-parallel way to get the final dendrogram.

A few GPU based implementations of parallel hierarchical clustering have also been proposed in literature [279, 280]. Apart from these a few Map/Reduce and Spark based implementations have also been proposed [281, 282, 283, 284].

7.1.4 Parallel subspace clustering algorithms

Subspace clustering algorithms are specifically designed for processing high dimensional datasets. It is possible that data points might have been drawn from multiple subspaces and membership of points to those subspaces is not known. Another problem associated with processing of high dimensional data is the "curse of dimensionality". The conventional similarity measures become unfit for processing such high dimensional data. Subspace clustering algorithms are a solution to the above problems. They cluster data into multiple subspaces and find a low dimensional subspace fitting each cluster.

There are two kinds of subspace clustering algorithms - *top-down* and *bottom-up*. The top-down subspace clustering algorithms produce highly disjoint clusters since they use partitioning based clustering approaches. A few top-down subspace algorithms include - PROCLUS [213], ORCLUS [214], FINDIT [38], δ-Clusters [285], COSA [286], and LAC [287]. Bottom-up subspace clustering is similar to finding frequent itemsets using apriori principle. Clusters are first found for each single dimension, and then dimensions are added for finding clusters in higher dimensions in the same way as that of apriori. Dimensions are added until cluster quality is preserved. The anti-monotonic property is used to prune away infrequent or irrelevant subspaces. Commonly used grid based bottom-up subspace clustering algorithms include CLIQUE [211], MAFIA [37], ENCLUS [212], SCHISM [288], and CBF [289]. Also, there are a few density based bottom-up subspace clustering algorithms that include - SUBCLUE [290], FIRES [291], DUSC [292], INSCY [293], and SUBSCALE [294].

Literature reveals very few approaches to parallel subspace clustering on distributed memory architectures. The first such approach is parallelization of MAFIA known as PMAFIA [225]. This algorithm is a data parallel algorithm in which data is distributed over multiple processing elements randomly and local computations are performed on each processing element. The results are then merged into a global output. A GPU based parallelization of MAFIA has also been presented in [227].

More recently a parallel framework [89] has been presented for grid-based bottom-up subspace clustering algorithms like CLIQUE, MAFIA, ENCLUS, SCHISM and CBF. This framework has five major steps- 1) gridding, 2) finding dense units, 3) candidate unit/-subspace generation for next iteration, 4) Steps 2 and 3 are repeated until no dense units are found, 5) cluster extraction. These steps are common to above bottom-up subspace clustering algorithms. The parallel framework first distributes the data randomly over the computing nodes and then every node executes steps 1, 2 & 3 iteratively. At each iteration, a local trie is generated at every node, which is communicated to the master to form a global trie for dense unit identification. This is repeated until the algorithm converges. Finally the clusters are extracted at the master node from the aggregates received.

The above approaches simply use random data distribution and do not rely on spatial locality. Hence we don't consider them for experimentation.

Apart from the above, the top-down subspace clustering algorithm LAC has been parallelized for shared memory architecture, which is known as PLAC [226]. A parallel version of SUBSCALE algorithm has also been presented for shared memory and GPU-based architectures [295]. A spark based parallelization of SUBCLUE algorithm, known as CLUS, is also presented in [228]. More recently, a grid-based parallel subspace clustering

Table 7.1: Data distribution strategies used by various parallel clustering algorithms

Algorithm	Year of Publication	kd-tree	Random Distribution	Others
Parallel K-means [28]	2011		✓	
[245]	2011		✓	
[246]	2013		✓	
PDBSCAN [238]	1999		✓	
[239]	2000		✓	
[252]	2001		✓	
[253]	2002		✓	
P-DBSCAN [256]	2010			projection-based
PDSDBSCAN-D [86]	2012	✓		
Pardicle [87]	2014	✓		
BD-CATS [223]	2015	✓		
HPDBSCAN [224]	2015			grid-based
GridDBSCAN-D [36]	2017	✓		
POpticsD [235]	2013		✓	
DOPTICS [120]	2015	✓		
Parallel RSNN [230]	2016	✓		
CLUMP [231]	2009		✓	
[276]	2005			grid-based
PINK [88]	2013		✓	
GridSLINK [32, 33]	2016			CD-Split
PMAFIA [225]	2000		✓	
[89]	2016		✓	

algorithm known as PSCEG [296] has also been presented for spark. A few MapReduce based parallel implementations are also proposed in literature [229, 297].

7.2 Data Distribution Methods

We now describe the data distribution strategies which include both existing methods (*Random* and *KD-Split*) and proposed methods (*Pbased-Split*, *PD-Split* and *CD-Split*). Most of the illustrated existing/proposed distribution strategies are only slightly different in their approach of partitioning. However, they cause a big effect on the overall execution of the parallel algorithms. We use the following terminologies: Let N be the size of the data, n be the total number of computing nodes or processors and d be the dimensionality of the dataset.

7.2.1 Random Partitioning

In random partitioning, data points are randomly divided to the computing nodes of the cluster. In practice, the first chunk of N/n data points are assigned to the first computing node, the next chunk to the second node and so on, which cannot be called as truly random partitioning. Load balancing is maintained in the distribution to achieve better

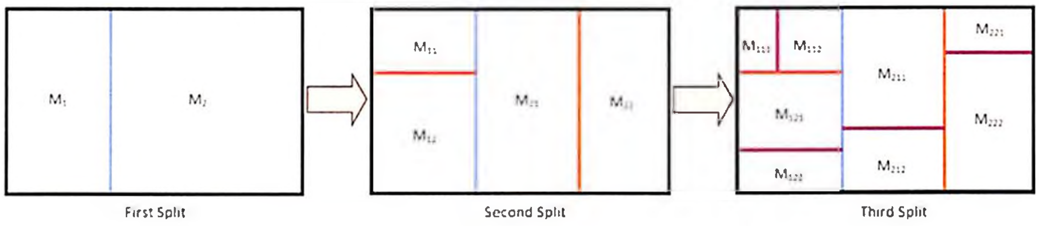


Figure 7.2: Kd-tree based data partitioning (KD-Split)

performance, i.e., each computing node gets equal number of data points. A few examples of the algorithms that use random partitioning can be found in - [88, 28, 235].

When random partitioning is used for density based and hierarchical clustering algorithms, we are not making best use of the inherent spatial pattern of their execution, leading to suffering of execution performance. It is better to use one of the spatial partitioning schemes explained below, as they capture inherent spatial patterns.

7.2.2 KD-Split

KD-Split or kd-tree based partitioning is the most commonly used spatial partitioning technique for distributing data to the computing nodes [36, 86, 120, 32, 230]. This technique recursively divides data among the computing nodes based on axis aligned split (see Figure 7.2). For every division, the splitting axis that has the largest spread is chosen and split is performed on the basis of the median for perfect load balancing (equal data points for every split). Recursive division continues until the total number of partitions is equal to the total number of computing nodes. Since load balancing is maintained at each split, each computing node will get equal number of data points. Figure 7.2 illustrates kd-tree based partitioning for n=8. It shows stage by stage splitting, where median for each split is chosen across the dimension that has largest spread.

7.2.3 Projection Based Split

Projection Based split (*Pbased-Split*), is our first proposed partitioning scheme. Initially, the axis with the largest spread is chosen. Then it recursively divides data into partitions on the basis of median. Each division is done along the same axis, unlike kd-tree where axis is chosen for every split. The recursive division continues until each partition or a cell contains a total number of points $< \tau$, where τ is parameter threshold. Figure 7.3 (Left

Hand Side) illustrates this split.

After the division is complete, all the cells formed are projected onto the axis chosen for splitting. The cells formed in Figure 7.3 (LHS) are projected over x-axis. This results in an ordering among the cells. Following this order, cells are packed together into non-overlapping groups (or partitions) in such a way that each group doesn't contain more than $\lceil N/n \rceil$ points (Figure 7.3 (RHS)). This scheme results in a load balancing which is very close to perfect load balancing. We can observe that the smaller the value of τ, the perfect the load balancing is.

This distribution strategy is an in independent generic method that can be applied to any parallel clustering algorithm. As it can be observed from the above description, the partitioning happens across only one dimension. This kind of partitioning reduces the number of machines to be communicated in steps 2 and 4 of any parallel clustering algorithm. This is because its boundaries overlap with lesser number of machines. However it may lead to increase in overall number of points transferred due to increase in perimeter of the boundaries. It is explained in the next subsection. Note that this method is more suitable to low-band width interconnects as it reduces the communication cost.

7.2.4 Parameterized Dimensional Split

Parameterized Dimensional Split or *PD-Split* is our second proposed partitioning scheme. This is specifically designed for parallel density based clustering algorithms [86, 36]. This partitioning scheme strives to minimize the communication overhead required during the local computations phase. A typical parallel density based clustering has the following execution layout:

- In step 1, data is distributed to the computing nodes using a spatial partitioning

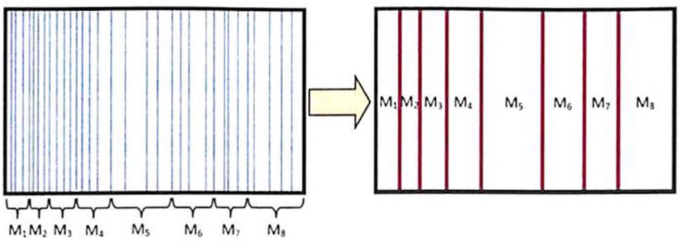

$M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8$
$M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8$

Figure 7.3: Projection based Split data partitioning (Pbased-Split)

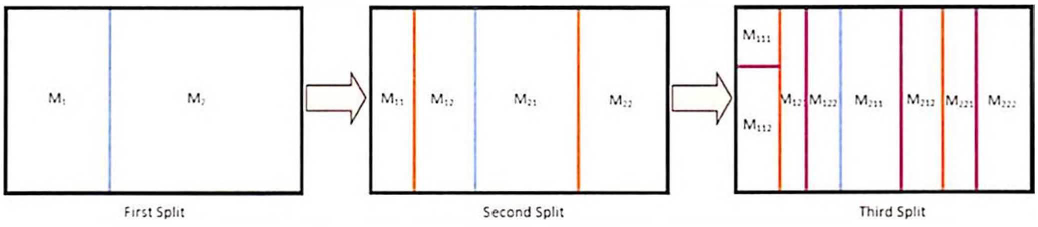


Figure 7.4: Parameterized Dimensional Split data partitioning (PD-Split)

scheme (typically KD-Split partitioning)

- In step 2, every computing node request for data points from other nodes, which are lying within ϵ-extended boundaries of the local node, where ϵ is a user defined parameter. This is depicted in Figure 7.5 on the next page for kd-tree partitioning, where Node M_{212} requests data points from Nodes M_{121}, M_{122}, M_{211} and M_{222}. These data points are required for computing exact ϵ-neighborhoods for the points lying near the boundaries of the local computing node. ϵ-neighborhoods of the points are required for DBSCAN and other density-based clustering algorithms in the next phase.

- In phase 3, local computations are performed where DBSCAN is performed on the local data with the help of additionally retrieved data for the neighboring computing nodes as explained previously.

- In phase 4, local clusterings are merged together to get global clustering.

Exploiting this execution layout, we try to minimize the communication overhead that occurs during phase 2 of the algorithm by changing the kd-tree partitioning scheme. Instead of computing the axis for splitting for each division, we let the splitting happen across the initially chosen dimension (like in Pbased-Split) until a threshold is reached. This time, however, threshold is on the width of the cell, along to the axis chosen. If a split is causing a cell's width to be $< 2\epsilon$, we then choose the next dimension for splitting, which has the current largest spread. Figure 7.4 illustrates this. In the first and the second splits, the splitting has occurred only along the x-axis. However, in the third recursive split, partition M_{11} was split along y-axis. This is because, the width of one of the cells resultant of splitting this partition along x-axis, is becoming lesser than 2ϵ. So the axis of split was changed.

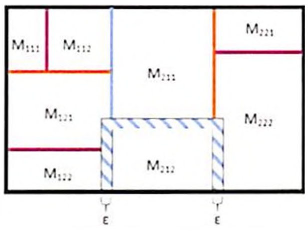


Figure 7.5: ϵ-extended regions for Computing Node M_{212} in case of kd-tree based split

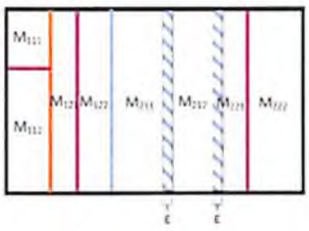


Figure 7.6: ϵ-extended regions for Computing Node M_{212} in case of PD-Split

Figure 7.5 and Figure 7.6 illustrate the the ϵ-extended strips (also known as *halo region*) for partition M_{112} for kd-tree based split and pd-split respectively. It is clear form the figure that the halo region spawn four partitions in case of kd-tree and only two partitions in case of PD-Split. When the dimensionality of the dataset increases, the number of nodes overlapping can even be more in kd-tree based partitioning as the axis for split keeps changing for every division. So, PD-split reduces the data required to be communicated in steps 2 & 4 of a parallel density-based clustering algorithm as it reduces the number of nodes to be approached for acquiring extra data points.

Note that the threshold on the width of each resultant partition has been chosen to be 2ϵ. This is because if the width of the partition becomes less than ϵ, the ϵ-extended strip might spawn to multiple partitions across the same axis. For example, in Figure 7.6, if width of partition M_{121} is lesser than ϵ, the ϵ-extended strip of M_{112} can spawn to machine M_{122} as well, which means that we are including all of M_{121} and some portion of M_{122} as well. This becomes huge communication cost. So, we restrict the width of each cell to be $> 2\epsilon$ and whenever a split can cause the width to go lesser than this value, we change our axis to split.

Note that such a restriction is not imposed on Pbased-Split which only splits across one axis. So, when ϵ value becomes large, Pbased-Split will have excessive communication as substantatiated by results presented in Figure 7.10 on page 179. Similarly for PD-Split also, it might happen that at larger values of ϵ that, although fetching points from lesser number of machines, the number of points fetched in total might exceed that for KD-Split. This is because large perimeter of intersection with the neighboring machines that has happened because of splitting along only one dimension. So, at larger values of ϵ, this problem could be present in PD-Split also, although not as severe as Pbased-Split. This is

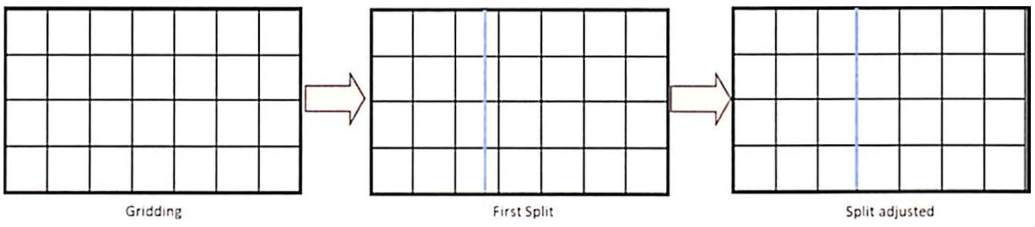


Figure 7.7: Sample division in CD-Split

substantiated by results presented in the above figure.

7.2.5 CD-Split

Cell based Dimensional Split or *CD-Split* is our third proposed partitioning scheme. This scheme has been specifically designed for grid based parallel SLINK (*dGridSLINK*) algorithm and has been introduced in our previous papers [33, 32]. dGridSLINK is the only parallel variant of SLINK that uses spatial distribution of data points.

The CD-Split partitioning is performed using gridding and median based split. It is similar to PD-Split, except that it additionally uses gridding. Initially a uniform virtual grid is overlaid on the entire data space, with an initially chosen cell size = $CellSizeinit/r$. $CellSizeinit$ is the cell size parameter of GridSLINK algorithm and $r(> 1)$ is a constant. For example, $CellSizeinit$ is calculated using the formula - $\sqrt[d]{\frac{RegionSize \times \tau}{N}}$, where $RegionSize$ is the volume of the data-space occupied by the points in the dataset, N is the size of the dataset and τ is a user defined threshold on maximum number of points we wish to keep in a cell. After gridding, we recursively split the data space into equal partitions by first splitting along one dimension, similar to PD-Split. Each split is a kd-tree like median split. However every time, the splitting axis is aligned with the nearest cell boundary as illustrated in Figure 7.7. The change in the dimension for splitting in case of CD-Split, however, is triggered by the cell size threshold, instead of ϵ-threshold like in PD-Split. The dimension for splitting is changed when the partition width can fit in only one cell across the current dimension. The total number of dimensions across which splitting is performed usually remains small, similar to PD-Split.

As mentioned before, CD-Split has been specifically designed for the dGridSLINK algorithm. The dGridSLINK algorithm internally performs local gridding in the local computations phase and performs the SLINK clustering using the local gridding. This local computations phase makes use of the initial global gridding performed and the partitioning performed with alignment to the grid boundaries (during the data distribution phase). This makes the local computations faster as inherently captures the design requirements of dGridSLINK. One can use KD-Split or Pbased-Split instead of CD-Split. However, the algorithm is expected to run slower for them as they don't do the split-axis alignment. This is substantiated by experiments presented in Section 7.4.3 on page 176. For more details of dGridSLINK algorithm, refer to [33].

7.3 Distribution Methods for very large datasets

The distribution methods described in the previous section, load the entire data into main memory for computing the splits of partitioning. However, while processing very large datasets (billions of floating points), the memory associated with the node performing the partitioning may not be sufficient enough to load the entire dataset. This makes those schemes unfit for distributing very large datasets. To handle such scenarios one can use sampling based techniques for data distribution. One such technique has been proposed in [223]. We name this technique as *A-KD-Split* and explain it as follows:

i. Randomly distribute all the data points to all the machines in the cluster.

ii. Randomly select a small fraction of data points from each machine and broadcast them to all other machines in the cluster.

iii. Every machine now has the same sample. Each machine now computes the first median for splitting over that sample.

iv. Every machine partitions the data into two sets, with one set on the left side of the median and the second on the right side of the median. The partition is performed along the axis that has maximum spread.

v. Then in a pair of two, machines exchange its left and right sets such that one machine gets the entire left half and the other gets entire right half.

vi. Now for all the machines that are on the left half, steps 2-6 are repeated recursively. They are also repeated for machines on the right half as well.

vii. Thus, this algorithm achieves disjoint partitioning in $\log n$ iterations, where n is the number of machines.

Note that this partitioning scheme may not lead to perfect load balancing. However, it is experimentally observed to give reasonably good load balancing. Also, note that the approximate versions of PD-Split (*A-PD-Split*) and CD-Split (*A-CD-Split*), are also designed in a similar fashion. In case of A-PD-Split, the nodes shall also have to additionally keep track of and communicate the dimension of splitting. In case of A-CD-Split, the initial virtual grid is to be calculated globally by inter-node communication, and a copy of the gridding information is to be broadcasted to each node. Then the splitting starts in a similar manner as that of A-KD-Split, except that the dimension across with the split has to happen shall change as per the cell size. Also, at every split, split-axis is aligned with the nearest cell boundary of the global grid. For the case of Pbased-Split, such an iterative distribution is not possible. The entire partitioning has to happen on the first taken sample and the data points are eventually distributed to their respective partitions, without any kind of iterative refinement. This may not result in good load balancing and hence we omit it for further discussion.

7.4 Experimental Results and Analysis

7.4.1 Experimental Setup

All experiments were conducted on the 32 nodes cluster infrastructure whose details are mentioned in Section 6.4 on page 141. All algorithms were implemented in C or C++ with MPI. The list of the datasets used for experimentation are given in Table 7.2 on the following page. The details of these datasets can be found in Section 2.3 on page 36.

The execution time for each experiment has been measured using `MPI_Wtime()` of `<mpi.h>` library. The default parameters chosen for experimentation are: for Pbased-Split, we choose the value of τ=1000; for approximate distributions based on sampling, we choose 10% points of the dataset as the sample. Note that for datasets of size > 20M, we have used approximate sampling based distributions.

We evaluate the proposed data distribution strategies in terms of (i) load balancing

Table 7.2: Details of datasets used for experimentation

Dataset	Size	Dimensionality	ϵ value for DBSCAN
3DSRN	434K	3	0.01
MPAGB8M3D	8M	3	2
MPAGD16M3D	16M	3	2
FOF57M3D	57M	3	3
MPAGD100M3D	100M	3	1
MPAHALO2.8M9D	2.8M	9	30

Table 7.3: Number of data points received by each computing node for various data distributions with variation in number of computing nodes (n), for FOF57M3D dataset

Distribution Scheme	Load Distribution	
	n=16	n=32
Random	3,561,887	1,780,944
KD-Split	3,561,887	1,780,944
PD-Split	3,561,887	1,780,944
Pbased-Split	3,541,062 - 3,571,329	1,721,712 - 1,813,961
CD-Split	3,498,032 - 3,638,541	1,597,254 - 1,862,171
A-KD-Split	3,397,251 - 3,795,134	1,584,754 - 1,922,658
A-PD-Split	3,344,652 - 3,786,249	1,571,113 - 1,911,904
A-CD-Split	3,285,412 - 3,799,763	1,523,624 - 1,924,521

achieved; and (ii) performance of various parallel spatial clustering algorithms. The results are presented as follows:

7.4.2 Load balancing achieved

Table 7.3 shows the load balancing achieved for each of the distribution strategy. Each value in the table denotes the number of data points received per computing node. As explained earlier, random partitioning, KD-Split and PD-Split achieve perfect load balancing. Pbased-split achieve near perfect load balancing because of the packing techniques as explained in Section 7.2.3 on page 169. Similarly, CD-Split also achieves near perfect load balancing as the splitting boundaries get aligned with grid/cell boundaries (as explained in Section 7.2.5 on page 173). Please note that similar load balancing has been observed for other datasets as well.

7.4.3 Performance of Parallel Spatial Clustering Algorithms

We now compare the performance of various parallel spatial clustering algorithms for the proposed distribution strategies. We compare for various versions of parallel DBSCAN, SNN and SLINK.

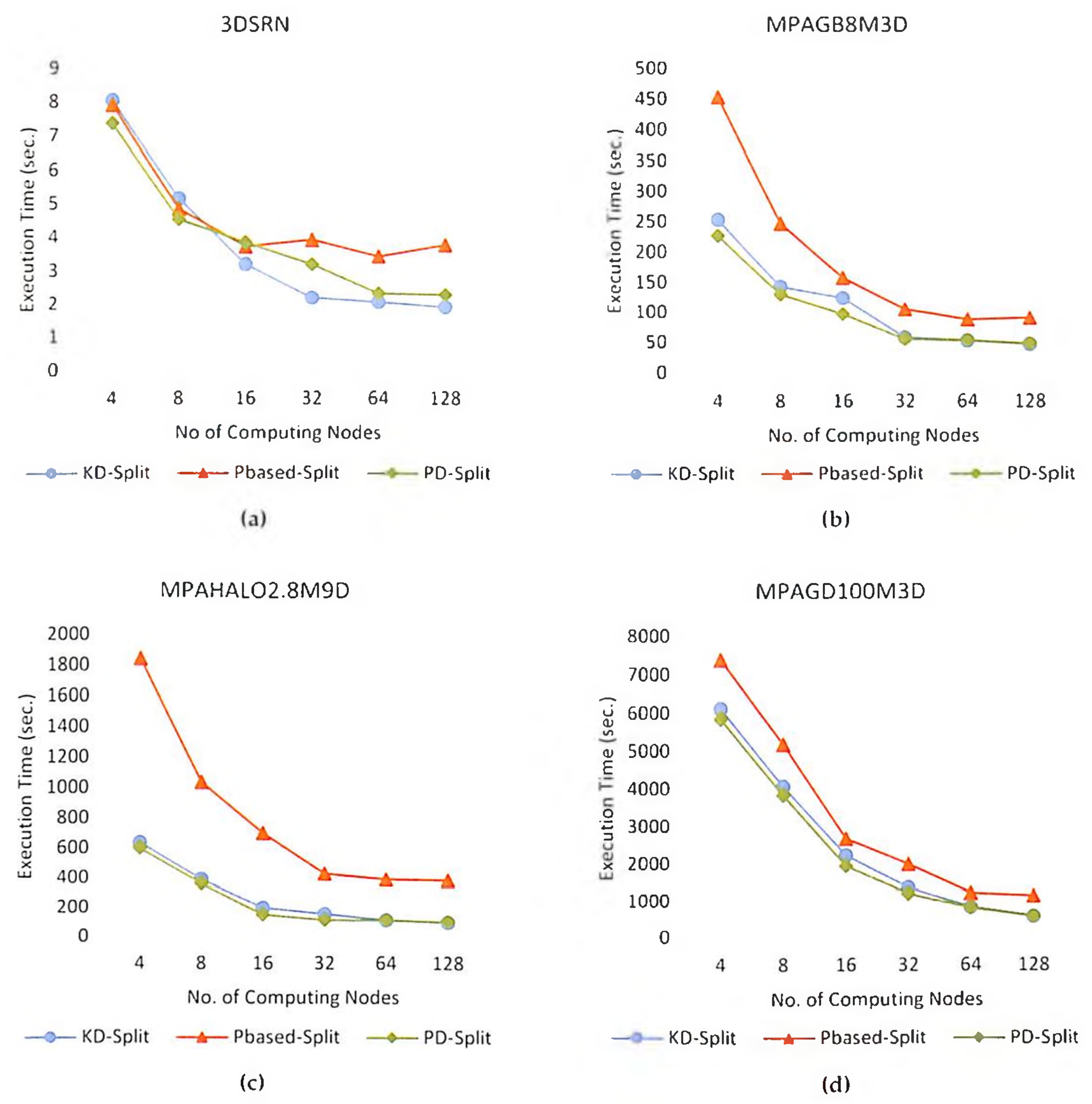


Figure 7.8: Performance of parallel GridDBSCAN algorithm for various data distributions

7.4.3.1 Parallel DBSCAN

We compare the performance of PDSDBSCAN-D [86] and GridDBSCAN-D [36] for various distribution strategies. The ϵ value for each dataset under experimentation has been given in Table 7.2 on page 176. The value of *Minpts* has been set to 5 for all datasets.

Figure 7.8 and Figure 7.9 on the next page present the performance of GridDBSCAN-D and PDSDBSCAN-D, for KD-Split, Pbased-Split and PD-Split distributions for various datasets executed over increasing number of computing nodes. The results show that PD-Split and KD-Split are competitive in execution performance. We can clearly observe that for lesser number of computing nodes, PD-Split is better than KD-Split. However, with

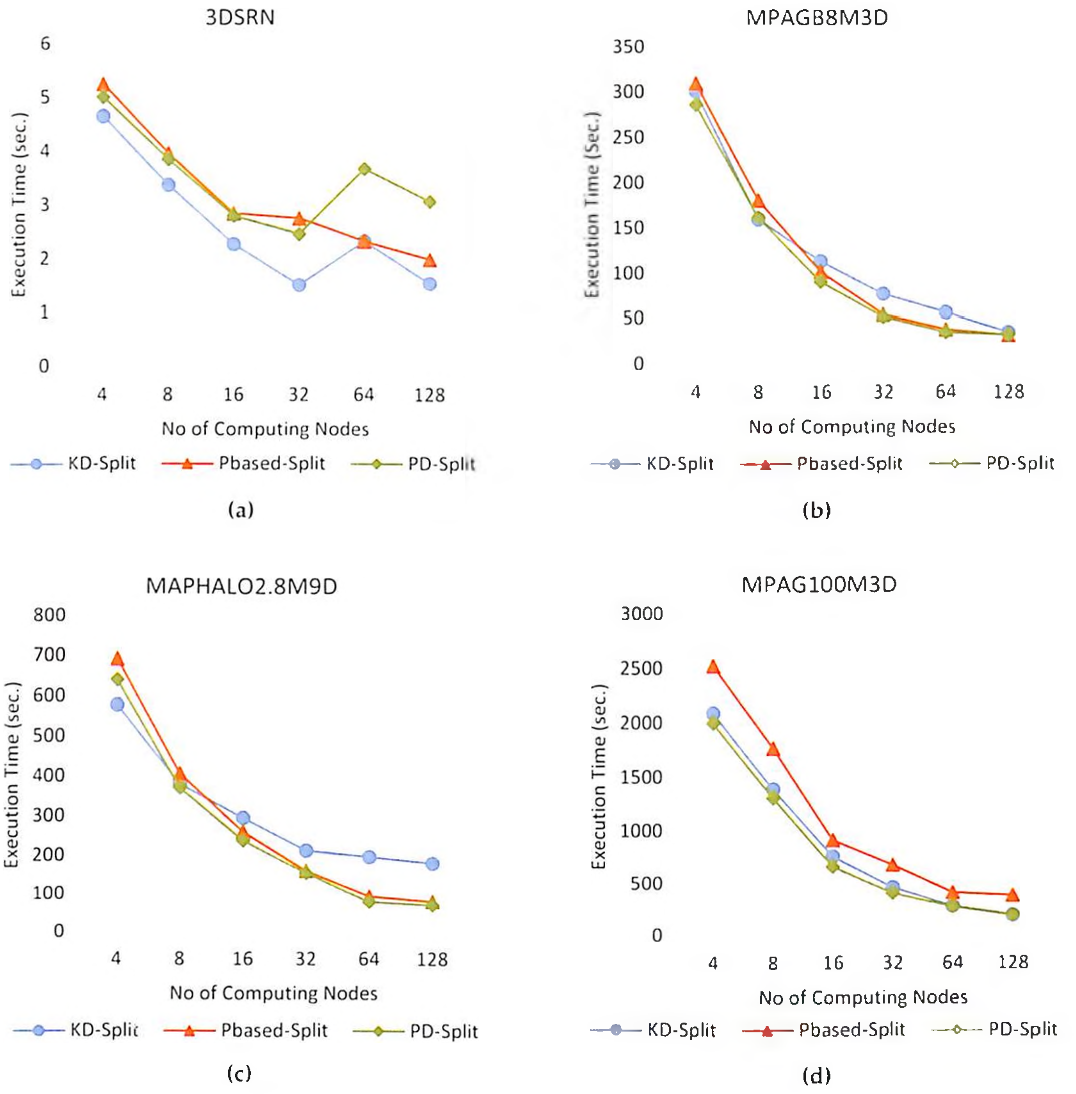


Figure 7.9: Performance of parallel PDSDBSCAN algorithm for various data distributions

increase in number of computing nodes, both them give almost the same performance, with KD-Split being slightly better (except for MPAHALO2.8M9D dataset). The result of MPAHALO2.8M9D dataset clearly show that PD-Split works much better than KD-Split for high dimensional data, even at higher number of computing nodes. This is because of reduced communication overhead in steps 2 and 4 of both the algorithms, as explained in Section 7.2.4 on page 170. This is also substantiated by the split-up time of various steps of the algorithms presented in Table 7.4 on the next page and Table 7.5 on the following page. PD-Split improves overall execution time as well as the execution time of each step of the algorithm. The results of 3DSRN dataset are erratic at higher number of computing nodes, because of insufficient data to be processed for such large number of processors.

Table 7.4: Split-up of execution times of various steps of GridDBSCAN-D for MPAGD100M3D dataset

	KD-Split	Pbased-Split	PD-Split
Data Distribution Step + Retrieval of Extra Points	26.58	37.23	19.33
Local Computations	1174.91	1673.6	1023.45
Merging Step	167.92	287.34	149.34
Total Time	1369.43	1998.23	1192.12

Table 7.5: Split-up of execution times of various steps of PDSDBSCAN-D for MPAGD100M3D dataset

	KD-Split	Pbased-Split	PD-Split
Data Distribution Step + Retrieval of Extra Points	26.58	37.23	19.33
Local Computations	376.23	508.72	305.53
Merging Step	92.51	138.01	79.23
Total Time	468.72	683.95	404.09

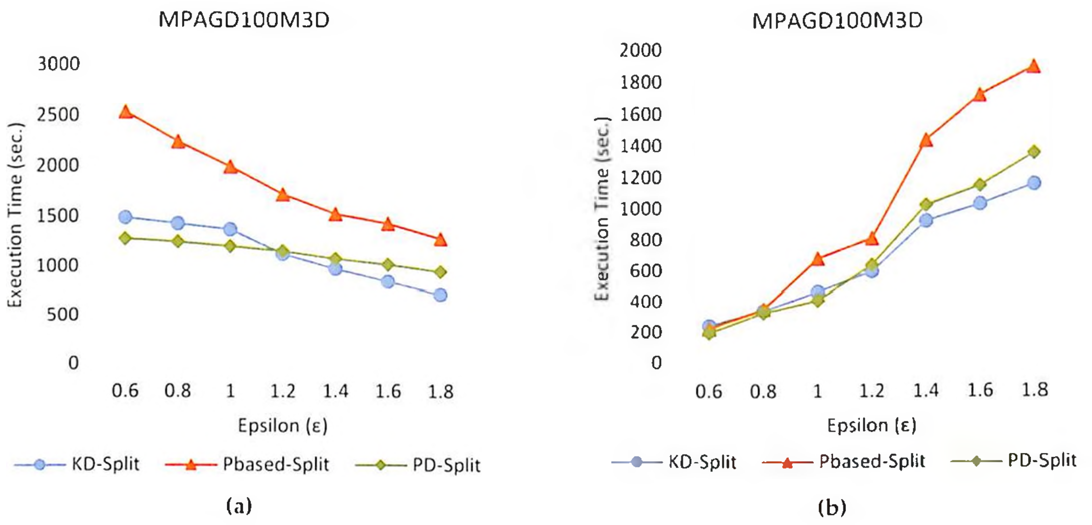


Figure 7.10: Performance of GridDBSCAN-D and PDSDBSCAN-D with variation in ϵ for various distributions over 32 computing nodes

Next, we conduct an experiment to measure the performance of both the parallel algorithms with variation in ϵ value. Figure 7.10a and Figure 7.10b present the results, which shows that PD-Split works better for lower values of ϵ. Whereas, KD-Split is found to dominate for higher values of ϵ. This is because at higher values of ϵ, the communication cost of Pbased-Split and PD-Split becomes higher. This is because the number of points lying within the ϵ-extended boundaries of the machines becomes large at higher ϵ. And it becomes more larger for Pbased-Split and PD-Split than KD-Split. This is also substantiated in Section 7.2.4 on page 170.

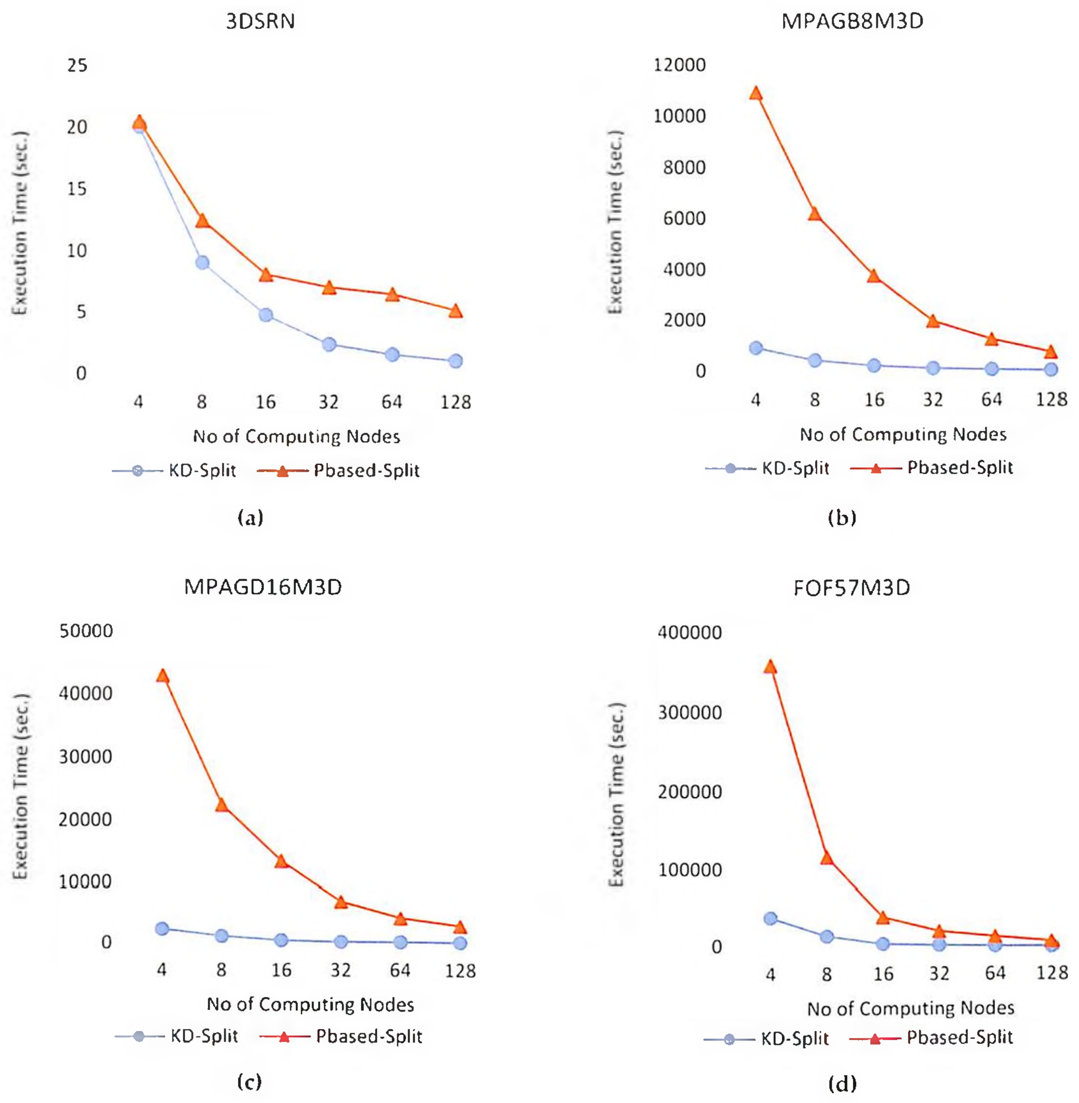


Figure 7.11: Performance of parallel dR-SNN algorithm for various data distributions with variation in number of computing nodes of the cluster

7.4.3.2 Parallel SNN

In this section, we evaluate the performance of dR-SNN algorithm [230] which is the only parallel SNN algorithm proposed for MPI based architectures. The values of the parameters chosen for experimentation are: k=30, ϵ=12 & $Minpts$=15, for all datasets. Note that ϵ of SNN is different from that of DBSCAN. It is a threshold on number of data points in case of SNN and a threshold on distance in case of DBSCAN. Figure 7.11 presents the execution time of dR-SNN for KD-Split and Pbased-Split distributions for various datasets executed over increasing number of computing nodes. The results clearly show that KD-Split approach has always been better than Pbased-Split. This is because of two reasons:

1) It uses K-NN queries inside the algorithm and K-NN query is more optimized for kd-trees, so naturally KD-Split is expected to do better in local computations phase; 2) The merging step in dR-SNN algorithm requires more communication in case of Pbased-Split. This is because in case of Pbased-Split the number of points that participate in the merging step is high. Both the arguments are substantiated by the split-up values presented in Table 7.6, which clearly shows the difference in local compuations step as well as merging step. Note that similar behaviour has been observed for different variations to the SNN algorithm parameters and other datasets as well.

Note that PD-Split and CD-Split are not applicable to dR-SNN algorithm as they are tailor-made for DBSCAN variants and GridSLINK respectively.

Table 7.6: Execution Time for various steps of dR-SNN algorithm for MPAGD16M dataset

	KD-Split	Pbased-Split
Data Distribution Step	49.81	63.84
Local Computations Step	171.86	5,192.13
Merging Step	27.40	1,661.48
Total Time	249.07	6,917.45

7.4.3.3 Parallel SLINK

Figure 7.12 on the following page presents the performance of dGridSLINK algorithm [33, 32] for KD-Split, Pbased-Split and CD-Split distributions for various datasets executed over increasing number of computing nodes. The value of τ, which dictates the initial cell size has been set to 300 as per the recommendations in the above papers. The results clearly show that CD-Split has always been better in all the cases. This is mainly because of the reduction of time in the global merging step which was possible by adjusts the partition boundaries to align with grid/cell boundaries. The split-up of execution time of various steps of the algorithm is presented in Table 7.7 on page 183 for MPAGD16M3D dataset. The results clearly show that CD-Split takes more time to distributed data. This is attributed to extra load of aligning splits with grid/cell boundaries. However, The time saved in local computations and merging step compensates for it. The merging time, especially is very low for CD-Split, because of the alignment of the split boundaries. On a whole, CD-Split is better than the remaining two. Similar behaviour has been observed for other datasets as well.

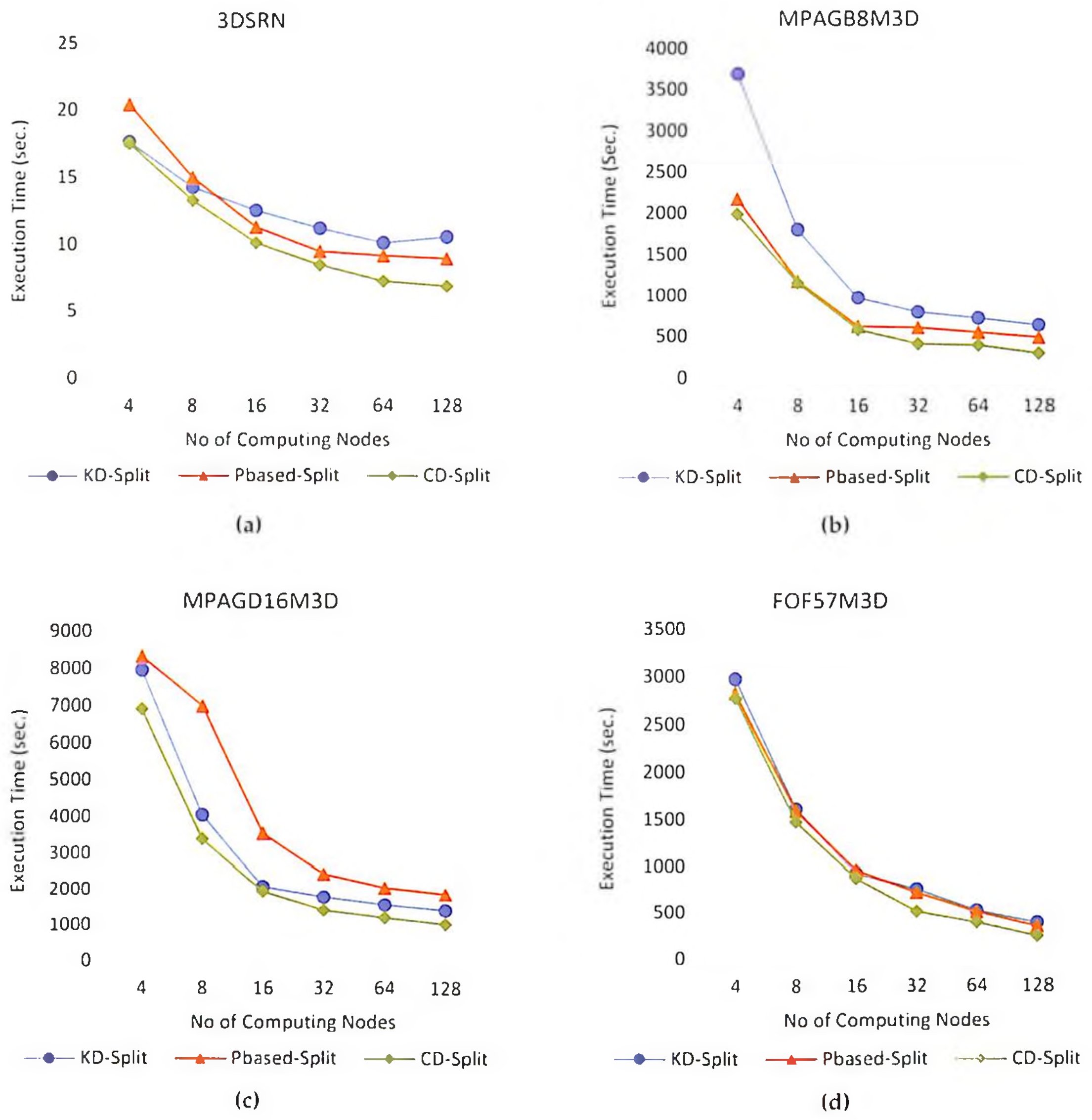


Figure 7.12: Performance of parallel GridSLINK algorithm for various data distributions

7.5 Discussion and Recommendations

Based on the above experimentation and results, we give the following recommendations regarding the usage of appropriate distribution methods for each of the above parallel clustering algorithms.

- For parallel DBSCAN (and DBSCAN like) algorithms, PD-Split and KD-tree based split are competitive. PD-Split is more suitable for smaller number of computing nodes and smaller values of ϵ. KD-Split is recommended to be used for larger values of ϵ.

- For parallel shared nearest neighbors clustering (dR-SNN), KD-Split always works bet-

Table 7.7: Execution Time for various steps of dGridSLINK algorithm for MPAGD16M dataset

	KD-Split	Pbased-Split	CD-Split
Data Distribution Step	49.81	63.84	59.23
Local Computations Step	1459.09	1961.98	1250.35
Merging Step	273.23	401.34	99.30
Total Time	1782.13	2427.17	1408.88

ter, as it K-NN based. Its usage is recommended at all times.

- For parallel GridSLINK algorithm, CD-Split has always been better than KD-Split and Pbased-Split. Its usage is recommended all the times.

- One can use Pbased-Split as a generic distribution scheme, free from parameters, when one wants to split across one dimension only. Pbased-Split also works good for high dimensional data in some cases (see Figure 7.9c on page 178).

7.6 Main Contributions

- We proposed three data distribution schemes - Pbased-Split, PD-Split and CD-Split, for distributing data over a cluster of computing nodes for executing MPI based parallel clustering algorithms.

- We also proposed approximate versions of the above schemes for distributing very large datasets that don't fit into the main memory for computing partition boundaries.

- We have also given appropriate recommendations for each of the distribution with respect to various parallel clustering algorithms.

- We have given very comprehensive literature survey of MPI based parallel clustering algorithms, with specific reference to the distribution methods they use.

7.7 Conclusions and Future Work

7.7.1 Conclusions

This chapter proposed three data distribution schemes - Pbased-Split, PD-Split and CD-Split, for distributing data over a cluster of computing nodes for executing MPI based parallel clustering algorithms. Data distribution is an important step of any data parallel clustering algorithm. To the best of our knowledge, data distribution is not yet discussed in existing literature.

The chapter also presented empirical evaluation of each of the distributions proposed for various parallel clustering algorithms that include - DBSCAN, SLINK and SNN, and gives suitable recommendations for usage of appropriate distribution scheme for the above algorithms. This chapter has also presented a very comprehensive review of parallel clustering algorithms.

7.7.2 Future Directions

Grid-based techniques can be exploited in future to design more efficient distribution strategies that are more efficient in run-time performance. Improving the run-time performance makes the parallel algorithms all the more faster and scalable.

Chapter 8

Data Distribution for dynamic incremental datasets

In this chapter we present DD-TREE which is a dynamic distributed data structure for indexing and distributing large incremental multi-dimensional datasets in a cluster of computing nodes. As discusses earlier, the distribution strategies proposed in the previous chapter are static, that they can not be incrementally updated with new set of data points. They shall have to re-do their entire distribution. DD-TREE addresses this issue as it supports dynamic incremental insertions. Before we present the proposed data structure, we give a brief review on the existing dynamic distributed data structures.

8.1 A Review on Distributed Data Structures

Distributed Data Structure (DDS) is a data structure that is used in a message passing system (typically a cluster of computing nodes). DDS is composed of a data organization scheme and a set of distributed access protocols to enable computing nodes to issue query and modification instructions and get appropriate responses. The data organiza-

---

- Jagat Sesh Challa, Poonam Goyal, Nikhil S., Aditya Mangla, Sundar Balasubramaniam, Navneet Goyal. *DDR-Tree: A dynamic distributed data structure for efficient data distribution among cluster nodes for spatial data mining algorithms.* In Proceedings of 2016 IEEE International Conference on Big Data (IEEE Big Data 2016), pp. 27-36, 5-8 December 2016, Washington DC, USA tion scheme acts like an index to the collection of local data structures that are stored at each computing node [298]. Many DDSs are proposed in literature for various domains including peer-peer network overlays, data analytics, social network mining, etc. [299, 298, 300, 301]. These data structures are typically used to index data for efficient query processing, routing, etc. Distributed versions of R-tree and its variants proposed in literature include - Parallel R-tree [302, 303], Distributed B-link tree [304], Distributed Random tree [299], Master-Client R-tree [305], Upgraded Parallel R-tree [306], SD-RTREE [307], etc. These data structures were originally proposed for database systems to improve the efficiency of various queries. Most of these focus on optimizing communication overheads and increasing degree of parallelism to get optimal query performance [302, 303, 304]. They achieve this by organizing data in such a way that multiple computing nodes can be simultaneously accessed to answer a query.

The above data structures were neither specifically designed for data distribution nor address any of the requirements of parallel spatial data mining algorithms. They don't target: preservation of spatial locality in their distribution, achieving good load balancing and giving optimal query performance at the same time. The above are the key requirements for any parallel spatial data mining algorithm [86, 235, 32, 36, 120, 33].

We now review the most recent dynamic distributed data structure - SD-RTREE. We use this structure for benchmarking the performance of the proposed DD-RTREE. SD-RTREE [307] is a hybrid structure based on AVL-tree [308] and R-tree [122]. Its structure is conceptually similar to that of a classical AVL tree, with its data organization principles borrowed from the R-tree spatial containment relationship. It is designed to reduce communication overheads in construction and querying. It supports dynamic insertions and shows good scalability. It supports both region and k-NN queries.

SD-RTREE, however, have a few drawbacks associated with its design. The re-distribution algorithm that handles node overflows, is based on k-NN search, which doesn't gaurantee good preservation of spatial locality. The re-distribution also happens point by point which makes it slow. Also, the data distribution of SD-RTREE doesn't guarantee good load balancing in practice. The communication cost involved in its construction is also high. So, it doesn 't suit our requirements.

The rest of this chapter is organized as follows: Section 8.2 on the next page presents

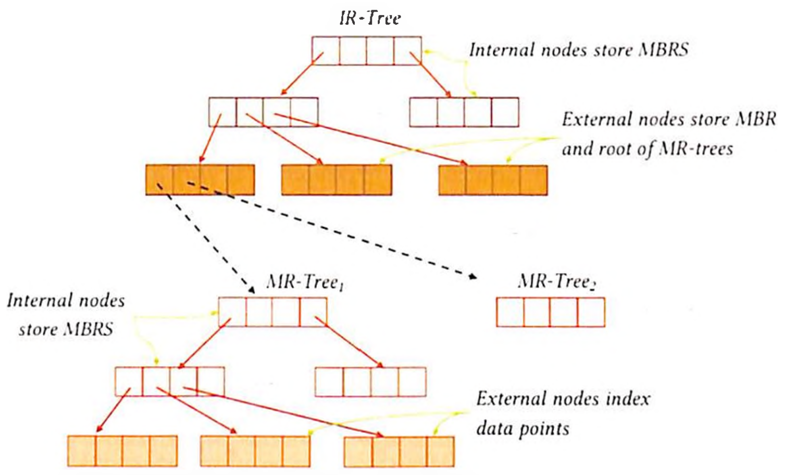

IR-Tree
Internal nodes store MBRS
External nodes store MBR and root of MR-trees
$MR\text{-}Tree_1$
$MR\text{-}Tree_2$
Internal nodes store MBRS
External nodes index data points

Figure 8.1: Structure of DD-Rtree

the DD-RTREE structure along with its operations and complexity analysis. Section 8.3 on page 193 presents the quality and performance evaluation. Section Section 8.5 on page 199 summarizes the main contributions of this chapter. Section Section 8.6 on page 200 concludes this chapter and gives future directions.

8.2 DD-Rtree

DD-RTREE is a dynamic distributed data structure that resides on a cluster of computing nodes. DD-RTREE is designed to distribute data across multiple computing nodes with the following objectives: maximizing spatial locality; achieving good load balance; minimizing inter-node communication for its construction, and minimizing execution time of queries and spatial data mining algorithms. DD-RTREE is first distributed spatial indexing structure which tries to achieve the above objectives. The design of DD-RTREE also makes it dynamic, i.e., data can be added incrementally and computing nodes can also be added incrementally, if required.

8.2.1 DD-Rtree design

The structure of DD-RTREE is illustrated in Figure 8.1. It comprises of R-trees at two levels. The first level R-tree is the index-R-tree (IR-TREE), which serves as the index to the entire structure and resides in a master computing node or a server from where all the instructions are issued. The second level comprises of multiple R-trees stored one each at each machine of the cluster (MR-TREE). MR-TREE indexes data points that belong to its machine. IR-TREE satisfies the following properties:

- Each node of IR-TREE has a minimum of Im and maximum of IM entries indexed in it, except the root which can have less than Im entries.
- Each internal node consists of MBRs which store the bounding information of all the objects indexed at their respective sub-trees.
- Each external node stores MBR information of all the points indexed in a machine. In other words, it stores the MBR of the root of an MR-TREE. It also stores the $machineID$ of the machine where that MR-TREE is stored and a count of points (cnt) indexed in it.
- Each external node also contains a buffer of a fixed capacity bc, that temporarily stores data points before pushing them into the corresponding MR-TREE.

MR-TREES are the R-trees with the native R-tree properties. Each machine has a capacity mc which is the maximum number of data points it can index.

Algorithm 8.1: DD-RTREE Construction

```
1 procedure CONSTRUCT-DD-RTREE ()
      Input  : List of Data Point $DL$
      Output: IR-TREE $ITree$ constructed
2     Initialize an empty IR-TREE $ITree$;
3     foreach point $p$ in $DL$ do
4         INSERT-IN-DD-RTREE ($p$, $ITree$);
5     end
6     foreach leaf $leaf$ in $ITree$ do
7         FLUSH-BUFFER ($leaf.buff$);
8     end
9     return ITree;
```

Algorithm 8.2: Insertion in DD-RTREE

```
1 procedure INSERT-IN-DD-RTREE ()
      Input  : Data Point $p$, IR-TREE $Itree$
      Output: $p$ inserted into $Itree$
2     $ILeaf$ = CHOOSE-LEAF ($p$, $ITree$);
3     Insert $p$ into $ILeaf.buffer$;
4     Update MBRs of $ITree$ in a bottom-up manner;
5     Increment $ILeaf.cnt$ by 1;
6     if $ILeaf.buffer$ is FULL then
7         Send FLUSH-BUFFER message to machine
            with ID = $ILeaf.machineID$ to empty the
            contents of $ILeaf.buffer$ into its
            MR-TREE;
8     end
```

Algorithm 8.3: Flush Buffer

```
1 procedure FLUSH-BUFFER ()
      Input  : Buffer $buff$
      Output: Points in $buff$ inserted into
                 machine'MR-TREE $Mtree$
2     foreach point $p$ in $buff$ do
3         INSERT-INTO-R-TREE ($p$, $Mtree$);
4     end
5     if no. of points in this machine exceeds $mc$ then
6         if there exists an empty machine in the
            cluster then
7             SPLIT-AND-ADJUST ();
8         else RE-DISTRIBUTE-DD-RTREE () ;
```

Algorithm 8.4: Re-Distribute DD-RTREE

```
1 procedure RE-DISTRIBUTE-DD-RTREE ()
      Input  : Machine $S$
      Output: Data points of $S$ re-distributed
2     Compute the proportion of points to be shifted
        to each overlapping node.;
3     if some node in them is full then
4         Recursively call RE-DISTRIBUTE-DD-RTREE
            over those machine to create space.;
5         Shift points based on overlap
            minimization.;
6     if sufficient points are not shifted then
7         Shift points based on $k$-NN;
```

8.2.2 DD-Rtree Construction

The pseudo codes of algorithms for construction of DD-Rtree are explained in Algorithm 8.1 on page 188, Algorithm 8.2 on page 188, Algorithm 8.3 on page 188 & Algorithm 8.4 on page 188. Initially an empty IR-Tree is created. Then data points in the data list DL are inserted into the DD-Rtree one after the other. In insertion, we first find the most appropriate leaf ($lLeaf$) of the IR-Tree to insert a data point p, by the usual R-tree recursive top down search using expansion area principles. We then store p in $lLeaf.buffer$ and update the MBRs of IR-Tree in a bottom up manner similar to that of R-tree. We also increment $lLeaf.cnt$, which indicates the number of points stored or to be stored in the corresponding machine. If $lLeaf.buffer$ reaches buffer capacity (bc), then the points indexed in the buffer are flushed into the corresponding machine, by inserting them into its MR-Tree. If at this point, the machine exceeds its capacity, it tries to identify if there is any other new machine in the cluster available by contacting the master. If, yes, the machine splits itself into two equal halves by the usual R-tree split algorithm and one of the halves is transmigrated to the new machine and two new MR-Trees are created. This would lead to MBR updates in the MR-Trees as well as the IR-Tree, which are done using a few MPI messages. If there is no free machine available in the cluster, then the machine performs re-distribution. In this process, we try to shift a few points from the current machine to few other machines so that some space is created for incoming data points. Re-distribution is explained in the next subsection. Finally, after all insertions finish, all the buffers of IR-Tree are flushed into their respective MR-Trees.

Re-distribution. Unlike SD-Rtree, where only one point is shifted out from a full computing node, we shift points in bulk, i.e. we shift multiple points in one re-distribution, creating more space for incoming points. This helps in reducing the communication overheads for subsequent insertions. The algorithm is as follows: when a computing node A is full, we first identify if there are any other computing nodes (B or C or both) whose MBRs are overlapping with that of A. If yes, we try to shift a maximum of τ points in total from the overlapping regions from A to their respective machines B or C. In practice $\tau = x \times bc$ where $x \in [0, 1]$. If any of B or C is full, then we first recursively apply re-distribution over that computing node to create space in it and then shift points from A into it. If however, we don't have sufficient space in the overlapping machines to shift τ points, or there are no overlapping computing nodes, we try to shift them to non-full machines that are not overlapping with A based on k-NN. In this, we compute min-distances from the centroid of these non-full overlapping computing nodes to A and order them in increasing order of min distance. Then depending on space availability in each of these machines, we greedily transfer points to them. For example if we have to shift points to node B having remaining space y, then we trigger y-NN using centroid of B over points in A and shift those y points from A to B. Similarly, we shift points to other non-overlapping machines. In practice, the k-NN based re-distribution is triggered very less number of times. So, it's the overlap based re-distribution strategy that suffices and ensures that spatial locality is not affected.

DD-Rtree advantages. We can see from the above discussion, that the design of DD-Rtree achieves minimal overlap among the bounding rectangles of the machines. This is because, all the algorithms governing construction of DD-Rtree are based on R-tree's construction principles. DD-Rtree exhibits good spatial locality and efficient query performance (verified by experiments in next section). Buffers attached to the leaves of IR-Tree enable reduction in communication overheads during the construction phase. Although the redistribution strategy of DD-Rtree involves high communication overhead, it is expected to be more efficient because the number of re-distributions occurring in total is quite less when compared to that of SD-Rtree. As a result, DD-Rtree has a lesser construction time. This has been verified by experiments (see next section). Also the redistribution strategy is based on the principles of minimizing overlap among the bounding rectangles of the machines when compared to that of SD-Rtree which is based on k-NN only. Thus, it gives better locality in distribution and efficient query performance. DD-Rtree serves as an efficient data distribution method to distribute data across computing nodes in a cluster and thereby improving the efficiency of the parallel spatial data mining algorithms.

8.2.3 Queries supported by DD-Rtree

DD-RTREE supports ϵ-neighborhood queries and k-NN queries. Queries are issued from the master or the server where the IR-TREE is stored.

Neighborhood Queries. The pseudo code explaining the execution of ϵ-neighborhood query over DD-RTREE is presented in Algorithm 8.5 on the following page and Algorithm 8.6 on the next page. We first construct an ϵ-extended region r by extending the coordinates of p in both directions across all dimensions. We then perform a region query over IR-TREE (*Itree*) similar to the top-down recursive search in an R-tree, to retrieve all the machines that overlap with r. Then for each of the leaves retrieved, we pass an MPI message (FORWARD-NBH-QUERY()) asking it to perform neighborhood query over its MR-TREE using r. The results of all of them are collected back at the master and are collectively returned. The number of MPI messages required to perform this query is double the number of machines visited. We can also minimize the messages by doing a sequential visit of all those leaves overlapping with r, reducing MPI messages to no. of machines visited + 1. But the first approach works faster for big datasets as it works in parallel.

Algorithm 8.5: NBH Query in DD-Rtree

```
procedure DDR-NBH-QUERY ()
    Input  : Query point $p$, $\epsilon$, IR-TREE $Itree$
    Output: Points lying in $\epsilon$-neighborhood of $p$
    Construct an $\epsilon$-extended region r of $p$;
    Perform a top-down recursive search over $Itree$
      to find the leaves of $Itree$ that overlap with r
      and store them in a Queue, $MQ$;
    foreach $leaf$ in $MQ$ do
        FORWARD-NBH-QUERY ($p$,$\epsilon$) to MR-TREE of
          $leaf$ and collect the results;
    end
```

Algorithm 8.6: Forwarding NBH Query

```
procedure FORWARD-NBH-QUERY ()
    Input  : Data point $p$, $\epsilon$
    Output: $tempList$ containing points of the
            MR-TREE lying in $\epsilon$ neighborhood of $p$
    R-NBH-QUERY ($p$, $\epsilon$, $Mtree$, $tempList$); //
      accumulates $\epsilon$-NBH of point $p$ lying in the
      MR-TREE of the current machine to $tempList$
    return $tempList$ to the master;
```

Algorithm 8.7: k-NN Query in DD-Rtree

```
procedure DDR-KNN-QUERY ()
    Input  : Query point $P$, $k$, a max-priority queue
             $NbhPQ$ of size $k$, IR-TREE $Itree$
    Output: $k$ nearest neighbors stored in $NbhPQ$
    Create two min-priority queues $PQ_{mmd}$ and
      $PQ_{md}$;
    foreach $machine$ $i \in ITree.leaves$ do
        Insert $machineID_i$ into $PQ_{mmd}$ with
          $minMaxdist$ and into $PQ_{md}$ with mindist
          from $p$ as keys;
    end
    Find the machine $S$ that contains $p$ from $Itree$;
    FORWARD-KNN-QUERY ($S$, $p$, $NbhPQ$, $PQ_{mmd}$,
      $PQ_{md}$) // makes an MPI call to machine $S$;
```

Algorithm 8.8: Forwarding k-NN Query

```
procedure FORWARD-KNN-QUERY ()
    Input  : Machine $S$, data point $p$, Priority
             Queues $NbhPQ$, $PQ_{mmd}$, $PQ_{md}$
    Output: $NbhPQ$ containing $k$ nearest neighbors
             of $p$ from machine $S$
    Remove $S$ from $PQ_{mmd}$ and $PQ_{md}$;
    Perform locally the $k$-NN search over MR-TREE
      of $S$;
    if $NbhPQ$ is empty then
        Insert all the $k$ nearest neighbors in
          $NbhPQ$ with distance from $p$ as keys;
    end
    else
        $tempDist$ = distance between $p$ and $k^{th}$
          nearest neighbor from $NbhPQ$;
        Insert only those neighbors that are at a
          distance $<$ $tempdist$ from $p$ and update
          $NbhPQ$;
        if $PQ_{mmd}$ is empty OR $PQ_{md}$ is empty then
            return NbhPQ;
        end
        ($id1$, $mmd$) $\leftarrow$ REMOVEMIN ($PQ_{mmd}$);
        ($id2$, $md$) $\leftarrow$ REMOVEMIN ($PQ_{md}$);
        if $mmd < tempDist$ then
            FORWARD-KNN-QUERY ($S_{id1}$, $p$,
              $NbhPQ$, $PQ_{mmd}$, $PQ_{md}$); //
              continuing search on machine $S_{id1}$
        end
        else if $md < tempDist$ then
            FORWARD-KNN-QUERY ($S_{id2}$, $p$,
              $NbhPQ$, $PQ_{mmd}$, $PQ_{md}$); //
              continuing search on machine
              hosting $S_{id2}$
        end
        else
            return $NbhPQ$; // return to master
        end
    end
```

Nearest Neighbor Queries. The k-NN query (Algorithm 8.7) uses one max priority queue NbhPQ of size k to store k nearest neighbors. It also uses two min priority queues - PQ_{mmd} and PQ_{md}, into which all the *machineID*s are inserted with their *minMaxdist* and *mindist* from p as keys respectively. We then find machine S that could contain p from *ITree*, by doing a top-down recursive search over it similar to R-tree. An MPI call is then made to S to execute FORWARD-KNN-QUERY and send all three priority queues to S. In the function FORWARD-KNN-QUEARY (Algorithm 8.8), being executed at S, we first remove S from both PQ_{mmd} and PQ_{md} and then perform a local k-NN search over MR-TREE of S. If S is the first machine of the cluster we are visiting, then we add all the k nearest neighbors to $NbhPQ$. Else we insert only those neighbors that are at a distance less than the distance between p and the current k^{th} nearest neighbor (*tempDist*) and the *NbhPQ* is then updated. After this, we do a REMOVEMIN() operation on both PQ_{mmd} and PQ_{md}

and store the retrieved (*machineID, distance*) pairs in (*id*1,*mmd*) and (*id*2,*md*) respectively. *mmd* is the distance at which there is at least one point in machine with machineID - id1 (S_{id1}). *md* is the minimum possible distance between *p* and any point in the machine with machineID - id2 (S_{id2}). If $mmd < tempDist$, then there is at least one data point in S_{id1} which is at a distance less than *tempDist* from *p*. So, we forward the search request to S_{id1}. If not, we check if $md < tempDist$, then we forward the request to S_{id2}. If *md* would have been greater than *tempDist*, we don't explore this machine. Now, if both the above criteria fail, we don't need to visit any more machines and we simply return the result. The number of MPI calls required for execution of this query is equal to number of machines visited + 1.

8.3 Performance Evaluation

We evaluate DD-RTREE with respect to (1) spatial locality, (2) communication cost (3) construction & querying time, and (4) performance of parallel spatial data mining algorithms it supports. We compare it with SD-RTREE and randomly distribution. We implement the IMCLIENT variant of SD-RTREE, where we have the image stored in a service providing server or the master. This is the most suitable for data distribution as we assume that the dataset is initially stored on this master. The details of the datasets used for experimentation are mentioned in Table 8.1 on the following page. The first four datasets are synthetic and the rest are real. SR500M2D & SR10M2D are randomly generated. Data in SN100M2D follow normal distribution. SC100M2D consists of synthetically generated well separated clusters equal to number of machines used for a particular experiment. SFONT1M11D, MPAHALO2.8M9D, MPAGD56M3D, MPAGD16M3D and FOF113M3D datasets are taken from Millennium data repository that contains astronomical data of galaxies in the sky [135]. These datasets are skewed in nature and do not follow any distribution. SBUS6M2D dataset contains samples of GPS traces of buses in Shanghai [136].

All the experiments were conducted on a cluster of 32 compute nodes whose details are mentioned in Section 6.4 on page 141. All the implementations were done in C with MPI library. In all our experiments, we make the following choices by default until and unless explicitly stated. We choose machine capacity (mc) in such a way that 5% of the

Table 8.1: Datasets used for Experimentation

S. No.	Dataset	Data Size	Dimensionality	Reference
1	SR500M2D	500M	2	-
2	SN100M2D	100M	2	-
3	SC100M2D	100M	2	-
4	SR10M2D	10M	2	-
5	SFONT1M11D	1M	11	[135]
6	MPAHALO2.8M9D	2.8M	9	[135]
7	MPAGD56M3D	56M	3	[135]
8	MPAGD16M3D	16M	3	[135]
9	FOF113M3D	113M 3	[135]	
10	SBUS6M2D	6M	2	[136]

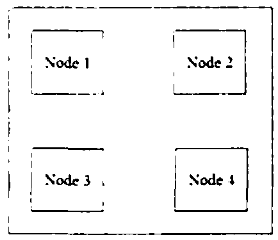


Figure 8.2: Sample Distribution

total capacity of all the machines remains vacant. We choose bc = 10% of the mc. The measurement of run-time has been done using `MPI_Wtime()` of `<mpi.h>` library.

8.3.1 Quality Evaluation

There are no specific measures reported in literature to evaluate the quality of data distribution in terms of spatial locality. So, we use various internal quality evaluation measures described in Appendix C on page 214 to evaluate the quality of DD-Rtree distribution. In order to check their appropriateness for evaluating quality of spatial locality, we have taken a synthetically generated data set of 10 million data points (two dimensions) containing four well separated partitions as given in Figure 8.2. We took this as the base dataset and generated a few more distributions to distort spatial locality by changing the membership of 10%, 20%, 30% randomly chosen data points from their original position to the next best partition. We also generated a dataset which has the membership assigned randomly. Table shows the values of all the measures for generated datasets. The results clearly indicate that the values of these measures deteriorate with increase in distortion of spatial locality. Thus, they are suitable for our evaluation. Similar results were obtained for other datasets with higher number of partitions, as well.

In our experiments, we have observed that all the measures shown in Table 8.2 on the

Dataset	BetaCV	Modularity	Norm. Cut	Davies-Bouldin	Norm. Hubert Stat.	Silhouette
SC1M2D	0.175	-0.208	3.886	0.078	0.562	0.987
SC1M2D_10%_man	0.303	-0.181	3.807	0.177	0.467	0.943
SC1M2D_20%_man	0.414	-0.159	3.741	0.246	0.399	0.904
SC1M2D_30%_man	0.507	-0.142	3.688	0.307	0.346	0.871
SC1M2D_rand	0.820	-0.091	3.519	0.7346	0.112	0.743

Table 8.2: Validating Quality Evaluation Measures

following page are behaving consistently with change in spatial locality. Hence in our subsequent presentation, we present our results with only two measures - *BetaCV* and *Silhouette Co-efficient* (see Table C.2 on page 219).

Distribution Quality. Table 8.8 on page 201 presents the values of quality measures for data distribution of DD-Rtree when compared with random distribution and distribution using SD-Rtree for various synthetic and real datasets. For synthetic random and synthetic normal datasets, the results show that the measures have always been consistently better for DD-Rtree than random distribution and SD-Rtree for different number of computing nodes used in the cluster. The quality of distribution for synthetic cluster dataset however, is slightly lower than that of SD-Rtree. This is because of difference in the re-distribution strategies used. Since the dataset has fully disjoint clusters, the k-NN based re-distribution strategy of SD-Rtree works better than the overlap based re-distribution strategy of DD-Rtree. The quality of distribution using DD-Rtree for all real datasets is found to be better than that of SD-Rtree. This shows the efficiency of DD-Rtree in handling skewed datasets. For SFONT1M11D and MPAGD2.8M9D, DD-Rtree performs much better than others. This is because at high dimensional space, overlap based re-distribution of DD-Rtree performs considerably better than k-NN based re-distribution of SD-Rtree.

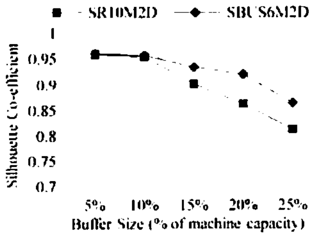


Figure 8.3: Silhouette Co-efficient with increase in buffer size

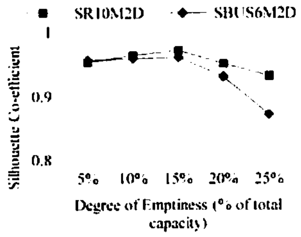


Figure 8.4: Silhouette Co-efficient with increase in degree of emptiness

Quality Analysis on Varying Factors. We analyze the quality of DD-Rtree with variation in buffer size and the degree of emptiness in the tree for 16 nodes in the cluster for SR10M2D and SBUS6M2D datasets. Results presented in Figure 8.3 on page 195 show that the quality of distribution deteriorates with increase in buffer size beyond 10% of machine capacity in both the cases. This is because large buffer size leads to infrequent flushes into MR-Tree. It also results in very large number of points being shifted in redistributions. Optimal quality is observed for buffer sizes between 5 and 10% of machine capacity. Similarly, results presented in Figure 8.4 on page 195 show that the quality of distribution initially improves and then deteriorates when we increase the degree of emptiness in the computing nodes. This is because high degree of emptiness leads to data being distributed in a skewed manner.

Data Structure	Range of no. of points in each machine
SD-Rtree	46,216 - 62,500
DD-Rtree	56,394 - 60,341

Table 8.3: Load Balancing for SR10M2D dataset

We also compare the load balancing achieved for DD-Rtree and SD-Rtree and results are presented in Table 8.3 for SR10M2D dataset for n=16, when degree of emptiness is 5%. The results show that DD-Rtree achieves better load balance than SD-Rtree. Similar results were obtained for other datasets as well.

8.3.2 Efficiency Evaluation

We evaluate DD-Rtree for execution time and MPI messages required for data distribution.

Table 8.4: Construction time of DD-Rtree vs SD-Rtree

Dataset	SR10M2D			SBUS6M2D		
No of Nodes	Data Struct.	Constr. Time	MPI Messages (approx..)	Data Struct.	Constr. Time	MPI Messages (approx..)
n=16	SD-Rtree	1787 sec.	25.3 M	SD-Rtree	1607 sec.	18.4 M
	DD-Rtree	1295 sec.	1.4 M	DD-Rtree	1064 sec.	1.2 M
n=32	SD-Rtree	1839 sec.	26.4 M	SD-Rtree	1672 sec.	19.6 M
	DD-Rtree	1363 sec.	1.6 M	DD-Rtree	1114 sec.	1.3 M

Construction of DD-Rtree. We measure construction time and number of MPI messages required for DD-Rtree and SD-Rtree on SR10M2D and SBUS6M2D datasets for 16 and 32 nodes. The results presented in Table Table 8.4 show that the execution time and the

Table 8.5: Construction time of DD-RTREE with variation in Buffer size

	Buffer Size	5%	10%	15%	20%	25%
SR10M2D	Construction Time	1462 sec.	1363 sec.	1284 sec.	1106 sec.	1085 sec.
	MPI messages	1.8 M	1.6 M	1.2 M	0.9 M	0.8 M
SBUS6M2D	Construction Time	1267 sec.	1114 sec.	1068 sec.	1027 sec.	992 sec.
	MPI messages	1.5 M	1.3 M	0.8 M	0.6 M	0.5 M

MPI messages required for DD-RTREE is less with respect to SD-RTREE. This is mainly attributed to buffering technique used to defer insertions and inserting them in bulk rather than point by point. This is also due to reduction in number of re-distributions for DD-RTREE.

We have also measured the construction time of DD-RTREE and number of MPI messages required, with variation in buffer size (% of machine capacity) for SR10M2D and SBUS6M2D datasets on 32 nodes. The results presented in Table 8.5 indicate that the number of MPI messages decrease as the buffer size increases. This is because when buffer size is small, the buffer is flushed very frequently and re-distribution routine is executed more number of times. However, we can see from Figure 8.3 on page 195, that quality of distribution is good when the buffer size is small. Therefore, we have taken buffer size to be 10% in all our experimentation.

Performance of Queries. We measure the average number of machines visited per query, average number of MPI messages and average execution time, for ϵ-neighborhood and k-NN queries when executed over DD-RTREE and SD-RTREE for SR100M2D and MPAGD56M3D datasets for 32 nodes. We have used 10% sample of the dataset as querying points and have computed averages. We choose ϵ=0.01 for SR100M2D dataset and ϵ=0.006 for MPAGD56M3D dataset for executing neighborhood queries. We choose k=20 for all k-NN queries. The results presented in Table Table 8.6 on the next page clearly indicate that all these parameters are better for DD-RTREE, proving its better spatial locality. We also observed that query performance of DD-RTREE is consistently maintained with variation in ϵ for neighborhood queries and variation in k for k-NN queries when compared to SD-RTREE.

Performance of Distributed DBSCAN. We perform simple version of distributed DBSCAN over SD-RTREE and DD-RTREE, to compare their performance over MPAGD16M3D

Table 8.6: Querying Performance of DD-RTREE and SD-RTREE

		SR100M2D			MPAGD56M		
		Average MPI Messages	Average Execution Time	Average number of Machines Visited	Average MPI Messages	Average Execution Time	Average number of Machines Visited
DD-Rtree	Nbh Query	3.18	0.051 sec.	1.59	2.9	0.043 sec.	1.45
	k-NN query	3.17	0.074 sec.	2.17	2.96	0.058 sec.	1.96
SD-Rtree	Nbh Query	3.44	0.069 sec.	1.72	3.26	0.056 sec.	1.63
	k-NN query	3.38	0.086 sec.	2.38	3.24	0.072 sec.	2.24

dataset over 32 machines. The ϵ was chosen to 0.01 and Min_Pts was chosen to be 5. Table 8.7 on the following page presents the summary of its execution. DBSCAN follows all four steps of a distributed algorithm explained in section 1. In step 1, we distributed data using suitable method. In this DD-RTREE takes less time when compared with SD-RTREE. This is attributed to reasons explained in the above experiments. In step 2, for every machine, we retrieve data points from other machines in the cluster which lie within ϵ-extended boundary of the current machine. In this step, the number of MPI messages required remains same in all cases. However, the execution time for DD-RTREE is lesser because the number of extra data points fetched from other machines is less due to its greater preservation of spatial locality in its distribution. Step 3 involves execution of local DBSCAN at each machine. In this step, the time require for local DBSCAN is less for DD-RTREE. This is because, preservation of spatial locality helps in reducing the search space for the neighborhood queries, when data is indexed in R-trees. This is also because of lesser number of extra points retrieved from other nodes. In step 4, we merge the results of all local DBSCAN to give the required global clustering. In this step, the time required for merging is almost the same for DD-RTREE and SD-RTREE. Thus, the above experiment shows that effective distribution of data using DD-Rtree enables reduction in communication complexity and thus improves the performance of parallel DBSCAN.

8.4 Discussion

DD-RTREE gains the following advantages by its design:

Table 8.7: Parallel DBSCAN using DD-Rtree and SD-Rtree

	Exec. time for SD-Rtree	Exec. time for DD-Rtree
Step 1	1948 sec.	1426 sec.
Step 2	481 sec.	304 sec.
Step 3	1671 sec.	1428 sec.
Step 4	150 sec.	139 sec.
Total Execution Time	**4250 sec.**	**3297 sec.**
	Number of data points retrieved from other machines	
Step 2	1.92 M	1.28 M

- The re-distribution strategy of DD-RTREE is a hybrid strategy based on overlap minimization and nearest neighbor search, unlike the SD-RTREE which is based only on nearest neighbor search. Nearest neighbor search alone cannot gaurantee good spatial locality and thus DD-RTREE turns out to be better in preserving spatial locality in its distribution.
- The communication overhead in construction of DD-RTREE is much lesser than that of SD-RTREE. This is primarily attributed to bulk insertions and bulk re-distributions (resultant of usage of buffers), unlike SD-RTREE that inserts and re-distributed point by point and hence becoming a communication bottle-neck. This leads to lesser construction time for DD-Rtree.
- Also, bulk loading helps in maintaining good load balancing in the machines, which is one of the key requirements for parallel spatial data mining algorithms.
- All the above factors of DD-RTREE resulted in improved performance of spatial queries and parallel DBSCAN.

8.5 Main Contributions

- We presented DD-RTREE which is a dynamic distributed data structure for indexing large and incremental datasets.
- DD-RTREE can be used for distributing data into the computing nodes of the cluster in such a way that it effectively preserves spatial locality in its distribution and achieves good load balancing
- Experimentally, we have shown that DD-RTREE gives better performance of spatial

queries and parallel DBSCAN, when compared with random distribution and SD-RTREE.

- Experiments also show that DD-RTREE has lesser communication overhead when compared to SD-RTREE and thus is efficient in its construction and querying.

8.6 Conclusions & Future Work

8.6.1 Conclusions

This chapter proposes DD-RTREE, which is a dynamic distributed data structure based on R-tree. DD-RTREE preserves spatial locality in its distribution, achieves good load balancing, exhibits less communication overhead in querying and construction, and improves the performance of parallel spatial data mining algorithms. DD-RTREE also supports efficient execution of ϵ-neighborhood and k-NN queries. The quality and efficiency evaluation together establishes the superiority of DD-RTREE with respect to SD-RTREE and random distribution.

8.6.2 Future Directions

DD-RTREE can be used to design highly efficient distributed framework for mining data streams. Also, the DD-RTREE strategy can very well be applied on other R-tree variants.

Table 8.8: Data distribution quality for varying number of computing nodes for various datasets

		n=16			n=32			n=64			n=128			n=256		
		rand	SDR	DDR	rand	SDR	DDR	rand	SDR	DDR	rand	SDR	DDR	rand	SDR	DDR
SR500M2D	BCV	0.715	0.526	0.439	0.632	0.518	0.426	0.698	0.516	0.428	0.702	0.511	0.403	0.697	0.506	0.411
	SIL	0.614	0.912	0.953	0.632	0.908	0.955	0.697	0.922	0.964	0.712	0.911	0.963	0.706	0.917	0.965
SN100M2D	BCV	0.765	0.699	0.698	0.712	0.624	0.614	0.703	0.617	0.598	0.71	0.599	0.572	0.685	0.548	0.516
	SIL	0.487	0.796	0.844	0.498	0.821	0.869	0.436	0.788	0.876	0.513	0.842	0.912	0.559	0.849	0.915
SC100M2D	BCV	0.833	0.411	0.451	0.795	0.396	0.432	0.768	0.347	0.386	0.778	0.301	0.941	0.724	0.282	0.304
	SIL	0.648	0.921	0.854	0.627	0.923	0.867	0.633	0.926	0.892	0.701	0.929	0.915	0.693	0.937	0.924
SFONT1M11D	BCV	0.514	0.398	0.248	0.534	0.375	0.244	0.527	0.368	0.214	0.486	0.347	0.196	0.447	0.329	0.204
	SIL	0.247	0.726	0.894	0.164	0.763	0.905	0.187	0.773	0.924	0.168	0.798	0.937	0.199	0.812	0.94
MPAHALO2.8M9D	BCV	0.628	0.583	0.473	0.604	0.562	0.436	0.593	0.518	0.401	0.562	0.501	0.386	0.579	0.483	0.372
	SIL	0.363	0.674	0.836	0.381	0.693	0.853	0.394	0.725	0.875	0.427	0.783	0.901	0.452	0.812	0.894
FOF113M3D	BCV	0.864	0.635	0.447	0.822	0.622	0.432	0.812	0.593	0.458	0.794	0.605	0.405	0.807	0.586	0.415
	SIL	0.294	0.764	0.889	0.386	0.793	0.892	0.357	0.813	0.914	0.429	0.818	0.927	0.414	0.827	0.942
MPAGD56M3D	BCV	0.62	0.455	0.412	0.637	0.441	0.399	0.587	0.428	0.394	0.641	0.427	0.367	0.623	0.428	0.338
	SIL	0.749	0.901	0.914	0.726	0.92	0.921	0.738	0.914	0.908	0.722	0.908	0.917	0.685	0.917	0.921
SBUS6M2D	BCV	0.512	0.458	0.345	0.539	0.469	0.314	0.521	0.431	0.324	0.566	0.413	0.305	0.547	0.407	0.298
	SIL	0.802	0.947	0.956	0.765	0.952	0.974	0.744	0.962	0.966	0.798	0.967	0.982	0.783	0.963	0.971

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The work done in this thesis primarily focuses on the following broad research topics - Data Mining, Data Streams, Data Structures and High Performance Computing. We specifically deal with the problems on development of efficient data indexing techniques for efficient spatial queries, mining variable speed streaming data and spatial data distribution for parallel data mining over distributed memory architectures. In the course of this thesis, we have innovatively used the concepts listed in the table below. The applicability of these concepts to the problems attempted in this thesis is summarized in Table 9.1 and have been briefly explained as follows:

- *Grid-R-tree.* This method uses a hybrid method of adaptive gridding and hierarchi-

Table 9.1: Key concepts used in proposed algorithms

Concept	Grid-R-tree	AnySC	AnyClus	Any-MP-Clus	AnyFI	MPAnyFI	DD-RTree	Static Distribution
Gridding and Adaptive Gridding	✓	✗	✗	✗	✗	✗	✗	✓
Hierarchical data structures	✓	✓	✓	✓	✓	✓	✓	✓
Spatial locality aware computations	✓	✓	✓	✓	✗	✗	✓	✓
Computational cost reduction	✓	✓	✓	✓	✓	✓	✓	✓
Handling variable stream speeds	✗	✓	✓	✓	✓	✓	✗	✗
Spatial Data Distribution	✗	✗	✗	✓	✗	✗	✓	✓
Buffering	✗	✓	✓	✓	✓	✓	✓	✗
Hierarchical Aggregation	✗	✓	✓	✓	✗	✗	✗	✗

cal structure, R-tree, in order to efficiently execute spatial queries used in spatial data mining algorithms. Its two-level design achieves reduction in search space for spatial queries and reduces the computations cost of clustering algorithms such as DBSCAN & OPTICS, as well as the K-NN classifier. The innovative use of hybrid concepts makes Grid-R-tree address the drawbacks of conventional structures like R-tree and kd-tree, arising due to large size and high dimensionality of the datasets (such as increase in overlap and height in R-tree & k-d-tree, respectively) and execute the queries more efficiently. The experimental analysis suggests that Grid-R-tree outperforms R-tree and kd-tree. The maximum speed up achieved is 42 for CellWiseNBH query, 21 for PointWiseNBH query and 25 for KNN query.

- AnyFI & MPAnyFI. These two methods are the first proposed methods for anytime frequent itemset mining of single-port and multi-port data streams, respectively. They use proposed hierarchical structure known as BFI-forest, which is a collection of BFI-trees. BFI-trees use buffers in their nodes to delay the processing of incompletely processed transactions arriving in variable speed streams. Experimental results suggest that AnyFI can handle stream speeds upto 60,000 transactions per second with recall close to 100%. The experiments also show the efficiency and scalability of MPAnyFI.
- AnySC. It is the first proposed anytime set-wise classification algorithm for data streams. It processes variable stream speeds using a proposed hierarchical structure known as CProf-forest, which is a collection of CProf-trees that are based on R-tree structure. CProf-trees contain buffers in their internal nodes to defer the processing of incompletely processed objects arriving in variable speed data streams. The training data is hierarchically aggregated in the form of CProf-trees to handle classification of test data arriving in the stream. AnySC makes best use of the hierarchical structure of the CProf-forest to get better accuracy when compared to the non-anytime set-wise classification algorithm (SC), even at SC's budget speed. This happens because of reduction in computational overhead by the conversion of the linear model to a hierarchical model. The experimental analysis shows that AnySC is able to process very high speed streams (upto 100k objects per second) and produce accurate results. The experimental results also show the applicabil-

ity of AnySC to the problem of community detection using text feeds from twitter, as well as simulation of website fingerprinting attack. AnySC performs the above applications where stream speed is varying, with good accuracy.

- AnyClus & AnyMPClus. These two problems use the proposed hierarchical structure known as AnyRTree, which is based on R-tree. AnyRTree also uses buffers to delay the processing of incompletely processed points arriving in variable speed streams, and also to handle noise and concept drift. AnyRTree is a hierarchical aggregation of micro-clusters using R-tree spatial containment principles. This preserves the spatial locality in the hierarchical arrangement of micro-clusters in the tree and hence leads to reduction in computation cost of the offline clustering algorithm applied over these micro-clusters. This also helps in producing purer and compact micro-clusters when compared to the existing methods. Any-MP-Clus uses spatial data distribution to efficiently merge the micro-clusters and then feeds them into a logarithmic tilted-time window framework. The experiments show that AnyClus and Any-MP-Clus can handle very high speed streams (>200k points per second) and produce micro-clusters of high purity ($\approx$1). The usage of noise buffers in the internal nodes at higher granularity also led AnyClus and Any-MP-Clus to handle noise and concept drift more effectively.
- DD-RTree & Static Data Distribution Strategies. The data distribution strategies proposed in this thesis are fully based on spatial locality principles and are specifically designed for those parallel data mining algorithms that exploit spatial locality. They achieve computational and communication cost reduction by use of spatial data distribution attained by following the scheme of hierarchical data indexing structures such R-tree and k-d-tree. DD-RTree additionally employs buffers to further reduce the communication overhead and uses bulk-loading. The experimental results suggests that the proposed static methods outperform the existing kd-tree based distribution scheme for density-based and hierarchical clustering algorithms. Appropriate recommendations for the usage of each of the distribution proposed is also given.

Notable Achievements of work done in this thesis:

- Proposed first of its kind solutions - Grid-R-tree, AnySC, Any-MP-Clus, AnyFI, MPAnyFI and Data Distribution Strategies.
- Effectively used hybrid concepts such as combination of grid and trees.
- Efficiently processed billions scale data.
- Efficiently processed high speed and variable speed streams.

9.2 Future Directions

An insight on future directions which we plan to pursue.

- More number of Grid-R-tree kind of hybrid structures can be made with combinations of multiple data structures.
- Anytime mining algorithms can be developed for anomaly detection and classifiers like SVM, Decision Tree, etc., for handling both large static data as well as streaming data. Anytime mining algorithms can also be developed for hierarchical clustering, subspace clustering, grid-based clustering, etc. over large static datasets.
- A "fast and slow framework" for anytime mining of data streams can be designed where two processes simultaneously capture the same stream, one using sampling and the other using anytime features. This is expected to improve the accuracy of the offline mining results produced for very high speed streams.
- The data distribution strategies can be extended to suit the parallel data mining algorithms proposed for Map/Reduce and Spark frameworks.
- Effective distributed frameworks based on MPI, Spark and Map/Reduce can be developed for distributed stream processing.

Appendix A

R-tree

R-tree [122] is a commonly used hierarchical indexing structure for indexing multi-dimensional spatial objects. R-tree and its variants are commonly used in spatial data mining algorithms for efficient execution of neighborhood and nearest neighbor queries, in logarithmic average time.

A.1 Structure of an R-tree

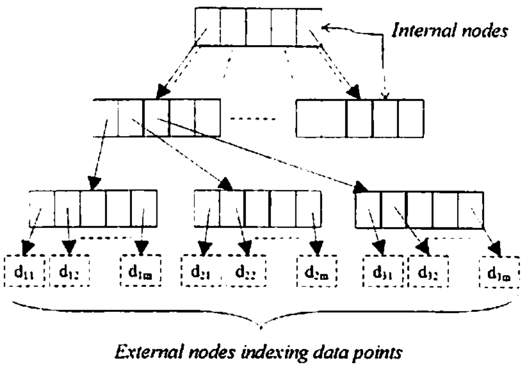


Figure A.1: R-tree: Structure

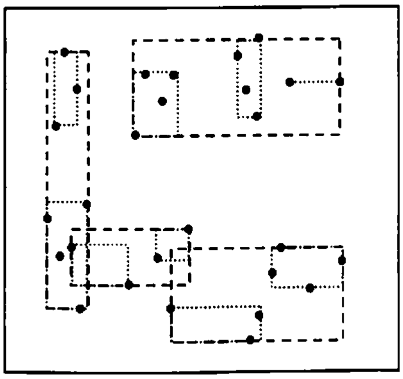


Figure A.2: Minimum Bounding Rectangles

Figure A.1 illustrates the structure of an R-tree. It has two kinds of nodes: *internal* and *external* (or *leaves*). It is defined as follows:

Definition A.1. R-tree is a height balanced multi-dimensional indexing structure having the following properties:

- Each node (both internal and external) contains between m and M entries ($m < M/2$). The root has at least one entry.
- An entry of an internal node stores the following entries: (i) a pointer *child* to the child sub-tree; (ii) a minimum bounding rectangle *MBR*; (iii) a pointer *next* to the next entry in the node.
- An entry of a leaf node stores d-dimensional data points.

An MBR stores bounding information of all the points indexed in the sub-tree rooted at it. Figure A.2 shows MBRs of the nodes of the R-tree formed while indexing data points. The MBR of a given entry bounds the MBRs present in all the entries indexed at its child node. This forms a hierarchy of MBRs in the tree.

Algorithm A.1: Insertion in R-tree

```
1  procedure Insert-In-R-tree ()
       Input  : data point $p$, R-tree node $node1$
       Output: $p$ inserted into tree rooted at $node1$
2      if $node1.type$ == INTERNAL then
3          $bestChild$ ← R-Pick-Child ($node1$, $p$);
4          Insert-In-R-tree ($bestChild$);
5      else if $node1.type$ == EXTERNAL then
6          Insert $p$ as a new entry in $node1$;
7          if $node1$ overflows then
8              R-Split-Node($node1$);
9          R-Update-MBR-Bottom-Up($node1$);
```

Algorithm A.2: R-Pick-Child

```
1  procedure R-Pick-Child ()
       Input  : R-tree node $node1$, data point $p$
       Output: $bestChild$ of $node1$ for insertion of $p$
2      $bestE$ ← $node1$'s first entry; $currE$ ← $bestE$;
3      while $currE \neq NULL$ do
4          if Expansion-Area ($bestE.MBR$, $p$) >
               Expansion-Area ($currE.MBR$, $p$) then
5              $bestE$ ← $currE$;
6          $currE$ ← $currE.next$
7      end
8      return $bestE.child$;
```

Algorithm A.3: R-Split-Node

```
1  procedure R-Split-Node ()
       Input  : R-tree node $nd$
       Output: Splits $nd$ into two and updates the tree
2      $e_1$ ← NULL; $e_2$ ← NULL; $te$ ← $nd.firstEntry$;
3      $nd_1$, $nd_2$ ← New-R-tree-Node ();
4      R-Pick-Seeds ($nd$, $e_1$, $e_2$);
5      while $te \neq$ NULL do
6          if No-Of-Filled-Entries ($nd_1$) < $m$ &
               No-Of-Filled-Entries ($nd_2$) < $m$ then
               // both $nd_1$ & $nd_2$ are underflown
7              if ExpansionArea ($nd_1$, $te.MBR$) <
                   ExpansionArea ($nd_2$, $te.MBR$) then
8                  Add-Entry-To-Node ($te$, $nd_1$);
9              else
10                 Add-Entry-To-Node ($te$, $nd_2$)
11         else if Num-Filled-Entries($nd_1$)< $m$ then
12             Add-Entry-To-Node ($te$, $nd_1$);
13         else
14             Add-Entry-To-Node ($te$, $nd_2$);
15         $te$ ← $te.next$;
16     end
17     Add-Node-To-Rtree-Node ($nd_1$, $nd.parent$);
18     Add-Node-To-Rtree-Node ($nd_2$, $nd.parent$);
19     if $nd.parent$ overflows then
20         R-Split-Node ($nd.parent$);
```

Algorithm A.4: R-Pick-Seeds

```
1  procedure R-Pick-Seeds ()
       Input  : R-tree node $nd$, entry pointers $e_1$ & $e_2$
       Output: $e_1$ & $e_2$, which are the pair of entries of
               $nd$ whose MBRs are farthest along any
               dimension. They are selected as initial
               seeds for splitting
2      foreach dimension $i$ do
3          $minE$ ← Min-Entry-Across-Dim($nd$, $i$);
4          $maxE$ ← Max-Entry-Across-Dim($nd$, $i$);
           // $minE$ & $maxE$ are pointers to entries
           of a R-tree node
5          $Arr[i].minEntry$ ← $minE$;
6          $Arr[i].maxEntry$ ← $maxE$;
7          $disN[i]$ ← Separation-Between-MBRs
               ($maxE.MBR$, $minE.MBR$);
8          $disN[i]$ ← $disN[i]$/Span($i$, $Arr[i]$); //
               normalizes the separation computed
9      end
10     foreach dimension $i$ do
11         if $disN[i] > dis$ then
12             $dis$ ← $disN[i]$;
13             $e_1$ ← $Arr[i].minEntry$;
14             $e_2$ ← $Arr[i].maxEntry$;
15         end
16     end
```

A.2 R-tree: Construction, Insertion and Deletion

R-tree is constructed by incremental dynamic insertions of a given list of data points. Insertion of a data point p happens in a *top-down recursive* fashion into the sub-tree rooted at a given node (*node*1), beginning from the root. Algorithm A.1 illustrates this. If the current node (*node*1) is an internal node, it picks the best child of the node (the child that has least expansion in its MBR's area after inclusion of p (Algorithm A.2), and makes a recursive call over that child (lines 2-4 of Algorithm A.1). If *node*1 is an external node, it simply inserts the data point into it as a new entry (line 6 of Algorithm A.1). If *node*1 overflows because of this insertion, i.e. the number of entries stored in it exceeds M, then it gets split into two (Algorithm A.3) and the newly created nodes are added to *node*1's parent (line 7 of Algorithm A.1). Subsequently, the MBRs of the tree are updated in a bottom up fashion as per the newly inserted data point (line 8 of Algorithm A.1). The average case time complexity of inserting a data point into R-tree is $O(\log_m N)$. Thus, average time complexity of R-tree construction is $O(N \log_m N)$, where N is the total number of data points inserted into in the tree.

The Split Node function (Algorithm A.3 on page 207) is the R-tree's linear split algorithm [122]. One can very well use quadratic or exponential split, instead. The linear split uses a linear cost seed selection algorithm (Algorithm A.4 on page 207). This seed selection algorithm, for each dimension, finds the entries of the node being split that have the lowest and the highest extreme rectangles respectively, and stores the pair in an array *Arr* of size *d* (lines 2-9 of Algorithm A.4 on page 207). It then computes the separation between the MBRs for each pair stored in *Arr* and stores the separation values in an array *disN* of size *d*. Then *disN* is normalized by dividing the separation found by the width of the entire set along the corresponding dimension. The pair of entries that have the highest normalized separation along any dimension are selected as initial seeds for splitting (lines 10-16 of Algorithm A.4 on page 207). The Split Node function then takes these two seeds and distributes the remaining entries to these seeds thus creating two nodes (lines 5-15 of Algorithm A.3 on page 207). Distribution happens with minimization of expansion area as the criteria. It also makes sure that none of the new nodes have underflow. The two new nodes are then attached to the parent of the older node, which is subsequently checked for overflow (lines 17-20 of Algorithm A.3 on page 207). If overflow occurs, it is handled in a similar way and the split can propagate up to the root leading to creation of new root. Thus in this way, R-tree grows upwards. For more details on Split Node and Pick Seeds functions, please refer to [122].

Deletion of a point p from R-tree involves finding it using point query and then its removal from the corresponding node. If this removal causes node under-flow (i.e. no. of entries become $< m$), it has to be handled by merging of nodes in a bottom up manner. Since deletion is out of scope for this thesis, we don't describe it further.

A.3 Distance Measures in an R-tree

We first present two distance measures used in R-trees.

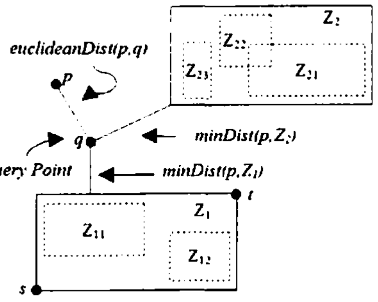


Figure A.3: Illustrating euclidean distance and min-distance

Definition A.2. ***Euclidean Distance.*** Given two d- dimensional points $p = (p_1, \cdots, p_d)$ and $q = (q_1, \cdots, q_d)$ (see Figure A.3), for $1 \leq i \leq d$,

$$euclideanDist(p,q) = \sqrt{\sum_{i=0}^{d}(p_i - q_i)^2}$$

Definition A.3. ***Min Distance.*** Given a d-dimensional query point $q = (q_1, \cdots, q_d)$ and an MBR $Z = (s,t)$ of an R-tree defined by two corner points s and t (as shown in Figure A.3

on page 208) such that $s = (s_1, \cdots, s_d)$, $t = (t_1, \cdots, t_d)$ and $s_i \leq t_i$ for $1 \leq i \leq d$, then

$$minDist(q, Z) = \sqrt{\sum_{i=0}^{d} |q_i - r_i|^2} \qquad where\ r_i = \left\{ \begin{array}{ll} s_i & if q_i < s_i \\ t_i & if q_i > t_i \\ q_i & otherwise \end{array} \right\}$$

$minDist(q, Z)$ is the minimum distance between a query point q and an MBR Z as illustrated in Figure A.3 on page 208 for d=2. It is the lower bound of the actual distance of any object lying in Z to q.

A.4 Queries Supported by R-tree

In this section we discuss various queries supported by R-tree. They include - *region queries (point, window & neighborhood queries)* and *nearest neighbor query*.

A.4.1 Point Query

Point query checks the existence of a given data point p in a dataset. The algorithm for point query over an R-tree executes in a top-down recursive fashion over the R-tree nodes, recursing into the children whose MBRs promise containment of p. In its traversal, if it finds p at any of the external nodes, it returns TRUE, else it returns FALSE (See Algorithm A.5) for pseudo code). The average complexity of point query over R-tree is $O(\log_m N)$.

Algorithm A.5: Point Query over R-tree

```
procedure POINT-QUERY-R-TREE ()
    Input  : data point p, R-tree root node
    Output: TRUE of p exists, FALSE otherwise
    boolean flag ← FALSE ;
    if node.type == EXTERNAL then
        foreach entry e indexed in node do
            if e == p then
                flag ← TRUE; break;
        end
    else if node.type == INTERNAL then
        foreach entry e indexed in node do
            if e.mbr contains p then
                flag ← POINT-QUERY-R-TREE (p,
                  e.child);
        end
    return flag;
```

Algorithm A.6: Window Query over R-tree

```
procedure WINDOW-QUERY-R-TREE ()
    Input  : region r, R-tree root node, data points
             list plist
    Output: plist containing data points lying in r
    if node.type == EXTERNAL then
        Add all the points indexed at node, that lie
          in r, to pList;
    end
    else if node.type == INTERNAL then
        foreach entry e of node do
            if e overlaps with r then
                WINDOW-QUERY-R-TREE (r,
                  e.child, plist);
            end
        end
    end
```

A.4.2 Window Query

Window query is a query which returns all the data points that lie in a d-dimensional window or a region r, from the entire data space. Figure A.4 on the next page illustrates the window query for d=2. The window query algorithm for R-tree also executes in a top-down recursive fashion, recursing itself into the children of a node whose MBRs overlap with the given region r. In its traversal, whenever it encounters an external node, it simply accumulates all the data points of this node that lie in region r into a temporary list and returns it. (See Algorithm A.6 for pseudo code). The average complexity of window query over R-tree is $O(\log_m N)$.

Algorithm A.7: ϵ-Neighborhood Query over R-tree

```
procedure ε-NEIGHBORHOOD-QUERY-R-TREE ()
    Input : data point $p$, $\epsilon$, R-tree root node, data points list plist
    Output: plist containing points lying within $\epsilon$ distance from $p$
    construct an $\epsilon$-extended region $r$ of $p$;
    tempList = WINDOW-QUERY-R-TREE ($r$, node);
    plist = points of templist within $\epsilon$-distance from $p$;
    return plist;
```

A.4.3 Neighborhood Query

Neighborhood query or ϵ-neighborhood query is a query which returns all the data points lying within an ϵ- distance from a given point p. Figure A.5 illustrates this query for d=2. The points lying in the circular region in the figure is ϵ-neighborhood of p. Neighborhood queries are extensively used in density based clustering algorithms like DBSCAN [34] and OPTICS [35].

The neighborhood query algorithm for an R-tree is presented in Algorithm A.7. In this, a region r (referred to as ϵ-extended region of p) is first constructed around p by extending its coordinates across all the dimensions by ϵ, in both positive and negative directions. Figure A.5 shows this region for 2-dimensional space. Then, a window query is executed over R-tree with r as the window. From the points returned by the window query, those that lie within ϵ distance from p are returned as ϵ-neighborhood of p. The average complexity of neighborhood query over R-tree is $O(\log_m N)$.

A.4.4 Nearest Neighbor Query

Nearest neighbor query or k-nearest neighbor (k-NN) query is a query that returns the k closest data points to a given query point p [121]. k-NN query for k=6 is illustrated in Figure A.6, where all the points within the circle form the k nearest neighbors of p. The best known algorithm for nearest neighbor search over R-tree is the *BF-kNN* [131], which uses a min-priority queue (PQ) that stores nodes of an R-tree as well as data points indexed in it. The key for insertion into priority queue is the *euclidean distance* for data points and *minDist* for nodes (or MBRs of nodes). BF-kNN is a greedy algorithm with minimum distance as the greedy choice. Algorithm A.8 on the next page presents its pseudo code. The BF-kNN algorithm first adds the root node of a given R-tree into PQ. Then in a loop it executes the following steps until k nearest neighbors are found. A REMOVE-MIN() operation is performed over PQ. If the min object is an internal node, it inserts all its indexed entries into PQ with their respective *minDist* from p as key values;

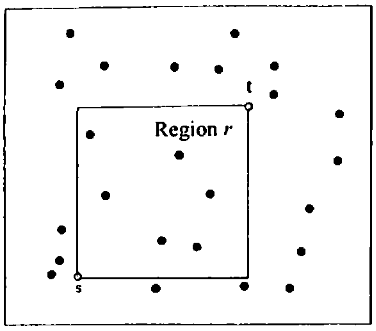


Figure A.4: Illustrating Window Query

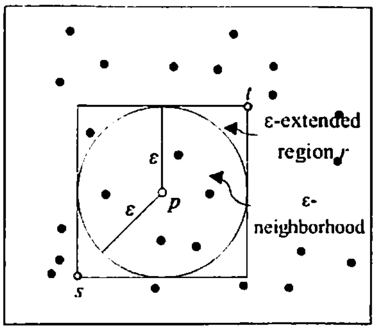


Figure A.5: Illustrating Neighborhood Query

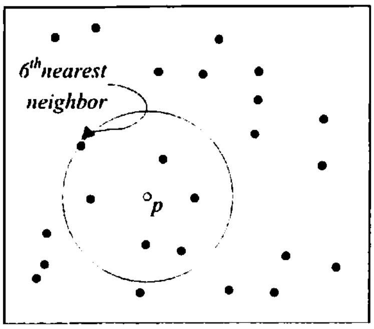


Figure A.6: Illustrating k-NN Query (k=6)

Algorithm A.8: k-Nearest Neighbor Query over R-tree

```
procedure k-NN-QUERY-R-TREE ()
    Input  : data point $q$, $k$, R-tree root node
    Output: $k$ nearest neighbors of $q$
    Initialize Empty Priority Queue $PQ$;  int $i$ = 1;
    Add node into $PQ$, with its minDistance from $p$ as the key;
    while $PQ$ not empty do
        element ele = REMOVE-MIN ($PQ$);
        if ele is internal node of R-tree then
            Add all its entries to $PQ$ with their respective minDist (from $q$) as keys;
        else if ele is external node of R-tree then
            Add all its entries to $PQ$ with their respective euclideanDist (from $q$) as keys;
        else if ele is a data point then
            report ele as $i^{th}$ nearest neighbor;  $i$++;
            if $i > k$ then
                return;
    end
```

If the min object is a penultimate node, it inserts all its indexed entries into PQ with their respective *euclideanDist* from p as key values. If the min object is a data point, it is marked as the i^{th} nearest neighbor. This step is repeated until $i > k$, with i initially set to 1.

The complexity of BF-kNN is dominated by the complexity of priority queue operations [131]. The number of objects inserted into the priority queue is $O(k + \sqrt{k})$ and cost of each insertion is $O(\log \sqrt{k})$, if PQ is implemented as a binary heap. Thus the average case complexity of BF-kNN algorithm is $O(k + \sqrt{k}).O(\log \sqrt{k}) = O(k \log k)$. However, the worst case complexity is $O(N)$, wherein all the nodes in the R-tree are added to the priority queue.

Appendix B

Tilted-Time Window FrameWork (TTWF)

Tilted-Time Window is a framework for storing summary statistics of a data stream. It is used to store the summary entire stream while giving greater weight-age to the recently arrived objects. TTWF is used by problems of anytime clustering and anytime FI mining in Chapter 6 on page 126 and Chapter 4 on page 60 respectively.

TTWF [309] is inspired by the fact that very often people are interested in looking at recent changes at finer granularity and the older changes at coarser granularity. It consists of a few windows whose cardinality is logarithmic in units of time. Consider the logarithmic TTWF shown in Figure B.1. Suppose the latest window w_1 stores the summary statistics of the stream received for last t_{in} units of time. Then, w_2 would store summary statistics for previous t_{in} units of time, w_3 would store for next $2t_{in}$ units of time, $4t_{in}$ units in w_4 and so on, growing at an exponential rate of 2. Essentially, w_1 and w_2 store summary statistics at finest granularity, and as the units of time represented by each window increases, the granularity becomes coarser. Note that each window w_i in TTWF has a temporary window tw_i representing the same units of time (as shown in Figure B.1), except for w_1.

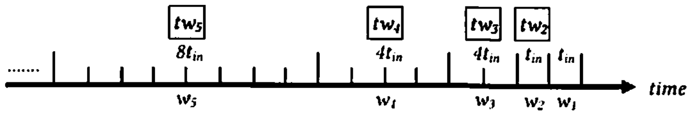


Figure B.1: Tilted-Time Window Framework

The above model can be useful in: (1) finding data patterns for a specific period of time represented by a specific contiguous subset of windows, while giving greater weightage to the recent windows; (2) specifically finding the period where a particular pattern exists in the stream; (3) perform weighted analysis on the windows representing a particular period, etc.

B.0.1 Maintenance of TTWF

Consider Figure B.1. TTWF assumes that objects are arriving in batches (of say t_{in} units of time). Lets start with an empty TTWF. When the first batch arrives (say F_1), the summary statistics from F_1 are stored in window w_1. When second batch arrives (F_2), the summary statistics of F_1 are moved from w_1 to w_2 and summary statistics of F_2 are put into w_1. When next batch arrives (F_3), summary statistics of F_1 are moved from w_2 to tw_2, summary

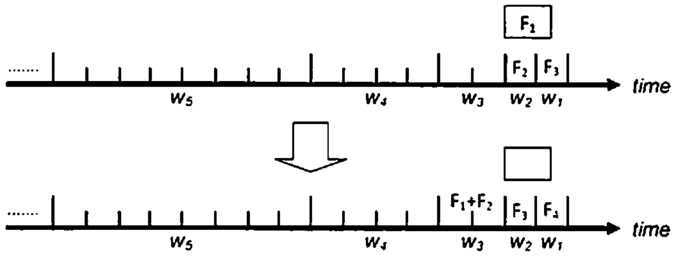


Figure B.2: Updating TTWF after receiving batch F4

statistics of F_2 are moved from w_1 to w_2 and summary statistics of F_3 are put into w_1. When the next batch arrives (F_4), we merge summary statistics of w_2 and tw_2 (F_1 and F_2) and put the merged statistics into w_3 as shown in Figure B.2. Then we move F_3 from w_1 to w_2, and put F_4 into w_1. Going this way, after receiving another 4 batches, summary statistics in w_3 ($F_1 + F_2$) will be placed in tw_3. And after another 4 batches, the summary statistics in w_3 and tw_3 will be merged and stored in w_4 as was done previously. And in this way the TTWF grows. We can clearly observe that total number of windows in TTWF at any point in time is of logarithmic order of units of time for which stream was received. If we consider w_1 to represent the stream received for an hour, then total number of windows for representing the entire stream for a month will be $\lceil(\log(1 \times 24 \times 30))\rceil + 1 = 11$ instead of $24 \times 30 = 720$ windows. Similarly for 1 year and 10 years it would respectively be equal to 15 and 18 windows respectively. This shows that tilted-time window representation is very compact and memory efficient. A user can request for a mining output for a time duration covered by a single or multiple windows. For this, we take summary statistics from those windows and perform offline processing over them.

Appendix C

Quality Evaluation Measures

In this chapter we described a few measures of assessing the quality of data distribution or partitioning or clustering. There are two kinds of measures - *external* & *internal*. External measures are used when the ground truth is available and Internal measures are used when ground truth is not available. We explain a few of these measures that are used in this thesis. They have been summarized in [310].

C.1 External Measures

Let $D = \{x_i\}_{i=1}^{n}$ be a dataset containing n data points of dimensionality d that are partitioned into k partitions. Let $y_i \in \{1,2,...,k\}$ be the ground-truth label (or ground-truth partition membership) for each point. The ground-truth partitioning is given as $T = \{T_1, T_2, ..., T_k\}$, where the partition T_j consists of all the points with label j , i.e., $T_j = \{x_i \in D \mid y_i = j\}$. Also, let $C = \{C_1, ..., C_r\}$ be a partitioning of the same dataset into r partitions, obtained via some clustering or a distribution algorithm, and let $\hat{y}_i \in \{1,2,...,r\}$ denote the obtained label for x_i.

External evaluation measures try to capture the extent to which points from the same ground-truth partition appear in the same obtained partition, and the extent to which points from different ground-truth partitions are grouped in different obtained partitions. All of the external measures rely on the $r \times k$ contingency table N, which is induced by a obtained partitioning C and the ground-truth partitioning T , defined as follows:

$$N(i,j) = n_{ij} = |C_i \cap T_j|$$

The count n_{ij} denotes the number of points that are common to an obtained partition C_i and ground-truth partition T_j . Further, let $n_i = |C_i|$ be the number of points in an obtained partition C_i, and let $m_j = |T_j|$ denote the number of points in ground-truth partition T_j . The contingency table can be computed from T and C in $O(n)$ time by examining the ground truth partitioning and obtained partitioning labels, y_i and $\hat{y}_i$, for each point $x_i \in D$ and incrementing the corresponding count $n_{y_i \hat{y}_i}$.

On the basis of the above discussion, we now describe a few external evaluation measures. Table C.1 on the following page summarizes the external measures used.

Matching Based Measures

Purity. Purity is a measure to quantify the extent to which an obtained partition C_i contains entities from only one ground-truth partition. In other words, it measures how

Table C.1: External Evaluation Measures

Measure	Formula	When is it better?
Purity	$\sum_{i=1}^{r} \frac{n_i}{n} purity_i = \frac{1}{n} \max_{j=1}^{k}\{n_{ij}\}$ where $purity_i = \frac{1}{n_i} \max_{j=1}^{k}\{n_{ij}\}$	High
Precision	$\frac{1}{r} \sum_{i=1}^{r} precision_i$ where $precision_i = \frac{1}{n_i} \max_{j=1}^{k}\{n_{ij}\}$	High
Recall	$\frac{1}{r} \sum_{i=1}^{r} recall_i$ where $recall_i = \frac{n_{ij_i}}{T_{j_i}}$	High
F-measure	$\frac{1}{r} \sum_{i=1}^{r} F_i$ where $F_i = \frac{2\, precision_i\, recall_i}{precision_i + recall_i}$	High
Jaccard Co-efficient	$\frac{TP}{TP+FN+FP}$	High
Pair-wise Precision	$\frac{TP}{TP+TP}$	High
Pair-wise Recall	$\frac{TP}{TP+TN}$	High

"pure" each obtained partition is. Purity of C_i can be defined as:

$$purity_i = \frac{1}{n_i} \max_{j=1}^{k}\{n_{ij}\}$$

The purity of an obtained partitioning C is defined as the weighted sum of the partition-wise purity values:

$$purity = \sum_{i=1}^{r} \frac{n_i}{n} purity_i = \frac{1}{n} \max_{j=1}^{k}\{n_{ij}\}$$

where the ratio $\frac{ni}{n}$ denotes the fraction of points in obtained partition C_i . The maximum value of purity is 1. The larger the purity of obtained partitioning C, the better the agreement with the ground-truth.

Precision. Given an obtained partition C_i, let j_i denote the ground-truth partition that contains the maximum number of points from C_i , that is, $j_i = \max_{j=1}^{k}\{n_{ij}\}$. The precision of a C_i is the same as its purity:

$$precision_i = \frac{1}{n_i} \max_{j=1}^{k}\{n_{ij}\} = \frac{n_{ij_i}}{n_i}$$

It measures the fraction of points in C_i from the majority partition T_{j_i}. The precision for the obtained partitioning C is the mean of partition-wise precision values:

$$Precision = \frac{1}{r} \sum_{i=1}^{r} precision_i$$

Recall. The recall of an obtained partition C_i is defined as

$$recall_i = \frac{n_{ij_i}}{|T_{j_i}|} = \frac{n_{ij_i}}{m_{j_i}}$$

where $m_{j_i} = |T_{j_i}|$. It measures the fraction of points in partition T_{j_i} shared in common with C_i. The recall for an obtained partitioning C is the mean of partition-wise recall values:

$$Recall = \frac{1}{r}\sum_{i=1}^{r} recall_i$$

F-measure. The F-measure (or F1-score) is the harmonic mean of the precision and recall values for each obtained partition. The F-measure for C_i can be written as:

$$F_i = \frac{2}{\frac{1}{precision_i} + \frac{1}{recall_i}} = \frac{2 \cdot precision_i \cdot recall_i}{precision_i + recall_i} = \frac{2 \cdot n_{ij_i}}{n_i + m_{j_i}}$$

The F-measure for an obtained partitioning C is the mean of partition-wise F-measure values:

$$F = \frac{1}{r}\sum_{i=1}^{r} F_i$$

F-measure tries to balance the precision and recall values across all the partitions. For a perfect partitioning, when $r = k$, the maximum value of the F-measure is 1.

Pair-Wise Measures

The pair-wise measures utilize the ground-truth and obtained partition labels over all pairs of points. Let $x_i, x_j \in D$ be any two points, with $i \neq j$. Depending on whether there is agreement between the obtained partition labels and ground-truth partition labels, there are four possibilities to consider:

- **True Positives**: x_i and x_j belong to the same partition in T, as well as in C. The number of all true positive pairs is given as:

$$TP = \left|\{(x_i, x_j) : y_i = y_j \text{ and } \hat{y}_i = \hat{y}_j\}\right|$$

- **False Negatives**: x_i and x_j belong to the same partition in T, but not in C. The number of all false negative pairs is given as:

$$FN = \left|\{(x_i, x_j) : y_i = y_j \text{ and } \hat{y}_i \neq \hat{y}_j\}\right|$$

- **False Positives**: x_i and x_j do not belong to the same partition in T, but they do in C. The number of all false positive pairs is given as:

$$FP = \left|\{(x_i, x_j) : y_i \neq y_j \text{ and } \hat{y}_i = \hat{y}_j\}\right|$$

- **True Negatives**: x_i and x_j neither belong to the same partition in T, nor do they in C. The number of all such true negative pairs is given as:

$$TN = \left|\{(x_i, x_j) : y_i \neq y_j \text{ and } \hat{y}_i \neq \hat{y}_j\}\right|$$

Because there are $N = \binom{n}{2} = \frac{n(n-1)}{2}$ pairs of points, we have the following identity:

$$N = TP + FN + FP + TN$$

Computation of the above four cases requires quadratic time. However, they can be computed more efficiently using the contingency table $N = n_{ij}$, with $1 \leq i \leq r$ and $1 \leq j \leq k$, in $O(n + rk)$ time [310]. We now describe a pairwise assessment measures based on the above four values:

Jaccard Co-efficient. The Jaccard Coefficient measures the fraction of true positive point pairs, but after ignoring the true negatives. It is defined as follows:

$$Jaccard = \frac{TP}{TP + FN + FP}$$

For a perfect partitioning C, the Jaccard takes the value 1, as in that case there are no FPs or FNs. It denotes the similarity in terms of the point pairs that belong together in both the obtained partitioning and the ground-truth partitioning, but ignores the point pairs that do not belong together.

Pairwise Accuracy. The pairwise accuracy for a partitioning C us defined as:

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Accuracy is a simple ratio of correctly predicted point pairs to all the point pairs.

Pairwise Precision & Recall. Pairwise precision and pairwise recall for a partitioning C are defined as:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN}$$

Precision measures the fraction of correctly partitioned point pairs, in the same obtained partition. Recall measures the fraction of correctly labeled points pairs in the same ground-truth partition.

C.2 Internal Measures

Internal evaluation measures are used when the ground-truth partitioning is not available, which is the typical scenario when clustering a dataset or distributing it to computing nodes. To evaluate the quality of the partitioning, internal measures utilize the notions of intra-partition similarity or compactness, when compared with notions of inter-partition separation, with usually a trade-off in maximizing both of them. The internal measures are based on the $n \times n$ distance matrix (proximity matrix) of all pairwise distances among the n points:

$$W = \left\{\delta(x_i, x_j)\right\}_{i,j=1}^{n}$$

where $\delta(x_i, x_j) = ||x_i - x_j||_2$ is the Euclidean distance between $x_i, x_j \in D$. Because W is symmetric and $\delta(x_i, x_i) = 0$, only the upper triangular elements of W (excluding the diagonal) are used in the internal measures. The proximity matrix W can also be considered as the adjacency matrix of the weighted complete graph G over the n points, that is, with nodes $V = \{x_i | x_i \in D\}$, edges $E = \{(x_i, x_j) | x_i, x_j \in D\}$, and edge weights $w_{ij} = W(i, j)$ for all $x_i, x_j \in D$.

Given an obtained partitioning $C = C_1, ..., C_k$ comprising $r = k$ partitions, with partition C_i containing $n_i = |C_i|$ points. The partitioning C can be considered as a k-way cut in G because $C_i \neq \phi$ for all i, $C_i \cap C_j = \phi$ for all i, j, and $\cup_i C_i = V$. Given any subsets $S, R \subset V$, define $W(S, R)$ as the sum of the weights on all edges with one vertex in S and the other in R, given as

$$W(S,R) = \sum_{x_i \in S} \sum_{x_j \in R} w_{ij}$$

Also, given $S \subseteq V$, we denote by $\overline{S}$ the complementary set of vertices, i.e., $\overline{S} = V - S$. The internal measures are based on various functions over the intra-partition and inter-partition weights. In particular, note that the sum of all the intra-partition weights over all partitions is given as

$$W_{in} = \frac{1}{2} \sum_{i=1}^{k} W(C_i, C_i)$$

We divide by 2 because each edge within C_i is counted twice in the summation given by $W(C_i, C_i)$. Also note that the sum of all inter-partition weights is given as

$$W_{out} = \frac{1}{2} \sum_{i=1}^{k} W(C_i, \overline{C_i}) = \sum_{i=1}^{k-1} \sum_{j>i} W(C_i, C_j)$$

Here too we divide by 2 because each edge is counted twice in the summation across partitions. The number of distinct intra-partition edges, denoted N_{in}, and inter-partition edges, denoted N_{out}, are given as

$$N_{in} = \sum_{i=1}^{k} \binom{n_i}{2} = \frac{1}{2} \sum_{i=1}^{k} n_i(n_i - 1)$$

$$N_{out} = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} n_i \cdot n_j = \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1, j \neq i}^{k} n_i \cdot n_j$$

Note that the total number of distinct pairs of points N satisfies the identity

$$N = N_{in} + N_{out} = \binom{n}{2} = \frac{1}{2} n(n-1)$$

Table C.2 on the next page summarizes a few internal quality evaluation measures. They are explained in detail as follows:

BetaCV Measure. The BetaCV measure is the ratio of the mean intra-partition distance to the mean inter-partition distance:

$$BetaCV = \frac{W_{in}/N_{in}}{W_{out}/N_{out}} = \frac{N_{out}}{N_{in}} \cdot \frac{W_{in}}{W_{out}} = \frac{N_{out}}{N_{in}} \cdot \frac{\sum_{i=1}^{k} W(C_i, C_i)}{\sum_{i=1}^{k} W(C_i, \overline{C_i})}$$

The smaller the BetaCV ratio, the better the partitioning, as it indicates that intra-partition distances are on average smaller than inter-partition distances.

Normalized Cut Measure. The normalized cut measure tries to minimize the sum of similarities from a partition C_i to other partition not in C_i, taking account the volume of

Table C.2: Internal Evaluation Measures

Measure	Formula	When is it better?
BetaCV	$\frac{W_{in}/N_{in}}{W_{out}/N_{out}}$	Low
Normalized Cut (NC)	$\sum_{i=1}^{k} \frac{W(C_i,\overline{C_i})}{W(C_i,V)}$	High
Modularity (Q)	$\sum_{i=1}^{k} \left(\frac{W(C_i,C_i)}{W(V,V)} - \left(\frac{W(C_i,V)}{W(V,V)} \right)^2 \right)$	Low
Davies-Bouldin Index (DB)	$\frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \{DB_{ij}\}$ where $DB_{ij} = \frac{\sigma_{\mu_i} + \sigma_{\mu_j}}{\delta(\mu_i,\mu_j)}$	Low
Silhouette Co-efficient (SIL)	$\frac{1}{n} \sum_{i=1}^{n} sil_i$ where $sil_i = \frac{\mu_{out}^{min}(x_i) - \mu_{in}(x_i)}{\max\{\mu_{out}^{min}(x_i), \mu_{in}(x_i)\}}$	High
Normalized Hubert Statistic (Γ^n)	$\frac{\sigma_{XY}}{\sqrt{\sigma_X^2 \sigma_Y^2}}$	High

the partition:

$$NC = \sum_{i=1}^{k} \frac{W(C_i,\overline{C_i})}{vol(C_i)} = \sum_{i=1}^{k} \frac{W(C_i,\overline{C_i})}{W(C_i,V)}$$

where $vol(C_i) = W(C_i, V)$ is the volume of obtained partition C_i, that is, the total weights on edges with at least one end in the partition. Since, we are using the proximity or distance matrix W, instead of the affinity or similarity matrix, the higher the normalized cut value the better.

Modularity. The modularity objective is measured as:

$$Q = \sum_{i=1}^{k} \left(\frac{W(C_i,C_i)}{W(V,V)} - \left(\frac{W(C_i,V)}{W(V,V)} \right)^2 \right)$$

where

$$W(V,V) = \sum_{i=1} kW(C_i,V) = \sum_{i=1} kW(C_i,C_i) + \sum_{i=1}^{k} W(C_i,\overline{C_i}) = 2(W_{in} + W_{out})$$

Modularity measures the difference between the observed and expected fraction of weights on edges within the partitions. Since we are using the distance matrix, the smaller the modularity measure the better the partitioning, which indicates that the intra-partition distances are lower than expected.

Davies-Bouldin Index. Let μ_i denote the partition mean, given as

$$\mu_i = \frac{1}{n_i} \sum_{x_j \in C_i} x_j$$

Further, let $_i$ denote the dispersion or spread of the points around the partition mean, given as

$$\sigma_{\mu_i} = \sqrt{\frac{\sum_{x_j \in C_i} \delta(x_j, \mu_i)^2}{n_i}} = \sqrt{var(C_i)}$$

where var(C_i) is the total variance of partition C_i. The Davies-Bouldin measure for a pair of partitions Ci and Cj is defined as the ratio

$$DB_{ij} = \frac{\sigma_{\mu_i} + \sigma_{\mu_j}}{\delta(\mu_i, \mu_j)}$$

DB_{ij} measures how compact the partitions are compared to the distance between the partition means. The Davies-Bouldin index is then defined as

$$DB_{ij} = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \{DB_{ij}\}$$

That is, for each obtained partition C_i, we pick the C_j that yields the largest DB_{ij} ratio. The smaller the DB value the better the partitioning, as it means that the partitions are well separated (i.e., the distance between partition means is large), and each partition is well represented by its mean (i.e., has a small spread).

Silhouette Coefficient. The silhouette coefficient is a measure of both cohesion and separation of partitions, and is based on the difference between the average distance to points in the closest partition and to points in the same partition. For each point x_i we calculate its silhouette co-efficient sil_i as

$$sil_i = \frac{\mu_{out}^{min}(x_i)\mu_{in}(x_i)}{\max\left\{\mu_{out}^{min}(x_i), \mu_{in}(x_i)\right\}}$$

where $\mu_{in}(x_i)$ is the mean distance from x_i to points in its own partition $\hat{y}_i$:

$$\mu_{in}(x_i) = \frac{\sum_{x_j \in C_{\hat{y}_i}, j \neq i} \delta(x_i, x_j)}{n_{\hat{y}_i} - 1}$$

and $\mu_{out}^{min}(xi)$ is the mean of the distances from x_i to points in the closest partition:

$$\mu_{out}^{min}(xi) = \min_{j \neq \hat{y}_i} \left\{ \frac{\sum_{y \in C_j} \delta(x_i, y)}{n_j} \right\}$$

The sil_i value of a point lies in the interval $[-1, +1]$. A value close to $+1$ indicates that x_i is much closer to points in its own partition and is far from other partitions. A value close to zero indicates that x_i is close to the boundary between two partitions. Finally, a value close to -1 indicates that x_i is much closer to another partition than its own partition, and therefore, the point may be mis-partitioned. The silhouette coefficient is defined as the mean sil_i value across all the points:

$$SIL = \frac{1}{n} \sum_{i=1}^{n} sil_i$$

A value close to $+1$ indicates a good partitioning.

Normalized Hubert Statistic. Let X and Y be two symmetric $n \times n$ matrices, and let $N = \binom{n}{2}$. Let $x,y \in R^N$ denote the vectors obtained by linearizing the upper triangular elements (excluding the main diagonal) of X and Y (e.g., in a row-wise manner), respectively. Let

μ_X denote the element-wise mean of x, given as

$$\mu_X = \frac{1}{N} \sum_{i=1} n - 1 \sum_{j=i+1} n X(i,j) = \frac{1}{N} x^T x$$

and let z_x denote the centered x vector, defined as

$$z_x = x - \mathbf{1} \cdot \mu_X$$

where $\mathbf{1} \in R^N$ is the vector of all ones. Likewise, let μ_Y be the element-wise mean of y, and z_y the centered y vector. The normalized Hubert statistic is defined as the element-wise correlation between X and Y,

$$\Gamma^n = \frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (X(i,j) - \mu_X)(Y(i,j) - \mu_Y)}{\sqrt{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (X(i,j) - \mu_X)^2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (Y(i,j) - \mu_Y)^2}} = \frac{\sigma_{XY}}{\sqrt{\sigma_X^2 \sigma_Y^2}}$$

where σ_X^2, and σ_Y^2 are the variances, and σ_{XY} the covariance, for the vectors x and y. The normalized Hubert statistic can be used as internal evaluation measure by letting $X = W$ be the pairwise distance matrix, and by defining Y as the matrix of distances between the partition means:

$$Y = \left\{ \delta(\mu_{\hat{y}_i}, \mu_{\hat{y}_j}) \right\}_{i,j=1}^{n}$$

Because both W and Y are symmetric, Γ_n is computed over their upper triangular elements. The higher its value, the better the partitioning.

References

[1] M. H. Dunham, *Data mining introductory and advanced topics*, 1st ed. Prentice Hall, 2003. ↑xiii, ↑1, ↑2

[2] Statista, "Number of internet users worldwide 2005-2018," 2018. [Online]. Available: https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/ ↑xiii, ↑3, ↑4

[3] Domo, "Data Never Sleeps 6," 2018. [Online]. Available: https://www.domo.com/learn/data-never-sleeps-6 ↑xiii, ↑3, ↑5

[4] IDC, "IDC Study: The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things Sponsored by EMC," 2018. [Online]. Available: https://www.emc.com/leadership/digital-universe/2014iview/index.htm ↑xiii, ↑4, ↑6

[5] P. Kudupu, "Web Snippets: 7 v's of big data," 2018. [Online]. Available: http://www.prathapkudupublog.com/2018/01/7-vs-of-big-data.html ↑xiii, ↑6, ↑7

[6] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Pearson Addison Wesley, 2005. ↑1, ↑2, ↑16, ↑80, ↑142, ↑153, ↑159, ↑164

[7] B. M. E. Moret, "Decision Trees and Diagrams," *ACM Computing Surveys*, vol. 14, no. 4, pp. 593–623, 12 1982. ↑2

[8] J. R. Quinlan, "Learning decision tree classifiers," *ACM Computing Surveys*, vol. 28, no. 1, pp. 71–72, 3 1996. ↑2

[9] S. B. Kotsiantis, "Decision trees: a recent overview," *Artificial Intelligence Review*, vol. 39, no. 4, pp. 261–283, 4 2013. ↑2

[10] H. Schütze, D. A. Hull, and J. O. Pedersen, "A comparison of classifiers and document representations for the routing problem," in *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '95*. New York, New York, USA: ACM Press, 1995, pp. 229–237. ↑2

[11] E. Wiener, J. O. Pedersen, and A. S. Weigend, "A Neural Network Approach to Topic Spotting," in *Proceedings of 4th annual symposium on document analysis and information retrieval (SDAIR'95)*, 1995, pp. 332–347. ↑2

[12] J. Furnkranz and G. Widmer, "Incremental reduced error pruning," in *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*. Morgan Kaufmann, CA, 1994, pp. 70–77. ↑2

[13] W. W. Cohen, "Fast Effective Rule Induction," in *Proceedings of the Twelfth International Conference on Machine Learning (ICML'95)*. Morgan Kaufmann, 1 1995, pp. 115–123. ↑2

[14] T. T. T. Nguyen, T. T. Nguyen, A. W.-C. Liew, and S.-L. Wang, "Variational inference based bayes online classifiers with concept drift adaptation," *Pattern Recognition*, vol. 81, pp. 280–293, 9 2018. ↑2, ↑57, ↑103

[15] D. Barbará, N. Wu, and S. Jajodia, "Detecting Novel Network Intrusions Using Bayes Estimators," in *Proceedings of the 2001 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 4 2001, pp. 1–17. ↑2

[16] I. Mani and I. Zhang, "KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction I BibSonomy," in *Proceedings of workshop on learning from imbalanced datasets (in ICML 2003)*. Morgan Kaufmann, 2003, pp. 1–7. ↑2

[17] V. N. Vapnik, *The nature of statistical learning theory*. Springer, 1995. ↑2

[18] N. Cristianini and J. Shawe-Taylor, *An introduction to support vector machines : and other kernel-based learning methods*. Cambridge University Press, 2000. ↑2

[19] G. Seni and J. F. Elder, "Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions," *Synthesis Lectures on Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 1–126, 1 2010. ↑2

[20] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up? Sentiment classification using machine learning techniques," in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing (EMNLP'02)*. Association for Computational Linguistics, 2002, pp. 79–86. ↑2

[21] C. Phua, D. Alahakoon, and V. Lee, "Minority report in fraud detection," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 50–59, 6 2004. ↑2

[22] A. Srivastava, A. Kundu, S. Sural, and A. Majumdar, "Credit Card Fraud Detection Using Hidden Markov Model," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 1, pp. 37–48, 1 2008. ↑2

[23] E. Blanzieri and A. Bryl, "A survey of learning-based techniques of email spam filtering," *Artificial Intelligence Review*, vol. 29, no. 1, pp. 63–92, 3 2008. ↑2

[24] D. Bazell and D. Aha, "Ensembles of Classifiers for Morphological Galaxy Classification," *The Astrophysical Journal*, vol. 548, no. 1, pp. 219–223, 2 2001. ↑2

[25] J. Khan, J. S. Wei, M. Ringnér, L. H. Saal, M. Ladanyi, F. Westermann, F. Berthold, M. Schwab, C. R. Antonescu, C. Peterson, and P. S. Meltzer, "Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks," *Nature Medicine*, vol. 7, no. 6, pp. 673–679, 6 2001. ↑2

[26] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, vol. 1*. USA, 1967, pp. 281–297. ↑2, ↑153

[27] S. Kumari, A. Maheshwari, P. Goyal, and N. Goyal, "Parallel Framework for Efficient k-means Clustering," in *Proceedings of the 8th Annual ACM India Conference on - Compute '15*. ACM, 2015, pp. 63–71. ↑2, ↑153, ↑159

[28] J. Zhang, G. Wu, X. Hu, S. Li, and S. Hao, "A Parallel K-Means Clustering Algorithm with MPI," in *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*. IEEE, 12 2011, pp. 60–64. ↑2, ↑153, ↑159, ↑168, ↑169

[29] H. Song, J.-G. Lee, and W.-S. Han, "PAMAE: Parallel k-Mediods Clustering with High Accuracy and Efficiency," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*. ACM New York, USA, 2017, pp. 1087–1096. ↑2, ↑153, ↑159

[30] R. Sibson, "SLINK: An optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1 1973. ↑2, ↑16, ↑153, ↑164

[31] W. Hendrix, M. M. Ali Patwary, A. Agrawal, W.-k. Liao, and A. Choudhary, "Parallel hierarchical clustering on shared memory platforms," in *2012 19th International Conference on High Performance Computing*. IEEE, 12 2012, pp. 1–9. ↑2, ↑154, ↑156, ↑165

[32] P. Goyal, S. Kumari, S. Sharma, V. Kishore, N. Goyal, and S. S. Balasubramaniam, "Spatial Locality Aware, Fast, and Scalable SLINK Algorithm for Commodity Clusters," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 9 2016, pp. 158–159. ↑2, ↑54, ↑166, ↑168, ↑169, ↑173, ↑181, ↑186

[33] P. Goyal, S. Kumari, S. Sharma, D. Kumar, V. Kishore, S. Balasubramaniam, and N. Goyal, "A Fast, Scalable SLINK Algorithm for Commodity Cluster Computing Exploiting Spatial Locality," in *2016 IEEE 18th International Conference on High Performance Computing and Communications*. IEEE, 2016, pp. 268–275. ↑2, ↑54, ↑153, ↑155, ↑156, ↑166, ↑168, ↑173, ↑174, ↑181, ↑186

[34] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 226–231. ↑2, ↑16, ↑17, ↑153, ↑160, ↑210

[35] M. Ankerst, M. M. Breunig, H. P. Kriegel, and J. Sander, "OPTICS: Ordering Points To Identify the Clustering Structure," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*. ACM New York, 1999, pp. 49–60. ↑2, ↑16, ↑17, ↑53, ↑153, ↑162, ↑210

[36] S. Kumari, P. Goyal, A. Sood, D. Kumar, S. Balasubramaniam, and N. Goyal, "Exact, Fast and Scalable Parallel DBSCAN for Commodity Platforms," in *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17*. ACM New York, 2017, pp. 1–10. ↑2, ↑8, ↑54, ↑140, ↑153, ↑155, ↑156, ↑161, ↑168, ↑169, ↑170, ↑177, ↑186

[37] S. Goil, H. Nagesh, and A. Choudhary, "MAFIA: Efficient and Scalable Subspace Clustering for Very Large Data Sets," in *Proceedings of 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1999, pp. 443–452. ↑2, ↑153, ↑167

[38] K.-G. Woo, J.-H. Lee, M.-H. Kim, and Y.-J. Lee, "FINDIT: a fast and intelligent subspace clustering algorithm using dimension voting," *Information and Software Technology*, vol. 46, no. 4, pp. 255–271, 3 2004. ↑2, ↑153, ↑166

[39] L. Parsons, E. Haque, and H. Liu, "Subspace clustering for high dimensional data," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 90–105, 6 2004. ↑2

[40] R. L. Breiger, S. A. Boorman, and P. Arabie, "An algorithm for clustering relational data with applications to social network analysis and comparison with multidimensional scaling," *Journal of Mathematical Psychology*, vol. 12, no. 3, pp. 328–383, 8 1975. ↑3

[41] S. Zhang, R.-S. Wang, and X.-S. Zhang, "Identification of overlapping community structure in complex networks using fuzzy c-means clustering," *Physica A: Statistical Mechanics and its Applications*, vol. 374, no. 1, pp. 483–490, 1 2007. ↑3

[42] A. Shepitsen, J. Gemmell, B. Mobasher, and R. Burke, "Personalized recommendation in social tagging systems using hierarchical clustering," in *Proceedings of the 2008 ACM conference on Recommender systems - RecSys '08*. New York, New York, USA: ACM Press, 2008, p. 259. ↑3

[43] G. Coleman and H. Andrews, "Image segmentation by clustering," *Proceedings of the IEEE*, vol. 67, no. 5, pp. 773–785, 1979. ↑3

[44] G. Stockman, S. Kopstein, and S. Benett, "Matching Images to Models for Registration and Object Detection via Clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 3, pp. 229–241, 5 1982. ↑3

[45] A. Tramacere and C. Vecchio, "Gamma-ray DBSCAN: a clustering algorithm applied to Fermi -LAT Gamma-ray data," *Astronomy & Astrophysics*, vol. 549, p. A138, 1 2013. ↑3

[46] X.-H. Gao, "Membership determination of open cluster NGC 188 based on the DBSCAN clustering algorithm," *Research in Astronomy and Astrophysics*, vol. 14, no. 2, pp. 159–164, 2 2014. ↑3

[47] R. Agrawal, T. Imieliński, A. Swami, R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, 1993. ↑3, ↑60, ↑79, ↑101

[48] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 55–86, 7 2007. ↑3

[49] C. C. Aggarwal and J. Han, *Frequent Pattern Mining*. Springer Publishing Company, 2014. ↑3

[50] P.-Y. Hsu, Y.-L. Chen, and C.-C. Ling, "Algorithms for mining association rules in bag databases," *Information Sciences*, vol. 166, no. 1-4, pp. 31–47, 10 2004. ↑3, ↑60, ↑79, ↑101

[51] M. Zaki, "Scalable algorithms for association mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 3, pp. 372–390, 2000. ↑3, ↑60, ↑62, ↑101

[52] J. Han, J. Pei, Y. Yin, J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00*, vol. 29, no. 2. New York, New York, USA: ACM Press, 2000, pp. 1–12. ↑3

[53] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee, "Sliding window-based frequent pattern mining over data streams," *Information Sciences*, vol. 179, no. 22, pp. 3843–3865, 11 2009. ↑3, ↑57, ↑60, ↑61, ↑62, ↑79, ↑101

[54] J. Blanchard, F. Guillet, R. Gras, and H. Briand, "Using Information-Theoretic Measures to Assess Association Rule Interestingness," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 2006, pp. 66–73. ↑3

[55] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Efficient mining of association rules using closed itemset lattices," *Information Systems*, vol. 24, no. 1, pp. 25–46, 3 1999. ↑3

[56] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, 7 2009. ↑3

[57] Z. He, X. Xu, and S. Deng, "Discovering cluster-based local outliers," *Pattern Recognition Letters*, vol. 24, no. 9-10, pp. 1641–1650, 6 2003. ↑3

[58] L. Portnoy, "Intrusion detection with unlabeled data using clustering," in *Proceedings of the ACM Workshop on Data Mining Applied to Security*, 2001, pp. 1–25. ↑3

[59] S. Hawkins, H. He, G. Williams, and R. Baxter, "Outlier Detection Using Replicator Neural Networks," in *Outlier detection using replicator neural networks (DaWaK 2002)*. Springer, Berlin, Heidelberg, 2002, pp. 170–180. ↑3

[60] D. Martinez, "Neural tree density estimation for novelty detection," *IEEE Transactions on Neural Networks*, vol. 9, no. 2, pp. 330–338, 3 1998. ↑3

[61] D. Barbará, N. Wu, and S. Jajodia, "Detecting Novel Network Intrusions Using Bayes Estimators," in *Proceedings of the 2001 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 4 2001, pp. 1–17. ↑3

[62] Ghosh and Reilly, "Credit card fraud detection with a neural-network," in *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences HICSS-94*. IEEE Comput. Soc. Press, 1994, pp. 621–630. ↑3

[63] S. Singh and M. Markou, "An approach to novelty detection applied to the classification of image regions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 4, pp. 396–406, 4 2004. ↑3

[64] CISCO, "Cisco Global Cloud Index: Forecast and Methodology, 2016 to 2021 White Paper," 2019. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html ↑4

[65] "Self-driving Cars Will Create 2 Petabytes Of Data, What Are The Big Da," 2019. [Online]. Available: https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172 ↑4

[66] R. Magoulas, "Roger Magoulas on Big Data - O'Reilly Radar," 2010. [Online]. Available: http://radar.oreilly.com/2010/01/roger-magoulas-on-big-data.html ↑5

[67] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004, pp. 1–13. ↑7

[68] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 5 2010, pp. 1–10. ↑7

[69] A. Ghoting, P. Kambadur, E. Pednault, and R. Kannan, "NIMBLE," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '11*. New York, New York, USA: ACM Press, 2011, pp. 334–342. ↑7

[70] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association Berkeley, 2012, pp. 1–14. ↑7

[71] M. P. Forum, "MPI: A Message-Passing Interface Standard," University of Tennessee, Tech. Rep., 1994. ↑7

[72] "Home - OpenMP," 2019. [Online]. Available: https://www.openmp.org/ ↑7, ↑8

[73] Intel, "Threading Building Blocks," 2017. [Online]. Available: https://www.threadingbuildingblocks.org/ ↑7

[74] Zhang Zhang, J. Savant, and S. Seidel, "A UPC runtime system based on MPI and POSIX threads," in *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*. IEEE, 2006, pp. 1–8. ↑7

[75] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty, "Fortran 2008 coarrays," *ACM SIGPLAN Fortran Forum*, vol. 34, no. 1, pp. 10–30, 4 2015. ↑7

[76] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," University of California at Berkeley, Berkeley CA, Tech. Rep., 1988. ↑7

[77] "PNNL: Global Arrays Toolkit," 2018. [Online]. Available: https://hpc.pnl.gov/globalarrays/ ↑7

[78] "The X10 Programming Language," 2019. [Online]. Available: http://x10-lang.org/ ↑7

[79] "Intel® Xeon PhiâĎć Processors," 2019. [Online]. Available: https://www.intel.in/content/www/in/en/products/processors/xeon-phi/xeon-phi-processors.html ↑8

[80] "CUDA Zone | NVIDIA Developer," 2019. [Online]. Available: https://developer.nvidia.com/cuda-zone ↑8

[81] "OpenCL Overview - The Khronos Group Inc," 2019. [Online]. Available: https://www.khronos.org/opencl/ ↑8

[82] "Homepage | OpenACC," 2019. [Online]. Available: https://www.openacc.org/ ↑8

[83] Q. He, F. Zhuang, J. Li, and Z. Shi, "Parallel Implementation of Classification Algorithms Based on MapReduce," in *International Conference on Rough Sets and Knowledge Technology (RSKT 2010)*. Springer, Berlin, Heidelberg, 10 2010, pp. 655–662. ↑8

[84] A. Srivastava, Eui-Hong Sam Han, V. Singh, and V. Kumar, "Parallel formulations of decision-tree classification algorithms," in *Proceedings. 1998 International Conference on Parallel Processing (Cat. No.98EX205)*. IEEE Comput. Soc, 1998, pp. 237–244. ↑8

[85] M. Joshi, G. Karypis, and V. Kumar, "ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE Comput. Soc, 1998, pp. 573–579. ↑8

[86] M. M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 11 2012, pp. 1–11. ↑8, ↑153, ↑155, ↑156, ↑161, ↑168, ↑169, ↑170, ↑177, ↑186

[87] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, "Pardicle: Parallel Approximate Density-Based Clustering," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 11 2014, pp. 560–571. ↑8, ↑153, ↑161, ↑168

[88] W. Hendrix, D. Palsetia, M. M. A. Patwary, A. Agrawal, W.-k. Liao, and A. Choudhary, "A scalable algorithm for single-linkage hierarchical clustering on distributed-memory architectures," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 10 2013, pp. 7–13. ↑8, ↑153, ↑154, ↑156, ↑165, ↑168, ↑169

[89] P. Goyal, S. Kumari, S. Singh, V. Kishore, S. S. Balasubramaniam, and N. Goyal, "A Parallel Framework for Grid-Based Bottom-Up Subspace Clustering," in *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 10 2016, pp. 331–340. ↑8, ↑153, ↑156, ↑167, ↑168

[90] Z.-g. Wang and C.-s. Wang, "A Parallel Association-Rule Mining Algorithm," in *Proceedings of International Conference on Web Information Systems and Mining (WISM 2012)*. Springer, Berlin, Heidelberg, 10 2012, pp. 125–129. ↑8

[91] M. H. Marghny and H. E. Refaat, "A new parallel association rule mining algorithm on distributed shared memory system," *International Journal of Business Intelligence and Data Mining*, vol. 7, no. 4, pp. 233–252, 2012. ↑8

[92] R. Agrawal and J. Shafer, "Parallel mining of association rules," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 962–969, 1996. ↑8

[93] "Apache Mahout," 2019. [Online]. Available: https://mahout.apache.org/ ↑8

[94] "MLlib | Apache Spark," 2019. [Online]. Available: https://spark.apache.org/mllib/ ↑8

[95] F. Cao, M. Estert, W. Qian, and A. Zhou, "Density-Based Clustering over an Evolving Data Stream with Noise," in *Proceedings of the 2006 SIAM International Conference on Data Mining*. SIAM, 2006, pp. 328–339. ↑8, ↑57, ↑126, ↑127, ↑129

[96] H.-F. Li, M.-K. Shan, and S.-Y. Lee, "DSM-FI: an efficient algorithm for mining frequent itemsets in data streams," *Knowledge and Information Systems*, vol. 17, no. 1, pp. 79–97, 10 2008. ↑8, ↑57, ↑61, ↑64, ↑76, ↑79

[97] H. Chen, L. Shu, J. Xia, and Q. Deng, "Mining frequent patterns in a varying-size sliding window of online transactional data streams," *Information Sciences*, vol. 215, pp. 15–36, 12 2012. ↑8, ↑57, ↑61, ↑62, ↑65, ↑75, ↑76, ↑79, ↑101

[98] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining Frequent Patterns in Data Streams at Multiple Time Granularities," in *Data Mining: Next Generation Challenges and Future Directions*. AAAI/MIT Press, 2003, pp. 191–212. ↑8, ↑57, ↑61, ↑62, ↑65, ↑79, ↑88, ↑101, ↑140

[99] Y.-N. Law and C. Zaniolo, "An Adaptive Nearest Neighbor Classification Algorithm for Data Streams," in *Proceedings of European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2005)*. Springer, Berlin, Heidelberg, 2005, pp. 108–120. ↑8, ↑57, ↑103

[100] K. Ueno, X. Xi, E. Keogh, and D.-j. Lee, "Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining," in *Sixth International Conference on Data Mining (ICDM'06)*. IEEE, 12 2006, pp. 623–632. ↑8, ↑9, ↑58, ↑110

[101] T. Seidl, I. Assent, P. Kranen, R. Krieger, and J. Herrmann, "Indexing density models for incremental learning and anytime classification on data streams," in *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*. New York, New York, USA: ACM Press, 2009, p. 311. ↑8, ↑58, ↑110

[102] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A Framework for Clustering Evolving Data Streams," in *Proceedings of 29th International Conference on Very Large Databases*. ACM New York, 2003, pp. 81–92. ↑8, ↑57, ↑126, ↑127, ↑129

[103] C. C. Aggarwal, J. Han, and P. S. Yu, "A Framework for Projected Clustering of High Dimensional Data Streams," in *Proceedings of the Thirtieth international conference on Very large data bases*. Morgan Kaufmann Publishers, 2004, pp. 852–863. ↑8, ↑57, ↑126, ↑127, ↑129

[104] P. Kranen, I. Assent, C. Baldauf, and T. Seidl, "The ClusTree: indexing micro-clusters for anytime stream mining," *Knowledge and Information Systems*, vol. 29, no. 2, pp. 249–272, 11 2011. ↑8, ↑9, ↑58, ↑127, ↑129, ↑131, ↑143

[105] H. Chen, "Mining top-k frequent patterns over data streams sliding window," *Journal of Intelligent Information Systems*, vol. 42, no. 1, pp. 111–131, 2 2014. ↑8, ↑62, ↑79

[106] C. H. Park, "Anomaly Pattern Detection on Data Streams," in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 1 2018, pp. 689–692. ↑8, ↑57

[107] Tran Manh Thang and Juntae Kim, "The Anomaly Detection by Using DBSCAN Clustering with Multiple Parameters," in *2011 International Conference on Information Science and Applications*. IEEE, 4 2011, pp. 1–5. ↑8, ↑153

[108] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 11 2017. ↑8, ↑57

[109] L. Rettig, M. Khayati, P. Cudre-Mauroux, and M. Piorkowski, "Online anomaly detection over Big Data streams," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 10 2015, pp. 1113–1122. ↑9, ↑57

[110] Y. Jiang, C. Zeng, J. Xu, School of Computer Science, Technology and Engineering, Nanjing University of Science, and Technology, C. Nanjing, dolphin.xu@njust.edu.cn, and Tao Li, "Real time contextual collective anomaly detection over multiple data streams," in *Proceedings of KDD Workshop on Outlier Detection and Description under Data Diversity*. ACM New York, USA, 2014, pp. 1–8. ↑9, ↑57

[111] C. Y. Sang and D. H. Sun, "Co-clustering over multiple dynamic data streams based on non-negative matrix factorization," *Applied Intelligence*, vol. 41, no. 2, pp. 487–502, 2014. ↑9, ↑127

[112] L. Tu, "Clustering on Multiple Data Streams," in *Advances in Intelligent and Soft Computing*. Springer, Berlin, Heidelberg, 2012, pp. 73–78. ↑9, ↑127

[113] M.-Y. Yeh, B.-R. Dai, and M.-S. Chen, "Clustering over Multiple Evolving Streams by Events and Correlations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 10, pp. 1349–1362, 10 2007. ↑9, ↑127

[114] D. Apiletti, E. Baralis, T. Cerquitelli, P. Garza, F. Pulvirenti, and L. Venturini, "Frequent Itemsets Mining for Big Data: A Comparative Analysis," *Big Data Research*, vol. 9, pp. 67–83, 9 2017. ↑9

[115] Z. Yu, X. Yu, Y. Liu, W. Li, and J. Pei, "Mining Frequent Co-occurrence Patterns across Multiple Data Streams," in *Proceedings of International Conference on Extending Database Technology (EDBT'15)*. Open Proceedings, 2015, pp. 73–84. ↑9, ↑57, ↑63

[116] P. Kranen, M. Hassani, and T. Seidl, "BT* - An Advanced Algorithm for Anytime Classification," in *Proceedings of the 24th international conference on Scientific and Statistical Database Management*. Springer-Verlag, 2012, pp. 298–315. ↑9, ↑58, ↑110

[117] M. Hassani, P. Kranen, and T. Seidl, "Precise anytime clustering of noisy sensor data with logarithmic complexity," in *Proceedings of Fifth International Workshop on Knowledge Discovery from Sensor Data - SensorKDD '11*. ACM New York, 2011, pp. 52–60. ↑9, ↑58, ↑127, ↑128, ↑129

[118] M. Hassani, P. Kranen, R. Saini, and T. Seidl, "Subspace anytime stream clustering," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management - SSDBM '14*. ACM, 2014, pp. 1–4. ↑9, ↑58, ↑127, ↑128

[119] I. Assent, P. Kranen, C. Baldauf, and T. Seidl, "AnyOut: Anytime Outlier Detection on Streaming Data," in *Proceedings of the 17th international conference on Database Systems for Advanced Applications - Volume Part I*. Springer-Verlag, 2012, pp. 228–242. ↑9, ↑58

[120] P. Goyal, S. Kumari, D. Kumar, S. Balasubramaniam, N. Goyal, S. Islam, and J. S. Challa, "Parallelizing OPTICS for Commodity Clusters," in *Proceedings of the 2015 International Conference on Distributed Computing and Networking - ICDCN '15*. New York, New York, USA: ACM Press, 2015, pp. 1–10. ↑16, ↑155, ↑156, ↑162, ↑168, ↑169, ↑186

[121] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1 1967. ↑16, ↑32, ↑50, ↑146, ↑210

[122] A. Guttman, "R-trees," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*. ACM New York, 1984, pp. 47–57. ↑16, ↑112, ↑130, ↑162, ↑186, ↑206, ↑208

[123] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications*. Springer Publishing Company, 2005. ↑16, ↑38, ↑49

[124] J. L. Bentley and J. Louis, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 9 1975. ↑16, ↑46

[125] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974. ↑16

[126] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems*, vol. 9, no. 1, pp. 38–71, 1 1984. ↑16, ↑46

[127] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database systems : the complete book*. Pearson Prentice Hall, 2009. ↑17, ↑47

[128] G. Li and J. Tang, "A New R-tree Spatial Index Based on Space Grid Coordinate Division," in *Proceedings of the 2011 International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE2011)*. Springer, Berlin, Heidelberg, 2011, pp. 133–140. ↑17, ↑47

[129] E. Schikuta, "Grid-clustering: an efficient hierarchical clustering method for very large data sets," in *Proceedings of 13th International Conference on Pattern Recognition*. IEEE, 1996, pp. 101–105. ↑21, ↑54

[130] W.-k. Liao, L. Ying, and A. Choudhary, "A grid-based Clustering Algorithm using Adaptive Mesh Refinement," in *Proceedings of the 7th Workshop on Mining Scienti c and Engineering Data Sets*, 2004. ↑21, ↑54

[131] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Transactions on Database Systems*, vol. 24, no. 2, pp. 265–318, 6 1999. ↑32, ↑33, ↑210, ↑211

[132] C. Faloutsos, T. Sellis, N. Roussopoulos, C. Faloutsos, T. Sellis, and N. Roussopoulos, "Analysis of object oriented spatial access methods," in *Proceedings of the 1987 ACM SIGMOD international conference on Management of data - SIGMOD '87*, vol. 16, no. 3. New York, New York, USA: ACM Press, 1987, pp. 426–439. ↑33

[133] "Vampir Trace Library," 2013. ↑36

[134] M. Kaul, B. Yang, and C. S. Jensen, "Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems," in *2013 IEEE 14th International Conference on Mobile Data Management*. IEEE, 6 2013, pp. 137–146. ↑37

[135] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars," *Nature*, vol. 435, no. 7042, pp. 629–636, 6 2005. ↑37, ↑142, ↑193, ↑194

[136] "SUVN Trace Data," 2012. [Online]. Available: http://wirelesslab.sjtu.edu.cn ↑37, ↑193, ↑194

[137] "KDD Cup 2004 Bio Dataset," 2004. [Online]. Available: http://cs.joensuu.fi/sipu/datasets/ ↑37

[138] J. Catlett, "Statlog (Shuttle) Data Set," 1993. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle) ↑37, ↑50

[139] R. Bhatt and A. Dhall, "Skin Segmentation Data Set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation ↑37, ↑50

[140] B. Borah and D. Bhattacharyya, "An improved sampling-based DBSCAN for large spatial databases," in *Proceedings of 2004 International Conference on Intelligent Sensing and Information Processing*. IEEE, 2004, pp. 92–96. ↑53, ↑54

[141] C.-F. Tsai and C.-W. Liu, "KIDBSCAN: A New Efficient Data Clustering Algorithm," in *International Conference on Artificial Intelligence and Soft Computing (ICAISC 2006)*. Springer, Berlin, Heidelberg, 2006, pp. 702–711. ↑53, ↑54

[142] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. New York, New York, USA: ACM Press, 1967, pp. 483–485. ↑53

[143] E. Schikuta and M. Erhart, "The BANG-clustering system: Grid-based data analysis," in *International Symposium on Intelligent Data Analysis (IDA 1997)*. Springer, Berlin, Heidelberg, 1997, pp. 513–524. ↑54

[144] W. Wang, J. Yang, and R. R. Muntz, "STING: A Statistical Information Grid Approach to Spatial Data Mining," in *Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann, 1997, pp. 186–195. ↑54, ↑153

[145] C.-F. Tsai and Chun-Yi Sung, "DBSCALE: An efficient density-based clustering algorithm for data mining in large databases," in *2010 Second Pacific-Asia Conference on Circuits, Communications and System*. IEEE, 8 2010, pp. 98–101. ↑54

[146] R. Jin and G. Agrawal, "Efficient decision tree construction on streaming data," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*. Washington, D.C.: ACM Press, New York, New York, USA, 2003, pp. 571–576. ↑57, ↑103

[147] D. Jankowski, K. Jackowski, and B. Cyganek, "Learning Decision Trees from Data Streams with Concept Drift," *Procedia Computer Science*, vol. 80, no. C, pp. 1682–1691, 2016. ↑57, ↑103

[148] Q. Xue, B.-w. Cao, Z. Chang-wei, Y. Ping-gang, and L. Yong-hong, "Study on Application of Bayesian Classifier Model in Data Stream," in *2010 International Conference on Computational and Information Sciences*. Chengdu, China: IEEE, 12 2010, pp. 1312–1315. ↑57, ↑103

[149] C. C. Aggarwal, *Data Classification: Algorithms and Applications*, 1st ed. Chapman & Hall / CRC, 2014. ↑57, ↑103

[150] D. K. Tasoulis, G. Ross, and N. M. Adams, "Visualising the Cluster Structure of Data Streams," in *Advances in Intelligent Data Analysis VII*. Springer Berlin-Heidelberg, 2007, pp. 81–92. ↑57, ↑126, ↑127, ↑129

[151] Y. Chen and L. Tu, "Density-based clustering for real-time stream data," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '07*. ACM New York, 2007, pp. 133–142. ↑57, ↑126, ↑127

[152] L. Wan, W. K. Ng, X. H. Dang, P. S. Yu, and K. Zhang, "Density-based clustering of data streams at multiple resolutions," *ACM Transactions on Knowledge Discovery from Data*, vol. 3, no. 3, pp. 1–28, 2009. ↑57, ↑126, ↑127

[153] C. C. Aggarwal and C. K. Reddy, *Data Clustering: Algorithms and Applications*, 1st ed. Chapman and Hall/CRC, 2013. ↑57

[154] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 346–357. ↑57, ↑61, ↑64

[155] M. Deypir, M. H. Sadreddini, and S. Hashemi, "Towards a variable size sliding window model for frequent itemset mining over data streams," *Computers & Industrial Engineering*, vol. 63, no. 1, pp. 161–172, 8 2012. ↑57, ↑61, ↑62

[156] J. Cheng, Y. Ke, and W. Ng, "A survey on algorithms for mining frequent itemsets over data streams," *Knowledge and Information Systems*, vol. 16, no. 1, pp. 1–27, 7 2008. ↑57

[157] S. Nasreen, M. A. Azam, K. Shehzad, U. Naeem, and M. A. Ghazanfar, "Frequent Pattern Mining Algorithms for Finding Associated Frequent Patterns for Data Streams: A Survey," *Procedia Computer Science*, vol. 37, pp. 109–116, 1 2014. ↑57

[158] A. Forestiero, "Self-organizing anomaly detection in data streams," *Information Sciences*, vol. 373, pp. 321–336, 12 2016. ↑57

[159] L. Tran, L. Fan, and C. Shahabi, "Distance-based outlier detection in data streams," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1089–1100, 8 2016. ↑57

[160] S. C. Tan, K. M. Ting, and T. F. Liu, "Fast anomaly detection for streaming data," in *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*. AAAI Press, 2011, pp. 1511–1516. ↑57

[161] Y. Xu, K. Wang, A. W.-C. Fu, R. She, and J. Pei, "Classification spanning correlated data streams," in *Proceedings of the 15th ACM international conference on Information and knowledge management - CIKM '06*. ACM New York, USA, 2006, pp. 132–141. ↑57

[162] Z. Wu, Y.-G. Jiang, X. Wang, H. Ye, and X. Xue, "Multi-Stream Multi-Class Fusion of Deep Networks for Video Classification," in *Proceedings of the 2016 ACM Conference on Multimedia Conference - MM '16*. ACM New York, USA, 2016, pp. 791–800. ↑57

[163] S. K. Greene, J. Huang, A. M. Abrams, D. Gilliss, M. Reed, R. Platt, S. S. Huang, and M. Kulldorff, "Gastrointestinal Disease Outbreak Detection Using Multiple Data Streams from Electronic Medical Records," *Foodborne Pathogens and Disease*, vol. 9, no. 5, pp. 431–441, 2012. ↑57

[164] Z. Qi, L. Jinze, and W. Wei, "Approximate clustering on distributed data streams," in *Proceedings - International Conference on Data Engineering*. IEEE, 2008, pp. 1131–1139. ↑57, ↑127

[165] W. Wu and L. Gruenwald, "Research issues in mining multiple data streams," in *Proceedings of the First International Workshop on Novel Data Stream Pattern Mining Techniques - StreamKDD '10*. ACM New York, 2010, pp. 56–60. ↑57, ↑127

[166] J. Gama, P. P. Rodrigues, and L. Lopes, "Clustering distributed sensor data streams using local processing and reduced communication," *Intelligent Data Analysis*, vol. 15, pp. 3–28, 2011. ↑57, ↑127

[167] P. P. Rodrigues and J. Gama, "Distributed clustering of ubiquitous data streams," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 5, no. 1, pp. 38–54, 2014. ↑57, ↑127

[168] A. Zhou, F. Cao, Y. Yan, C. Sha, and X. He, "Distributed data stream clustering: A fast EM-based approach," in *Proceedings - International Conference on Data Engineering*. IEEE, 2007, pp. 736–745. ↑57, ↑127

[169] J. Guo, P. Zhang, J. Tan, and L. Guo, "Mining frequent patterns across multiple data streams," in *Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM '11*. New York, New York, USA: ACM Press, 2011, pp. 2325–2328. ↑57, ↑62

[170] B. P. Jaysawal and J.-W. Huang, "Mining Frequent Progressive Usage Patterns Across Multiple Mobile Broadcasting Channels," in *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. Springer, Cham, 2014, pp. 149–155. ↑57, ↑62

[171] J. Guo, "A Study on Distributed Frequent Co-occurrence Patterns Algorithms across Multiple Data Streams," *Journal of Software*, vol. 11, no. 12, pp. 1191–1198, 12 2016. ↑57, ↑63

[172] D. Amagata and T. Hara, "Mining Top-k Co-Occurrence Patterns across Multiple Streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 10, pp. 2249–2262, 10 2017. ↑57, ↑63

[173] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, J. Ingram, and S. E. Seker, "Online anomaly detection for multi-source VMware using a distributed streaming framework," *Software: Practice and Experience*, vol. 46, no. 11, pp. 1479–1497, 11 2016. ↑57

[174] C. Zhang, H. Yan, S. Lee, and J. Shi, "Multiple profiles sensor-based monitoring and anomaly detection," *Journal of Quality Technology*, vol. 50, no. 4, pp. 344–362, 10 2018. ↑57

[175] P. Kranen, S. Günnemann, S. Fries, and T. Seidl, "MC-Tree: Improving Bayesian Anytime Classification," in *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*. Springer, Berlin, Heidelberg, 2010, pp. 252–269. ↑58, ↑110

[176] S. Esmeir and S. Markovitch, "Interruptible anytime algorithms for iterative improvement of decision trees," in *Proceedings of the 1st international workshop on Utility-based data mining - UBDM '05*. ACM New York, USA, 2005, pp. 78–85. ↑58

[177] S. Schlobach, E. Blaauw, M. El Kebir, A. Ten Teije, F. Van Harmelen, S. Bortoli, M. Hobbelman, K. Millian, Y. Ren, S. Stam, P. Thomassen, R. Van Het Schip, and W. Van Willigem, "Anytime classification by ontology approximation," in *Proceedings of the First International Conference on New Forms of Reasoning for the Semantic Web: Scalable, Tolerant and Dynamic - Volume 291*. CEUR-WS.org Aachen, Germany, 2007, pp. 57–71. ↑58

[178] B. Hui, Y. Yang, and G. I. Webb, "Anytime classification for a pool of instances," *Machine Learning*, vol. 77, no. 1, pp. 61–102, 10 2009. ↑58

[179] C. I. Lemes, D. F. Silva, and G. E. Batista, "Adding Diversity to Rank Examples in Anytime Nearest Neighbor Classification," in *Proceedings of 13th International Conference on Machine Learning and Applications*. IEEE, 12 2014, pp. 129–134. ↑58

[180] S. T. Mai, I. Assent, and M. Storgaard, "AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. ACM New York, USA, 2016, pp. 1025–1034. ↑58

[181] S. T. Mai, I. Assent, and A. Le, "Anytime OPTICS: An Efficient Approach for Hierarchical Density-Based Clustering," in *Proceedings of International Conference on Database Systems for Advanced Applications*. Springer, Cham, 2016, pp. 164–179. ↑58

[182] Shichao Zhang and Chengqi Zhang, "Anytime mining for multiuser applications," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 32, no. 4, pp. 515–521, 7 2002. ↑58, ↑63

[183] H.-F. Li and S.-Y. Lee, "Mining frequent itemsets over data streams using efficient window sliding techniques," *Expert Systems with Applications*, vol. 36, no. 2, pp. 1466–1477, 3 2009. ↑60, ↑61, ↑62, ↑65

[184] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery*, vol. 8, no. 1, pp. 53–87, 1 2004. ↑60, ↑61, ↑68, ↑79, ↑101

[185] L. Troiano and G. Scibelli, "Mining frequent itemsets in data streams within a time horizon," *Data & Knowledge Engineering*, vol. 89, pp. 21–37, 1 2014. ↑61, ↑62

[186] C. Raïssi, P. Poncelet, and M. Teisseire, "Towards a new approach for mining frequent itemsets on data stream," *Journal of Intelligent Information Systems*, vol. 28, no. 1, pp. 23–36, 2 2007. ↑61

[187] "Zipf Distribution." [Online]. Available: https://en.wikipedia.org/wiki/Zipf'slaw ↑84

[188] N. T. Gridgeman, "Lam'e ovals," *The Mathematical Gazette*, vol. 54, no. 387, pp. 31–37, 1970. ↑84

[189] P. Stein, "A Note on the Volume of a Simplex," *The American Mathematical Monthly*, vol. 73, no. 3, pp. 299–301, 1966. ↑86

[190] "Market-Basket Synthetic Data Generator - CodePlex Archive." [Online]. Available: https://archive.codeplex.com/?p=synthdatagen ↑90

[191] "Frequent Itemset Mining Dataset Repository." [Online]. Available: http://fimi.ua.ac.be/data/ ↑90

[192] "MSNBC.com Anonymous Web Data Data Set." [Online]. Available: http://archive.ics.uci.edu/ml/datasets/msnbc.com+anonymous+web+data ↑90

[193] R. O. Duda, P. E. P. E. Hart, and D. G. Stork, *Pattern classification*. Wiley, 2001. ↑91

[194] "Curve Fitting." [Online]. Available: https://en.wikipedia.org/wiki/Curvefitting ↑98

[195] C. C. Aggarwal, "The setwise stream classification problem," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14*. New York, New York, USA: ACM Press, 2014, pp. 432–441. ↑103, ↑104, ↑106, ↑117, ↑118, ↑124

[196] W. Silva, Á. Santana, F. Lobato, and M. Pinheiro, "A methodology for community detection in Twitter," in *Proceedings of the International Conference on Web Intelligence - WI '17*. Leipzig, Germany: ACM Press, New York, NY, USA, 2017, pp. 1006–1009. ↑104

[197] M. Liberatore and B. N. Levine, "Inferring the source of encrypted HTTP connections," in *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*. Alexandria, Virginia, USA: ACM Press, New York, NY, USA, 2006, pp. 255–263. ↑105, ↑118

[198] S. Jhaver, "Large Scale Data Mining with Applications in Social Computing," Ph.D. dissertation, University of Texas, Dallas, 2014. ↑105, ↑106

[199] M. Reed, P. Syverson, and D. Goldschlag, "Anonymous connections and onion routing," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 4, pp. 482–494, 5 1998. ↑105

[200] J. A. Blackard and D. J. Dean, "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables," *Computers and Electronics in Agriculture*, vol. 24, no. 3, pp. 131–151, 12 1999. ↑117, ↑142

[201] "Stanford Postagger," 2018. [Online]. Available: ttps://nlp.stanford.edu/software/tagger.shtml ↑117

[202] X. H. Dang, V. Lee, W. K. Ng, A. Ciptadi, and K. L. Ong, "An EM-Based Algorithm for Clustering Data Streams in Sliding Windows," in *Proceedings of International Conference on Database Systems for Advanced Applications (DASFAA 2009)*. Springer, Berlin, Heidelberg, 2009, pp. 230–235. ↑126

[203] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Databases Method for Very Large," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. ACM New York, 1996, pp. 103–114. ↑126, ↑153

[204] A. Balzanella and R. Verde, "Summarizing and detecting structural drifts from multiple data streams," in *Studies in Classification, Data Analysis, and Knowledge Organization*. Springer-Verlag, 2013, pp. 105–112. ↑127

[205] J. Beringer and E. Hüllermeier, "Online clustering of parallel data streams," *Data & Knowledge Engineering*, vol. 58, no. 2, pp. 180–204, 8 2006. ↑127

[206] Z. R. Hesabi, T. Sellis, and X. Zhang, "Anytime concurrent clustering of multiple streams with an indexing tree," in *Proceedings of the 4th International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications - Volume 41*. JMLR.org, 2015, pp. 19–32. ↑128

[207] "KDD Cup 1999 Data," 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html ↑142

[208] H. Yigit, "ABC-based distance-weighted kNN algorithm," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 27, no. 2, pp. 189–198, 3 2015. ↑146

[209] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for K-medoids clustering," *Expert Systems with Applications*, vol. 36, no. 2, pp. 3336–3341, 3 2009. ↑153

[210] R. Jarvis and E. Patrick, "Clustering Using a Similarity Measure Based on Shared Near Neighbors," *IEEE Transactions on Computers*, vol. C-22, no. 11, pp. 1025–1034, 11 1973. ↑153, ↑163

[211] R. Agrawal, J. Gehrke, D. Gunopulos, P. Raghavan, R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 94–105, 6 1998. ↑153, ↑167

[212] C.-H. Cheng, A. W. Fu, and Y. Zhang, "Entropy-based subspace clustering for mining numerical data," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '99*. New York, New York, USA: ACM Press, 1999, pp. 84–93. ↑153, ↑167

[213] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, J. S. Park, C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park, "Fast algorithms for projected clustering," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data - SIGMOD '99*. New York, New York, USA: ACM Press, 1999, pp. 61–72. ↑153, ↑166

[214] C. C. Aggarwal, P. S. Yu, C. C. Aggarwal, and P. S. Yu, "Finding generalized projected clusters in high dimensional spaces," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00*. New York, New York, USA: ACM Press, 2000, pp. 70–81. ↑153, ↑166

[215] G. Sheikholeslami, S. Chatterjee, and A. Zhang, "WaveCluster: a wavelet-based clustering approach for spatial data in very large databases," *The VLDB Journal The International Journal on Very Large Data Bases*, vol. 8, no. 3-4, pp. 289–304, 2 2000. ↑153

[216] A. Mukhopadhyay and U. Maulik, "Unsupervised Satellite Image Segmentation by Combining SA Based Fuzzy Clustering with Support Vector Machine," in *2009 Seventh International Conference on Advances in Pattern Recognition*. IEEE, 2 2009, pp. 381–384. ↑153

[217] S. Madeira and A. Oliveira, "Biclustering algorithms for biological data analysis: a survey," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 1, no. 1, pp. 24–45, 1 2004. ↑153

[218] K. A., S. R. Knollmann, S. I. Muldrew, F. R. Pearce, M. A. Aragon-Calvo, Y. Ascasibar, P. S. Behroozi, D. Ceverino, S. Colombi, J. Diemand, K. Dolag, B. L. Falck, P. Fasel, J. Gardner, S. Gottlöber, C.-H. Hsu, F. Iannuzzi, A. Klypin, Z. LukiÂt'c, M. Maciejewski, C. McBride, M. C. Neyrinck, S. Planelles, D. Potter, V. Quilis, Y. Rasera, J. I. Read, P. M. Ricker, F. Roy, Springel, V. Stadel, J. Stinson, G. P. M. Sutter, V. Turchaninov, D. Tweed, G. Yepes, M. Zemp, and M., "Haloes gone MAD: The Halo-Finder Comparitions Project," *Monthly Notics of the Royal Astronomical Society*, vol. 415, pp. 2293–2318, 2011. ↑153

[219] S. Huo, "Detecting Self-Correlation of Nonlinear, Lognormal, Time-Series Data via DBSCAN Clustering Method, Using Stock Price Data as Example," Ph.D. dissertation, Ohio State University, 2011. ↑153

[220] Y. Jiang and J. Zhang, "Parallel K-Medoids clustering algorithm based on Hadoop," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*. IEEE, 6 2014, pp. 649–652. ↑153

[221] W. Zhao, H. Ma, and Q. He, "Parallel K-Means Clustering Based on MapReduce," in *Proceedings of the 1st International Conference on Cloud Computing*. Springer-Verlag, 2009, pp. 674–679. ↑153, ↑159

[222] R. Jin, A. Goswami, and G. Agrawal, "Fast and exact out-of-core and distributed k-means clustering," *Knowledge and Information Systems*, vol. 10, no. 1, pp. 17–40, 7 2006. ↑153

[223] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "BD-CATS: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. New York, New York, USA: ACM Press, 2015, pp. 1–12. ↑153, ↑157, ↑161, ↑168, ↑174

[224] M. Götz, C. Bodenstein, and M. Riedel, "HPDBSCAN: Highly Parallel DBSCAN," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments - MLHPC '15*. ACM New York, USA, 2015, pp. 1–10. ↑153, ↑156, ↑161, ↑168

[225] H. Nagesh, S. Goil, and A. Choudhary, "A scalable parallel subspace clustering algorithm for massive data sets," in *International Conference on Parallel Processing*. IEEE, 2000, pp. 477–484. ↑153, ↑167, ↑168

[226] H. Nazerzadeh, M. Ghodsi, and S. Sadjadian, "Parallel Subspace Clustering," in *Proceedings of the 10th Annual Conference of Computer Society of Iran*. IEEE, 2005, pp. 1–8. ↑153, ↑167

[227] A. Adinetz, J. Kraus, J. Meinke, and D. Pleiter, "GPUMAFIA: Efficient Subspace Clustering with MAFIA on GPUs," in *Proceedings of European Conference on Parallel Processing*. Springer, Berlin, Heidelberg, 2013, pp. 838–849. ↑153, ↑167

[228] B. Zhu, A. Mara, and A. Mozo, "CLUS: Parallel Subspace Clustering Algorithm on Spark," in *Proceedings of East European Conference on Advances in Databases and Information Systems*. Springer, Cham, 2015, pp. 175–185. ↑153, ↑167

[229] R. L. Ferreira Cordeiro, C. Traina, A. J. Machado Traina, J. López, U. Kang, and C. Faloutsos, "Clustering very large multi-dimensional datasets with MapReduce," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '11*. New York, New York, USA: ACM Press, 2011, pp. 690–698. ↑153, ↑168

[230] S. Kumari, S. Maurya, P. Goyal, S. S. Balasubramaniam, and N. Goyal, "Scalable Parallel Algorithms for Shared Nearest Neighbor Clustering," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 12 2016, pp. 72–81. ↑153, ↑155, ↑156, ↑163, ↑168, ↑169, ↑180

[231] V. Olman, Fenglou Mao, Hongwei Wu, and Ying Xu, "Parallel Clustering Algorithm for Large Data Sets with Applications in Bioinformatics," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 6, no. 2, pp. 344–352, 4 2009. ↑153, ↑154, ↑156, ↑165, ↑168

[232] Q. Qian, S. Zhao, C.-J. Xiao, and C.-L. Hung, "Multi-level Grid Based Clustering and GPU Parallel Implementations," in *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*. IEEE, 6 2017, pp. 397–402. ↑153

[233] C. Xiaoyun, C. Yi, Q. Xiaoli, Y. Min, and H. Yanshan, "PGMCLU: A novel parallel grid-based clustering algorithm for multi-density datasets," in *2009 1st IEEE Symposium on Web Society*. IEEE, 8 2009, pp. 166–171. ↑153

[234] C. Deng, J. Song, R. Sun, S. Cai, and Y. Shi, "GRIDEN: An effective grid-based and density-based spatial clustering algorithm to support parallel computing," *Pattern Recognition Letters*, vol. 109, pp. 81–88, 7 2018. ↑153

[235] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "Scalable parallel OPTICS data clustering using graph algorithmic techniques," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM New York, USA, 2013, pp. 1–12. ↑155, ↑156, ↑162, ↑168, ↑169, ↑186

[236] P. Goyal, S. Kumari, D. Kumar, S. Balasubramaniam, and N. Goyal, "Parallelizing OPTICS for multicore systems," in *Proceedings of the 7th ACM India Computing Conference on - COMPUTE '14*. ACM New York, USA, 2014, pp. 1–6. ↑155, ↑156, ↑162

[237] B. Welton, E. Samanas, and B. P. Miller, "Mr. Scan: Extreme scale density-based clustering using a tree-based network of GPGPU nodes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM Press, 2013, pp. 1–11. ↑156, ↑162

[238] X. Xu, J. Jäger, and H.-P. Kriegel, "A Fast Parallel Clustering Algorithm for Large Spatial Databases," *Data Mining and Knowledge Discovery*, vol. 3, no. 3, pp. 263–290, 1999. ↑156, ↑160, ↑168

[239] A. Zhou, S. Zhou, J. Cao, Y. Fan, and Y. Hu, "Approaches for scaling DBSCAN algorithm to large spatial databases," *Journal of Computer Science and Technology*, vol. 15, no. 6, pp. 509–526, 11 2000. ↑156, ↑160, ↑168

[240] X. Li, "Parallel algorithms for hierarchical clustering and cluster validity," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 11, pp. 1088–1092, 1990. ↑156, ↑164

[241] C.-H. Wu, S.-J. Horng, and H.-R. Tsai, "Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses," *Journal of Parallel and Distributed Computing*, vol. 60, no. 9, pp. 1137–1153, 9 2000. ↑156, ↑164

[242] Z. Du and F. Lin, "A novel parallelization approach for hierarchical clustering," *Parallel Computing*, vol. 31, no. 5, pp. 523–527, 5 2005. ↑156, ↑164

[243] P. S. Bradley, O. L. Mangasarian, and W. N. Street, "Clustering via concave minimization," in *Proceedings of the 9th International Conference on Neural Information Processing Systems*. MIT Press, 1996, pp. 368–374. ↑159

[244] J. C. Dunn, "A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters," *Journal of Cybernetics*, vol. 3, no. 3, pp. 32–57, 1 1973. ↑159

[245] J. Kumar, R. T. Mills, F. M. Hoffman, and W. W. Hargrove, "Parallel k-Means Clustering for Quantitative Ecoregion Delineation Using Large Data Sets," *Procedia Computer Science*, vol. 4, pp. 1602–1611, 1 2011. ↑159, ↑168

[246] M.-F. F. Balcan, S. Ehrlich, and Y. Liang, "Distributed k-means and k-median Clustering on General Topologies," in *Advances in Neural Information Processing Systems 26 (NIPS 2013)*. Neural Information Processing Systems Foundation, Inc., 2013, pp. 1–9. ↑159, ↑168

[247] M. N. Joshi, "Parallel K-Means Algorithm on Distributed Memory Multiprocessors," University of Minnesota, Twin Cities, Tech. Rep., 2003. ↑159

[248] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," in *Proceedings of the VLDB Endowment*. VLDB Endowment, 3 2012, pp. 622–633. ↑159

[249] K. D. Garcia and M. C. Naldi, "Multiple Parallel MapReduce k-Means Clustering with Validation and Selection," in *2014 Brazilian Conference on Intelligent Systems*. IEEE, 10 2014, pp. 432–437. ↑159

[250] B. Wang, J. Yin, Q. Hua, Z. Wu, and J. Cao, "Parallelizing K-Means-Based Clustering on Spark," in *2016 International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 8 2016, pp. 31–36. ↑159

[251] V. Santhi and R. Jose, "Performance Analysis of Parallel K-Means with Optimization Algorithms for Clustering on Spark," in *Proceedings of International Conference on Distributed Computing and Internet Technology (ICDCIT'18)*. Springer, Cham, 2018, pp. 158–162. ↑159

[252] D. Arlia and M. Coppola, "Experiments in Parallel Clustering with DBSCAN," in *Proceedings of the 7th International Euro-Par Conference on Parallel Processing*. Springer, 2001, pp. 326–331. ↑160, ↑168

[253] M. Coppola and M. Vanneschi, "High-performance data mining with skeleton-based structured parallel programming," *Parallel Computing*, vol. 28, no. 5, pp. 793–813, 5 2002. ↑160, ↑168

[254] E. Januzaj, H.-P. Kriegel, and M. Pfeifle, "DBDC: Density Based Distributed Clustering," in *Proceedings of 2004 International Conference on Extending Database Technology (EDBT '04)*. Springer, Berlin, Heidelberg, 2004, pp. 88–105. ↑160

[255] S. Brecheisen, H.-P. Kriegel, and M. Pfeifle, "Parallel Density-Based Clustering of Complex Objects," in *Proceedings of 2006 Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2006)*. Springer, Berlin, Heidelberg, 2006, pp. 179–188. ↑160

[256] M. Chen, X. Gao, and H. Li, "Parallel DBSCAN with Priority R-tree," in *2010 2nd IEEE International Conference on Information Management and Engineering*. IEEE, 2010, pp. 508–511. ↑160, ↑168

[257] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data - SIGMOD '90*. New York, New York, USA: ACM Press, 1990, pp. 322–331. ↑160

[258] B. Welton and B. P. Miller, "Mr. Scan: A Hybrid / Hybrid Extreme Scale Density Based Clustering Algorithm," Northwestern University, Tech. Rep., 2015. ↑162

[259] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering," *Procedia Computer Science*, vol. 18, pp. 369–378, 1 2013. ↑162

[260] C.-C. Chen and M.-S. Chen, "HiClus: Highly Scalable Density-based Clustering with Heterogeneous Cloud," *Procedia Computer Science*, vol. 53, pp. 149–157, 1 2015. ↑162

[261] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan, "MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 12 2011, pp. 473–480. ↑162

[262] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 83–99, 2 2014. ↑162

[263] B.-R. Dai and I.-C. Lin, "Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 6 2012, pp. 59–66. ↑162

[264] Y. Yu, J. Zhao, X. Wang, Q. Wang, and Y. Zhang, "Cludoop: An Efficient Distributed Density-Based Clustering for Big Data Using Hadoop," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, pp. 1–13, 6 2015. ↑162

[265] X. Hu, L. Liu, N. Qiu, D. Yang, and M. Li, "A MapReduce-based improvement algorithm for DBSCAN," *Journal of Algorithms & Computational Technology*, vol. 12, no. 1, pp. 53–61, 3 2018. ↑162

[266] Y. Gu, X. Ye, F. Zhang, Z. Du, R. Liu, and L. Yu, "A parallel varied density-based clustering algorithm with optimized data partition," *Journal of Spatial Science*, vol. 63, no. 1, pp. 93–114, 1 2018. ↑162

[267] H. Song and J.-G. Lee, "RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning," in *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*. ACM New York, USA, 2018, pp. 1173–1187. ↑162

[268] I. Cordova and T.-S. Moh, "DBSCAN on Resilient Distributed Datasets," in *2015 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 7 2015, pp. 531–540. ↑162

[269] D. Han, A. Agrawal, W.-K. Liao, and A. Choudhary, "A Novel Scalable DBSCAN Algorithm with Spark," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 5 2016, pp. 1393–1402. ↑162

[270] G. Luo, X. Luo, T. F. Gooch, L. Tian, and K. Qin, "A Parallel DBSCAN Algorithm Based on Spark," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. IEEE, 10 2016, pp. 548–553. ↑162

[271] A. Lulli, M. Dell'Amico, P. Michiardi, and L. Ricci, "NG-DBSCAN," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 157–168, 11 2016. ↑162

[272] F. Huang, Q. Zhu, J. Zhou, J. Tao, X. Zhou, D. Jin, X. Tan, and L. Wang, "Research on the Parallelization of the DBSCAN Clustering Algorithm for Spatial Data Mining Based on the Spark Platform," *Remote Sensing*, vol. 9, no. 12, pp. 1–33, 12 2017. ↑162

[273] C. Zewen and Z. Yao, "Parallel Text Clustering Based on MapReduce," in *2012 Second International Conference on Cloud and Green Computing*. IEEE, 11 2012, pp. 226–229. ↑163

[274] S. Wang and C. F. Eick, "MR-SNN: Design of parallel Shared Nearest Neighbor clustering algorithm using MapReduce," in *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)(*. IEEE, 3 2017, pp. 312–315. ↑163

[275] N. Meghwal and S. M, "Parallel Implementation of Shared Nearest Neighbor Clustering Algorithm," SERC, Indian Institute of Science, Bangalore, India, Tech. Rep., 2015. ↑163

[276] S. Rajasekaran and Sanguthevar, "Efficient parallel hierarchical clustering algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 497–502, 6 2005. ↑165, ↑168

[277] M. Dash, H. Liu, P. Scheuermann, and K. L. Tan, "Fast hierarchical clustering and its validation," *Data & Knowledge Engineering*, vol. 44, no. 1, pp. 109–138, 1 2003. ↑165

[278] M. Dash, S. Petrutiu, and P. Scheuermann, "pPOP: Fast yet accurate parallel hierarchical clustering using partitioning," *Data & Knowledge Engineering*, vol. 61, no. 3, pp. 563–578, 6 2007. ↑165

[279] S. Kim and D. C. Wunsch, "A GPU based Parallel Hierarchical Fuzzy ART clustering," in *The 2011 International Joint Conference on Neural Networks*. IEEE, 7 2011, pp. 2778–2782. ↑166

[280] S. A. Rylov and I. A. Pestunov, "Fast hierarchical clustering of multispectral images and its implementation on NVIDIA GPU," *Journal of Physics: Conference Series*, vol. 1096, no. 1, p. 012039, 9 2018. ↑166

[281] C. Jin, M. A. Patwary, A. Agrawal, W. Hendrix, W.-k. Liao, and A. N. Choudhary, "DiSC : A Distributed Single-Linkage Hierarchical Clustering Algorithm using MapReduce," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM, 2013, pp. 1–11. ↑166

[282] C. Jin, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, "Incremental, distributed single-linkage hierarchical clustering algorithm using mapreduce," in *Proceedings of the Symposium on High Performance Computing*. Society for Computer Simulation International, 2015, pp. 83–92. ↑166

[283] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, "A Scalable Hierarchical Clustering Algorithm Using Spark," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*. IEEE, 3 2015, pp. 418–426. ↑166

[284] G. Mazzeo and C. Zanilo, "The parallelization of acomplex hierarchical clustering algorithm: faster unsu-pervised learning on larger data sets," University of California, Los Angeles, Tech. Rep., 2016. ↑166

[285] Jiong Yang, Wei Wang, Haixun Wang, and P. Yu, "δ-clusters: capturing subspace correlation in a large data set," in *Proceedings 18th International Conference on Data Engineering*. IEEE Comput. Soc, 2002, pp. 517–528. ↑166

[286] J. H. Friedman and J. J. Meulman, "Clustering objects on subsets of attributes," *Journal of Royal Statistical Society*, vol. 66, no. 4, pp. 815–849, 2004. ↑166

[287] C. Domeniconi, D. Papadopoulos, D. Gunopulos, and S. Ma, "Subspace Clustering of High Dimensional Data," in *2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 517–521. ↑166

[288] K. Sequeira and M. Zaki, "SCHISM: A New Approach for Interesting Subspace Mining," in *Fourth IEEE International Conference on Data Mining (ICDM'04)*. IEEE, 2004, pp. 186–193. ↑167

[289] J.-W. Chang and D.-S. Jin, "A new cell-based clustering method for large, high-dimensional data in data mining applications," in *Proceedings of the 2002 ACM symposium on Applied computing - SAC '02*. ACM Press, 2002, pp. 503–507. ↑167

[290] K. Kailing, H.-P. Kriegel, and P. Kroger, "Density-Connected Subspace Clustering for High-Dimensional Data *," in *4th SIAM International Conference on Data Mining*. SIAM, 2004, pp. 246–257. ↑167

[291] H. Kriegel, P. Kroger, M. Renz, and S. Wurst, "A Generic Framework for Efficient Subspace Clustering of High-Dimensional Data," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 2005, pp. 250–257. ↑167

[292] I. Assent, R. Krieger, E. Müller, and T. Seidl, "DUSC: Dimensionality Unbiased Subspace Clustering," in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 10 2007, pp. 409–414. ↑167

[293] I. Assent, R. Krieger, E. Muller, and T. Seidl, "INSCY: Indexing Subspace Clusters with In-Process-Removal of Redundancy," in *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 12 2008, pp. 719–724. ↑167

[294] A. Kaur and A. Datta, "A novel algorithm for fast and scalable subspace clustering of high-dimensional data," *Journal of Big Data*, vol. 2, no. 1, p. 17, 12 2015. ↑167

[295] A. Datta, A. Kaur, T. Lauer, and S. Chabbouh, "Exploiting multiâĂŞcore and manyâĂŞcore parallelism for subspace clustering," *International Journal of Applied Mathematics and Computer Science*, vol. 29, no. 1, pp. 81–91, 2019. ↑167

[296] B. Zhu, B. Ordozgoiti, and A. Mozo, "PSCEG: An unbiased Parallel Subspace Clustering algorithm using Exact Grids," in *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. i6doc.com Publishers, 2016, pp. 581–586. ↑168

[297] Z. Gao, Y. Fan, K. Niu, and Z. Ying, "MR-Mafia: Parallel Subspace Clustering Algorithm Based on MapReduce for Large Multi-dimensional Datasets," in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 1 2018, pp. 257–262. ↑168

[298] E. N. Adriano Di Pasquale, "Scalable Distributed Data Structures: A Survey," in *3rd International Workshop on Distributed Data and Structures (WDAS 2000)*, 2000, pp. 87–111. ↑186

[299] B. Kröll, P. Widmayer, B. Kröll, and P. Widmayer, "Distributing a search tree among a growing number of processors," *ACM SIGMOD Record*, vol. 23, no. 2, pp. 265–276, 6 1994. ↑186

[300] T. P. Hayes, J. Saia, and A. Trehan, "The forgiving graph: a distributed data structure for low stretch under adversarial attack," in *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC '09*. New York, New York, USA: ACM Press, 2009, pp. 121–130. ↑186

[301] M. T. Goodrich, M. J. Nelson, and J. Z. Sun, "The rainbow skip graph: a fault-tolerant constant-degree distributed data structure," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA 2006)*. SIAM, 2006, pp. 384–393. ↑186

[302] I. Kamel, C. Faloutsos, I. Kamel, and C. Faloutsos, "Parallel R-trees," in *Proceedings of the 1992 ACM SIGMOD international conference on Management of data - SIGMOD '92*, vol. 21, no. 2. New York, New York, USA: ACM Press, 1992, pp. 195–204. ↑186

[303] E. G. Hoel and H. Samet, "Data-Parallel R-Tree Algorithms," in *1993 International Conference on Parallel Processing (ICPP'93)*. IEEE, 8 1993, pp. 47–50. ↑186

[304] T. Johnson and A. Colbrook, "A distributed data-balanced dictionary based on the B-link tree," in *Proceedings of Sixth International Parallel Processing Symposium*. IEEE Comput. Soc. Press, 1992, pp. 319–324. ↑186

[305] B. Schnitzer and S. Leutenegger, "Master-client R-trees: a new parallel R-tree architecture," in *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*. IEEE Comput. Soc, 1999, pp. 68–77. ↑186

[306] L. Shuhua, Z. Fenghua, and S. Yongqiang, "A Design of Parallel R-tree on Cluster of Workstations," in *International Workshop on Databases in Networked Information Systems (DNIS 2000)*. Springer, Berlin, Heidelberg, 2000, pp. 119–133. ↑186

[307] C. du Mouza, W. Litwin, and P. Rigaux, "Large-scale indexing of spatial data in distributed repositories: the SD-Rtree," *The VLDB Journal*, vol. 18, no. 4, pp. 933–958, 8 2009. ↑186

[308] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009. ↑186

[309] Y. Chen, G. Dong, J. Han, B. Wah, and J. Wang, "Multi-Dimensional Regression Analysis of Time-Series Data Streams," in *Proceedings of the 28th international conference on Very Large Data Bases*. ACM New York, 2002, pp. 323–334. ↑212

[310] M. J. Zaki and W. Meira Jr, *Data Mining and Analysis*. 1st Edition, 2014. ↑214, ↑217

Publications

Conference Papers

i. **Jagat Sesh Challa**, Poonam Goyal, Vijay M Giri, Dhananjay Mantri, Navneet Goyal, "AnySC: Anytime Set-wise Classification of Variable Speed Data Streams", In Proceedings of 2018 IEEE International Conference on Big Data (IEEE Big Data 2018), pp. 967-974, 10-13 December 2018, Seattle, WA, USA.

ii. Poonam Goyal, **Jagat Sesh Challa**, Shivin Srivastava, Navneet Goyal, "AnyFI: An Anytime Frequent Itemset Mining Algorithm for Data Streams", In Proceedings of 2017 IEEE International Conference on Big Data (IEEE Big Data 2017), 11-14 December 2017, Boston, MA, USA.

iii. **Jagat Sesh Challa**, Poonam Goyal, Nikhil S., Aditya Mangla, Sundar Balasubramaniam, Navneet Goyal, "DDR-Tree: A dynamic distributed data structure for efficient data distribution among cluster nodes for spatial data mining algorithms", In Proceedings of 2016 IEEE International Conference on Big Data (IEEE Big Data 2016), 5-8 December 2016, Washington DC, USA.

Journal Papers

i. **Jagat Sesh Challa**, Poonam Goyal, Ajinkya Kokandakar, Dhananjay Mantri, Pranet Verma, Sundar Balasubramaniam, Navneet Goyal, "A New Micro-Cluster based Approach for Anytime Clustering of Data Streams that handles Noise and Concept Drift", submitted for review in *Journal of Experimental & Theoretical Artificial Intelligence* (**TETA**), Taylor & Francis. (Revision submitted)

ii. Poonam Goyal, **Jagat Sesh Challa**, Shivin Srivastava, Navneet Goyal, "Anytime Frequent Itemset Mining of Transactional Data Streams", submitted for review in *Big Data Research*, Elsevier.

iii. Poonam Goyal, **Jagat Sesh Challa**, Dhruv Kumar, Navneet Goyal, Sundar Balasubramaniam, "Grid-R-tree: A data structure for efficient neighborhood and nearest neighbor queries in data mining", submitted for review in *Journal of Data Science & Analytics* (**JDSA**), Springer.

iv. **Jagat Sesh Challa**, Poonam Goyal, Nikhil S, Amogh Sharma, Shan Balasubramaniam, Navneet Goyal, "Experiments on Data Distribution Strategies for Parallel Spatial Clustering Algorithms", to be submitted to *ACM Transactions on Knowledge Discovery from Data* (**TKDD**), ACM.

v. **Jagat Sesh Challa**, Poonam Goyal, Navneet Goyal, "Anytime Data Mining: A Comprehensive Survey", to be submitted to *Wiley Inter-disciplinary Reviews on Data Mining and Knowledge Discovery* (**WIRE-DMKD**), Wiley.

Biographies

Brief Biography of the Candidate

Jagat Sesh Challa is a PhD Student in the Department of Computer Science & Information Systems at Birla Institute of Technology & Science (BITS), Pilani, Rajasthan, India. He has completed MSc.(Tech) Information Systems in 2010 and Masters of Engineering degree in Software Systems in 2012, both from the same department at BITS Pilani. His research interests are in the area of Data Mining, Streaming Data, Anytime Mining, Data Indexing Structures, Data Distribution strategies and High Performance Computing. He has a total of 9 publications which are accepted in reputed international conferences and journals. He is a reviewer for ACM Transactions on Knowledge Discovery from Data. He is a student member of IEEE. He is well experienced in teaching activities like conducting tutorials and laboratory sessions for various courses like Data Structures & Algorithms, Design & Analysis of Algorithms, Computer Programming, Logic in Computer Science, and Discrete Structures for Computer Science. Contact him at jagatsesh@pilani.bits-pilani.ac.in. http://universe.bits-pilani.ac.in/pilani/jagatsesh/profile

Brief Biography of the Supervisor

Prof. Poonam Goyal is currently an Associate Professor and Head in the Department of Computer Science & Information Systems at Birla Institute of Technology & Science, Pilani, Rajasthan, India. She has completed her Ph.D. in Numerical Analysis (Applied Mathematics) Department of Mathematics University of Roorkee (now IIT, Roorkee), Roorkee, India in 1995, ME in Software Systems, Department of Computer Science & Information Systems from BITS-Pilani, Pilani, India in 2002. She has received the IBM Scalable Data Analytics Innovation Faculty Award 2010 in the area of Scalable Data Analytics for the research concept titled "Developing a Smart Crop Management System using Data Analytics". She is actively involved with the Advanced Data Analytics and Parallel Technologies (ADAPT) Lab and she is the Convenor of the Web Intelligence and Social Computing (WiSOC) Lab. Her current research interests are in the area of Data Mining, High Performance Computing, Solution for Big Data Analytics, Information/Image Retrieval, Social Media Analytics, and Stream Mining. She has published more than 35 research papers in reputed Internal Conferences and Journals. She is a reviewer for various journals like ACM TKDD, IEEE Transactions SMC, T&F TETA, etc. Contact her at poonam@pilani.bits-pilani.ac.in. http://universe.bits-pilani.ac.in/pilani/poonam/profile

Brief Biography of the Co-Supervisor

Prof. Anil Maheshwari is currently a Professor and Graduate Director in the Department of Computer Science at Carleton University, Ottawa, Canada. He is also an Adjunct Professor at Birla Institute of Technology and Sciences, Pilani (India) since 2007. He has completed his Ph.D. in Computer Science from Tata Institute of Fundamental Research, Mumbai, India in 1993. He later worked as a Postdoctoral fellow at the Max Planck Institute for Informatics, Germany (1993-1994) and at the Carleton University, Ottawa, Canada

(1994-1995). He is currently working in the Computational Goemetry lab at Carleton University, primarily working in the areas of Design and Analysis of Algorithms for problems in computational geometry, graphs and discrete mathematics. He has received a number of research grants from various funding sources like MITACS, DFAIT, NSERC, etc. He has published over 200 papers in various peer reviewed journals and conferences of international repute. He has also served as the PC member for various National and International Conferences. Contact him at anil@scs.carleton.ca. http://people.scs.carleton.ca/ maheshwa/