الجمهورية الجزائرية الديمقراطية الشعبية

Republique Algérienne Démocratique Populaire

وزارة التعليم العالي والبحث العلمي

Ministère de l'Enseignement Supérieur de la Recherche Scientifique

Université 20 août 1955 Skikd

جامعة 20 أوت 1955 – سكيكدة

Faculte des Sciences
Departement d'informatique

كلية العلوم
قسم الإعلام الآلي

الرقم: .... / ق/ إ/ أ/ ك.ع2026/

Autorisation de Dépôt de Mémoire de Master

Je soussigné : .....K.hettabi..Karima.....

Certifie que l'étudiant(e) : .....Khettabi..Zahra.....

Spécialité : ....Intelligence.artificielle.....

Ayant soutenu le projet intitulé : ..Indexing..Continous..Data.in.IoT.Environment

A apporté les corrections nécessaires sur son manuscrit de Master.

جامعة 20 أوت 1955 - سكيكدة
قسم الإعلام الآلي
كلية العلوم

Signature de l'encadreur

PEOPLES DEMOCRATIC REPUBLIC OF ALGERIA
الجمهورية الجزائرية الديمقراطية الشعبية
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
وزارة التعليم العالي والبحث العلمي

University of 20 Août 1955 - Skikda
Faculty of Sciences
Department of Computer Science



Master Thesis

Submitted in partial fulfillment of the requirements for the degree of
Master of Computer Science

Specialization: Artificial Intelligence

Indexing Continuous Data in IoT Environment

Presented by:
Zahra KHETTABI

Supervised by:
Mrs. Karima KHETTABI

Thesis committee:

M. - Chairman
M. - Examiner

June, 2026

DEDICATION

To Allah, for the strength and the light upon my path

To my beloved parents, for their endless love and support

To myself, for the resilience and for never giving up

ACKNOWLEDGEMENTS

I would like to express my deepest and most sincere gratitude to my supervisor, **Mrs. Karima KHETTABI**, for her extraordinary dedication and the immense effort she invested in this work. This thesis is, in many ways, a reflection of her unwavering commitment and the countless hours she spent guiding me through every challenge. I am profoundly grateful for her ability to provide both the intellectual freedom to explore and the firm, thoughtful direction needed to succeed has been a cornerstone of my academic growth.

I am also truly honored to have this work examined by such distinguished jury members. I would like to thank each of them for their time, consideration, and the expertise they dedicate to reviewing this thesis.

My sincere thanks go as well to the faculty of the Computer Science Department, whose mentorship has shaped my academic journey and built the foundation upon which this work stands.

ABSTRACT

The Internet of Things (IoT) has emerged as one of the most transformative paradigms in modern computing, generating continuous, high-volume, and multi-dimensional data streams from interconnected devices. With the rapid growth of such data, efficient indexing mechanisms have become essential, particularly in Fog Computing environments, where computational and memory resources are significantly more constrained than in centralized cloud infrastructures.

The k-Nearest Neighbor (kNN) search is a fundamental operation widely used in IoT applications, including anomaly detection, intelligent monitoring, and real-time analytics. However, executing kNN queries over continuously growing data streams presents significant challenges, such as increased search latency, structural degradation of indexing schemes over time, and higher memory and energy consumption in resource-limited environments.

To address these challenges, this study proposes a hierarchical indexing architecture tailored for Fog-based IoT systems. The proposed approach consists of two complementary phases. In the first phase, data reduction is achieved using the BIRCH algorithm through the construction of a Clustering Feature (CF) tree. This step generates compact micro-clusters that summarize homogeneous regions of the data space, effectively reducing the overall computational complexity while preserving the intrinsic spatial structure of the dataset.

In the second phase, an independent binary indexing tree is constructed for each micro-cluster. This separation between clustering and indexing ensures improved data homogeneity within each index structure, resulting in shallower trees and reduced traversal cost during kNN search operations.

The proposed methodology exploits the cluster-based partitioning to restrict the search space from the global dataset to localized cluster-specific subspaces. Additionally, kNN queries can be executed in parallel across clusters, further enhancing search efficiency. Experimental results demonstrate that this hierarchical approach significantly reduces search time and computational overhead while maintaining scalability and adaptability to continuous data streams.

These findings confirm that integrating cluster-based data reduction techniques with distributed per-cluster indexing constitutes an effective and scalable solution for real-time data indexing in fog-based IoT environments.

Keywords: Continuous IoT data stream, clustering, indexing, BIRCH Algorithm, parallel kNN query, fog-cloud computing.

RÉSUMÉ

LInternet des Objets (IoT) sest imposé comme lun des paradigmes les plus influents de linformatique moderne, générant des flux de données continus, volumineux et multidimensionnels à partir dappareils interconnectés. Face à cette croissance rapide des données, le développement de mécanismes dindexation efficaces devient indispensable, en particulier dans les environnements de Fog Computing, caractérisés par des ressources computationnelles et mémoires plus limitées que celles des infrastructures Cloud centralisées.

La recherche des k plus proches voisins (k-Nearest Neighbor kNN) constitue une opération fondamentale dans de nombreuses applications IoT, telles que la détection danomalies, la surveillance intelligente et lanalyse en temps réel. Cependant, lexécution de requêtes kNN sur des flux de données continus et évolutifs soulève plusieurs défis majeurs, notamment laugmentation du temps de recherche, la dégradation progressive des structures dindexation, ainsi quune consommation accrue de mémoire et dénergie dans des environnements à ressources limitées.

Afin de répondre à ces contraintes, cette étude propose une architecture dindexation hiérarchique adaptée aux environnements IoT basés sur le Fog Computing. Lapproche proposée repose sur deux phases complémentaires. La première phase consiste en une réduction des données à laide de lalgorithme BIRCH, via la construction dun arbre Clustering Feature (CF-Tree). Cette étape permet de générer des micro-agrégats (micro-clusters) représentant des régions homogènes de lespace des données, réduisant ainsi la complexité computationnelle globale tout en préservant la structure spatiale intrinsèque du jeu de données.

La seconde phase consiste à construire un arbre dindexation binaire indépendant pour chaque micro-agrégat obtenu. Cette séparation entre les phases de regroupement et dindexation garantit une meilleure homogénéité des données au sein de chaque structure dindex, conduisant à des arbres moins profonds et à une réduction du coût de parcours lors des opérations de recherche kNN.

La méthodologie proposée exploite la partitionnement clusterisé afin de restreindre lespace de recherche, passant dun espace global à des sous-espaces locaux associés à chaque cluster. De plus, les requêtes kNN peuvent être exécutées de manière parallèle au niveau des clusters, améliorant ainsi lefficacité globale du système. Les résultats expérimentaux démontrent que cette approche hiérarchique permet de réduire significativement le temps de recherche et la surcharge computationnelle, tout en garantissant la scalabilité et ladaptabilité aux flux de données continus.

Ces résultats confirment que lintégration de techniques de réduction de données basées sur le regroupement avec des structures dindexation distribuées par cluster constitue une solution efficace et évolutive pour lindexation en temps réel dans les environnements IoT supportés par le Fog Computing.

Mots clés: Flux de données IoT continu, clustering, indexation, BIRCH Algorithm, requête k-plus proches voisins en parallèle, fog-cloud computing.

ملخص

برز إنترنت الأشياء (IoT) كأحد أكثر المجالات تأثيرًا في الحوسبة الحديثة، حيث تنتج الأجهزة المتصلة تدفقات بيانات مستمرة وعالية الكثافة ذات طبيعة متعددة الأبعاد. ومع التوسع المتزايد في حجم هذه البيانات، أصبحت الحاجة ملحّة إلى تطوير آليات فهرسة فعّالة تدعم البحث السريع في بيئات الحوسبة الضبابية Fog Computing، والتي تتميز بموارد حسابية محدودة مقارنةً بالبنى السحابية التقليدية.

تُعد خوارزمية البحث عن أقرب الجيران kNN – Neighbor k-Nearest من أكثر العمليات استخدامًا في تطبيقات إنترنت الأشياء، مثل كشف الشذوذ، المراقبة الذكية، والتحليل اللحظي للبيانات. غير أن تنفيذ kNN على بيانات متدفقة ومتنامية باستمرار يواجه تحديات جوهرية، أبرزها ارتفاع تكلفة البحث الخطي، تدهور أداء هياكل الفهرسة التقليدية مع الزمن، وزيادة استهلاك الذاكرة والطاقة في البيئات ذات القيود الحسابية.

لمعالجة هذه الإشكالية، تقترح هذه الدراسة بنية فهرسة هرمية موجهة لبيئة Fog تعتمد على مرحلتين تكميليتين. في المرحلة الأولى، يتم تقليص البيانات باستخدام خوارزمية BIRCH من خلال بناء شجرة Feature Clustering، (CF-Tree) والتي تُمكّن من تلخيص البيانات في صورة عناقيد دقيقة micro-clusters تمثل مجموعات متجانسة من النقاط ذات القرب المكاني. يسمح هذا التقليل بخفض التعقيد الحسابي للمرحلة اللاحقة دون فقدان البنية المكانية الأساسية للبيانات.

في المرحلة الثانية، يتم إنشاء شجرة فهرسة ثنائية مستقلة لكل عنقود، بحيث تُستخدم كبنية بحث محلية لدعم عمليات kNN بكفاءة. يهدف هذا الفصل بين مرحلتي التجميع والفهرسة إلى تقليل عمق الأشجار الناتجة وتحسين تجانس البيانات داخل كل فهرس، مما يحدّ من عدد العقد التي يتم زيارتها أثناء البحث.

تعتمد المنهجية المقترحة على تقليص مجال البحث من الفضاء الكلي إلى فضاءات محلية محدودة داخل العناقيد، وقد أظهرت النتائج التجريبية أن هذا النهج يُسهم في تقليل زمن البحث وعدد المقارنات الحسابية، مع الحفاظ على قابلية التوسّع والاستجابة الفورية لتدفقات البيانات المستمرة.

تؤكد هذه النتائج أن الدمج بين تقنيات التقليل العنقودي عبر BIRCH والفهرسة الثنائية الموزعة لكل عنقود يشكل مقاربة فعالة وملائمة لبيئات إنترنت الأشياء المعتمدة على الحوسبة الضبابية.

كلمات مفتاحية: تدفق بيانات إنترنت الأشياء المستمر، التجميع ، الفهرسة ، استعلام الجار الأقرب (KNN) ، حوسبة الضباب-السحابة .

Contents

Dedication . . . . . **II**

Acknowledgments . . . . . **III**

Abstract . . . . . **IV**

Contents . . . . . **X**

List of Figures . . . . . **X**

List of Tables . . . . . **X**

Introduction **2**
1 Contexte 2
2 Motivation 2
3 Objective And Contributions 3
4 Thesis Organization 4

Chapitre 1 : Internet of Things (IoT) **6**
1 Introduction 6
2 IoT Definition 6
3 IoT Functional Blocks 6
4 IoT Architecture 7
5 IoT Applications 9
5.1 Smart Cities 9
5.2 Industrial IoT 9
5.3 Smart Homes 10
5.4 Healthcare 11
6 IoT Challenges 11
6.1 Security and privacy 11
6.2 Interoperability 11
6.3 Data Management 12
6.4 Energy Consumption 12
6.5 Scalability 12
6.6 Latency and Real-Time Requirements 12
7 Cloud Computing 12
7.1 Cloud Computing Definition 12
7.2 Cloud Computing Architecture 12

7.3 Cloud computing challenges . . . 13
8 Fog Computing Fundamentals . . . 14
8.1 Fog Computing Definition . . . 14
8.2 Fog Computing Architecture . . . 14
9 Big IoT Data . . . 16
9.1 Big IoT Data Definition . . . 16
9.2 Big IoT Data Characteristics . . . 16
10 Conclusion . . . 17

Chapitre 2 : Indexing and Clustering Methods . . . 20
1 Introduction . . . 20
2 Multidimensional vs. Metric Space Concepts . . . 20
2.1 Metric Space Definition . . . 20
2.2 Multidimensional Definition . . . 20
2.3 Distance Functions . . . 21
2.3.1 Minkowski Distances . . . 21
2.3.2 Euclidean Distance . . . 22
2.3.3 Manhattan Distance . . . 23
2.3.4 Chebyshev Distance . . . 23
2.4 Concepts of Ball and Hyperplane . . . 24
2.4.1 Ball Partitioning . . . 24
2.4.2 Hyperplane Partitioning . . . 25
3 IoT Data Indexing Methods . . . 26
3.1 Overview of Big IoT Data and Indexing Challenges . . . 26
3.2 Taxonomy of IoT Indexing Methods . . . 26
3.2.1 Traditional Indexing Methods . . . 26
3.2.2 Metric-Based Indexing Methods . . . 27
3.2.3 Stream-Based Indexing Approaches . . . 28
3.2.4 Clustering-Based Indexing Approaches . . . 29
3.3 Evolution toward Fog-Based Indexing Architectures . . . 30
3.4 Data Reduction through Incremental Clustering . . . 30
3.5 Benchmarking Against Fog-Cloud BH-Tree Approaches . . . 31
3.6 Limitations of Existing Approaches . . . 31
3.7 Research Gap and Motivation . . . 32
4 Clustering Techniques for IoT Data . . . 32
4.1 Clustering Definition . . . 32
4.2 Types of Clustering Algorithms . . . 32
4.2.1 Partitioning Clustering Algorithms . . . 33
4.2.2 Density-based Algorithms . . . 34
4.2.3 Grid-Based Clustering Algorithms . . . 35
4.2.4 Model-Based Clustering Algorithms . . . 36
4.2.5 Hierarchical Clustering Algorithms . . . 38
4.3 Use of Clustering in Fog-Based Indexing . . . 39
5 Advantages of Continuous IoT Data Streams . . . 41
6 Conclusion . . . 42

Chapitre 3 : Proposed indexing method . . . 43

1 Introduction . . . 44
2 System Architecture Overview . . . 44
2.1 Data Ingestion Mechanism . . . 45
2.2 Sliding Window Model . . . 46
2.3 Data Transformation . . . 46
3 Fog Node Design . . . 46
3.1 Cluster Distribution Strategy . . . 46
3.2 Distributed Execution Model . . . 47
4 Incremental Clustering Implementation . . . 47
4.1 Incremental BIRCH Model . . . 47
4.2 Cluster Extraction Strategy . . . 48
4.3 Cluster Filtering Mechanism . . . 49
5 Metric Tree Index Construction . . . 49
5.1 Index Construction . . . 49
5.2 Tree Structure Definition . . . 50
5.3 Recursive Tree Construction . . . 50
5.4 Pivot Selection Strategy . . . 50
5.5 Degeneracy Handling . . . 51
6 kNN Query Processing . . . 51
6.1 Search Algorithm on Metric Tree . . . 51
6.2 Distance Computation . . . 52
6.3 Candidate Management . . . 52
6.4 Leaf Processing . . . 53
7 Parallel kNN over Trees . . . 53
7.1 Forest-Based Search . . . 53
7.2 Candidate Tree Selection . . . 54
7.3 Distributed Result Aggregation . . . 54
7.4 Cloud-Level Analytics . . . 55
8 Conclusion . . . 55

Chapitre 4 : Experimental Evaluation . . . 57
1 Introduction . . . 57
2 Experimental Setup . . . 57
2.1 Hardware Configuration . . . 57
2.2 Experimental Parameters . . . 57
2.3 Dataset Description . . . 58
2.4 Evaluation Metrics . . . 58
3 Clustering Evaluation . . . 59
4 Metric-Tree Construction . . . 61
4.1 Number of Generated Trees . . . 62
4.2 Structural Complexity Analysis . . . 64
5 Search-Space Reduction And Traversal Analysis . . . 65
5.1 Efficiency Ratio . . . 65
5.2 Distance Reduction . . . 66
5.3 Traversal Reduction . . . 68
5.4 Traversal Complexity . . . 69

6 Conclusion 69

Conclusion **71**

List of Figures

1 IoT architecture with three layers 8
2 IoT architecture with five layers 9
3 Cloud computing architecture 13
4 Fog computing architecture 15
5 Illustration of distance between two points in a metric space. 20
6 Data points represented in a three-dimensional feature space 21
7 Ball partitioning using hyperspherical regions 24
8 Hyperplane partitioning using axis-aligned splits 25
9 Overview of the BIRCH clustering process and CF-tree structure . . . 41
10 Streaming IoT Data Processing Pipeline Architecture 45
11 Impact of k on calculated distances across the evaluated datasets . . . 68
12 Impact of k on visited leaves across the evaluated datasets 69

List of Tables

1 Hardware Configuration 57
2 Experimental Parameters 58
3 Summary of the datasets used in the implementation 58
4 Sample Clustering Results on IoT Telemetry Sensor Dataset 59
5 Sample Clustering Results on Industrial IoT Dataset 60
6 Sample Clustering Results on The Intel Lab Dataset 60
7 Sample Clustering Results on PM2.5 Air Quality Dataset 61
8 Sample Indexing Results on The Intel Lab Dataset 62
9 Sample Indexing Results on Industrial IoT Dataset 63
10 Sample Indexing Results on IoT Telemetry Dataset 63
11 Sample Indexing Results on PM2.5 Air Quality Insights Dataset 64
12 Intel Lab: Traversal Statistics by k 66
13 IoT Telemetry: Traversal Statistics by k 66
14 Industrial IoT: Traversal Statistics by k 67
15 PM2.5 Air Quality Insights: Traversal Statistics by k 67

Introduction

1 Contexte

The rapid development of Internet of Things (IoT) technologies means that connected devices are now generating huge amounts of real-time data. This generated data requires efficient means of storing, processing and analysing it, particularly when searching for similarity queries. The most significant challenge in this environment is quickly accessing relevant data due to its vast quantity, diversity and frequent changes.

Indexing is one of the primary mechanisms used to organize and facilitate access to big data. Indexing aims to reduce the search space and the number of computational operations, allowing for faster execution of queries, especially those based on similarity between objects. In terms of data heterogeneity in size, type and dimension, the metric space is proposed as the best compromise, since only the distances between data are used, regardless of their types and dimensions [83].

To improve the management of large volumes of IoT data,fogcloud architectures are currently employed. The hierarchical fog architecture comprises three layers [56]: the terminal layer, the fog layer and the cloud layer (see Figure 1). The terminal layer (also known as the basic layer) contains different types of IoT devices, such as sensors, vehicles and mobile devices, which are distributed in various locations. This layer generates and transfers data to the fog layer. The fog layer is an intermediate layer between the terminal layer and the cloud layer. It provides IoT users with services such as data processing and storage via fog nodes. It is distributed and close to the devices. The cloud layer is responsible for large-scale and complex computingźź, and is characterised by different service discovery mechanismsźš and permanent data storage[56] .

Due to the dynamic nature of IoT data and its changing distribution over time, the demand for analytics solutions capable of managing the heterogeneity and high velocity of this data . Clustering algorithms are used in this situation to classify data into homogeneous groups based on criteria such as similarity. This facilitates indexing and enables parallel processing during both index construction and similarity searches.

This study aims to propose effective solutions for indexing streaming IoT data and improving the performance of similarity queries. It is based on a general approach that combines concepts such as indexing, metric spaces, fog computing and clustering.

2 Motivation

With the proliferation of smart devices, IoT systems are generating massive volumes of real-time spatio-temporal data. According to recent projections, over 75 billion IoT devices will be connected globally by 2025, producing more than 300 zettabytes of data annually [42]. However, traditional cloud-centric architectures are increasingly insufficient to meet the low-latency and high-reliability demands of modern IoT applications, especially in time-sensitive domains such as health monitoring, smart grids,

and autonomous vehicles [33].

The problem of data latency has emerged as a serious bottleneck. Studies indicate that cloud-only systems can experience up to 40% higher latency compared to architectures using local or edge processing. To overcome these limitations, fog computing has emerged as a complementary paradigm that brings computation and storage closer to the edge of the network, enabling faster data processing and reducing the strain on centralized cloud infrastructure [17].

Research has demonstrated that by using fog nodes, network speeds can be made up to 90% quicker, and energy usage can be decreased by up to 60% compared to using the cloud only . These improvements mean that fog computing is perfect for IoT applications that need to be very fast and are used by lots of people.

However, it is difficult to efficiently collect and retrieve different types of data from the Internet of Things (IoT) in fog-cloud environments. Current methods have problems with using too much energy, not being able to adapt to more users, and slow query performance [86]. This means we need a new way of indexing, which should use adaptive clustering and intelligent fog-level indexing to support parallel, real-time kNN queries. The new indexing strategy should also use less energy and be faster.

3 Objective And Contributions

The Internet of Things (IoT) is growing rapidly, which means more and more data is being generated by sensors and smart devices all over the place and at all times. It is very important that this data can be retrieved rapidly and efficiently, as this is needed for real-time analytics and decision-making in applications where delay is not an option, such as smart cities, environmental monitoring and intelligent transportation systems. However, current cloud-centric processing systems usually have problems with delayed communication and higher energy usage.

The primary objective of this thesis is to design and implement an efficient indexing approach large-scale IoT spatio-temporal data within fog–cloud environments. By leveraging fog-based resources, the proposed approach aims to bring computation closer to the data sources, thereby reducing query latency, improving data organization, and supporting efficient nearest neighbor search over continuously generated IoT datasets.

More specifically, this research pursues the following objectives:

- The study aims to address the limitations of current cloud-based indexing methods when used with large amounts of data from the Internet of Things (IoT).
- The second is to use an adaptive clustering mechanism to efficiently organise different types of IoT data at the fog layer.
- The aim is to develop an indexing structure that will be able to support efficient k-Nearest Neighbor (kNN) queries.
- This is an approach for searching for continuous IoT data using kNN queries.
- To evaluate the effectiveness of the proposed approach using real-world IoT datasets, analyzing its performance in terms of query efficiency, scalability, and computational cost.

4 Thesis Organization

This thesis is organized into four main chapters that progressively introduce the research context, present the theoretical foundations, describe the proposed implementation, and evaluate the performance of the proposed indexing approach.

Chapter 1 Internet of Things Fundamentals:The Basics of the Internet of Things This chapter tells you what the Internet of Things is. It shows the most important parts of IoT systems, how they are built, and how they are used in places like smart cities, healthcare, and industry. The chapter talks about the main problems with IoT systems. These include how to manage data, how to make the systems bigger, how much energy they use, and how long they take to respond. Also, the basics of cloud computing and fog computing are explained to show how they can manage large amounts of data from the Internet of Things.

Chapter 2 Indexing and Clustering Methods:This looks at the theory behind ways of organising data. It introduces the ideas of multidimensional and metric spaces, similarity measures, and distance functions used for similarity search. The chapter also looks at different ways to cluster data and explains why they are important for dealing with large amounts of data from the Internet of Things. Finally, it talks about how clustering can improve how well indexes work for continuous IoT data streams.

Chapter 3 Proposed Indexing method: This chapter presents a hierarchical indexing approach for managing continuous IoT data streams in fog environments. It combines incremental clustering using BIRCH with the construction of metric index trees to organize data efficiently. The approach also supports distributed and parallel kNN query processing while reducing the search space through cluster-based pruning.

Chapter 4 Experimental Evaluation: evaluates the performance of the proposed approach using real IoT datasets. This chapter describes the experimental environment, datasets, and evaluation metrics. It analyzes the results obtained in terms of clustering quality, index construction efficiency, and kNN query performance, and discusses the impact of different system parameters on overall system performance.

Chapter 1

Internet of Things (IoT)

1 Introduction

The Internet of Things (IoT) is one of the most transformative paradigms in modern computing. It enables the interconnection of billions of devices, sensors and systems via the internet. The IoT provides the foundation for smart environments such as smart cities, healthcare systems, industrial automation and intelligent transportation. By integrating sensing, communication, and computational capabilities, the IoT facilitates real-time data collection, processing, and decision-making across heterogeneous and distributed infrastructures.

2 IoT Definition

The Internet of Things (IoT) comprises a system of interrelated computing devices, mechanical and digital machines, objects, animals and people. All of these are given unique identifiers (UIDs) and the ability to transfer data over a network, eliminating the need for human-to-human or human-to-computer interaction. The IoT enables physical objects to collect and exchange data, creating a seamlessly connected environment in which devices can communicate and act intelligently.

It is a network of physical objects such as devices, vehicles and buildings that are embedded with electronics, software, sensors and network connectivity. This enables these objects to collect and exchange data.

3 IoT Functional Blocks

A IoT system is composed of a number of functional blocks to facilitate various utilities to the system. These blocks are device, communication, service, management, security and application [62].

- The devices block provides monitoring , detecting, actuating, and surveillance activities. Devices exchange data with other connected devices and applications, or collect data from other devices. They also process data locally or send data to centralized servers or cloud-based application back-endsto process data or perform some tasks locally and other tasks within the IoT infrastructure depending on temporal and spatial constraints. IoT devices may also be of various types, for example, wearable sensors, smart watches, LED lights, automobiles, and industrial machines [62].
- The communication block ensures communication across devices and with distributed servers. IoT communication protocols generally function in the data link layer, network layer, transport layer and application layer [62].
- The services block in a IoT system supports different types of functions including device control, data management, device modeling, data delivery and device recovery services [62].
- The management block delivers various functions to govern an IoT system in order to research the IoT systems underlying governance [62].

- The security of the IoT system providing functions such as authentication, permission, confidentiality, message integrity and data security [62].
- The application block is the most critical in terms of users as it works as an interface that delivers the necessary modules to control and supervise various aspects of the IoT system [62].

4 IoT Architecture

The rapid development of the Internet of Things (IoT) has been greatly influenced by the integration of related technologies, such as cloud and edge computing and big data analytics. These developments have led to remarkable improvements in sensor and actuator functionality, as well as the miniaturisation of devices with embedded networking capabilities. Consequently, the number of IoT devices has increased exponentially to reach the billions or even trillions depending on the application domain and geographic scale. [87].

To efficiently manage the vast amounts of data generated by such diverse and distributed environments, a layered architecture has proven to be the most adaptable and scalable solution. The three-layer architecture model is widely accepted as the foundational framework for IoT systems. It comprises the perception, network and application layers.[81] [63] [33].

1. Perception Layer The perception layer, also referred to as the sensing layer, is the lowest level in the IoT stack. Its role is to capture data from the physical environment using various devices, including RFID tags, sensors, GPS modules and camera systems. It transforms this environmental data into digital signals for further processing and transmits them to the next layer. [11]

2. Network Layer The network layer acts as a bridge between the perception and application layers. It receives data collected by sensors and ensures its transmission via communication protocols and technologies such as Zigbee, 5G, NB-IoT and Wi-Fi. Gateways, routers and cloud interfaces may also be included in this layer, playing a critical role in ensuring secure and reliable data transfer. [80].

3. Application Layer The application layer sits at the top of the stack and processes the received information, providing services that are tailored to user needs. The specific domains covered by this layer vary depending on the use case, and include smart homes, smart cities, intelligent transportation, precision agriculture, and e-healthcare. Its role is to ensure that IoT-generated data is meaningful and valuable for decision-making and automation. [81] [79].

   To improve abstraction and support complex services, several researchers have suggested extending the basic three-layer model to include a middleware layer and a business layer. This five-layer architecture improves scalability, interoperability and the integration of external systems and services. [80][45].

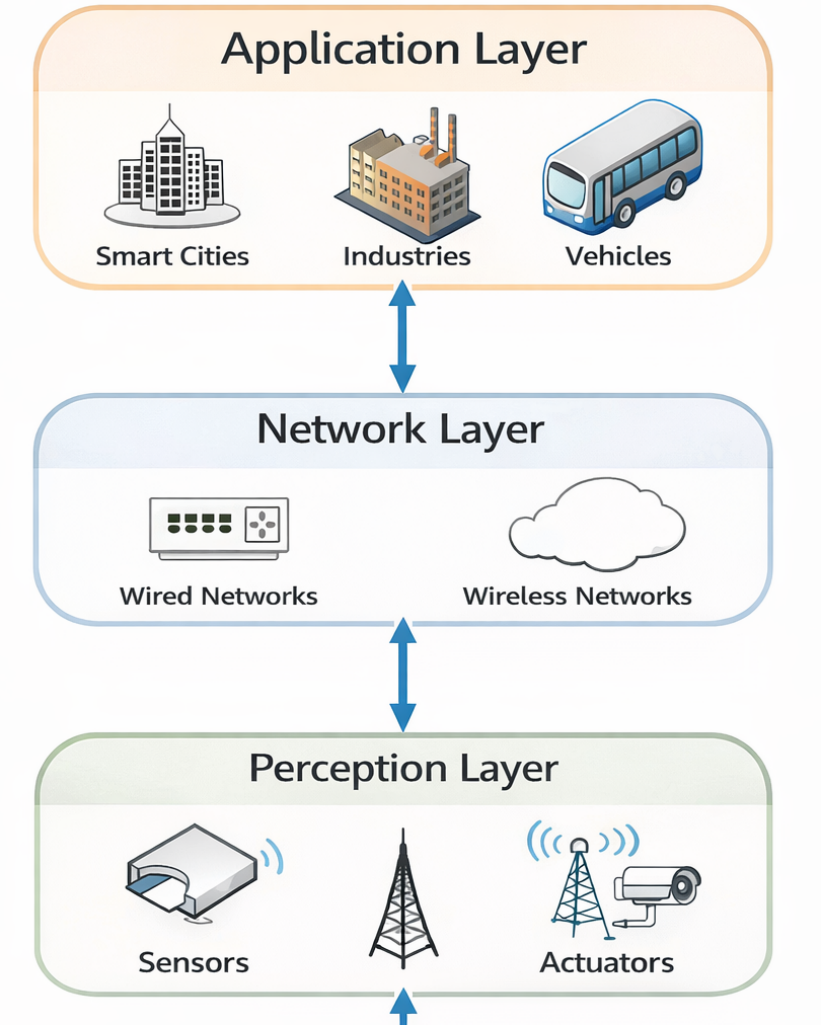


Figure 1: IoT architecture with three layers

4. Middleware Layer This layer acts as a service management layer, linking applications to the underlying infrastructure. It provides application programming interfaces (APIs), data filtering, semantic analysis and service discovery, ensuring that heterogeneous Internet of Things (IoT) services can operate cohesively. The middleware also handles data storage and processing based on context and incoming data streams, as well as decision-making.[80].

The business layer manages all the lower layers and provides tools for data analysis, service orchestration and the enforcement of business logic. It is essential for decision-making, policy definition and preserving user privacy. By analysing information from the middleware and comparing it across layers, the business layer enables the optimisation of services and strategic system adjustments. [33][45].

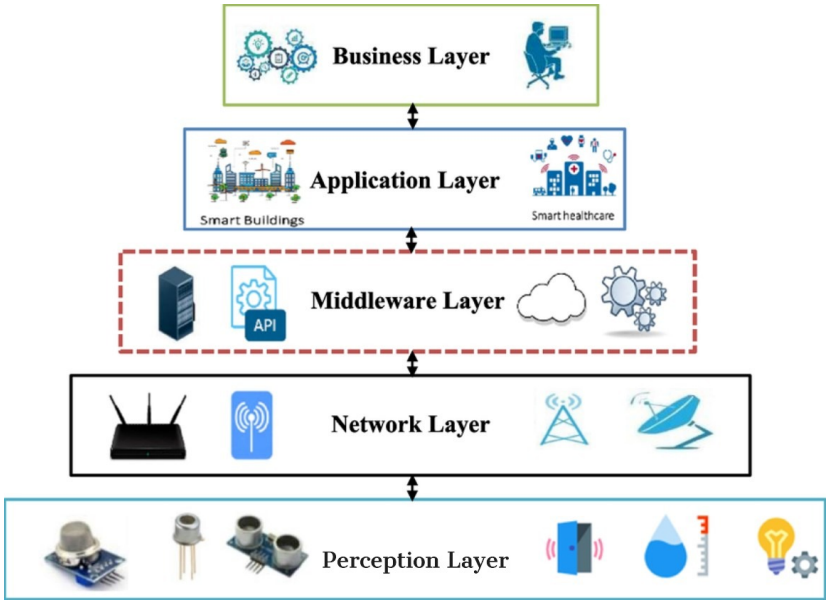


Figure 2: IoT architecture with five layers

5 IoT Applications

The Internet of Things (IoT) has transformed traditional sectors by enabling real-time data collection and analysis, automation, and intelligent decision-making. IoT applications span numerous industries, increasing efficiency, safety and personalisation. The main IoT application domains are detailed below, organised by sector:

5.1 Smart Cities

Smart cities are a key application area within the Internet of Things (IoT) paradigm. They integrate a large network of connected devices, sensors and intelligent systems to improve urban living. Through the use of real-time data acquisition and analysis, smart cities seek to optimise infrastructure, streamline public services, and promote sustainable development. Smart city applications span transportation systems, waste management, environmental monitoring, public safety and energy grids, all of which operate cohesively through interoperable, often AI-enhanced, IoT infrastructures. [64]

Despite their potential, smart cities face significant challenges related to scalability, data privacy, interoperability and energy efficiency. Emerging research proposes solutions such as green IoT, edge AI (TinyML) and privacy-preserving frameworks, including differential privacy and blockchain.[59]

As a strategic component of digital transformation, smart cities embody the convergence of the IoT, AI and urban governance and are central to developing resilient, citizen-centred urban ecosystems.

5.2 Industrial IoT

The integration of internet-connected devices and sophisticated data analytics in industrial contexts such as manufacturing, logistics, energy and agriculture is known as

the Industrial Internet of Things (IIoT). IIoT emphasises machine-to-machine communication, predictive maintenance, and automation to increase operational efficiency, reduce costs, and enhance safety. In contrast, consumer IoT focuses on user-facing applications such as wearables and smart homes.

IIoT uses sensors, actuators and edge devices to link tangible industrial assets, such as robots, conveyor belts and turbines, to the digital realm. These systems collect data in real time and transmit it to cloud or fog computing infrastructures via networks for analysis and decision-making purposes. The essential elements of the IIoT are:

- Sensors and Actuators: For monitoring equipment conditions and interacting with the physical environment.
- Edge and Fog Computing: To reduce latency and enable local data processing.
- AI and Machine Learning: For fault prediction, optimization, and decision support.
- Cybersecurity Mechanisms: To protect critical infrastructure from attacks.

Applications of IIoT span various sectors:

- In manufacturing, IIoT supports smart factories and Industry 4.0 principles through real-time production monitoring.
- In energy, it facilitates grid optimization and asset performance management.
- In logistics, it improves supply chain visibility and reduces delays via connected fleet systems.

"IIoT offers unprecedented opportunities to transform traditional industries through digitalization, yet it also introduces challenges in interoperability, data privacy, and system integration" [50].

5.3 Smart Homes

Smart Homes represent a prominent application domain within the Internet of Things (IoT), where various interconnected devices and systems are embedded in residential environments to enhance comfort, energy efficiency, security, and automation. The functionality of such environments is contingent on the seamless integration of sensors, actuators, network connectivity, and intelligent control systems, which collectively facilitate the real-time monitoring and management of household functions.

The core components of a smart home typically include:
Environmental sensors (e.g., temperature, humidity, air quality)

- Smart appliances (e.g., refrigerators, washing machines)
- Security systems (e.g., smart locks, surveillance cameras)
- Energy management systems (e.g., smart thermostats, solar integration)
- Voice-controlled assistants (e.g., Alexa, Google Assistant)

Data from these devices is often processed locally through edge devices or remotely via cloud platforms. Artificial intelligence and machine learning techniques are increasingly used to enable predictive and adaptive automation, such as learning user preferences or detecting anomalies.

5.4 Healthcare

The Internet of Things (IoT) is being used in more and more areas of healthcare. This has led to the Internet of Medical Things (IoMT), which is a network of medical devices and applications that collect, analyse and send health data in real time. These technologies allow doctors to monitor patients, manage long-term health problems, provide smart diagnoses and act quickly to treat patients. For example, smartwatches and biosensors can monitor things like heart rate, oxygen levels and glucose. If they spot anything unusual, they'll automatically let healthcare providers know, meaning people won't have to go to hospital as much and will get better faster. IoMT devices can also help hospitals run more efficiently by tracking equipment, managing stock, and making workflows more streamlined. But using IoMT also creates problems with data privacy, cybersecurity and following the rules. As healthcare systems become more digital, the IoMT is very important in making healthcare more accessible, predictive and personalised.

6 IoT Challenges

The Internet of Things (IoT) is growing quickly, but there are some big problems that are stopping it from being used as much as it could be. These challenges can be put into three groups: technical, security-related, or related to how well the system can be used by other systems.

6.1 Security and privacy

Security is one of the biggest worries when it comes to IoT systems. The large number of connected devices, which often have limited processing power, makes them vulnerable to cyberattacks. If people can't trust the data, or if it isn't kept safe, then the whole system could be at risk.

6.2 Interoperability

The fact that there is no standard means that devices and platforms do not work well together. Devices from different manufacturers often use their own special communication rules, which makes it hard to work together and limits how many devices can be used at once. We are working to make things more consistent, but this is being done in different ways.

6.3 Data Management

IoT devices generate large amounts of different types of data at high speeds [12, 34]. Managing, storing, processing and analysing this data as it happens is very difficult, especially when there are not many cloud or edge computing resources available [18, 68].

6.4 Energy Consumption

Many IoT devices are battery-powered and are expected to operate for extended periods [87]. Keeping energy use low is a big worry, especially for sensors in hard-to-reach places where it costs a lot to fix or it's just not practical [7].

6.5 Scalability

As the number of IoT devices grows to billions, it becomes a big design challenge to make sure that systems can grow to support this expansion without getting worse in how well they work or how reliable they are [34, 12].

6.6 Latency and Real-Time Requirements

Some IoT applications, like self-driving cars or healthcare monitoring, need to process data in real time with very little delay [68]. This is difficult to do in places where there are not a lot of resources [67].

7 Cloud Computing

7.1 Cloud Computing Definition

Cloud computing is a way of accessing a shared set of computing resources (like networks, servers, storage, applications and services) whenever you need them. These resources can be set up and ready to use very quickly, with minimal management or interaction from the service provider.

The model is all about making sure it's available and includes five key features (on-demand self-service, wide network access, sharing resources, quick adaptability and measured service), three service models (IaaS, PaaS and SaaS) and four ways it can be used (private, public, hybrid and community clouds).

7.2 Cloud Computing Architecture

Cloud computing architecture defines how computing resources are organised and delivered over the internet. It is typically structured into three key layers: Infrastructure, Platform and Software layers, also known as IaaS, PaaS and SaaS respectively.

- Infrastructure as a Service (IaaS):Provides virtualized physical computing resources such as servers, storage, and networking.

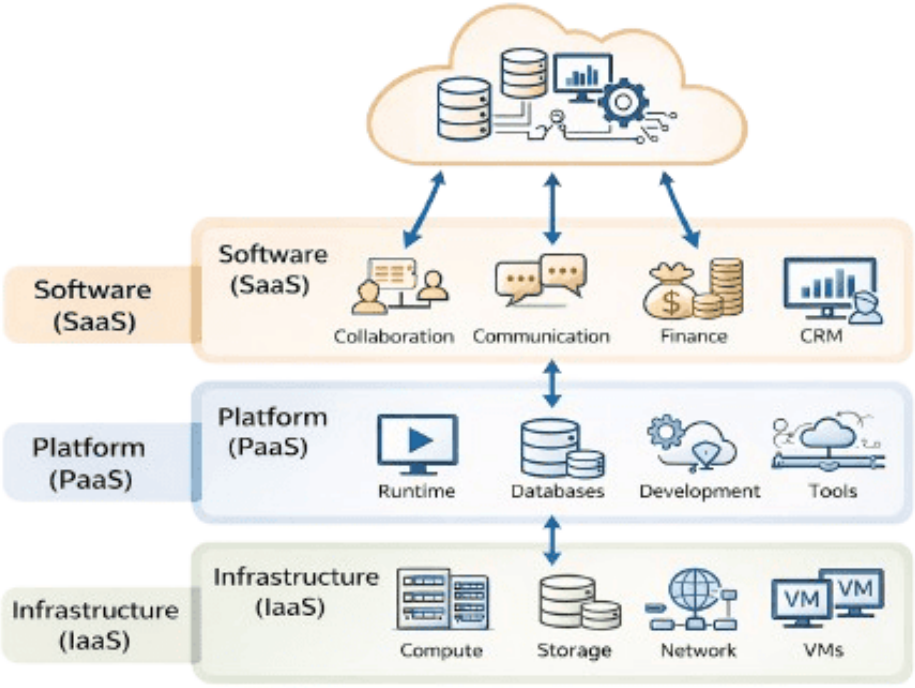


Figure 3: Cloud computing architecture

- Platform as a Service (PaaS):Provides runtime environments, development tools, and database management.
- Software as a Service (SaaS):Delivers software applications over the internet, accessible through web browsers.

7.3 Cloud computing challenges

Integrating cloud computing with the Internet of Things (IoT) provides scalability, remote accessibility and advanced analytics capabilities. However, it also introduces challenges affecting performance, security and reliability. Key challenges include:

Security and Privacy

IoT devices often collect personal or industrial data. When this data is sent to the cloud, it can be intercepted, accessed without permission and attacked by hackers. Multi-tenant cloud environments make it more likely that data will be stolen, and the fact that you don't know where your data is stored could mean that you don't meet the necessary standards.

Latency and real-time processing

Many IoT applications, such as self-driving cars or industrial control systems, need to process data very quickly. Moving large amounts of data to one main cloud server can make the system slower.

Bandwidth and network constraints

Continuous data streaming from large numbers of devices can overload networks, increasing operational costs and reducing performance,

Data Management and Heterogeneity

IoT data comes in many different formats (such as structured, semi-structured and unstructured), which means it needs to be integrated and processed in a complex way. Cloud platforms must be able to handle large amounts of different types of data quickly.

Reliability and Availability

IoT-cloud systems are dependent on stable internet connections, so they are vulnerable to outages. If you can't access the cloud or your internet connection is down, you might not be able to do important things with your IoT devices. Interoperability: IoT ecosystems consist of different devices, protocols and platforms. Integration with cloud services often requires standard interfaces and software solutions.

8 Fog Computing Fundamentals

8.1 Fog Computing Definition

Fog computing is a type of distributed computing that brings computation and data storage closer to the network edge, where many Internet of Things (IoT) devices are located [18, 68]. This means you don't need to rely on the cloud so much for these tasks that use a lot of resources. This makes things work better and faster [67].

Mist computing takes this even further. It brings computation and data storage closer to the edge using devices such as Mist computing servers [61]. These are low-power servers that can be used in large numbers [40].

8.2 Fog Computing Architecture

Fog computing is a way of storing information that is spread across many different devices, such as phones and sensors. It is a way of connecting the cloud and all the devices that are connected to the internet, like smart fridges and thermostats. It puts processing, storage and networking resources closer to the data sources. This reduces delay, saves bandwidth and allows for instant decision-making [18, 24, 68]. The building is usually made up of three main parts:

The Edge Layer is the closest to the physical environment and consists of IoT devices such as sensors, actuators, RFID tags, embedded controllers and mobile devices. These components are responsible for getting data, filtering it in advance and, in some cases, basic analysis [34]. Edge devices often need help from nearby fog nodes for more complex tasks because they don't have a lot of computing power or energy [67].

Fog Layer This is a layer between the edge and the cloud. It is made up of fog nodes, which can include IoT gateways, routers, switches, microservers, and even dedicated fog clusters used at base stations or local data centres [40]. These fog nodes

are used for tasks that require a lot of computing power, such as analysing data, machine learning and storing temporary data. The fog layer can operate on its own, making sure important applications like self-driving cars, industrial process control and real-time video analysis keep working even when there are problems with the cloud [84]. This layer also makes sure that data is encrypted before it is sent, and that it follows the rules about privacy.

The way these layers work together makes them easy to expand and resistant to failure. By processing data closer to where it is generated, the fog computing architecture reduces network congestion, supports applications that are sensitive to latency, and enables faster feedback loops [68]. For example, in smart transportation systems, fog nodes installed at traffic intersections can process video feeds locally to detect congestion or accidents in milliseconds, while only sending summaries or reports of anything unusual to the cloud for long-term analysis [87]. In the same way, in the industrial IoT, special computer parts called fog nodes in manufacturing plants can check and control machines as they happen.

What's more, how fog computing is set up can be different depending on what it's being used for. It can either be set up so that fog nodes directly serve local edge devices (a flat topology), or so that multiple levels of fog nodes aggregate and process data before it reaches the cloud (a hierarchical topology) [24]. This flexibility makes fog computing particularly useful in areas such as healthcare monitoring, environmental sensing, and smart energy grids.

The idea behind fog computing architecture is to make the most of all the different parts of a network, making sure that the computing power is used in the best way. This helps to balance how well the network performs, how much it costs, how much energy it uses, and how private the data is. It works well with cloud computing, creating a strong system that can handle the needs of next-generation IoT ecosystems [18, 40].

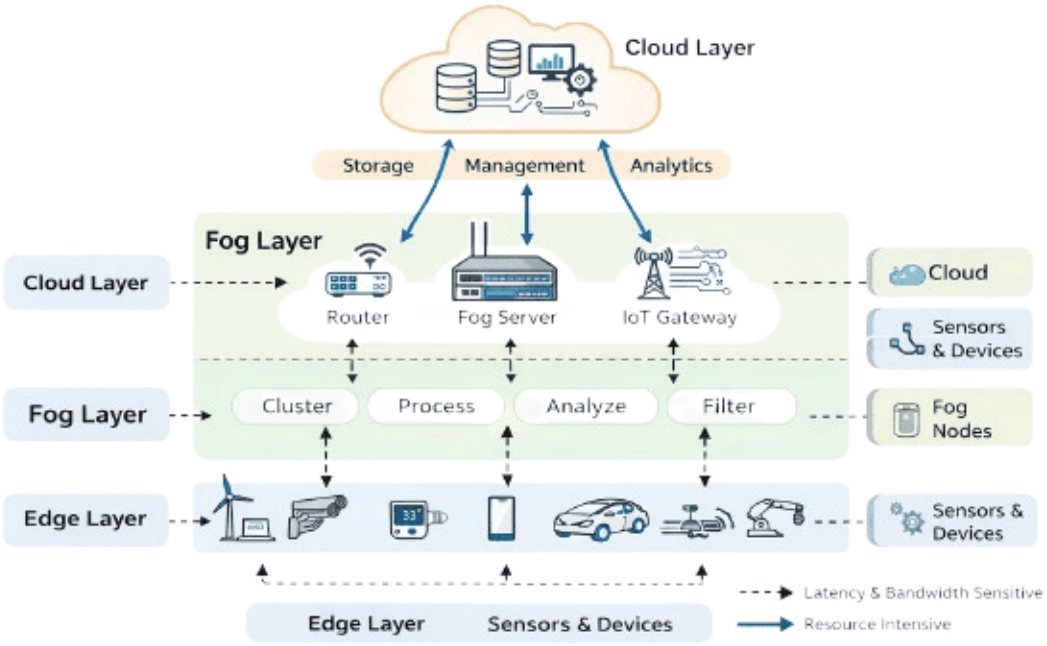


Figure 4: Fog computing architecture

9 Big IoT Data

9.1 Big IoT Data Definition

Big IoT Data is a term used for very large sets of data from many different sources that are produced by IoT devices [34, 12]. These devices are connected to the internet and create lots of data very quickly and in many different formats. This data is often very complicated and needs special software to store it, process it, and analyse it [54]. Big IoT Data is different from traditional big data. It is real-time or near-real-time, aware of context and often distributed geographically [68]. This means it requires low-latency analytics and adaptive data management strategies across cloud and fog computing layers [18, 67].

This type of data is also difficult to trust because of noise, redundancy and incompleteness [69]. It is also difficult to use to help make decisions in areas like smart cities, healthcare, industrial automation and intelligent transportation, because only a small part of the data is relevant [87].

9.2 Big IoT Data Characteristics

The concept of big data in IoT environments is mainly driven by the massive and continuous generation of data from distributed sensors, devices, and connected systems. Unlike traditional datasets, IoT data are not only large in size but also dynamic, heterogeneous, and generated at high speed. These characteristics introduce significant challenges in terms of storage, processing, and real-time analysis [34, 12].

In IoT systems, data are produced in a streaming manner rather than as static datasets. This continuous generation makes it difficult to rely on traditional batch processing systems, which are not designed to handle real-time data flows. As a result, modern big data solutions, including cloud and fog computing, are required to support scalable storage and low-latency processing [54, 9].

Big IoT data are commonly described using the well-known “V” characteristics. In the context of this work, four key dimensions are particularly relevant: velocity, variety, veracity, and value.

1. Velocity

Velocity refers to the speed at which data are generated and processed. In IoT environments, data streams are produced continuously by sensors and devices, often in real time. This high-speed data flow requires systems capable of processing information immediately as it arrives.

In applications such as industrial monitoring or smart healthcare, delays in processing may lead to critical consequences. Therefore, real-time or near real-time processing becomes essential. Technologies such as stream processing frameworks and in-memory computing systems are commonly used to handle high-velocity data [71, 5].

From an indexing perspective, velocity introduces a major challenge: it is not feasible to index all incoming data points individually in real time. This limitation motivates the use of incremental clustering and data reduction techniques, as proposed in this thesis, in order to maintain efficient query processing under continuous data flow.

2. Variety

Variety refers to the diversity of data formats generated in IoT systems. Data may include structured sensor readings, semi-structured logs, and unstructured content such as images or video streams [23].

This heterogeneity complicates data integration and processing, as different data types require different storage and analysis techniques. Technologies such as NoSQL databases and data lakes are often used to manage this diversity due to their flexibility in handling multiple data formats [21].

In the context of similarity search and indexing, variety also affects the choice of distance metrics and data representation. For example, numerical sensor data can be processed using Euclidean distance, while other types of data may require specialized similarity measures. This diversity reinforces the need for flexible indexing structures capable of handling heterogeneous data.

3. Veracity

Veracity refers to the quality, accuracy, and reliability of data. IoT data are often collected from distributed and heterogeneous sources, which may introduce noise, missing values, or inconsistencies [69].

Low-quality data can negatively impact both clustering and indexing processes. For instance, noisy data may lead to incorrect cluster formation, which in turn affects the efficiency of indexing structures and the accuracy of kNN queries. Therefore, ensuring data quality or incorporating robust processing techniques is an important aspect of IoT data management.

4. Value

Value represents the useful insights that can be extracted from data. The ultimate goal of IoT systems is not only to collect data but also to transform it into actionable knowledge that supports decision-making [55, 3].

In this context, efficient indexing and query processing play a crucial role in enabling fast access to relevant data. Without appropriate indexing mechanisms, extracting value from large-scale IoT data becomes computationally expensive and time-consuming. Therefore, improving query efficiency directly contributes to enhancing the value of IoT data.

10 Conclusion

The Internet of Things represents a dynamic and continuously evolving environment characterized by the generation of large-scale, high-velocity, and heterogeneous data streams. These characteristics introduce several challenges related to data management, scalability, latency, and processing efficiency.

Cloud and fog computing paradigms provide complementary solutions to these challenges. While cloud computing offers scalable storage and processing capabilities, fog computing enables low-latency data processing closer to data sources. This distributed architecture is particularly suitable for real-time IoT applications.

However, the effectiveness of such systems depends heavily on the ability to efficiently organize, index, and query the generated data. Traditional methods are not sufficient to handle the complexity of continuous IoT streams, which highlights the need for advanced techniques that combine data reduction, indexing, and distributed

processing.

These observations motivate the approach proposed in this thesis, which integrates clustering-based data reduction with metric indexing and fog-based parallel processing to support efficient kNN query execution in streaming IoT environments.

Chapter 2

Indexing and Clustering Methods

1 Introduction

The Internet of Things (IoT) is growing very quickly, which has led to a lot of different types of data being created. This is often called Big IoT Data. Managing, organising, and efficiently retrieving this data have become significant challenges in modern distributed environments. Indexing methods are really important for letting you search and find things quickly, while clustering techniques are essential for putting similar data together and making things easier to compute.

2 Multidimensional vs. Metric Space Concepts

2.1 Metric Space Definition

A *metric space* is a fundamental mathematical structure in which the concept of distance between elements is formally defined. Formally, a metric space is a pair (X, d), where X is a non-empty set of objects and $d : X \times X \to \mathbb{R}_{\geq 0}$ is a distance function, or *metric*, that satisfies the following properties for all $x, y, z \in X$ [19]:

1. **Non-negativity:** $d(x, y) \geq 0$
2. **Identity of indiscernibles:** $d(x, y) = 0 \iff x = y$
3. **Symmetry:** $d(x, y) = d(y, x)$
4. **Triangle inequality:** $d(x, z) \leq d(x, y) + d(y, z)$

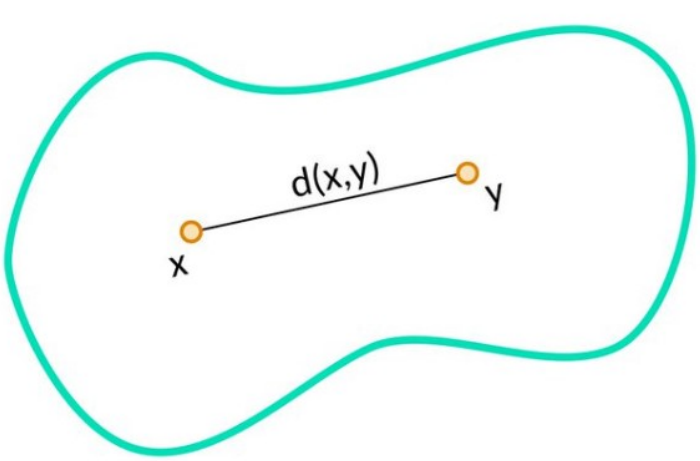


Figure 5: Illustration of distance between two points in a metric space.

2.2 Multidimensional Definition

Multidimensional data refers to datasets where each observation or object is described by multiple attributes, often referred to as dimensions or features, which can be numerical, categorical, or mixed in nature. In computational contexts, these dimensions form

a feature space, where each dimension represents a distinct measurable variable. Multi-dimensional representations are fundamental in data analysis, pattern recognition, and machine learning, as they enable complex phenomena to be modeled and compared in high-dimensional spaces. However, as dimensionality increases, data processing faces the well-known curse of dimensionality, which impacts indexing, similarity search, and computational efficiency[38]

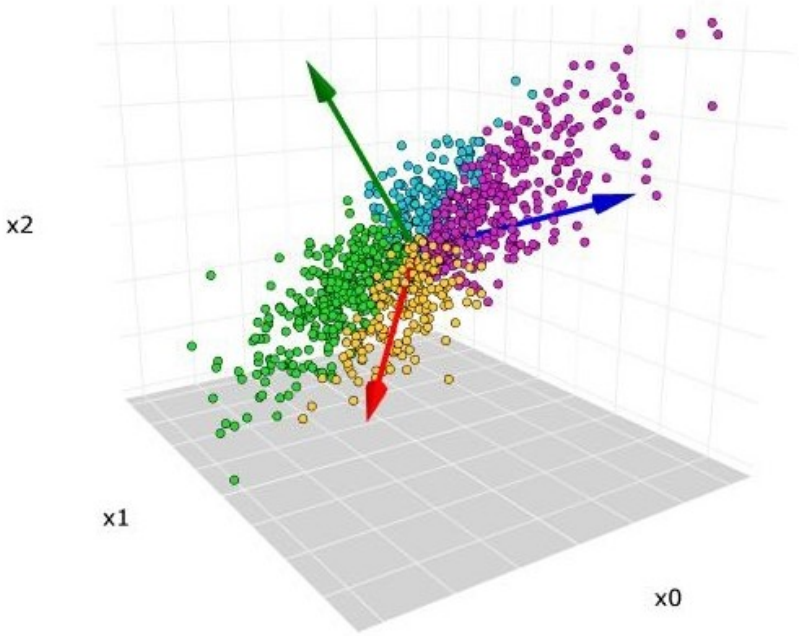


Figure 6: Data points represented in a three-dimensional feature space

2.3 Distance Functions

Distance functions are mathematical tools used to measure the similarity or dissimilarity between objects. They play a fundamental role in various applications such as pattern recognition, machine learning, and data mining [27, 43]. These metrics, typically derived from object coordinates, can be categorized into discrete and continuous distance functions. Discrete distance functions produce values from a limited set, while continuous distance functions yield values from a potentially infinite range . Furthermore, distance functions can also be distinguished based on their computational complexity, which directly affects their efficiency in large-scale applications [41].

2.3.1 Minkowski Distances

The *Minkowski distances* represent a family of metric functions widely used in mathematics, data analysis, and pattern recognition . They are often referred to as L_p *metrics*, where each specific case is determined by a parameter p. These metrics provide a generalized framework for measuring distances in an n-dimensional space, encompassing well-known special cases such as the *Manhattan*, *Euclidean*, and *Chebyshev* distances [15].

Formally, the Minkowski distance between two vectors:

$$X \in \mathbb{R}^n = (x_1, x_2, \ldots, x_n), \quad Y \in \mathbb{R}^n = (y_1, y_2, \ldots, y_n)$$

is defined as:

$$L_p[X, Y] = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

This formulation adapts to different geometric interpretations depending on the value of p [22]:

1. **For** $p = 1$ — Corresponds to the *Manhattan distance*:

$$L_1[X, Y] = \sum_{i=1}^{n} |x_i - y_i|$$

2. **For** $p = 2$ — Corresponds to the *Euclidean distance*:

$$L_2[X, Y] = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

3. **For** $p \to \infty$ — Corresponds to the *Chebyshev distance*:

$$L_\infty[X, Y] = \max_{1 \leq i \leq n} |x_i - y_i|$$

Intermediate values of p provide smooth transitions between these geometric interpretations, making the Minkowski distance highly adaptable for applications in clustering, similarity search, and high-dimensional data analysis [2].

2.3.2 Euclidean Distance

The Euclidean distance is one of the most widely used similarity measures in data analysis, machine learning, and indexing structures. It represents the shortest straight-line distance between two points in a multidimensional space, and it is derived from the classical geometry of Euclidean space [26].

Given two data points $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$, the Euclidean distance is defined as:

$$d(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

This metric has been found to be particularly suitable for continuous numerical data, such as sensor readings in IoT environments, where each dimension represents a physical measurement (e.g. temperature, humidity, or pressure). Its geometric interpretation renders it intuitive and straightforward to apply in a multitude of real-world scenarios [37].

However, it should be noted that the Euclidean distance is subject to certain limitations. In high-dimensional spaces, the model's discriminative power is known to decrease due to the so-called "curse of dimensionality" , whereby distances between points tend to become similar. Furthermore, the square and square root operations increase computational cost, which may impact performance in large-scale or real-time systems such as IoT streaming environments.

Notwithstanding these limitations, the Euclidean distance remains a fundamental metric and is frequently employed as a baseline in similarity search and kNN-based applications.

2.3.3 Manhattan Distance

The Manhattan distance, also referred to as the L1 distance, measures the distance between two points as the sum of the absolute differences of their coordinates. It is called "Manhattan" distance because it reflects the distance traveled along grid-like paths, similar to navigating city blocks [26].

Formally, the Manhattan distance between two points X and Y is defined as:

$$d(X, Y) = \sum_{i=1}^{n} |x_i - y_i|$$

Compared to the Euclidean distance, the Manhattan distance is computationally simpler because it avoids square and square root operations. This makes it more efficient for large-scale datasets and streaming environments where performance is critical.

Another important advantage is its robustness in high-dimensional spaces. Unlike Euclidean distance, which can be affected by large deviations in certain dimensions, the Manhattan distance treats each dimension independently and reduces the influence of extreme values [37].

Due to these properties, the Manhattan distance is often used in applications involving sparse data, discrete attributes, or situations where computational efficiency is more important than geometric accuracy.

2.3.4 Chebyshev Distance

The Chebyshev distance, also known as the L_∞ distance, measures the distance between two points as the maximum absolute difference along any dimension. In other words, it considers only the largest deviation between corresponding components of the two points [26].

It is defined as:

$$d(X, Y) = \max_i |x_i - y_i|$$

This metric is particularly useful in scenarios where the overall distance is determined by the most significant difference in any single dimension. For example, in monitoring systems, a large deviation in one sensor reading may be more critical than small variations across multiple sensors.

The Chebyshev distance is computationally efficient, as it requires only comparison operations rather than summations or square roots. It is commonly used in grid-based

environments, image processing, and applications where maximum tolerance thresholds are important [37].

However, its main limitation is that it ignores the contributions of other dimensions, which may lead to loss of information in certain contexts. Therefore, it is typically used in specific applications where the maximum deviation is the primary concern.

2.4 Concepts of Ball and Hyperplane

Partitioning is a key concept in designing storage and indexing structures. It involves dividing the search space into smaller, easier-to-manage parts. This means that when a search is made, only the information that is needed is looked at. This makes the search faster [66, 16].

In metric spaces, data objects are not associated with explicit coordinates, which makes traditional geometric partitioning methods unsuitable [25]. To do this, a representative object is selected as a reference point. This object is called the *pivot*. The distance between each object and the pivot is calculated, and a threshold distance is chosen to divide the dataset into two subsets [72, 85].

There are two common ways to divide up metric spaces: *ball partitioning* and *hyperplane partitioning* [25, 16, 88].

2.4.1 Ball Partitioning

A *ball* generalizes the concept of a disc in the Euclidean plane or a sphere in three-dimensional space to any metric space [88]. It is defined by a pivot object p and a radius r, which together determine the boundary within which objects are grouped [72, 85].

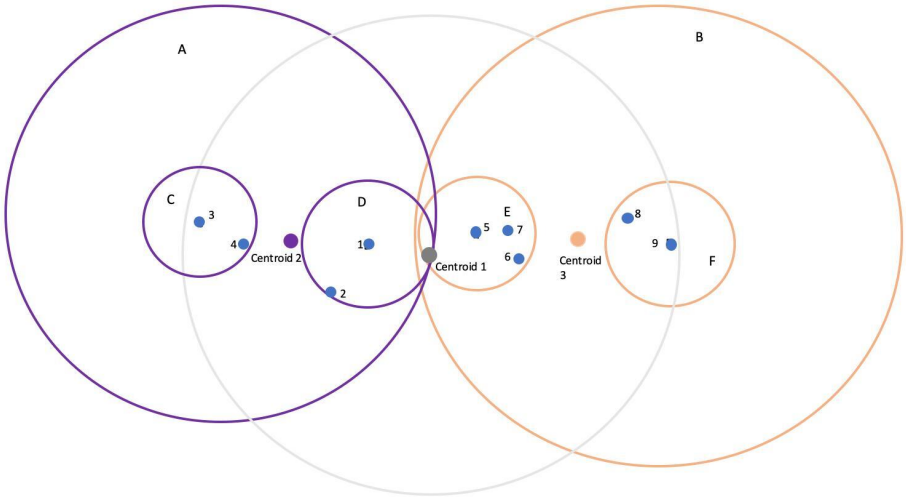


Figure 7: Ball partitioning using hyperspherical regions

Formally, let $M = (O, d)$ be a metric space, where O is the set of objects and d is the distance function. A ball, denoted as $\text{Ball}(p, r)$, is defined as:

$$S_1 = \{o \in O \mid d(p, o) \leq r\} \tag{1}$$

$$S_2 = \{o \in O \mid d(p, o) > r\} \tag{2}$$

Here, S_1 contains all objects within the radius r from the pivot p, while S_2 contains all objects outside that radius. The value $r$also referred to as the *covering radius*is often chosen to balance the two subsets [25]. When the pivot lies near the median distance, objects located exactly at the median can be assigned to either subset in a balanced yet arbitrary manner to maintain symmetry [66].

Ball partitioning is widely used in similarity search and indexing structures for high-dimensional or coordinate-free data, as it allows efficient pruning of irrelevant subsets during query processing [16, 88].

2.4.2 Hyperplane Partitioning

The concept of *hyperplane partitioning* is employed to divide the set of objects O into two subsets using two distinct pivots p_1 and p_2, chosen arbitrarily, such that $d(p_1, p_2) > 0$ [85, 72]. Once the pivots are selected, each object $o \in O$ is assigned to either subset S_1 or S_2 according to its relative distances to the pivots [25].

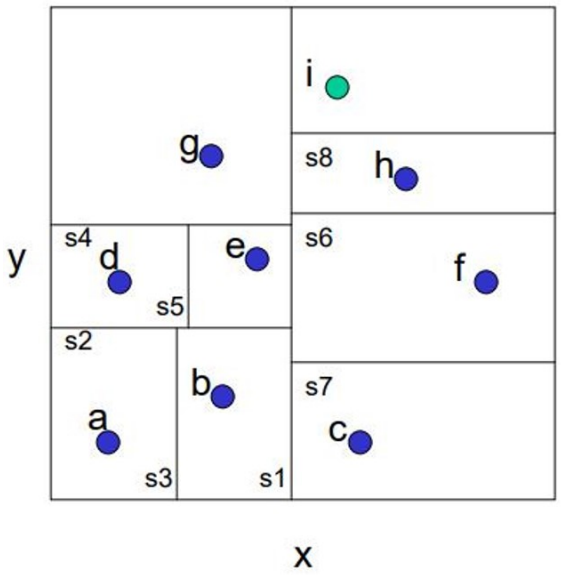


Figure 8: Hyperplane partitioning using axis-aligned splits

Generalized Hyperplane Partitioning : Let $M = (O, d)$ be a metric space, where O is the set of objects and d is the distance function. Let $(p_1, p_2) \in O^2$ be two pivot objects with $d(p_1, p_2) > 0$. The generalized hyperplane $H(p_1, p_2)$ partitions the space into two subsets as follows:

$$S_1 = \{o \in O \mid d(p_1, o) \leq d(p_2, o)\}$$

$$S_2 = \{o \in O \mid d(p_1, o) > d(p_2, o)\}$$

The generalized hyperplane method has the advantage of eliminating overlap between data, ensuring that no objects are shared between the subsets [88]. However, unlike the ball partitioning approach, it does not guarantee a balanced distribution of

objects [16]. Therefore, selecting well-adapted pivots to achieve balance remains an important and challenging research problem [66].

Comparison:

Ball partitioning groups data based on distance similarity, forming hyperspherical regions, while hyperplane partitioning divides the space using axis-aligned splits. As demonstrated in Figures 7 and 8, the former adapts more effectively to the intrinsic structure of the data, whereas the latter imposes rigid geometric partitions.

In the context of this work, the proposed indexing approach is closer to ball partitioning, since it relies on distance computations and metric-tree structures rather than axis-aligned splits.

3 IoT Data Indexing Methods

3.1 Overview of Big IoT Data and Indexing Challenges

The rapid proliferation of the Internet of Things (IoT) has given rise to a substantial generation of heterogeneous and incessant data. The data sets under consideration manifest the well-known characteristics of big data, namely volume, velocity and variety. Consequently, the storage and processing of these data sets is highly challenging. Recent studies have indicated that IoT systems continuously generate spatio-temporal data, necessitating efficient management and fast retrieval mechanisms [75].

It is evident that traditional cloud-based architectures are constrained in their capacity to manage such data, a limitation that can be attributed to the inherent high latency and bandwidth constraints of such architectures. The advent of fog computing has been instrumental in addressing the aforementioned issues by establishing an intermediate layer that facilitates processing operations in closer proximity to the data sources. This development has been shown to result in a substantial reduction in latency and enhancement of response times, as evidenced by numerous studies in the field [77].Furthermore, the necessity for immediate decision-making in real-time IoT applications, such as healthcare monitoring and industrial automation, underscores the imperative for efficient indexing techniques.

3.2 Taxonomy of IoT Indexing Methods

IoT indexing techniques can be categorized into four main groups: traditional, metric-based, stream-based, and clustering-based approaches.

3.2.1 Traditional Indexing Methods

Traditional indexing methods, such as B-tree and R-tree, were initially developed for static and disk-based database systems. These structures rely on hierarchical organization and data partitioning to facilitate efficient search operations. The B-tree organizes data in a balanced manner based on key values, making it suitable for exact-match queries, while the R-tree is designed for spatial data indexing by grouping nearby objects into minimum bounding rectangles.

Despite the evident efficacy of these methodologies in traditional database environments, their applicability to the management of data streams generated by the Internet of Things (IoT) remains restricted. A primary challenge is the inherent dynamism, continuity, and high dimensionality of IoT data. It is evident that conventional indexing structures are not engineered to accommodate frequent updates and continuous insertions, which are prevalent in streaming scenarios. Consequently, the maintenance of the index structure becomes computationally expensive over time.

Furthermore, spatial indexing methods such as R-tree are susceptible to the problem of overlapping regions. This overlap gives rise to multiple traversal paths during query processing, which increases the search cost and reduces efficiency. This issue becomes more pronounced as the dimensionality of the data increases, which is often the case in IoT applications involving multiple sensors and attributes.

A further limitation is scalability. It is evident that conventional indexing methodologies were not conceptualised to function within distributed environments, such as fog or edge computing architectures. The conventional assumption is that of centralized storage, a system which introduces latency and bandwidth issues when dealing with geographically distributed IoT devices.

Therefore, while traditional indexing methods provide a foundational understanding of hierarchical data organization, they are not well-suited for real-time IoT data streams. These limitations motivate the adoption of more advanced approaches, such as metric-based and clustering-based indexing, which are better adapted to dynamic, high-volume, and distributed data environments.

3.2.2 Metric-Based Indexing Methods

Metric-based indexing structures, such as M-tree and VP-tree, are widely used for similarity search in IoT environments because they organize data according to a distance function rather than a fixed spatial representation. This makes them more flexible than traditional indexing methods when the data are heterogeneous or multi-dimensional. In these structures, the triangle inequality is exploited to eliminate irrelevant regions during search, which reduces the number of distance computations required to answer kNN queries.

The M-tree is a representative metric access method that organizes objects around routing entries and uses distance bounds to prune subtrees during similarity search. Similarly, the VP-tree partitions the space around selected pivots and recursively separates objects according to their distance from these pivots. Both approaches are useful when the main objective is to accelerate nearest-neighbor retrieval in a metric space, but their performance depends strongly on the distribution of the data and the quality of the partitioning strategy.

Recent research has proposed more specialized metric structures such as the Binary Hyperplane Tree (BH-tree), which partitions the data space using two pivots and a hyperplane-like separation rule. In the fog-cloud method described in [46], BH-tree construction is combined with clustering in order to reduce search time and support parallel kNN processing. The paper shows that organizing data into clusters before indexing can improve pruning efficiency and reduce the cost of similarity search in streaming IoT settings. This idea is relevant to the present thesis because it confirms

that metric-based indexing becomes more effective when it is applied to compact and homogeneous groups of data rather than to the full raw stream.

A recurring issue in tree-based metric indexing is overlap between regions or subtrees. When overlap is high, the search algorithm must explore several branches, which increases the number of visited nodes and reduces pruning efficiency. For this reason, recent studies have emphasized overlap optimization as an important factor in improving kNN performance in IoT systems. In practice, the effectiveness of metric indexing depends not only on the choice of tree structure, but also on how well the data are grouped before tree construction and how tightly the nodes are organized.

Despite their advantages, metric-based indexing methods are not fully sufficient for continuous IoT streams. They may degrade when the data distribution changes over time, and they usually require additional mechanisms to remain efficient in dynamic environments. This limitation explains why metric trees are often combined with clustering or distributed processing in fog-based systems. In this thesis, this observation motivates the use of incremental clustering before indexing, so that the metric tree is built over compact clusters instead of the raw stream.

3.2.3 Stream-Based Indexing Approaches

Stream-based methods are specifically designed to process continuous and evolving data, making them suitable for IoT environments where data arrive in real time [14, 53]. These approaches operate incrementally, processing each data point as it arrives while maintaining a compact summary of the stream.

CluStream CluStream is one of the earliest and most widely used stream clustering algorithms [1]. It follows a two-phase model consisting of an online phase and an offline phase. In the online phase, incoming data are summarized into micro-clusters, which store statistical information such as the number of points, linear sum, and squared sum. In the offline phase, these micro-clusters are used to generate final clusters using traditional clustering methods such as k-means.

This approach allows efficient handling of large data streams with limited memory. However, CluStream assumes spherical cluster shapes and may not perform well when clusters have irregular structures [6].

DenStream DenStream extends the idea of micro-clusters by introducing a density-based approach [20]. It maintains two types of micro-clusters: potential clusters and outlier clusters. A fading function is used to reduce the importance of older data over time, allowing the model to adapt to evolving data distributions.

Compared to CluStream, DenStream is more robust to noise and can detect clusters of arbitrary shapes [53]. However, it requires careful parameter tuning and may incur additional computational cost due to the maintenance of outlier structures [6].

ClusTree ClusTree organizes micro-clusters in a hierarchical tree structure, enabling fast insertion and retrieval operations [49]. It is designed to process data streams in a single pass and can dynamically adapt to concept drift. The tree structure ensures that the clustering model remains up-to-date and balanced over time.

This approach improves update efficiency and adaptability, but it still focuses on data summarization rather than precise query processing [6].

Limitations for kNN Query Processing Despite their effectiveness in summarizing streaming data, these methods are not optimized for direct similarity search. They represent data using aggregated summaries (micro-clusters), which may lead to a loss of fine-grained information. As a result, performing exact kNN queries on raw data points is not straightforward.

Moreover, these approaches do not provide explicit indexing structures designed for efficient nearest-neighbor search. Therefore, additional indexing mechanisms are required to support fast query processing, which may introduce extra complexity.

Discussion In summary, stream-based methods are well-suited for handling high-velocity IoT data and capturing evolving patterns. However, their primary objective is data summarization rather than efficient query execution. This limitation motivates the integration of clustering with dedicated indexing structures, as proposed in this thesis, in order to support efficient kNN search in streaming environments.

3.2.4 Clustering-Based Indexing Approaches

Clustering-based indexing approaches aim to reduce the search space by grouping similar data points into compact clusters before applying indexing techniques. This strategy is particularly effective in IoT environments, where data are generated continuously and in large volumes. By operating on clusters instead of individual data points, the system can significantly reduce computational cost and improve query efficiency.

Clustering for Data Reduction One of the main motivations behind clustering-based indexing is data reduction. In streaming IoT scenarios, indexing every incoming data point is computationally expensive and often unnecessary. Clustering allows the system to represent groups of similar data points using compact summaries, which reduces both memory usage and search complexity.

This reduction directly impacts kNN query processing, as the number of candidate points to be examined is significantly decreased. Instead of searching the entire dataset, the query can be restricted to relevant clusters, improving efficiency.

The BIRCH Algorithm The BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm is one of the most widely used clustering methods for large-scale and streaming data. It incrementally builds a Clustering Feature (CF) tree, where each node summarizes a set of data points using statistical information such as the number of points, linear sum, and squared sum [89].

Integration with Indexing Structures Recent studies have shown that combining clustering with indexing can significantly improve query performance. Instead of building a single global index, clusters can be indexed independently, leading to smaller and more efficient index structures. This approach reduces the depth of trees, limits unnecessary traversal, and improves pruning efficiency during kNN search [76].

In fog-based architectures, clusters can also be distributed across multiple nodes, enabling parallel index construction and query execution. This further enhances scalability and reduces response time.

Comparison with Density-Based Clustering Alternative clustering methods, such as DBSCAN, are also used in IoT environments. Unlike BIRCH, DBSCAN groups data based on density and can detect clusters of arbitrary shapes. However, it is generally more computationally expensive and less suitable for continuous data streams.

In contrast, BIRCH provides a more scalable and incremental solution, making it better adapted to streaming scenarios where data arrive continuously.

Limitations Despite its advantages, clustering-based indexing also has limitations. The performance depends heavily on parameter selection, such as clustering thresholds and minimum cluster size. Poor parameter choices may lead to unbalanced clusters or loss of important information.

Additionally, clustering introduces an abstraction layer that may reduce the precision of query results, especially when clusters are too coarse. This trade-off between efficiency and accuracy must be carefully managed.

Discussion In summary, clustering-based indexing provides an effective way to reduce the search space and improve query efficiency in IoT data streams. By combining data reduction with indexing, this approach addresses the limitations of both traditional and metric-based methods. This observation motivates the approach proposed in this thesis, which integrates incremental clustering with metric indexing to support efficient kNN query processing in fog environments.

3.3 Evolution toward Fog-Based Indexing Architectures

The limitations of centralized cloud processing have led to the adoption of fog-based indexing architectures. In such architectures, data processing is distributed across edge, fog, and cloud layers.

Fog computing enables real-time data processing by reducing latency and network congestion. It also allows parallel execution of queries across multiple nodes, improving scalability and performance [74].

Several approaches have been proposed to integrate indexing with fog computing. For instance, BCCF-tree structures use clustering-based partitioning to improve search efficiency in distributed environments [73]. Similarly, recent works propose hybrid clustering-indexing frameworks to optimize query execution in fog-cloud systems.

3.4 Data Reduction through Incremental Clustering

Given the continuous and high-volume nature of IoT data streams, indexing all incoming data points is computationally infeasible. Therefore, data reduction techniques play a critical role in improving system efficiency.

Incremental clustering algorithms, such as BIRCH and DBSCAN, group data into compact clusters, which are then indexed instead of individual data points. This significantly reduces memory usage and computational cost.

Studies show that clustering-based data reduction improves query performance and reduces energy consumption in fog environments [46].

3.5 Benchmarking Against Fog-Cloud BH-Tree Approaches

A relevant reference for positioning the proposed framework is the fogcloud indexing method based on DBSCAN clustering and BH-tree construction presented in [46]. In that approach, the system is organized into a multi-stage pipeline operating at the fog layer. First, the incoming data stream is partitioned into clusters using a density-based method (DBSCAN), which groups data points according to local density. Second, a decision mechanism based on the coefficient of variation determines whether new data should be integrated into existing structures or trigger the creation of new ones. Finally, the resulting clusters are indexed using BH-tree structures, enabling parallel kNN query execution across multiple fog nodes.

This design demonstrates the importance of combining clustering and indexing in fog environments, particularly for reducing the effective search space and improving query responsiveness. However, it also introduces additional complexity due to the use of density-based clustering and multi-level decision mechanisms, which may be sensitive to parameter tuning and data distribution.

In contrast, the approach proposed in this thesis adopts a simpler and more modular pipeline. Instead of DBSCAN, it relies on incremental BIRCH clustering to generate compact micro-clusters in a single pass over the data stream. Each cluster is then indexed independently using a binary metric tree, avoiding the need for complex integration rules between clusters and index structures.

The comparison between the two approaches is therefore conceptual rather than strictly numerical. Both aim to improve kNN query performance by reducing the search space through clustering and enabling distributed processing in the fog layer. However, the proposed method emphasizes structural simplicity, incremental processing, and easier interpretability of performance metrics.

From an evaluation perspective, the most relevant comparison criteria include the number of generated clusters, the number of indexing trees, the cost of distance computations, and query execution time. These indicators provide a consistent basis for assessing how effectively each approach reduces computational overhead while maintaining scalability in streaming IoT environments [89, 46].

3.6 Limitations of Existing Approaches

Despite the advances in IoT indexing techniques, several limitations remain:

- Traditional methods are not suitable for dynamic and high-dimensional data.
- Metric-based methods may suffer from performance degradation over time.
- Stream-based approaches focus on summarization rather than efficient query processing.

- Clustering-based methods are sensitive to parameter tuning.
- Distributed fog-based systems introduce challenges in synchronization and load balancing.

These limitations highlight the need for hybrid solutions that integrate multiple techniques.

3.7 Research Gap and Motivation

The analysis of existing methods reveals that most approaches address only specific aspects of the problem. Few solutions integrate clustering, metric indexing, and distributed processing in a unified framework.

Recent research emphasizes the importance of combining clustering and indexing within fog architectures to support efficient kNN queries over streaming data [46]. However, existing solutions still face challenges related to scalability, stability, and adaptability.

Therefore, there is a need for a hybrid indexing framework that integrates:

- Incremental clustering for data reduction,
- Metric-based indexing for efficient similarity search,
- Distributed processing in fog environments.

This gap motivates the proposed approach presented in this thesis.

4 Clustering Techniques for IoT Data

4.1 Clustering Definition

Clustering is the process of organizing a set of data objects into groups, called clusters, such that objects within the same cluster are more similar to each other than to those in other clusters, according to a specific similarity or distance measure [44, 43]. It is a core technique in unsupervised learning, enabling the discovery of inherent patterns and structures in datasets without prior labeling [15]. Clustering methods can be broadly categorized into partition-based, hierarchical, density-based, and model-based approaches, each suited to different data characteristics and application domains [37, 3].

4.2 Types of Clustering Algorithms

There are different types of clustering algorithms. They can be put into groups based on how they organise data [44, 43]. Each category has its own way of finding groups of things in the data. This could be by splitting the data space, building hierarchical structures, detecting dense regions, or using statistical and grid-based models [37, 3]. The next few sections explain the main types of clustering algorithms and what makes them different [13].

4.2.1 Partitioning Clustering Algorithms

Partitioning clustering algorithms divide a given dataset into k disjoint subsets, where each subset is treated as an individual cluster [43, 37]. The process is guided by an objective function that evaluates the quality of the partitioning. One of the most common objective functions tries to make the *sum of the squares* of the differences between points and the centre of their group as small as possible [51, 52]. This is formally defined as:

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \|p - m_i\|^2 \tag{3}$$

where p represents a data point belonging to cluster C_i, and m_i denotes the centroid (mean) of cluster C_i. The main goal is to produce compact and well-separated clusters by minimizing E [15].

Among the well-known partitioning methods are **k-means** and **Fuzzy C-Means (FCM)** .

K-Means Algorithm The k-means algorithm is one of the most widely used ways of dividing things up, because its simple and works well [43, 37]. The process starts by picking random points from the dataset to be the first cluster centres [52]. Each piece of data is then linked to the closest centroid based on a measurement of how far away they are (often the Euclidean distance) [51]. After assigning all points, the centroids are recalculated as the mean position of all points within each group. This process is repeated until the centroids are stable, meaning they dont change significantly between iterations [15].

However, the performance and quality of the clustering produced by k-means depends heavily on the choice of initial centroids [10]. Poor initialization may lead to suboptimal clustering solutions or convergence to local minima. To address this issue, the algorithm is often executed multiple times with different initializations, selecting the solution with the lowest error [43]. While this approach is manageable for small or medium-sized datasets, it becomes computationally expensive for large-scale data, such as Internet of Things (IoT) data streams [3].

Fuzzy C-Means (FCM) Algorithm Fuzzy C-Means (FCM) is a type of k-means algorithm that allows for more flexibility . Instead of forcing each piece of data to belong to only one group, FCM gives each piece a degree of membership to all groups, with values ranging from 0 to 1. This makes it softer and more flexible [39].

The FCM procedure starts by creating a fuzzy membership matrix. Each entry u_{ij} in the matrix shows how close point p_j is to cluster C_i. The cluster centroids are then calculated based on these membership values, and the objective function is defined to account for the fuzziness :

$$J_m = \sum_{i=1}^{k} \sum_{j=1}^{n} (u_{ij})^m \|p_j - m_i\|^2 \tag{4}$$

where $m > 1$ is the fuzzification parameter controlling the level of cluster fuzziness [65]. At each iteration, both the membership matrix and centroids are updated. The process continues until the change in the objective function J_m between iterations falls below a predefined stopping threshold.

By allowing overlapping clusters, FCM is particularly useful in applications where data points may not have clear boundaries [60]. However, similar to k-means, it can be computationally intensive for large datasets, and its performance depends on the choice of initial parameters and the fuzzification coefficient m [3].

4.2.2 Density-based Algorithms

Density-based clustering algorithms aim to discover clusters within a dataset by identifying regions of high data point density that are separated by regions of low density [28, 37]. The fundamental idea underlying these algorithms is that, for each object within a cluster, its neighborhooddefined by a radius parameter *eps*must contain at least a minimum number of points denoted by *MinPts* [28]. Based on this principle, data points are categorized into three types: *core points*, *border points*, and *noise points*. Core points are directly density-connected to other core points, thereby forming the backbone of clusters, while border points lie on the periphery of clusters and noise points are considered as outliers that do not belong to any cluster [43].

These algorithms are particularly powerful in detecting clusters of arbitrary shapes and in handling noise effectively [28]. Moreover, they operate in a single pass over the data, making them efficient for large datasets. Among the most notable density-based clustering algorithms are the **Density-Based Spatial Clustering of Applications with Noise (DBSCAN)** and its hierarchical extension **Ordering Points To Identify the Clustering Structure (OPTICS)** [28, 8].

Density-Based Spatial Clustering of Applications with Noise (DBSCAN): DBSCAN is a connectivity-based algorithm that categorizes points into three types: *core*, *border*, and *noise* [28]. The key parameters are:

- **eps**: the neighborhood radius threshold.
- **MinPts**: the minimum number of points required to form a dense region.

The algorithm proceeds as follows:

1. The density of a point x_i is defined as the number of points k_i within radius *eps* around x_i.
2. If $k_i > MinPts$, x_i is a core point.
3. Two points are directly connected if their distance is less than *eps*.
4. Two points are density-connected if they are connected through a chain of core points.
5. A border point has fewer than *MinPts* points within *eps* but lies within the neighborhood of a core point.

6. A noise point is any point that is neither a core nor a border point.

Using these definitions, DBSCAN identifies the transitive closure of density-connected points to form clusters [28].

Ordering Points To Identify the Clustering Structure (OPTICS): OPTICS is an extension of DBSCAN designed to address its limitations when clusters have varying densities [8, 28]. This algorithm computes an augmented ordering of points that captures the intrinsic clustering structure of the dataset at different density thresholds in a single pass. The resulting ordering can be represented as a hierarchical cluster tree, since higher-density clusters are always contained within lower-density ones [8]. OPTICS uses the same parameters as DBSCAN (*eps* and *MinPts*) and proceeds as follows:

1. Select a point and determine if it is a core point by checking whether it has at least *MinPts* points within *eps*.

2. For each core point c, create an edge to another point within the core distance of c.

3. Select connected sets of core points to form clusters, progressively extending them based on density reachability.

OPTICS is particularly useful when the dataset contains clusters with highly variable densities, as it can reveal the clustering structure without explicitly producing a partition for a fixed parameter setting [43].

4.2.3 Grid-Based Clustering Algorithms

Grid-based clustering algorithms operate by partitioning the data space into a finite number of non-overlapping cells that form a grid structure [37]. The clustering process is then performed on this grid rather than directly on the data points, which allows for significant improvements in computational efficiency, particularly for large datasets [13]. The advantage of this method is that the processing time is typically dependent on the number of cells, not the number of data points. Clusters are formed based on the density and distribution of data points within the grid cells [37]. To enhance clustering accuracy, grid-based methods can incorporate subspace clustering and hierarchical clustering strategies.

Prominent examples of grid-based clustering algorithms include:

- **STatistical INformation Grid-based method (STING)** [78]
- **CLustering In QUEst (CLIQUE)** [4]
- **Merging of Adaptive Intervals Approach to Spatial Data Mining (MAFIA)** [58]

STatistical INformation Grid-based method (STING) STING is a grid-based clustering algorithm particularly suitable for spatial databases [78, 37]. It employs a hierarchical grid structure where the data space is recursively divided into rectangular cells. Initially, spatial data is stored in cells at the top level, and each cell is subsequently divided into four equal child cells, each corresponding to a quadrant of the parent cell [78]. For each cell, statistical parameters such as mean, standard deviation, and distribution type are precomputed and stored. The algorithm estimates the probability of each cell being relevant to a query or cluster [13]. If a cell is deemed relevant, the same analysis is applied to its children. The clustering result is formed by identifying regions composed of adjacent relevant cells. This approach is computationally efficient because it does not require scanning all individual data points once the statistical summaries are computed [37].

CLustering In QUEst (CLIQUE) CLIQUE is a tool that organises data into groups based on its numerical values [4]. It uses a bottom-up approach to identify dense regions in subspaces of the data. The process starts by dividing each dimension into equal parts, which creates grid cells. A cell is considered *dense* if it contains more data points than a user-defined density threshold, denoted by τ [4]. The cells in the lower-dimensional subspaces are combined to identify the dense units in the higher-dimensional subspaces. The dense units are then joined together to form groups. This approach is efficient for finding groups in different parts of high-dimensional data without needing the user to specify the relevant dimensions in advance [37].

Merging of Adaptive Intervals Approach to Spatial Data Mining (MAFIA) MAFIA is an extension of the CLIQUE algorithm that improves clustering quality by employing adaptive cell intervals instead of fixed-width cells [58]. The algorithm constructs an adaptive grid by reading data blocks into memory and calculating histograms for each dimension. A *dominance factor* α is used to determine which bins are considered dense. Specifically, bins that are at least $r\alpha$ times denser than the average are marked as candidate dense units (CDUs) [58]. These CDUs are then recursively combined to form higher-dimensional dense units. Adjacent CDUs are merged to produce the final clusters. By adapting the cell sizes to the distribution of data, MAFIA reduces the number of empty or low-density cells and produces more accurate clustering results, especially in datasets with varying density [3].

4.2.4 Model-Based Clustering Algorithms

Model-based clustering algorithms assume that the dataset is generated from an underlying model, where each cluster corresponds to a particular data distribution or structural pattern. These methods aim to determine the best model parameters that best fit the observed data. Data points are connected in different ways. Some of these ways include statistics, ideas and groups.[31].

In general, two primary approaches exist in model-based clustering:

1. **Neural approach:** Based on artificial neural networks, where clusters are represented as neurons and the system learns through iterative adaptation of connection weights.

2. **Statistical approach:** Based on probabilistic models and statistical measures to represent and evaluate cluster structures.

Among these, two representative algorithms are discussed below. The Self-Organizing Map (SOM) [47] is presented as a neural approach, whereas the COBWEB algorithm [29] is presented as a statistical approach.

Self-Organizing Map (SOM) The Self-Organizing Map (SOM) [48] is one of the most widely used neural network-based clustering methods. It represents each cluster as a neuron, and the input data are also considered as neurons. The network is composed of two layers:

- An **input layer**, corresponding to the dimensions of the input vectors.
- An **output layer** (map), arranged typically in a two-dimensional grid where each unit corresponds to a neuron and is associated with an n-dimensional weight vector:
$$m = (m_1, m_2, \ldots, m_n)$$
where n is the dimensionality of the input space.

Initially, the weight vectors are randomly initialized. The SOM is trained iteratively in the following manner:

1. Randomly select an input vector $\mathbf{x}$ from the dataset.
2. Compute the distance between $\mathbf{x}$ and all weight vectors using a distance measure (commonly the Euclidean distance).
3. Identify the *Best Matching Unit* (BMU) whose weight vector is closest to $\mathbf{x}$.
4. Update the BMU's weight vector and those of its neighboring units to move them closer to $\mathbf{x}$ in the input space.

Through this competitive learning process [48], the SOM preserves the topological relationships between data points and enables visualization of high-dimensional data. Map units may also be grouped post-training to form larger, interpretable clusters.

Model-Based Clustering via COBWEB The COBWEB algorithm [30] represents a statistical approach to model-based clustering. It incrementally constructs a hierarchical clustering structure in the form of a classification tree, where each node corresponds to a cluster described by a probabilistic profile [32].

COBWEB operates as an *incremental learner*, meaning that upon insertion of a new data object, the classification tree is updated dynamically. This process can significantly alter the tree's structure, potentially reorganizing multiple nodes to better represent the updated dataset.

The core decision criterion in COBWEB is the *Category Utility* (CU) measure , defined as:
$$CU = \sum_k P(C_k) \sum_i \sum_j P(A_i = V_{ij} \mid C_k)^2$$
where:

- $P(C_k)$ is the probability of cluster C_k.
- A_i represents the i-th attribute.
- V_{ij} is the j-th possible value of attribute A_i.
- $P(A_i = V_{ij} \mid C_k)$ denotes the conditional probability of A_i taking the value V_{ij} given that the object belongs to cluster C_k.

The CU measure balances two competing objectives:

1. **Maximizing intra-class similarity:** Ensuring that members of the same cluster share similar characteristics.
2. **Maximizing inter-class dissimilarity:** Ensuring that different clusters are as distinct as possible.

COBWEB uses CU to decide whether to create a new cluster, merge existing clusters, split clusters, or insert a data object into an existing cluster. This makes it well-suited for streaming and evolving datasets where incremental updates are necessary

4.2.5 Hierarchical Clustering Algorithms

Hierarchical clustering is a clustering technique that organizes data into a nested hierarchy of clusters, allowing visualization through a dendrogram [44, 37]. This method is based on a *connectivity approach*, where clusters are formed according to the proximity or similarity between data objects, typically measured using a distance matrix [57]. The process can be constructed either *bottom-up* (agglomerative) or *top-down* (divisive), and proceeds step-by-step to refine the cluster structure [82].

In the **agglomerative** approach, each data object starts as an individual cluster. Pairs of clusters are successively merged based on a linkage criterion, such as *single-link* (minimum distance), *complete-link* (maximum distance), or *average-link* (mean distance between points) [70]. The merging continues until all points belong to a single cluster or until a desired number of clusters is reached.

Conversely, the **divisive** approach starts with all data objects in a single large cluster. The cluster is recursively split into smaller clusters according to a dissimilarity measure, until the desired number of clusters is obtained. This approach is less common than agglomerative clustering due to its higher computational complexity [43].

Well-known hierarchical clustering algorithms include:

- Clustering Using REpresentatives [35]
- Robust Clustering algorithm for Categorical attributes [36]
- Balanced Iterative Reducing and Clustering using Hierarchies[90]

Clustering Using REpresentatives CURE is an agglomerative hierarchical clustering algorithm that aims to overcome the limitations of centroid-based and all-points-based methods by using a set of well-dispersed representative points for each cluster [35]. The algorithm follows a divisive clustering strategy: initially, each data point is treated as a separate cluster. For each cluster, a set of representative points is selected and then *shrunk* toward the cluster centroid by a predefined shrinking factor [35]. During each iteration, two clusters with the closest pair of representative points are merged, and the process continues until the target number of clusters is achieved. This strategy allows CURE to handle clusters with non-spherical shapes and varying sizes more effectively [3].

Robust Clustering Algorithm for Categorical Attributes ROCK is an agglomerative hierarchical clustering algorithm specifically designed for categorical data [36, 37]. Unlike numerical distance measures, ROCK introduces the concept of *links*a measure of similarity based on the number of shared neighbors between points [36]. The algorithm begins with each data point as an individual cluster and computes the link values between all pairs of clusters. A heap structure is maintained to store and update these link values. In each iteration, the pair of clusters with the highest value of a predefined criterion function (based on links) is merged. This process continues until the desired number of clusters is formed [43]. ROCK is particularly effective for clustering categorical data where distance metrics may not be meaningful [3].

4.3 Use of Clustering in Fog-Based Indexing

Fog computing is a way of storing data that sits between cloud data centres and IoT devices. It provides low-latency services and processes data close to where it is collected [18, 84]. But the huge amount of different IoT data created at the edge needs good ways to organise it so that it can be found and used quickly [12]. Clustering algorithms are very important in this situation. They help to group similar data together and reduce redundancy [43].

By using clustering techniques, fog nodes can sort incoming IoT data into meaningful groups before storage, improving retrieval efficiency. For example, k-means partitions data based on similarity, while DBSCAN detects outliers and removes noise before indexing [51, 28]. This step reduces cloud workload and ensures that only relevant, structured information is transmitted.

Clustering also helps to distribute data evenly across fog nodes, enabling efficient workload balancing. This improves system performance in terms of latency, bandwidth usage, and energy efficiency, which are critical in IoT environments. Therefore, integrating clustering into fog-based indexing enhances data management and supports scalability in dynamic IoT ecosystems.

Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) BIRCH is an incremental hierarchical clustering algorithm specifically designed to handle large-scale and continuously evolving datasets. Unlike traditional clustering approaches that require multiple scans over the entire dataset, BIRCH constructs a

compact and memory-efficient representation of the data through a structure known as the Clustering Feature (CF) tree.

The core idea of BIRCH lies in summarizing the dataset using clustering features rather than storing individual data points. Each node in the CF-tree contains a set of CF entries, where each entry is defined by a triplet (N, LS, SS) representing the number of points, the linear sum, and the squared sum of the data points within a cluster. This compact representation allows the algorithm to incrementally absorb new data while preserving essential statistical properties.

The construction of the CF-tree is performed in a single pass over the data. As each new data point arrives, it is inserted into the closest leaf entry based on a distance threshold. If the insertion violates the threshold constraint, the tree dynamically grows by splitting nodes, ensuring that the structure remains balanced and scalable. This incremental behavior makes BIRCH particularly suitable for streaming environments where data arrives continuously.

In subsequent phases, the CF-tree can be optionally rebuilt using a larger threshold to further condense the data. Finally, a global clustering algorithm, such as k-means, can be applied to the leaf entries to produce the final clusters. Additional refinement steps may reassign data points to their nearest centroids to improve clustering quality. [90].

The importance of BIRCH in this work extends beyond its clustering capability. In the context of the proposed indexing framework presented in Chapter 3, BIRCH plays a critical role as a data reduction mechanism. Instead of indexing raw IoT data streams, which would be computationally expensive and inefficient, the system first applies incremental BIRCH clustering to group incoming data into compact and homogeneous micro-clusters.

These micro-clusters serve as the fundamental units for subsequent processing. After the clustering phase, each cluster is treated as an independent subset of the data and is used to construct a local metric indexing structure. In other words, BIRCH transforms the continuous data stream into a set of structured partitions that can be efficiently indexed and distributed across fog nodes.

This design choice directly impacts the efficiency of the kNN query processing stage. By reducing the number of indexed elements and organizing the data into coherent clusters, the search space is significantly minimized. Consequently, the number of distance computations, visited nodes, and overall query execution time are reduced.

Therefore, BIRCH is not only used as a clustering algorithm in this thesis but also as a foundational component that enables scalable indexing and efficient similarity search in fog-based IoT environments.

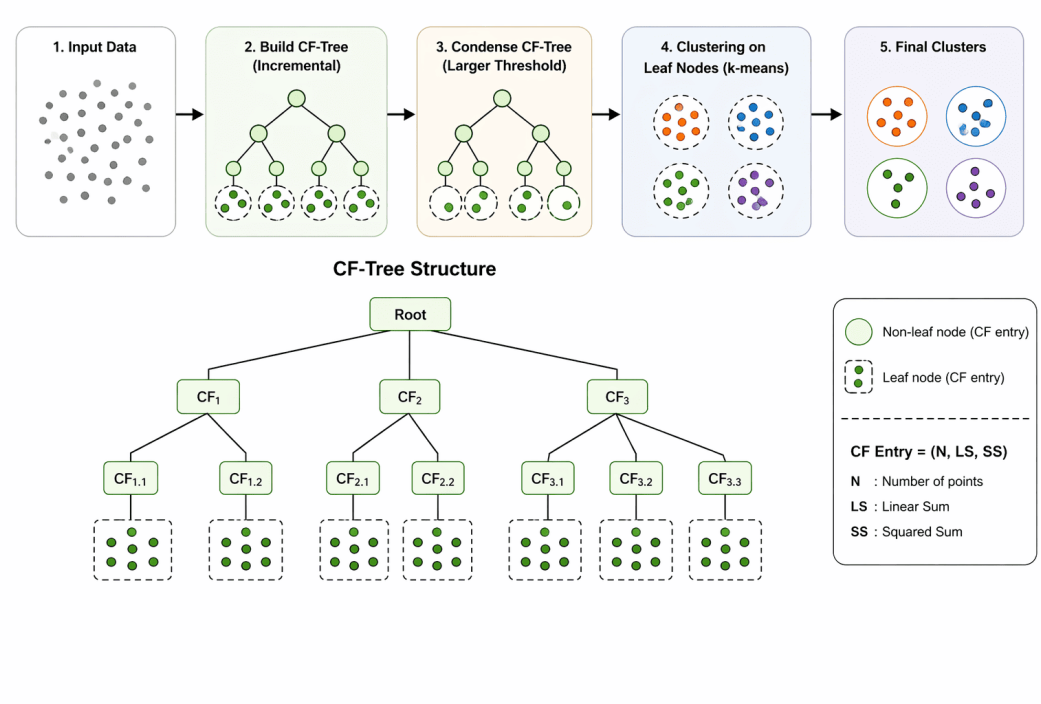


Figure 9: Overview of the BIRCH clustering process and CF-tree structure

5 Advantages of Continuous IoT Data Streams

IoT data streams are now a key part of modern intelligent systems, especially in places where sensors and devices produce data all the time. Traditional data processing systems work in batches, which means they wait to process data until there is enough data to fill a batch. Streaming systems, on the other hand, process data as soon as it is generated. They don't need to store data temporarily or analyse it offline. This feature is very important for making IoT-based infrastructures more responsive and adaptable. [34, 12].

One of the best things about having data that is always changing is that you can process it as it happens. In this case, data are analysed as soon as they arrive, which means that systems can react to events as they happen. This is very important in situations where delays can have serious consequences. For example, in smart healthcare systems, checking patients' vital signs regularly can spot problems early on and help to solve them straight away. In industry, real-time data processing helps to predict when machines will fail by identifying any unusual behaviour. These capabilities show how important streaming models are for making decisions quickly. [71, 5].

Another important advantage is that streaming systems can be scaled up or down as needed. IoT environments are full of devices that generate large amounts of data very quickly. Traditional systems often find it hard to deal with such large amounts of data. On the other hand, stream-processing frameworks are designed to manage continuous flows of data with minimal delay, enabling the system to expand as the number of data sources increases. This property is especially important in large-scale projects, like smart cities, where thousands of sensors are used at the same time. [54].

You can also use continuous IoT data streams to do predictive analytics. By collect-

ing and analysing data over time, systems can spot patterns, trends and connections that aren't visible in static datasets. This means we can plan for things like equipment failures, traffic jams, or changes to the environment. In the Industrial IoT and environmental monitoring, it's really useful to be able to predict things. This can save money and make systems more reliable [3].

As well as helping to predict things, streaming data helps to manage resources efficiently. As data are processed bit by bit, systems can change how they use their resources based on how much work they have at the moment. For example, we can change how much computing power, network speed and storage space we use in real time to match the amount of data coming in. This clever use of resources means that fewer are used and that the system as a whole works better. This is really important in fog and edge computing environments where resources are limited. [68].

Another key benefit of continuous data streams is their ability to support systems that adapt and are aware of context. IoT systems can adjust their behaviour in response to changing conditions by continuously observing their environment. This means that we can develop smart services that are exactly what the user needs and that fit in with the environment. For example, smart homes can automatically change the temperature and lighting based on whether anyone is home and what they like. Intelligent transportation systems can also change traffic signals based on real-time traffic conditions. [87, 67].

But there are also some problems with this. The need for real-time processing puts strict limits on how long things take to process and how efficiently they are processed. Also, because data moves so fast and in such large amounts, it is important to have good ways to organise and find it, otherwise performance will get worse. These challenges show that we need better ways to manage data, like putting similar data together and using lots of computers to store information in fog environments, as we suggest in this thesis.

In short, IoT data streams have lots of advantages. These include being able to respond straight away, being able to grow, being able to predict what will happen, using resources efficiently and being able to adapt. These properties make streaming so important for modern IoT systems, especially for applications that need low latency and continuous monitoring. At the same time, they show how important it is to design efficient ways of collecting and processing information that can be used when there is a limit on the amount of data that can be collected. This is the main focus of this work.

6 Conclusion

indexing and clustering methods emerge as complementary and indispensable tools for structuring, retrieving, and analyzing IoT data in both cloud and fog computing environments. They form a crucial foundation for achieving scalability, adaptability, and efficiency in IoT systems.

Chapter 3

Proposed indexing method

1 Introduction

This chapter presents the implementation details of the proposed system for processing IoT streaming data and executing kNN queries. The implementation is directly derived from the developed source code, focusing on the integration of streaming ingestion, incremental clustering, distributed indexing, and parallel query processing. The objective is to transform continuous sensor data into structured clusters, index them efficiently and support fast similarity queries.

2 System Architecture Overview

The implemented system follows a pipeline architecture composed of sequential processing stages. The execution flow is explicitly defined in the main pipeline, where each component is instantiated and connected. The pipeline consists of the following stages:

- Raw IoT records are received through the `EdgeGateway` component, where the `receive()` function converts incoming observations into numerical vectors and forwards them to the streaming layer.
- The `StreamBroker` component manages data transmission between ingestion and processing stages using the `publish()` and `consume()` functions.
- Streaming records are grouped into overlapping windows using the SlidingWindow module. The append() function stores records inside the internal buffer, while the to_matrix() function converts generated snapshots into numerical matrices.
- Each generated window is normalized using `StandardScaler` before clustering and similarity search operations.
- The normalized windows are distributed across multiple fog nodes through the `FogClusterManager` component using the `process_parallel()` function.
- Each `FogNode` executes incremental clustering using the `IncrementalBIRCH` model through the `update()` and `get_clusters()` functions.
- Small clusters are removed using the `MIN_CLUSTER_SIZE` threshold in order to reduce unstable partitions and unnecessary indexing overhead.
- The remaining clusters are indexed locally using metric trees constructed by the `build_tree()` function with adaptive pivot selection using `adaptive_pivot()`.
- Local similarity search is executed using the `MetricTreeKNN` class through the `search()` function.
- Candidate neighbors are maintained using the `KNNState` structure through the `try_add_squared()` and `worst_dist_sq()` functions.

- Distributed query execution is coordinated by the `ForestKNNParallel` component, where the `_select_candidate_trees()` function selects candidate trees and the `search()` function launches parallel kNN searches across distributed metric trees.
- The partial search results returned by fog nodes are merged using the merge_from() function to produce the final global kNN result.
- Execution statistics and performance metrics are stored through the `CloudAnalytics` component using the `log()` function.

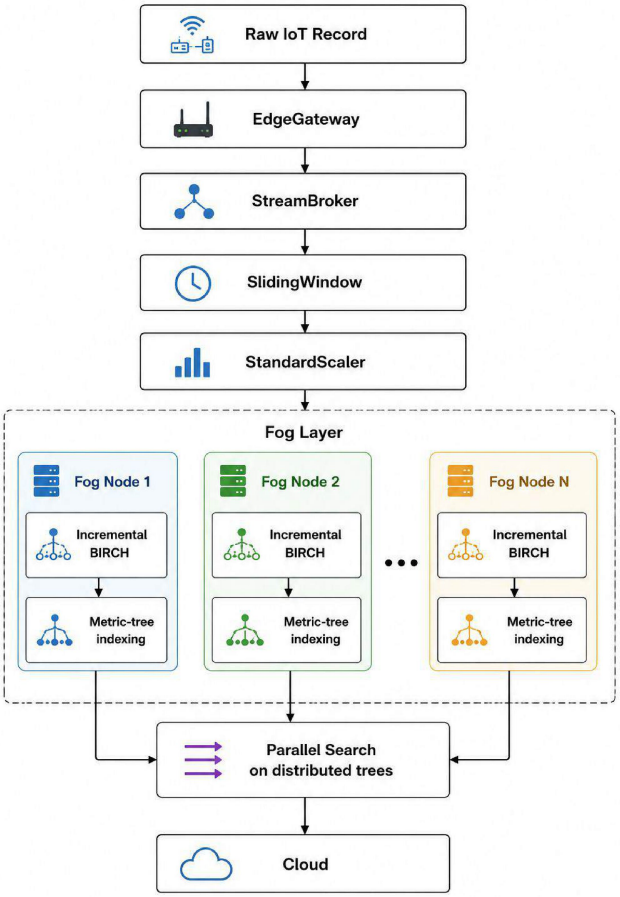


Figure 10: Streaming IoT Data Processing Pipeline Architecture

2.1 Data Ingestion Mechanism

The data ingestion process is implemented using two main components: the 'EdgeGateway' and the 'StreamBroker'. The gateway receives incoming data and immediately forwards it to the broker without applying any transformation or filtering.

The ‘StreamBroker‘ uses a queue-based mechanism to store incoming data. Each record is inserted using a ‘publish‘ operation and retrieved using a ‘consume‘ operation. This ensures a FIFO (First-In-First-Out) data flow.

2.2 Sliding Window Model

After ingestion, the stream is organised using a sliding window mechanism. This collects incoming records until the predefined size is reached. Once full, the buffer produces a snapshot and applies a step-size shift so that the window can advance through the stream.

This approach is well-suited to continuous IoT data, as it allows the system to process data incrementally rather than waiting for the entire dataset to become available. It also preserves temporal continuity, since consecutive windows may overlap depending on the selected step size. In the current implementation, the snapshot is extracted as a list and later converted into a matrix for numerical processing. Therefore, this stage can be described as a streaming segmentation mechanism rather than a fog-specific computation stage. Its role is to provide structured windows that can be efficiently passed to the fog layer for clustering.

2.3 Data Transformation

Before clustering, the windowed snapshot is transformed into a numerical matrix. The implementation first selects only numeric attributes from the input dataset and removes invalid values such as infinite or missing entries. This preprocessing step ensures that the stream contains only usable numerical vectors.

This detail is important because the clustering stage is sensitive to the scale of the input variables. Without standardization, attributes with larger numeric ranges could dominate the clustering process and distort the similarity relationships between records. For this reason, the transformation stage should be understood as a preprocessing step that prepares the data for fog-level clustering, even though it is executed within the pipeline before the clustering component itself. It belongs conceptually to the edge-fog boundary, but functionally it supports the fog layer.

3 Fog Node Design

Each FogNode contains a local clustering model, a collection of metric trees, a list of cluster centroids, and a profile centroid that summarizes the nodes local data distribution. This structure allows each fog node to act as an independent processing unit for indexing and search.

3.1 Cluster Distribution Strategy

The FogClusterManager partitions the current window and distributes the resulting clusters to the available fog nodes. The distribution strategy combines cluster-to-node similarity with load balancing so that clusters are assigned to the most appropriate fog node while maintaining a balanced execution load across the distributed environment.

3.2 Distributed Execution Model

The proposed architecture adopts a distributed execution model designed to support continuous IoT stream processing within a fog-computing environment. Instead of relying on centralized cloud-side computation, the processing workflow is distributed across multiple fog nodes located closer to the data sources. This distributed organization enables scalable stream ingestion, localized index construction, and parallel similarity search while reducing communication overhead and query latency. The execution model is specifically designed to support continuous indexing and parallel kNN processing over dynamically evolving IoT data streams.

4 Incremental Clustering Implementation

4.1 Incremental BIRCH Model

The proposed architecture integrates an incremental clustering layer based on the BIRCH algorithm in order to continuously organize incoming IoT streams before distributed metric indexing. Unlike traditional batch-oriented clustering approaches, the clustering process operates incrementally on streaming windows and continuously adapts to evolving data distributions without rebuilding the clustering structure from scratch.

The IncrementalBIRCH component is initialized using the Birch model from the sklearn-cluster library with:

```
Birch(
    threshold=BIRCH_THRESHOLD,
    n_clusters=None
)
```

The `threshold` parameter controls cluster granularity and directly affects the compactness of the generated partitions. Consequently, it influences the number of distributed metric trees generated during the indexing stage and impacts the overall traversal complexity of the search space. Setting `n_clusters=None` preserves the fully incremental behavior of the clustering process and avoids imposing a fixed number of clusters on the evolving stream.

Instead of relying on the traditional `fit()` mechanism, the proposed implementation performs continuous updates using `partial_fit()`, allowing newly arrived streaming windows to be incrementally integrated into the existing CF-tree structure. This update strategy enables continuous stream adaptation while reducing recomputation overhead and preserving low-latency processing inside the fog layer.

After each incremental update, the internal clustering representation is transformed into explicit stream partitions using the `predict()` mechanism. Each data point receives a cluster label, and points sharing identical labels are regrouped into localized cluster representations through dictionary-based aggregation.

The generated clusters are not used as final analytical outputs, but rather as distributed stream partitions for localized metric-tree construction. This organization stage reduces search-space fragmentation and prepares the continuous stream for scalable distributed indexing and parallel kNN traversal.

Small and unstable micro-clusters are filtered using the MIN_CLUSTER_SIZE threshold before index construction. This filtering stage reduces unnecessary tree-generation overhead, prevents excessive fragmentation of the distributed indexing space, and improves traversal stability during query execution.

The implementation of the proposed incremental clustering component is presented in Listing X.

```
import numpy as np
from sklearn.cluster import Birch

class IncrementalBIRCH:
    def __init__(self, threshold=0.3):
        self.model = Birch(
            threshold=threshold,
            n_clusters=None)
        self.fitted = False

    def update(self, data):
        data = np.asarray(data, dtype=float)
        if data.size == 0:
            return
        self.model.partial_fit(data)
        self.fitted = True

    def get_clusters(self, data):
        data = np.asarray(data, dtype=float)
        if not self.fitted or len(data) == 0:
            return []
        labels = self.model.predict(data)
        clusters = {}
        for i, label in enumerate(labels):
            clusters.setdefault(label, []).append(data[i])
        return list(clusters.values())
    @staticmethod
    def centroids(clusters):
        return [
            np.mean(np.asarray(c, dtype=float), axis=0)
            for c in clusters
            if len(c) > 0]
```

4.2 Cluster Extraction Strategy

The cluster extraction stage converts the internal BIRCH representation into explicit searchable partitions suitable for distributed metric-tree indexing. After the prediction

phase, data points assigned to identical labels are regrouped into localized cluster structures representing coherent regions of the streaming space.

This transformation produces independent cluster partitions that are subsequently distributed across fog nodes for local metric-tree construction. Each retained cluster becomes an independent indexing region participating in localized kNN traversal and distributed parallel search.

The extraction strategy therefore acts as an intermediate transformation layer between incremental stream organization and distributed metric indexing.

4.3 Cluster Filtering Mechanism

Not every cluster produced by BIRCH is kept for indexing. A filtering step removes clusters whose size is below a minimum threshold.

This filtering step is important for two reasons. First, it reduces the number of trees that must be created, which lowers memory and processing overhead. Second, it helps avoid indexing clusters that are too sparse to support effective pruning during search. In other words, the filtering phase improves the quality of the cluster set that will be forwarded to the distributed indexing stage. It is therefore a structural optimization step inside the fog layer, not a cloud-level task.

5 Metric Tree Index Construction

5.1 Index Construction

After the clustering and filtering stages, the remaining clusters are distributed across multiple fog nodes for local index construction. Instead of building a single centralized indexing structure, the proposed architecture adopts a distributed indexing strategy in which each fog node independently constructs its own set of metric trees from the clusters assigned to it. Parallel index construction is coordinated by the FogClusterManager component using ThreadPoolExecutor. The process-parallel() function partitions the streaming window into multiple cluster groups and dispatches them concurrently to the available fog nodes. Each node executes the complete local indexing pipeline independently, including cluster processing, metric-tree construction, and local structure initialization. This design transforms the indexing stage into a distributed parallel operation rather than a sequential centralized task. As a result, multiple metric trees can be constructed simultaneously across different fog nodes, significantly reducing the total index construction latency for large streaming windows. Inside each fog node, the assigned clusters are indexed using the recursive build-tree() function. Since each fog node processes only a subset of the global data, the resulting trees are smaller, shallower, and computationally less expensive to construct compared

to a global monolithic tree. This distributed partitioning also improves memory locality and reduces the probability of structural degradation caused by continuously growing IoT streams. The parallel indexing strategy further improves scalability because the workload is distributed among several fog nodes instead of relying on a single processing unit. As the volume of streaming IoT data increases, additional fog nodes

can be integrated into the architecture to absorb the indexing workload without modifying the overall processing pipeline. Moreover, distributing the index structures across multiple fog nodes naturally supports the next stage of the system, namely parallel kNN search over distributed trees. Since each fog node already maintains its own local metric trees, similarity queries can later be executed concurrently across several distributed indexes, reducing query latency and minimizing the explored search space. Another important advantage of the proposed distributed indexing approach is fault isolation. Since each fog node maintains independent local trees, failures affecting one node do not invalidate the entire indexing system. The remaining fog nodes continue operating and answering local similarity queries independently, which improves the robustness of the overall fog-based architecture. Therefore, parallel index construction constitutes a fundamental component of the proposed distributed fog processing pipeline, enabling scalable, low-latency, and adaptive indexing for continuous IoT data streams.

5.2 Tree Structure Definition

The indexing structure is implemented as a binary metric tree. Each node contains two pivot points (‘pivot1‘ and ‘pivot2‘), references to left and right child nodes, and a set of points if the node is a leaf.

Additionally, leaf nodes store a NumPy representation of their points (‘points-np‘) to support efficient vectorized computations.

5.3 Recursive Tree Construction

The tree is constructed recursively using the build-tree function. The construction process stops when the number of points in a cluster is less than or equal to a predefined leaf size.

At each step, the dataset is partitioned into two subsets based on the distance to the pivots. This results in a hierarchical structure where each node represents a partition of the data.

5.4 Pivot Selection Strategy

The tree construction uses an adaptive pivot selection strategy. The variance of the cluster is computed, and the selection method changes according to the dispersion of the points. If the variance is high, the system uses a farthest-point heuristic; otherwise, it uses random selection. This mechanism remains in the fog layer because it is part of local index construction.

The purpose of this strategy is to improve the quality of the split without making the construction phase overly expensive. Choosing pivots adaptively helps the tree create more meaningful partitions, which in turn improves pruning during search. The pivot strategy is therefore not a separate stage, but a local optimization embedded in the fog indexing process.

5.5 Degeneracy Handling

During tree construction, the algorithm may encounter cases where one of the partitions becomes empty or nearly empty. In such situations, the node is turned into a leaf to prevent infinite recursion. This is a necessary safeguard in the fog layer, because the data distribution inside a cluster may not always be favorable for a balanced binary split.

This degeneracy handling step ensures that the tree remains stable even when clusters are small, sparse, or unevenly distributed. Without this check, the recursive construction process could produce invalid or excessively deep trees. The code also stores the squared radii of the subtrees for pruning during search, although the main role of these values is revealed in the query stage.

6 kNN Query Processing

Similarity search in the proposed architecture is executed in a distributed manner across multiple fog nodes. Each `FogNode` performs local kNN search on its own metric-tree indexes before participating in the global distributed query execution process.

Local search operations are implemented using the `MetricTreeKNN` component. The `search()` function traverses the local metric trees iteratively using a stack-based depth-first traversal strategy. Instead of scanning all indexed points, the search process selectively explores only the relevant subtrees according to pivot distances and subtree covering radii.

Distance computations are performed using squared Euclidean distance in order to avoid repeated square-root operations during traversal. This optimization reduces computational overhead while preserving the relative ordering of distances required for nearest-neighbor evaluation.

The search process maintains candidate neighbors using the `KNNState` structure. The `try_add_squared()` function dynamically updates the top-k nearest neighbors stored in a heap-based structure, while `worst_dist_sq()` provides the current pruning threshold used during traversal.

Tree pruning is performed using lower-bound estimates derived from the distances between the query point and the node pivots together with the subtree covering radii `r1_sq` and `r2_sq`. Subtrees that cannot contain closer neighbors than the current candidates are skipped during traversal. This pruning mechanism significantly reduces the number of explored nodes, visited leaves, and computed distances.

When a leaf node is reached, all contained points are processed locally and evaluated against the current candidate set. The local nearest-neighbor candidates generated inside each fog node are later forwarded to the distributed search layer, where global result aggregation is performed across multiple distributed trees.

6.1 Search Algorithm on Metric Tree

The local kNN search algorithm is implemented through the `MetricTreeKNN.search()` function. The traversal process begins from the root node and proceeds iteratively using a stack-based depth-first strategy.

For internal nodes, the algorithm computes the distances between the query point and the pivots `pivot1` and `pivot2`. These distances are then used to estimate lower bounds for the left and right subtrees according to the corresponding covering radii.

Based on these lower-bound estimates and the current pruning threshold returned by `worst_dist_sq()`, the algorithm determines whether each subtree should be explored or pruned. The subtree having the closest pivot to the query is explored first in order to improve early candidate refinement and strengthen pruning efficiency during traversal.

Unlike exhaustive search, this strategy avoids visiting irrelevant branches and reduces the search space explored inside each local metric tree.

6.2 Distance Computation

Distance evaluation inside the search procedure relies entirely on squared Euclidean distance. Using squared distances avoids repeated square-root operations while preserving the ordering required for nearest-neighbor comparison.

For internal nodes, distances are computed between the query point and the pivot points stored in each node. These distances are used to estimate subtree lower bounds and drive pruning decisions.

For leaf nodes, distance computation is vectorized using NumPy operations. Instead of evaluating points individually, all points contained in the leaf are processed simultaneously through matrix operations. This vectorized computation reduces traversal overhead and improves execution efficiency inside the fog environment.

The total number of computed distances is also recorded during execution and later used as a performance metric for evaluating pruning effectiveness and search efficiency.

6.3 Candidate Management

Candidate neighbors are managed through the `KNNState` structure. This structure maintains the current top-k nearest neighbors discovered during traversal using a heap-based priority queue.

The `try_add_squared()` function inserts or replaces candidate neighbors according to their squared distance values. Whenever a newly evaluated point is closer than the current worst candidate stored in the heap, the heap is updated dynamically.

The `worst_dist_sq()` function returns the largest distance currently stored among the top-k candidates. This value acts as a dynamic pruning threshold during traversal and directly influences subtree elimination decisions.

In addition to candidate maintenance, the `KNNState` structure also records execution statistics such as:

- the number of computed distances,
- the number of visited leaves,
- and the final top-k candidate neighbors.

These statistics are later used during experimental evaluation to analyze the efficiency of pruning and distributed search execution.

6.4 Leaf Processing

When traversal reaches a leaf node, all points stored inside the leaf are evaluated locally against the query point. Since leaf nodes contain a limited number of points determined by the configured leaf size, exhaustive local evaluation remains computationally efficient.

The points are processed in vectorized batches using NumPy arrays, allowing simultaneous computation of multiple distances instead of sequential point-by-point evaluation. The resulting distances are then forwarded to the `try_add_squared()` function to update the current candidate set.

This local leaf processing strategy minimizes traversal overhead while preserving exact nearest-neighbor evaluation inside each metric tree. Combined with subtree pruning, it enables efficient local kNN execution before the distributed aggregation stage performed across multiple fog nodes.

7 Parallel kNN over Trees

The proposed architecture extends kNN search to a distributed fog environment where multiple metric trees are processed concurrently across several fog nodes. Instead of relying on a centralized global index, the system distributes the indexed clusters across independent `FogNode` instances, where each node maintains its own local metric-tree structures.

Distributed query execution is coordinated by the `ForestKNNParallel` component. The search procedure operates on a forest of distributed metric trees generated from the clustering stage. Each fog node independently performs local similarity search on its assigned trees before contributing partial results to the global query execution process.

The distributed search workflow is divided into three main stages:

- candidate fog-node and tree selection,
- parallel local tree traversal,
- distributed result aggregation.

This architecture reduces the explored search space, improves scalability, and minimizes query latency by avoiding exhaustive traversal of all distributed indexes.

7.1 Forest-Based Search

The distributed indexing structure is organized as a forest of independent metric trees distributed across multiple fog nodes. Each tree corresponds to a retained cluster generated during the incremental clustering stage and represents a localized search region inside the distributed fog environment.

Unlike centralized indexing approaches, the proposed architecture does not merge all trees into a single global structure. Instead, each FogNode maintains its own local collection of trees stored in the trees structure. This organization enables decentralized

query execution and improves scalability by partitioning the global search space into smaller independent local regions.

During query execution, local trees are traversed independently using `MetricTreeKNN`.

Each fog node performs local nearest-neighbor evaluation on its own distributed trees before forwarding partial candidates to the global aggregation stage.

This forest-based organization also improves fault isolation and distributed scalability. Since the trees are physically distributed across several fog nodes, the failure or overload of one node does not invalidate the entire indexing system.

7.2 Candidate Tree Selection

Before launching the distributed search process, the system reduces the search space by selecting only the most relevant fog nodes and local metric trees. This selection stage is implemented by the `_select_candidate_trees()` function inside the `ForestKNNParallel` component.

The selection procedure operates hierarchically in two stages.

First, fog nodes are ranked according to the distance between the query vector and the node-level `profile_centroid`. These centroids summarize the local data distribution maintained by each fog node. Only the closest fog nodes defined by the `top_nodes` parameter are retained for the next stage.

Second, inside each selected fog node, local metric trees are ranked according to the distance between the query point and the corresponding cluster centroids. Only the closest trees determined by the `top_trees_per_node` parameter are selected for traversal.

This hierarchical candidate selection mechanism significantly reduces unnecessary tree exploration and limits distributed search execution to the most relevant regions of the indexed data space. As a result, the number of computed distances and visited leaves is reduced while maintaining efficient nearest-neighbor retrieval.

After candidate selection, the selected trees are processed concurrently , allowing parallel local kNN execution across distributed fog nodes.

7.3 Distributed Result Aggregation

After local searches are completed, each fog node returns a partial set of nearest-neighbor candidates generated from its local metric trees. These local candidate sets are then merged to produce the final global kNN result.

Result aggregation is performed using the `merge_from()` function implemented inside the `KNNState` structure. The merging procedure combines the local heaps generated by distributed fog nodes into a single global top-k candidate set.

During aggregation, candidate neighbors are re-evaluated according to their squared distance values. Only the globally closest neighbors are retained in the final heap structure. The aggregation stage also combines execution statistics such as:

- the number of computed distances,
- the number of visited leaves,

This distributed aggregation mechanism allows the architecture to execute scalable parallel kNN search while preserving exact nearest-neighbor retrieval across distributed metric trees.

7.4 Cloud-Level Analytics

The cloud layer is the final stage of the pipeline. Its role is not to perform indexing or similarity search, but to collect the metrics produced during processing.

8 Conclusion

This chapter presented the proposed indexing approch designed for continuous IoT data streams in fog computing environments. The approach combines incremental clustering using the BIRCH algorithm with the construction of independent metric trees for each cluster, forming a hierarchical indexing architecture.

The main contribution of this design lies in separating data reduction from indexing. By grouping incoming data into compact and homogeneous clusters, the system reduces the search space before applying indexing structures. This leads to shallower trees and limits unnecessary traversal during kNN query execution.

The integration of a sliding window mechanism ensures that the system can handle continuous and evolving data streams while maintaining temporal consistency. In addition, distributing clusters across multiple fog nodes enables parallel index construction and query processing, improving scalability.

However, the proposed method also introduces certain trade-offs. The overall performance depends on parameter selection, such as window size, clustering threshold, and leaf size. Moreover, maintaining multiple index trees increases structural complexity, especially in highly dynamic data scenarios.

In summary, the proposed approach provides a structured and modular solution for IoT data indexing by combining clustering, metric indexing, and distributed processing. It establishes a clear basis for evaluating the effectiveness of the method, which is analyzed in the following chapter.

Chapter 4

Experimental Evaluation

1 Introduction

This chapter presents the experimental evaluation of the proposed approach.The objective is to assess its effectiveness in reducing search cost in streaming environments through the integration of clustering, indexing. The evaluation focuses on three main aspects: the number of distance computations, the number of visited leaf nodes, and the overall pruning efficiency.

2 Experimental Setup

This section presents the experimental environment, datasets, execution parameters, and evaluation metrics used to assess the proposed indexing architecture for continuous IoT streams. The experiments evaluate the effectiveness of incremental clustering, metric-tree indexing, pruning efficiency, and parallel kNN query processing in fog environments.

2.1 Hardware Configuration

All experiments were conducted on a personal computing environment using Python and multi-threaded execution through `ThreadPoolExecutor` to simulate distributed fog processing.

Table 1: Hardware Configuration

Component	**Specification**
Processor	Intel(R) Core(TM) i5-7300U CPU @ 2.60 GHz
RAM	8 GB
Graphics	Intel HD Graphics 620
Operating System	Windows 10 64-bit
Programming Language	Python

2.2 Experimental Parameters

The suggested system processes continuous IoT streams and the experiments evaluate the impact of different parameter settings on traversal reduction, pruning efficiency, and distributed query performance.

Table 2: Experimental Parameters

Parameter	Value
Window Processing Model	Sliding Window
Window Size	Dataset-dependent
Clustering Model	Incremental BIRCH
Minimum Cluster Size	Configured Threshold
Indexing Structure	Distributed Metric Trees
Search Algorithm	kNN Search
k Values	{5, 10, 15, 20, 50, 70}
Distance Metric	Squared Euclidean Distance
Parallel Execution	ThreadPoolExecutor

2.3 Dataset Description

The evaluation was conducted using multiple IoT-oriented datasets representing different stream sizes, density distributions, and clustering complexities. The datasets include environmental sensing, industrial monitoring, and telemetry streams.

Large datasets such as Intel Lab, Industrial IoT, and IoT Telemetry were used to evaluate scalability and distributed traversal behavior under large search spaces. The PM2.5 dataset was used to evaluate indexing performance on smaller and denser streams.

Table 3: Summary of the datasets used in the implementation

Dataset	Domain	Size	Dim	Streaming
Industrial IoT	Smart factories	1000	3	100
IoT Telemetry Sensor	Generic IoT	405184	6	600
Intel Lab Data	Health monitoring	2313682	4	500
PM2.5 Air Quality Insights	Environmental Monitoring	561	8	75

2.4 Evaluation Metrics

The experimental evaluation focuses on clustering behavior, traversal reduction, pruning effectiveness, and distributed kNN query performance.

The main evaluation metrics include:

- Number of generated clusters
- Number of metric trees
- Calculated distances
- Visited leaves
- Traversal reduction efficiency

To evaluate pruning effectiveness, the following efficiency ratio was used:

$$\text{Efficiency Ratio} = \frac{\text{Calculated Distances}}{\text{Total Points}} \tag{5}$$

Lower efficiency values indicate stronger pruning effectiveness and better search-space reduction.

3 Clustering Evaluation

This section evaluates the behavior of the Incremental BIRCH clustering stage during continuous IoT stream processing. The objective is to analyze how streaming data is incrementally partitioned before distributed metric-tree indexing.

The clustering process operates on sliding windows using the `partial_fit()` update mechanism. After clustering, sparse partitions are filtered according to a minimum cluster-size threshold in order to reduce indexing overhead and tree fragmentation.

The generated clusters directly influence the number of distributed metric trees, the granularity of search-space partitioning, and the traversal complexity during kNN query execution.

IoT Telemetry Dataset: The IoT Telemetry dataset generated a high number of clusters because telemetry streams contain continuous fluctuations and evolving sensor values.

As the stream evolves, the clustering process continuously detects new local regions and creates additional clusters to adapt to these changes.

This explains the higher clustering variability observed across streaming windows.

Table 4: Sample Clustering Results on IoT Telemetry Sensor Dataset

Window	Points	Clusters	Cluster Time (s)
1	600	12	0.249754
2	600	12	0.223475
3	600	12	0.265228
4	600	12	0.212794
5	600	12	0.194099
		...	
1346	600	21	0.940163
1347	600	20	0.916797
1348	600	22	0.944466
1349	600	15	0.949152
Average	**600.00**	**13.72**	**0.58512**

Industrial IoT Dataset: The Industrial IoT dataset produced several clusters with higher variability because industrial sensors generate dynamic and evolving measurements.

Different machine activities create changing local patterns inside the stream, which forces the clustering stage to generate multiple localized partitions.

This increased the number of generated clusters and improved distributed indexing granularity.

Table 5: Sample Clustering Results on Industrial IoT Dataset

Window	Points	Clusters	Cluster Time (s)
1	100	40	0.079319
2	100	49	0.071979
3	100	50	0.068160
4	100	56	0.071117
5	100	62	0.075003
6	100	60	0.074078
7	100	66	0.074707
8	100	63	0.070215
9	100	62	0.072350
10	100	65	0.085995
Average	**100.00**	**57.30**	**0.074292**

The Intel Lab Dataset: The Intel Lab dataset generated a relatively large number of clusters because the stream contains changing sensor measurements across windows. As the data changes over time, BIRCH creates several clusters to organize similar points together. However, the variability remained moderate, which explains why the clustering structure stayed relatively stable without excessive fragmentation.

Table 6: Sample Clustering Results on The Intel Lab Dataset

Window	Points	Clusters	Cluster Time (s)
1	500	18	2.277994
2	500	44	0.174778
3	500	26	0.179141
4	500	29	0.192981
5	500	39	0.194825
		...	
9249	500	90	1.561786
9250	500	93	2.112000
9251	500	109	1.447670
9252	500	49	1.661618
9253	500	12	1.375168
Average	**500.00**	**50.90**	**1.117796**

The PM2.5 Air Quality Insights Dataset: The PM2.5 dataset generated fewer and more stable clusters because the stream structure is more compact and less dynamic than the other datasets.

The sensor measurements remained relatively close across windows, so the clustering process did not need to generate many new partitions.

As a result, the indexing structure remained stable with lower fragmentation.

Table 7: Sample Clustering Results on PM2.5 Air Quality Dataset

Window	Points	Clusters	Cluster Time (s)
1	75	27	0.052062
2	75	29	0.047105
3	75	35	0.045744
4	75	36	0.068862
5	75	37	0.065013
		...	
15	75	41	0.077637
16	75	35	0.075725
17	75	36	0.087715
18	75	37	0.090449
19	75	37	0.093192
20	75	34	0.081955
Average	**75.00**	**34.35**	**0.0783**

Overall, the differences between the datasets are explained by the nature of their stream distributions. Heterogeneous and evolving streams such as Industrial IoT and IoT Telemetry produce more clusters because their local densities change over time, while compact streams such as PM2.5 generate fewer and more stable clusters. The filtering stage then removes sparse clusters to stabilize the indexing architecture and reduce unnecessary tree generation.

4 Metric-Tree Construction

This section evaluates the distributed metric-tree indexing stage constructed from the retained clusters generated by the Incremental BIRCH model. After cluster filtering, each retained cluster is transformed into an independent metric-tree structure within the fog layer.

The objective of this stage is to organize continuous IoT streams into localized searchable structures that reduce traversal complexity and improve pruning efficiency during distributed kNN query execution.

4.1 Number of Generated Trees

The number of generated metric trees depends directly on the clustering behavior of the streaming windows, since each retained cluster produces an independent indexing structure.

Intel Lab Dataset The Intel Lab dataset produced a large number of metric trees across the evaluated streaming windows.

This happened because the sensor measurements were a bit different over time, which made the clustering stage create a lot of small groups. Each retained cluster produced its own metric-tree structure, so the number of generated trees increased.

This meant that the indexing architecture was able to divide the search space more finely while keeping the structure relatively stable. The generated trees stayed pretty much the same across windows, which meant that they didn't get too fragmented and made it easy to search for things locally.

Table 8: Sample Indexing Results on The Intel Lab Dataset

Window	Trees	Tree Build Time (s)
1	10	0.064117
2	10	0.028843
3	10	0.145910
4	11	0.081430
5	12	0.076996
	...	
9249	6	0.021398
9250	7	0.037076
9251	7	0.021517
9252	10	0.035575
9253	6	0.046418
Average	**8.90**	**0.055928**

Industrial IoT Dataset The Industrial IoT dataset created lots of metric trees spread across the stream execution.

This was because the machine was being run in different ways and the sensors were giving different measurements. This meant that the clustering stage found several localised partitions. As a result, extra metric trees were created to organise these partitions on their own.

The indexing structure that resulted from this process made it easier to organise searchable regions into smaller parts. This local organisation made it easy to search and helped keep things stable when kNN was being used.

Table 9: Sample Indexing Results on Industrial IoT Dataset

Window	Trees	Tree Build Time (s)
1	19	0.007345
2	16	0.002862
3	13	0.001887
4	12	0.006042
5	7	0.003860
6	6	0.001668
7	9	0.002695
8	9	0.003102
9	8	0.001745
10	8	0.005647
Average	**10.70**	**0.003685**

IoT Telemetry Dataset This is a set of data from the Internet of Things (IoT).

The IoT Telemetry dataset produced one of the highest numbers of generated metric trees among all the datasets that were evaluated.

This happened because the telemetry stream showed continuous changes and more variability across different periods of time. As the local data distribution changed, the clustering stage repeatedly created new local clusters, which directly increased the number of metric-tree structures.

This made the indexing space more fragmented and complicated. However, the larger number of trees also allowed the architecture to adapt to changing stream regions and maintain local search execution despite the evolving data distribution.

Table 10: Sample Indexing Results on IoT Telemetry Dataset

Window	Trees	Tree Build Time (s)
1	12	0.305354
2	12	0.313026
3	12	0.170596
4	12	0.274093
5	12	0.234162
	...	
1345	14	0.544061
1346	13	0.588237
1347	14	0.561225
1348	12	0.578684
1349	13	0.537194
Average	**12.80**	**0.410263**

PM2.5 Air Quality Insights Dataset The PM2.5 dataset produced fewer metric trees and demonstrated the most consistent tree-generation behaviour.

This happened because the sensor measurements were mostly the same across the different streaming windows. This meant that there were fewer localised clusters during the clustering stage. This meant that fewer metric trees were needed to index the retained partitions.

Table 11: Sample Indexing Results on PM2.5 Air Quality Insights Dataset

Window	Trees	Tree Build Time (s)
1	11	0.007628
2	12	0.003868
3	9	0.001646
4	11	0.004266
5	9	0.003189
	...	
15	8	0.001110
16	9	0.002655
17	10	0.001569
18	9	0.003965
19	12	0.002873
20	7	0.001196
Average	**9.83**	**0.002998**

The smaller number of trees meant that indexing was less fragmented and kept the indexing structure compact. This made it easier for people to search for things in a local way, and it made the indexing process simpler.

The results show that the number of metric trees generated is directly related to the clustering behaviour. Datasets with more variability generated more clusters and therefore more indexing trees, whereas compact and homogeneous datasets produced fewer and more stable indexing structures. This confirms that clustering is the main way that search space is divided up, how indexing is done in detail, and how metric-tree structures are organised.

4.2 Structural Complexity Analysis

The structural complexity of the distributed indexing space is reflected through traversal behavior, visited leaves, and distance computations during kNN query execution.

Datasets producing larger numbers of distributed trees generally exhibited higher traversal complexity due to increased partition diversity and heterogeneous local densities.

The Industrial IoT and IoT Telemetry datasets produced the highest traversal complexity, with mean visited leaves exceeding 8 leaves per query and mean calculated distances approaching 150 distance evaluations. This behavior indicates broader traversal requirements across distributed indexing regions.

In contrast, the Intel Lab dataset maintained lower traversal complexity despite generating relatively large numbers of trees. The mean number of visited leaves remained close to 5 leaves with approximately 102 calculated distances per query, indicating effective pruning and localized traversal behavior.

The PM2.5 dataset exhibited the lowest structural complexity due to its compact stream distribution, generating fewer trees and reduced traversal overhead.

Overall, the results indicate that the distributed metric-tree architecture maintains manageable traversal complexity while preserving localized indexing behavior across evolving IoT streams.

5 Search-Space Reduction And Traversal Analysis

This section evaluates the effectiveness of the proposed distributed metric-tree architecture in reducing the explored search space during kNN query execution. The objective is to analyze how the distributed indexing structure minimizes unnecessary traversal and distance computations across continuous IoT streams.

Unlike the previous clustering and indexing stages, the traversal behavior is directly influenced by the neighborhood parameter k. Increasing the value of k expands the search region and requires the traversal process to explore larger portions of the distributed indexing space.

The evaluation focuses on:

- Distance reduction
- Traversal reduction
- Traversal complexity
- Search efficiency

5.1 Efficiency Ratio

To evaluate search-space reduction effectiveness, the following efficiency ratio was used:

$$Efficiency\ Ratio = \frac{Calculated\ Distances}{Total\ Points}$$

This metric estimates the fraction of the original search space explored during query execution. Lower efficiency values indicate stronger pruning effectiveness and better localization of the traversal process.

The experimental results show that increasing the value of k generally increased the efficiency ratio due to the larger traversal regions required to retrieve additional nearest neighbors.

The Intel Lab and IoT Telemetry datasets exhibited progressive increases in efficiency values as k increased. This behavior indicates that larger neighborhood requests reduced pruning effectiveness and expanded the explored search space.

In contrast, the Industrial IoT dataset maintained the lowest efficiency ratio among the evaluated datasets with a nearly stable value close to 0.132. This result demonstrates that the distributed indexing architecture successfully eliminated a large portion of irrelevant search regions during traversal.

The PM2.5 dataset also preserved stable efficiency behavior due to its compact stream organization and highly localized metric-tree partitions.

Overall, the results confirm that the proposed distributed metric-tree architecture significantly reduces the explored search space during continuous kNN execution while maintaining stable pruning performance across different streaming environments.

5.2 Distance Reduction

The number of calculated distances represents the computational cost of kNN query execution. Larger values of k generally increase the explored search region and require additional distance evaluations during traversal.

Tables 12–15 summarize the average traversal behavior for different values of k across the evaluated datasets.

Table 12: Intel Lab: Traversal Statistics by k

k	Distances	Leaves	Efficiency	Query Time
5	112.06	5.88	0.2241	0.001888
10	114.71	6.07	0.2294	0.002138
15	117.00	6.23	0.2340	0.002080
20	118.90	6.36	0.2378	0.002151
50	124.40	6.75	0.2488	0.002291
70	125.44	6.83	0.2509	0.002347

The Intel Lab dataset exhibited a clear increase in traversal cost as k increased. The average number of calculated distances increased from approximately 112 evaluations for $k = 5$ to more than 125 evaluations for $k = 70$. This behavior indicates that larger neighborhood requests expand the explored search region and reduce pruning effectiveness.

Table 13: IoT Telemetry: Traversal Statistics by k

k	Distances	Leaves	Efficiency	Query Time
5	150.74	8.62	0.2512	0.004110
10	156.53	9.02	0.2609	0.003972
15	160.49	9.32	0.2675	0.004112
20	163.05	9.50	0.2717	0.004417
50	168.02	9.89	0.2800	0.004153
70	168.02	9.89	0.2800	0.004355

The IoT Telemetry dataset produced the highest traversal complexity among the evaluated datasets. Increasing the value of k significantly expanded the number of

explored regions and distance evaluations due to the heterogeneous stream distribution and higher local density variability.

Table 14: Industrial IoT: Traversal Statistics by k

k	Distances	Leaves	Efficiency	Query Time
5	13.20	3.80	0.1320	0.002791
10	13.20	3.80	0.1320	0.004152
15	13.20	3.80	0.1320	0.003383
20	13.20	3.80	0.1320	0.003285
50	13.20	3.80	0.1320	0.002528
70	13.20	3.80	0.1320	0.002310

The Industrial IoT dataset maintained stable traversal behavior across all values of k. The number of calculated distances and visited leaves remained nearly constant, indicating that the distributed indexing structure preserved highly localized traversal regions independently of the neighborhood size.

Table 15: PM2.5 Air Quality Insights: Traversal Statistics by k

k	Distances	Leaves	Efficiency	Query Time
5	16.70	3.85	0.2227	0.001095
10	16.70	3.85	0.2227	0.001340
15	16.70	3.85	0.2227	0.001273
20	16.70	3.85	0.2227	0.001283
50	16.70	3.85	0.2227	0.001294
70	16.70	3.85	0.2227	0.001267

The PM2.5 dataset also preserved stable traversal behavior across all tested values of k. The compact stream structure generated highly localized search regions with limited traversal expansion.

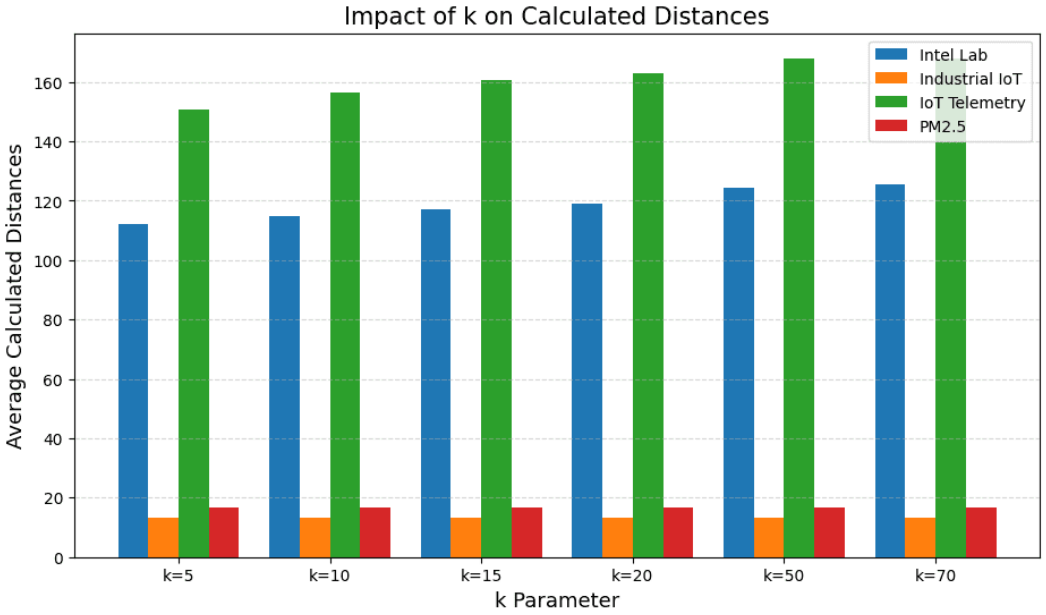


Figure 11: Impact of k on calculated distances across the evaluated datasets

The results show that increasing the value of k progressively increases the number of calculated distances for heterogeneous datasets such as Industrial IoT and IoT Telemetry.

5.3 Traversal Reduction

Traversal reduction is evaluated through the number of visited leaves during distributed kNN execution. Lower leaf exploration indicates that the indexing structure successfully localizes the search process and limits unnecessary branch traversal.

The experimental results show that increasing the value of k gradually expanded the explored traversal region for heterogeneous datasets such as Intel Lab and IoT Telemetry. Larger neighborhood requests required the traversal process to visit additional candidate regions in order to retrieve a broader set of nearest neighbors.

The IoT Telemetry dataset produced the highest leaf exploration among the evaluated datasets. The average number of visited leaves increased from approximately 8.6 leaves for $k = 5$ to nearly 10 leaves for larger values of k. This behavior reflects the heterogeneous spatial distribution of the stream and the presence of evolving local density regions.

In contrast, the Industrial IoT and PM2.5 datasets maintained nearly stable traversal behavior across all tested values of k. The generated metric-tree partitions remained sufficiently compact to preserve localized traversal and efficient pruning performance.

Overall, the results confirm that the proposed distributed metric-tree architecture effectively limits unnecessary traversal while preserving scalable neighborhood exploration during continuous query execution.

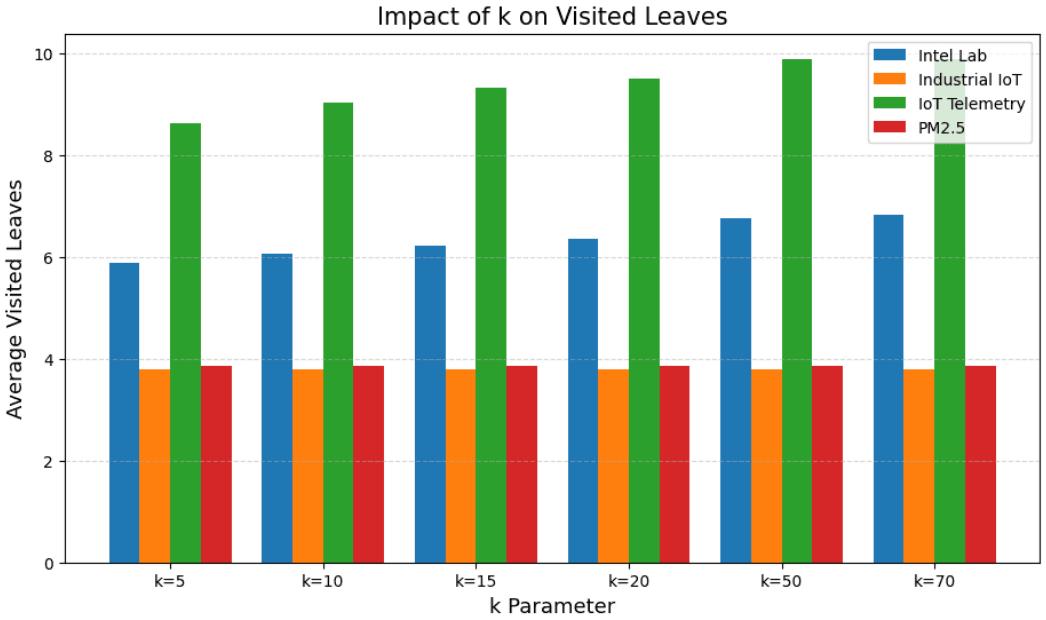


Figure 12: Impact of k on visited leaves across the evaluated datasets

The number of visited leaves increased progressively with larger values of k, indicating broader traversal regions during distributed kNN execution.

5.4 Traversal Complexity

The complexity of traversal shows how hard it is for the search process to find its way through the distributed metric-tree structure during kNN execution. The results of the experiment show that data sets which are more varied and have more differences between clusters tend to generate wider regions which need to be explored more. This behaviour was most obvious in the IoT Telemetry dataset, which always produced higher numbers of visited leaves and distance calculations as (k) increased. In contrast, the Industrial IoT and PM2.5 datasets maintained stable traversal behaviour due to their more compact and less fragmented indexing structures. The complexity of traversal is affected not only by the size of the neighbourhood (k), but also by how the clusters are organised during the indexing phase.

6 Conclusion

This chapter evaluated the proposed indexing architecture using multiple real-world IoT datasets under continuous streaming conditions. The experimental analysis focused on clustering behavior, distributed metric-tree construction, search-space reduction, traversal efficiency, and parallel kNN execution.

The results show that the integration of incremental BIRCH clustering with metric-tree indexing significantly reduces the effective search space before query execution. Cluster-based partitioning improved data organization inside the fog layer and limited unnecessary traversal across unrelated regions of the indexing space.

The constructed metric trees maintained relatively low structural complexity while supporting efficient pruning during kNN search. The traversal analysis demonstrated that the proposed indexing strategy reduced both the number of calculated distances and the number of visited leaves compared to exhaustive search. The pruning mechanism became particularly effective for moderate k values, where localized cluster traversal minimized exploration overhead.

The evaluation also confirmed the scalability of the distributed fog architecture. Parallel query execution across multiple trees reduced query latency and allowed the workload to be distributed among fog nodes without centralized processing bottlenecks. In addition, the continuous window-based processing model enabled the indexing pipeline to adapt incrementally to evolving IoT streams without rebuilding the global indexing structure.

However, the experiments also revealed several limitations. The efficiency of the indexing process remains sensitive to clustering parameters, window configuration, and cluster distribution balance. Large k values increase traversal complexity and reduce pruning effectiveness due to the expansion of the candidate search space. Overall, the experimental results confirm that the proposed approach provides an effective framework for continuous IoT data indexing by combining incremental stream partitioning, metric indexing, and parallel search execution within the fog layer.

Conclusion

This thesis addressed the problem of efficient k-nearest neighbor (kNN) query processing over continuous IoT data streams within fog computing environments. The rapid growth of IoT applications has led to the generation of large-scale, high-velocity, and heterogeneous data, which poses significant challenges for traditional data management systems. In particular, centralized indexing approaches struggle to meet the requirements of low latency, scalability, and real-time responsiveness, especially in distributed IoT infrastructures.

To address these challenges, this work proposed a hybrid indexing framework that integrates incremental clustering, metric-based indexing, and distributed processing across fog nodes. The proposed approach is based on a pipeline architecture that processes data streams through successive stages, including data ingestion, sliding window management, clustering, indexing, and query execution. This design allows the system to operate continuously while maintaining a structured representation of the incoming data.

A key component of the proposed framework is the use of the BIRCH algorithm for incremental clustering. This choice enables the system to group incoming data into compact and representative clusters in a single pass, without requiring full data storage or repeated processing. By summarizing data into clusters, the system reduces the number of elements that need to be indexed, which directly contributes to lowering computational cost. This clustering stage acts as a data reduction mechanism that simplifies subsequent indexing operations.

Following clustering, each cluster is independently indexed using a binary metric tree structure. This indexing approach partitions the data space based on distance comparisons and supports efficient pruning during kNN search. The use of independent trees for each cluster avoids the complexity of maintaining a global index and allows the system to adapt more easily to variations in data distribution. In addition, the distribution of clusters across multiple fog nodes enables parallel processing, which improves scalability and reduces query response time.

The experimental evaluation was conducted on several real-world datasets representing different IoT scenarios. The results showed that the proposed approach is able to reduce the number of distance computations and limit the number of visited nodes during query execution. These findings confirm that clustering-based data reduction combined with metric indexing can effectively reduce the search space. As a result, the query execution time remains relatively low and stable, even when the data volume increases. This demonstrates that the proposed method is suitable for continuous and high-velocity data streams.

Another important observation is the role of pruning mechanisms in improving search efficiency. By exploiting distance bounds within the metric tree structure, the system avoids exploring irrelevant branches, which reduces unnecessary computations. This behavior is reflected in the reduced number of visited leaves and the improved

pruning efficiency observed in the experimental results. These indicators provide a more detailed understanding of how the indexing structure contributes to overall performance.

Despite these positive results, the study also highlights several limitations. First, the performance of the proposed method is sensitive to parameter selection, including window size, clustering thresholds, and tree configuration parameters. Inappropriate parameter values may lead to unbalanced clusters or inefficient indexing structures. Second, the use of multiple independent trees introduces additional structural complexity, particularly in dynamic environments where data distribution evolves over time. Third, the current evaluation focuses primarily on computational metrics and does not include other important factors such as energy consumption, communication overhead, or fault tolerance, which are critical in fog computing systems.

In addition, the proposed framework does not explicitly address issues related to concept drift, where the statistical properties of the data change over time. In such cases, the clustering and indexing structures may need to be updated or adapted dynamically to maintain their effectiveness. This limitation suggests that further improvements are required to enhance the adaptability of the system in long-term streaming scenarios.

Overall, this work demonstrates that combining clustering, metric indexing, and distributed fog processing provides a practical and effective solution for managing continuous IoT data streams. The proposed framework improves query efficiency by reducing the search space and enabling parallel processing, while maintaining a relatively simple and modular structure. At the same time, the results emphasize that achieving optimal performance requires careful parameter tuning and consideration of system constraints.

Future work may focus on several directions. One important extension is the development of adaptive parameter tuning mechanisms that can automatically adjust clustering and indexing parameters based on data characteristics. Another direction is the integration of energy-aware and resource-aware strategies to improve the efficiency of fog nodes. Additionally, dynamic load balancing techniques can be explored to better distribute computational tasks across nodes.

Further research may also consider extending the framework to support more complex query types, such as range queries or similarity joins, as well as handling heterogeneous and high-dimensional data. Finally, incorporating mechanisms to detect and adapt to concept drift would enhance the robustness of the system in evolving IoT environments.

In conclusion, the proposed approach provides a structured contribution to the field of IoT data indexing by demonstrating how clustering, indexing, and distributed processing can be effectively combined to support efficient kNN query processing in streaming contexts.

Bibliography

[1] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 81--92, 2003.

[2] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *ICDT*, pages 420--434, 2001.

[3] Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering: Algorithms and Applications*. CRC Press, 2013.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD*, pages 94--105, 1998.

[5] Tyler et al. Akidau. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale data processing. *VLDB*, 8(12):1792--1803, 2015.

[6] Tajudeen Akanbi Akinosho, Elias Tabane, and Zenghui Wang. A comparative analysis of data stream clustering algorithms. *International Journal of Computing*, 22(4):439--446, 2023.

[7] Ian F. et al. Akyildiz. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393--422, 2002.

[8] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD*, pages 49--60, 1999.

[9] Michael et al. Armbrust. A view of cloud computing. *Communications of the ACM*, 53(4):50--58, 2010.

[10] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of SODA*, pages 1027--1035, 2007.

[11] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787--2805, October 2010.

[12] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787--2805, 2010.

[13] Pavel Berkhin. A survey of clustering data mining techniques. *Grouping Multidimensional Data*, pages 25--71, 2006.

[14] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Jesse Read, Philipp Kranen, Hannes Kremer, Timm Jansen, and Thomas Seidl. Moa: A real-time analytics open-source framework. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 617--620. Springer, 2011.

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[16] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322--373, 2001.

[17] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *MCC Workshop*, 2012.

[18] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop*, pages 13--16, 2012.

[19] Dmitri Burago, Yuri Burago, and Sergei Ivanov. *A Course in Metric Geometry*. American Mathematical Society, 2001.

[20] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the SIAM International Conference on Data Mining*, pages 328--339, 2006.

[21] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12--27, 2011.

[22] Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300--307, 2007.

[23] Min et al. Chen. Big data: A survey. *Mobile Networks and Applications*, 19(2):171--209, 2014.

[24] Mung Chiang and Tao Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854--864, 2016.

[25] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. *VLDB*, pages 426--435, 1997.

[26] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer, 2009.

[27] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley, 2001.

[28] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of KDD*, pages 226--231, 1996.

[29] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139--172, 1987.

[30] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139--172, 1987.

[31] Chris Fraley and Adrian E. Raftery. Model-based clustering, discriminant analysis, and density estimation. *Journal of the American Statistical Association*, 97(458):611--631, 2002.

[32] John H. Gennari, Pat Langley, and Douglas Fisher. Models of incremental concept formation. *Artificial Intelligence*, 40(1-3):11--61, 1989.

[33] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645--1660, September 2013.

[34] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645--1660, 2013.

[35] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. In *SIGMOD*, pages 73--84, 1998.

[36] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. In *ICDE*, pages 512--521, 1999.

[37] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2011.

[38] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2012.

[39] Richard J. Hathaway and James C. Bezdek. Fuzzy c-means clustering of incomplete data. *IEEE Transactions on Systems, Man, and Cybernetics*, 17(5):735--744, 1987.

[40] Pengfei et al. Hu. Fog computing: Survey and research challenges. *IEEE Access*, 5:20786--20801, 2017.

[41] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. *STOC*, 1998.

[42] IoT Analytics. State of iot 2025: Number of connected iot devices growing 14% to 21.1 billion globally. `https://iot-analytics.com/number-connected-iot-devices/`, 2025. Accessed: 2026-04-10.

[43] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651--666, 2010.

[44] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264--323, 1999.

[45] R. Khan, S. U. Khan, R. Zaheer, and S. Khan. Future internet: The internet of things architecture, possible applications and key challenges. In *Proc. 10th Int. Conf. Frontiers of Information Technology (FIT)*, pages 257--260, 2012.

[46] Karima Khettabi et al. Efficient iot data stream indexing in fog-cloud. *Big Data and Cognitive Computing*, 2023.

[47] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464--1480, 1990.

[48] Teuvo Kohonen. *Self-Organizing Maps*. Springer, 3rd edition, 2001.

[49] Philipp Kranen, Ira Assent, Corinna Baldauf, and Thomas Seidl. The clustree: Indexing micro-clusters for anytime stream mining. *Knowledge and Information Systems*, 29(2):249--272, 2011.

[50] J. Lee, Y. Zhang, and H. Kim. Industrial internet of things: Architecture, applications, and challenges. *IEEE Transactions on Industrial Informatics*, 21(3):450--465, 2025.

[51] Stuart Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129--137, 1982.

[52] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium*, pages 281--297, 1967.

[53] Spyros Mansalis, Eirini Ntoutsi, Nikos Pelekis, and Yannis Theodoridis. An evaluation of data stream clustering algorithms. *Statistical Analysis and Data Mining*, 11(4):167--187, 2018.

[54] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*. Manning Publications, 2015.

[55] Viktor Mayer-Schönberger and Kenneth Cukier. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. Houghton Mifflin Harcourt, 2013.

[56] Mithun Mukherjee, Rakesh Matam, Lei Shu, Leandros Maglaras, Mohamed Amine Ferrag, Nikumani Choudhury, and Vikas Kumar. Security and privacy in fog computing: Challenges. *IEEE access*, 5:19293--19304, 2017.

[57] Daniel Müllner. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378*, 2011.

[58] H. Srinivasa Nagesh, Sanjay Goil, and Alok Choudhary. Mafia: Efficient and scalable subspace clustering for very large data sets. In *KDD Workshop on High Performance Data Mining*, 1999.

[59] J. Ortiz, L. Fernandez, and P. Nguyen. Tinyml in smart cities. *Future Internet*, 2025.

[60] Nikhil R. Pal and James C. Bezdek. A review on fuzzy clustering and its applications. *Pattern Recognition*, 28(9):1277--1294, 1995.

[61] Jüri-Sören et al. Preden. The benefits of self-awareness and attention in fog and mist computing. In *ICC Workshop*, pages 1096--1101, 2015.

[62] P. P. Ray. A survey on internet of things architectures. *Journal of King Saud University - Computer and Information Sciences*, 30(3):291--319, 2018.

[63] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70--95, February 2016.

[64] A. Rita, J. Smith, and K. Lee. A comprehensive review of iot standards for smart cities. *MDPI Smart Cities*, 8(4), 2025.

[65] Timothy J. Ross. *Fuzzy Logic with Engineering Applications*. Wiley, 2010.

[66] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[67] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30--39, 2017.

[68] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637--646, 2016.

[69] Weisong et al. Shi. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637--646, 2016.

[70] Peter H. A. Sneath and Robert R. Sokal. *Numerical Taxonomy*. Freeman, 1973.

[71] Michael et al. Stonebraker. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 42(1):42--47, 2013.

[72] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175--179, 1991.

[73] Unknown. Efficient indexing for iot massive data based on cloud-fog computing. *ResearchGate*, 2019.

[74] Unknown. A distributed computing approach to iot data processing: Edge, fog, and cloud analytics framework. *ResearchGate*, 2020.

[75] Unknown. A survey on big iot data indexing: Potential solutions and open issues. *ResearchGate*, 2020.

[76] Unknown. Clustering and parallel indexing of big iot data in fog-cloud computing. *ResearchGate*, 2021.

[77] Unknown. An overview of fog data analytics for iot applications. *MDPI*, 2023.

[78] Wei Wang, Jiong Yang, and Richard Muntz. Sting: A statistical information grid approach to spatial data mining. In *VLDB*, pages 186--195, 1997.

[79] M. Weyrich and C. Ebert. Reference architectures for the internet of things. *IEEE Software*, 33(1):112--116, January 2016.

[80] F. Wortmann and K. Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221--224, June 2015.

[81] L. Da Xu, W. He, and S. Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233--2243, November 2014.

[82] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645--678, 2005.

[83] Chengcheng Yang, Dong Deng, Shuo Shang, and Ling Shao. Efficient locality-sensitive hashing over high-dimensional data streams. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1986--1989. IEEE, 2020.

[84] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. *Proceedings of the Workshop on Mobile Big Data*, pages 37--42, 2015.

[85] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311--321, 1993.

[86] A. Yousefpour et al. Fog computing: Towards minimizing latency in iot applications. *IEEE Internet of Things Journal*, 2018.

[87] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22--32, February 2014.

[88] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*. Springer, 2006.

[89] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference*, 1996.

[90] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference*, pages 103--114, 1996.