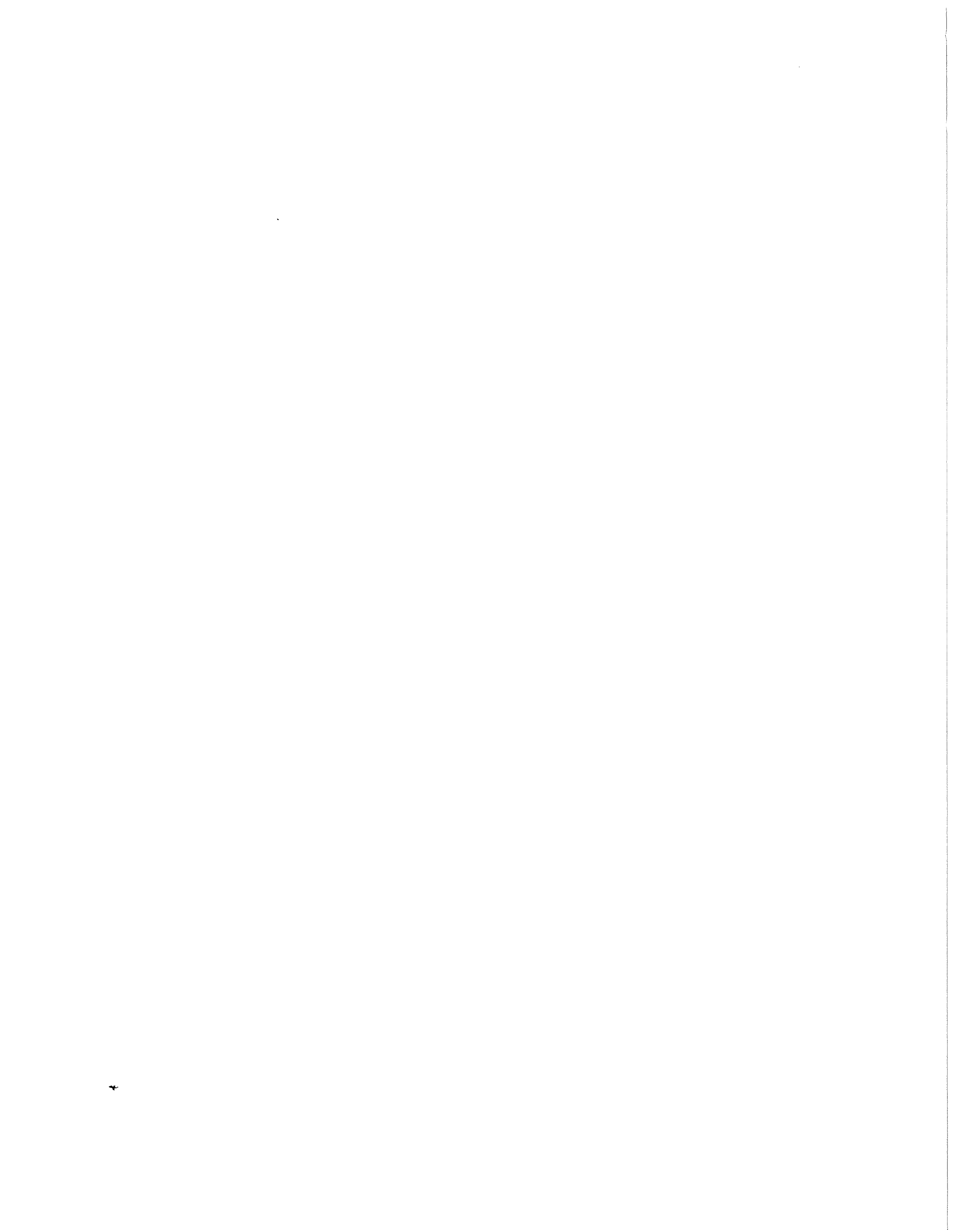
Data Clustering For Very Large Datasets Plus Applications

Tian Zhang

Technical Report #1355

November 1997



DATA CLUSTERING FOR VERY LARGE DATASETS PLUS APPLICATIONS

By

Tian Zhang

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

1997

© Copyright by Tian Zhang 1997

All Rights Reserved

Abstract

Data clustering is an important way of exploring data, and has been shown to be useful in many domains such as data classification and image processing. Recently, there is a growing emphasis on exploratory analysis of *very large* datasets to discover useful patterns and correlations among attributes. It is called *data mining*, and data clustering is regarded as a particular branch. However existing data clustering methods do not adequately address the problem of processing very large datasets with limited resources (e.g., running time and memory). As the dataset size increases, they do not scale up well in terms of memory requirement, running time, and result quality.

In this thesis, with a new in-memory data structure called CF-tree serving as a data distribution summarization, an efficient and scalable data clustering method is proposed, and implemented in a system called BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies). With various patterns of synthetic datasets, its performance is studied and compared with other methods in terms of memory requirement, running time, quality, stability and scalability. Finally, BIRCH is applied to solve two real world problems: (1) building an iterative and interactive pixel classification tool, and (2) generating initial codebook for image compression. Its performance on these real datasets is also compared with other methods.

Acknowledgements

First of all, I would like to devote my Ph.D thesis to the two most important persons in my life: my mother Kaiguang and my daughter Angela. In the past year, my life has been filled with both tears and joys: first my mother left me after an arduous fight with cancer, then my daughter arrived just like an Angel full of comforts. Seeing both of them, one leaving and one arriving, I have understood what life is all about. I am very grateful to my husband Jian for being with me all the time and giving me the support I need to finish my Ph.D.

My advisor, Raghu Ramakrishnan, has been the major force for me to stay concentrated on my Ph.D topic. I am very fortunate to meet such an open-minded advisor: he accepted me as his student even though he knew that I had a very different background. He gave me the freedom to explore new topics (just as he said "do what other people are not doing"), and at the same time, he pushed me to focus on and deepen the work. Also from him, I gained tremendously a lot on my research, writing and presentation skills. Besides all of these, he is like a friend with whom I can talk not only my career objective, but also the stories of my mother and my daughter.

I am very thankful to the fact that I have another advisor, Miron Livny. So I have got the unusual advantage of double resources that made my Ph.D study most efficient. Miron is always full of suggestions and comments on my work that I could never use up. To me, his encouragement is just as earnest as his criticism is. However either way has equally benefited my work.

I took my first database course from Jeffrey Naughton, and I would like to thank him for leading me to this fascinating field, and for serving on my committee at the end. Jude Shavlik has always been very encouraging and helpful to me during the whole period of my Ph.D study in Madison. I attribute my knowledge in machine learning to his splendid lectures, and I appreciate his help and strong support when I was on the job market too. At one point, I have been so interested in logic programming and Kenneth

Kunen has spent a lot of time showing me the interesting problems and solutions in this area. Although my thesis is not about logic, but all those intelligent discussions are very valuable experiences to me. I had the opportunity to work on a project with Yannis Ioannidis in my first year in Madison and I have learned from him the way of describing a complex problem with very simple terms. I had a very interesting discussion with Olvi Mangasarian about optimal sequence trends. Eric Bach provided many references in graph theory to me. John Strikwerda helped me solve problems related to the linkage of FORTRAN and C codes.

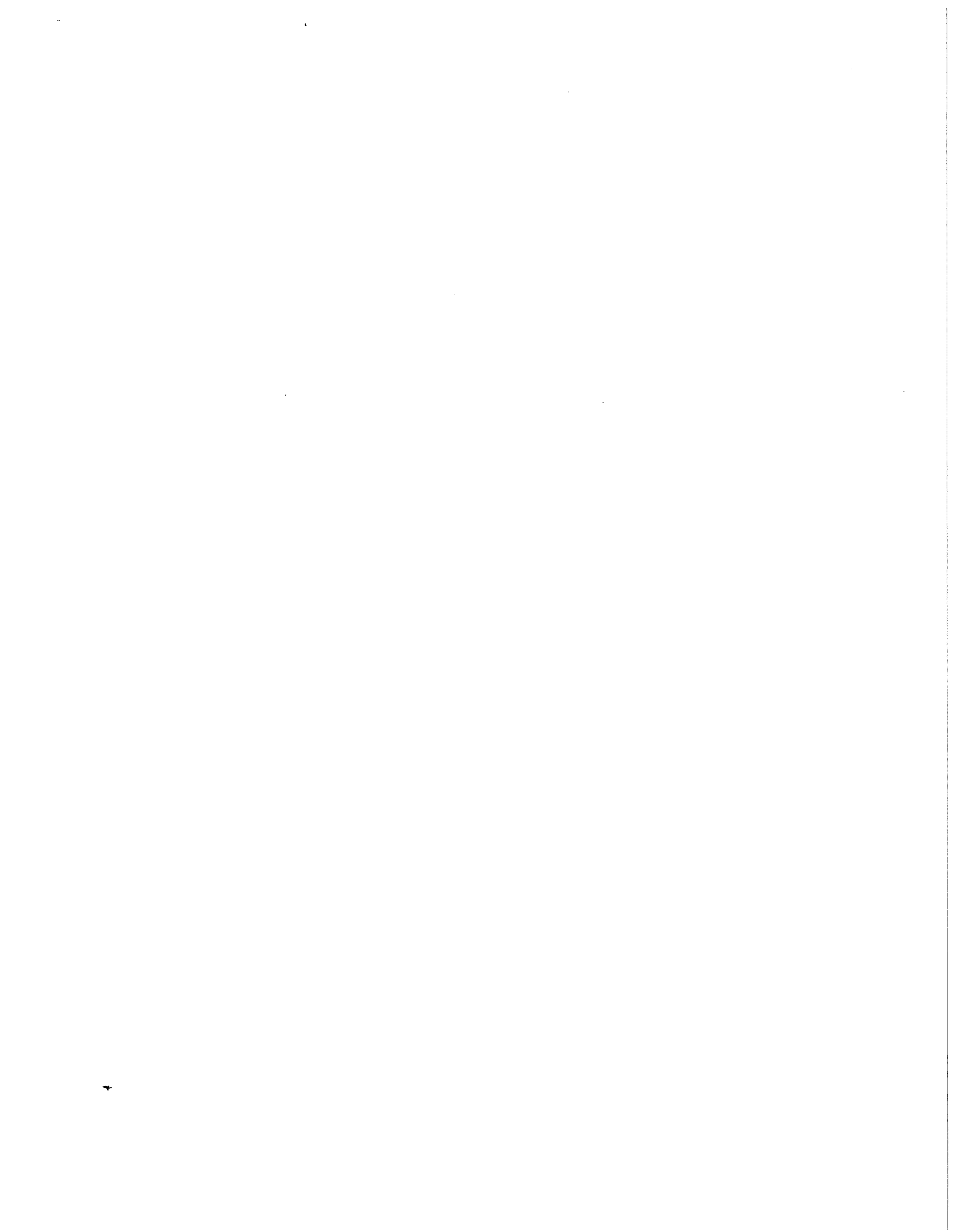
Wei-Yin Loh of Statistics Department was on my preliminary committee and oral defense committee. He helped me identify the relationship between my work and the most current related statistical literatures. He has shown me the most abstract statistical theory in such an intuitive way. John Norman of Soil Sciences Department was another external examiner on my committee. He has continuously shown great interest and enthusiasm in my work, and gave me detailed suggestions for improving my work and my thesis. His student, Chris Kucharik, provided the access to their images for conducting my experiments.

I would like to thank the other members in my advisors' group for the pleasant collaborations: Kent Wenger provided me the data input code and kept improving it; Jussi Myllymaki showed me how to use DEVISE in great details; Shaun Flisakowski explained how to overload the C++ memory allocation part; Viresh Ratnakar provided images and image compression algorithms for conducting my experiments, and he also explained the meanings of various quality measurements used in image compression. When I first joined the group, Praveen Seshadri has given me a lot of advices and help to start the work.

It is a great pleasure being a member of the resourceful database group in the department, and I have enjoyed the interesting communications with the other students including Joseph Albert, Navin Kabra, Jignesh Patel, Kristin Rufte, Shivakumar Venkataraman, Janet Wiener, Jiebing Yu, and Yihong Zhao. I would also like to thank Lorene Webber for helping me understand and fulfill the department and graduate school's

Ph.D procedures in a timely way.

It is time to say good-bye, but I would like to say keeping in touch instead because I know I will miss Madison and all of you for a long time.



List of Figures

1 Overview of Thesis 3

2 A CF-tree Example 10

3 Effect of Split, Merge and Resplit 13

4 Rebuilding CF-tree 16

5 Overview of BIRCH Clustering Algorithm 25

6 Flow Chart of Phase 1 28

7 Flow Chart of Phase 2 35

8 Flow Chart of Phase 3 37

9 Flow Chart of Phase 4 39

10 Time Scalability with respect to Increasing Number of Points per Cluster (n) 51

11 Quality Stability with respect to Increasing Number of Points per Cluster (n) 51

12 Time Scalability with respect to Increasing Number of Clusters (K) . . . 52

13 Quality Stability with respect to Increasing Number of Clusters (K) . . . 52

14 Time Scalability with respect to Increasing Dimension (d) 53

15 Quality Stability with respect to Increasing Dimension (d) 53

16 Interactive and Iterative Pixel Classification Tool 64

17 An Example of DAG Used to Track History 66

18 An Example of MVI image 67

19 1st Run: Separate Trees and Sky 68

20 2nd Run: Separate Branches, Shadows and Sunlit Leaves 69

21 Another Example of MVI image 74

22 Lena Compressed with BIRCH Codebook 79

23 Lena Compressed with CLARANS Codebook 79

24 Lena Compressed with LBG Codebook 79

25 Baboon Compressed with BIRCH Codebook 79

26 Baboon Compressed with CLARANS Codebook 79

27 Baboon Compressed with LBG Codebook 79

28 Data of DS1 95

29 Data of DS2 95

30 Data of DS3 95

31 Intended Clusters of DS1 95

32 Intended Clusters of DS2 95

33 Intended Clusters of DS3 95

34 BIRCH Clusters of DS1 96

35 BIRCH Clusters of DS2 96

36 BIRCH Clusters of DS3 96

37 BIRCH Clusters of DS1o 96

38 BIRCH Clusters of DS2o 96

39 BIRCH Clusters of DS3o 96

40 CLARANS Clusters of DS1 97

41 CLARANS Clusters of DS2 97

42 CLARANS Clusters of DS3 97

43 CLARANS Clusters of DS1o 97

44 CLARANS Clusters of DS2o 97

45 CLARANS Clusters of DS3o 97

46 KMEANS Clusters of DS1 98

47 KMEANS Clusters of DS2 98

48 KMEANS Clusters of DS3 98

49 KMEANS Clusters of DS1o 98

50 KMEANS Clusters of DS2o 98
51 KMEANS Clusters of DS3o 98
52 BIRCH Clusters of DS4 with Outlier Options On 99
53 BIRCH Clusters of DS5 with Outlier Options On 99
54 BIRCH Clusters of DS6 with Outlier Options On 99
55 BIRCH Clusters of DS4 with Outlier Options Off 99
56 BIRCH Clusters of DS5 with Outlier Options Off 99
57 BIRCH Clusters of DS6 with Outlier Options Off 99

List of Tables

1 Data Generation Parameters and Their Values or Ranges Experimented . 43
2 BIRCH Parameters and Their Default Values 45
3 Datasets Used as Base Workload 46
4 BIRCH Performance on Base Workload 48
5 CLARANS Performance on Base Workload 48
6 KMEANS Performance on Base Workload 48
7 Sensitivity to Initial Threshold 57
8 Sensitivity to Page Size 59
9 Effects of Outlier Options 59
10 Effects of Phase 3 Algorithms 60
11 Effects of Memory Size 61
12 Effects of Distance Metrics 62
13 BIRCH,CLARANS,KMEANS on Pixel Classification 73
14 BIRCH, CLARANS and LBG on Image Compression 78

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

2 Background 4

2.1 Metric Attributes 4

2.2 Measurements for one cluster 4

2.3 Measurements between two clusters 5

2.4 Measurements of Clustering Quality 6

2.5 Weighting and Shifting 6

3 CF and CF-tree 8

3.0.1 Clustering Feature (CF) 8

3.0.2 CF Representativity 9

3.0.3 CF Additivity 9

3.1 CF-tree 9

3.1.1 CF-tree Definition 10

3.1.2 Insertion Algorithm 11

3.1.3 Anomalies 14

3.1.4 Rebuilding Algorithm 15

3.1.5 Reducibility 17

3.1.6 CF-tree versus Multi-Dimensional Histogram 17

3.1.7 Generalizing CF-tree 18

4 Data Clustering 19

4.1 Previous Work 19

4.1.1 Probability-Based 19

4.1.2 Distance-Based 21

4.2 Contributions and Limitations of BIRCH 23

4.3 BIRCH Clustering Algorithm 24

4.3.1 Overview 24

4.3.2 Phase 1 29

4.3.3 Phase2 34

4.3.4 Phase 3 36

4.3.5 Phase 4 39

4.3.6 Memory Management 40

5 Performance of BIRCH Data Clustering **41**

5.1 Analysis 41

5.2 Synthetic Dataset Generator 42

5.3 Parameters and Default Setting 44

5.4 Base Workload Performance 45

5.5 Other Methods on Base Workload 48

5.6 Scalability and Stability 54

5.6.1 Increasing the Number of Points per Cluster (n) 54

5.6.2 Increasing the Number of Clusters (K) 55

5.6.3 Increasing the Dimension (d) 56

5.7 Sensitivity to Parameters 56

6 BIRCH Applications **63**

6.1 Pixel Classification Tool 63

6.1.1 Motivation 63

6.1.2 Pixel Classification Tool 64

6.1.3 Example of Using the Tool 66

6.1.4 User Evaluation 71

6.1.5 BIRCH,CLARANS and KMEANS on Pixel Classification 72

6.2 Codebook Generation in Image Compression 75
6.2.1 Motivation 75
6.2.2 BIRCH,CLARANS and LBG on Image Compression 75

7 Implementation Issues **80**
7.1 Code Components 80
7.1.1 Utility 80
7.1.2 Data Clustering 80
7.1.3 Density Estimation 81
7.1.4 Data Input and Preparation 81
7.2 BIRCH Execution 81
7.2.1 Input of BIRCH 82
7.2.2 Output of BIRCH 83

8 Conclusions and Future Research **84**
8.1 Data Clustering 84
8.2 Density Estimation 85
8.2.1 Previous Work 85
8.2.2 CF-kernel Method 88

A Proof of CF Representativity Theorem **90**

B CF-tree Insertion Algorithm **92**

C CF-tree Rebuilding Algorithm **94**

D Base Workload **95**

E BIRCH on Base Workload **96**

F CLARANS on Base Workload **97**

G KMEANS on Base Workload **98**

H Effects of Outlier Options 99

Bibliography 100

Chapter 1

Introduction

This thesis presents system BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies), a system that provides an efficient and scalable data clustering method for very large datasets with a limited amount of resources (e.g., running time and memory).

In this thesis, *data clustering* refers to the problem of dividing N data points into K groups to minimize the intra-group difference, such as the sum of the squared distances from the cluster centers. Usually given a very large set of multidimensional data points, the data space is not uniformly occupied by the data points. Through data clustering, one can identify the sparse and the crowded regions, and hence discover the overall distribution patterns or the correlations among data attributes, which may be used to guide the application of more rigorous analysis procedures. It is a very practical subject and has been studied for many years. Many methods have been developed and applied to various domains, including data classification and image compression [41, 13]. However, it is also a very difficult subject because theoretically, it is a *nonconvex discrete*[35] optimization problem. Due to an abundance of local minima, there is typically no way to find the global minimal solution without trying all possible partitions, which is infeasible except when N and K are extremely small.

Recently, there is a growing emphasis on exploratory analysis of very large datasets to discover useful patterns. Organizations are investing heavily on "data warehousing" to collect data in a form suitable for extensive exploratory analysis and there has been extensive research on exploratory data analysis or "data mining" algorithms [3, 5, 20, 48]. Data clustering is regarded as a particular branch of data mining. Here the important issue is that for a data mining method to be successful in a database environment, it must scale up well in terms of memory requirement, running time and quality as the dataset

size increases. So besides the mentioned difficulty, we have additional database-oriented constraints: *The amount of memory available is limited (typically, much smaller than the dataset size) whereas the dataset can be any large. We want to minimize the I/O cost involved in clustering the dataset.* However, prior work in data clustering did not adequately address the problem of processing very large datasets with a limited amount of resources (e.g., running time and memory). So generally as the dataset size increases, they could not scale up well in terms of memory requirements, running time, and result quality.

We will present a data clustering system named BIRCH and demonstrate that it is especially suitable for clustering very large datasets. First it tries to escape from local minima by clustering on a data summarization instead of on the data itself. Second it tries to cluster any large dataset as well as the given amount of memory allows. Third its running time and I/O cost is linear with the the dataset size: a *single scan* of the dataset yields a good clustering, and one or more additional passes can (optionally) be used to improve the quality further.

By (1) evaluating BIRCH's running time, memory usage, clustering quality, stability and scalability, (2) comparing BIRCH with other existing algorithms, and (3) applying BIRCH to real world problems, we argue that BIRCH is the best available clustering method for handling very large datasets. BIRCH's architecture also offers opportunities for parallel and concurrent clustering. It is also possible to interactively and dynamically tune the performance based on knowledge gained about the dataset over the course of the execution.

The thesis is organized in the following way. Chapter 2 presents some background materials needed for discussing data clustering in BIRCH. Chapter 3 introduces the clustering feature (CF) concept, the CF-tree structure, as well as some related algorithms, theorems and proofs. The differences between the CF-tree and the multi-dimensional histogram are also discussed briefly. Chapter 4 first surveys the previous data clustering work, and then discusses how it is approached in BIRCH by using the CF-tree structure. All the details of BIRCH data clustering algorithm will be addressed in this

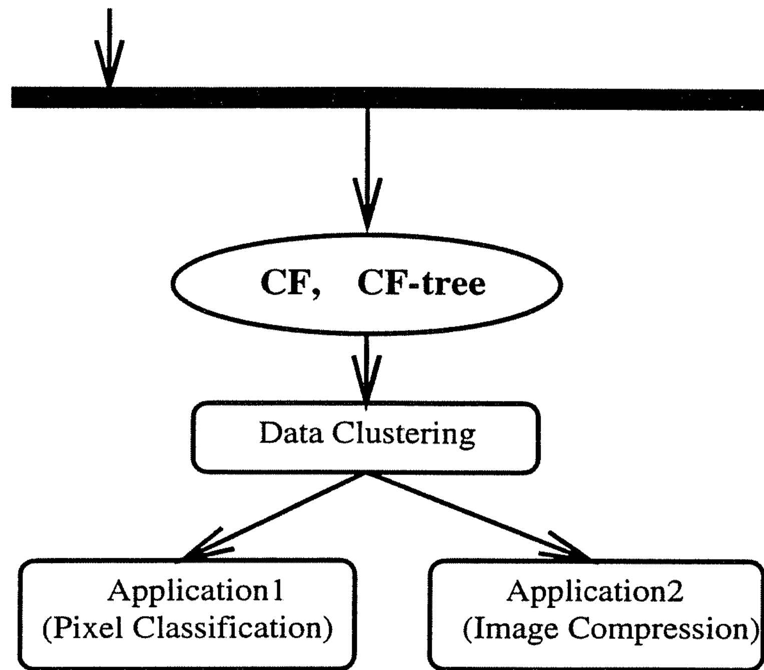


Figure 1: Overview of Thesis

chapter. Chapter 5 presents the performance evaluation of the BIRCH, CLARANS[45], and KMEANS[32] on synthetic datasets. Chapter 6 presents two applications of BIRCH data clustering algorithm. They are also intended to show how BIRCH, CLARANS and KMEANS perform on those real datasets. Chapter 7 discusses the implementation issues in BIRCH, and it serves as a technical documentation for using BIRCH. Chapter 8 presents our conclusions and points out directions for future research. In this chapter we will explain how BIRCH can be generalized to solve density estimation problem for very large datasets efficiently, what additional work is needed to improve it further. Figure 1 gives an overview of the contents of the thesis: the major components and their relationships:

1. CF and CF-tree: concepts, theorems and algorithms;
2. BIRCH data clustering algorithm based on CF-tree;
3. Applications of BIRCH data clustering algorithm.

Chapter 2

Background

2.1 Metric Attributes

The major focus of this chapter is to show all the measurements that are used in the BIRCH clustering algorithm. Generally, there are two types of attributes involved in the dataset: *metric* and *nonmetric*. Informally, a *metric* attribute is one whose values can be represented or mapped by explicit coordinates in a *Euclidean* vector space; a *nonmetric* attribute is one whose values can not be represented or mapped by explicit coordinates in a *Euclidean* vector space. In this thesis, BIRCH considers metric attributes only, just as KMEANS does. So each tuple of d attributes can be represented as a d-dimensional vector. I assume that the readers are familiar with the terminology of the d-dimensional vector space, such as vector addition, vector subtraction, and vector dot product[12]. Following are the definitions of various measurements that will be used in the descriptions of algorithms in this thesis.

2.2 Measurements for one cluster

We begin by defining centroid, radius and diameter for one cluster. Given N d-dimensional data points in a cluster: $\{\vec{X_i}\}$ where $i = 1, 2, ..., N$, the **centroid** $\vec{X0}$, **radius** R and **diameter** D of the cluster are defined as:

$$\vec{X0} = \frac{\sum_{i=1}^{N} \vec{X_i}}{N} \tag{2.1}$$

$$R = [\frac{\sum_{i=1}^{N} (\vec{X_i} - \vec{X0})^2}{N}]^{\frac{1}{2}} \tag{2.2}$$

$$D = [\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(\vec{X_i} - \vec{X_j})^2}{N(N-1)}]^{\frac{1}{2}} \tag{2.3}$$

In the above definitions, R measures the average distance from member points of a cluster to the centroid; D measures the average pairwise distance within a cluster. They are two alternative measurements of the scatterness of the cluster around the centroid.

2.3 Measurements between two clusters

Next between two clusters, we define 5 alternative distances for measuring their closeness.

Given the centroids of two clusters: $\vec{X0_1}$ and $\vec{X0_2}$, the **centroid Euclidean distance** $D0$ and **centroid Manhattan distance** $D1$ of the two clusters are defined as:

$$D0 = [(\vec{X0_1} - \vec{X0_2})^2]^{\frac{1}{2}} \tag{2.4}$$

$$D1 = |\vec{X0_1} - \vec{X0_2}| = \sum_{i=1}^{d} |\vec{X0_1}^{(i)} - \vec{X0_2}^{(i)}| \tag{2.5}$$

Given N_1 d-dimensional data points in a cluster: $\{\vec{X_i}\}$ where $i = 1, 2, ..., N_1$, and N_2 data points in another cluster: $\{\vec{X_j}\}$ where $j = N_1+1, N_1+2, ..., N_1+N_2$, the **average inter-cluster distance** $D2$, **average intra-cluster distance** $D3$ and **variance increase distance** $D4$ of the two clusters are defined as:

$$D2 = [\frac{\sum_{i=1}^{N_1}\sum_{j=N_1+1}^{N_1+N_2}(\vec{X_i}-\vec{X_j})^2}{N_1 N_2}]^{\frac{1}{2}} \tag{2.6}$$

$$D3 = [\frac{\sum_{i=1}^{N_1+N_2}\sum_{j=1}^{N_1+N_2}(\vec{X_i}-\vec{X_j})^2}{(N_1+N_2)(N_1+N_2-1)}]^{\frac{1}{2}} \tag{2.7}$$

$$D4 = [\sum_{k=1}^{N_1+N_2}(\vec{X_k} - \frac{\sum_{l=1}^{N_1+N_2}\vec{X_l}}{N_1+N_2})^2 - \sum_{i=1}^{N_1}(\vec{X_i} - \frac{\sum_{l=1}^{N_1}\vec{X_l}}{N_1})^2 - \sum_{j=N_1+1}^{N_1+N_2}(\vec{X_j} - \frac{\sum_{l=N_1+1}^{N_1+N_2}\vec{X_l}}{N_2})^2]^{\frac{1}{2}} \tag{2.8}$$

The average intra-cluster distance $D3$ is actually the diameter D of the merged cluster. For the sake of clarity, we treat $\vec{X0}$, R and D as properties of a single cluster, and $D0$, $D1$, $D2$, $D3$ and $D4$ as properties between two clusters, and state them separately.

2.4 Measurements of Clustering Quality

Assume N data points are clustered into K clusters. Cluster i, $i = 1, ..., K$, contains N_i data points, has a radius of R_i and a diameter of D_i. $\sum_{i=1}^{K} N_i = N$. We define the following 4 alternative clustering quality measurements: **weighted average cluster radius square** $Q1$ (or $\bar{R}$), **weighted average cluster diameter square** $Q2$ (or $\bar{D}$), **weighted total cluster radius square** $Q3$, and **weighted total cluster diameter square** $Q4$ as below.

$$Q_1 = \frac{\sum_{i=1}^{K} N_i R_i^2}{\sum_{i=1}^{K} N_i} \tag{2.9}$$

$$Q_2 = \frac{\sum_{i=1}^{K} N_i (N_i - 1) D_i^2}{\sum_{i=1}^{K} N_i (N_i - 1)} \tag{2.10}$$

$$Q_3 = \sum_{i=1}^{K} N_i R_i^2 \tag{2.11}$$

$$Q_3 = \sum_{i=1}^{K} N_i (N_i - 1) D_i^2 \tag{2.12}$$

2.5 Weighting and Shifting

We can optionally pre-process data by weighting and/or shifting the data along different dimensions without affecting the relative placement of data points (That is if point A is left to point B, then after weighting and shifting, point A is still left to point B.). For example, to normalize the data, one can shift the data by the mean value along each dimension, and then weight the data by the inverse of the standard deviation on

each dimension. In general, such data pre-processing is a debatable semantic issue. On one hand, it avoids biases caused by some dimensions. For example, the dimensions with large spread dominate the distance calculations in the clustering process. On the other hand, it is inappropriate if the spread is indeed due to natural differences of clusters. Since prep-rocessing the data in such a manner is considered independent of the clustering algorithm in this thesis, we will assume that the user who uses the data clustering algorithm on his/her data knows how to pre-process the data.

Chapter 3

CF and CF-tree

BIRCH chooses to summarize a dataset as a set of subclusters to reduce the scale of the problem. So the following questions about the summarization follow immediately:

1. How much summary information should be kept for each subcluster?
2. How is the summary information organized?
3. How efficiently is the organization maintained?

This whole chapter is devoted to address the above questions while it presents the **Clustering Feature** (CF) concept, the **CF-tree** structure, as well as their related algorithms and theorems. They are at the core of BIRCH's incremental and scalable computation.

3.0.1 Clustering Feature (CF)

A **Clustering Feature** (CF) is a triple summarizing the information that we maintain about a subcluster of data points: (1) the number of data points, (2) the linear sum of data points, and (3) the square sum of data points.

Definition 1 (CF Definition): *Given N d-dimensional data points in a cluster:* $\{\vec{X_i}\}$ *where* $i = 1, 2, ..., N$*, the* **Clustering Feature** *(CF) entry of the cluster is defined as a triple:* $\mathbf{CF} = (N, \vec{LS}, SS)$*, where N is the number of data points in the cluster,* $\vec{LS}$ *is the linear sum of the N data points, i.e.,* $\sum_{i=1}^{N} \vec{X_i}$*, and SS is the square sum of the N data points, i.e.,* $\sum_{i=1}^{N} \vec{X_i}^2$*.*

This triple not only summarizes the distribution inside the subcluster, but also can be used to measure the closeness between two subclusters, and to measure the overall

clustering quality. Following is the theorem with respect to the representativity of the CF entries.

3.0.2 CF Representativity

Theorem 2 (CF Representativity Theorem): *Given the CF entries of clusters, all the measurements defined in Chapter 2 can be computed accurately.*

The proof is given in Appendix A with some mathematical transformations of the definition formulae. So one can think of a subcluster as a set of data points, but only the CF entry is stored as the summary. This CF summary is not only efficient because it takes much less space than all the data points in the subcluster, but also sufficient because it is enough to support computing all the measurements that we need for making decisions in our system accurately.

The following theorem with respect to the additivity of the CF entries further shows that the CF entries of subclusters can be stored and computed incrementally and accurately as clusters are merged, or new data points are inserted.

3.0.3 CF Additivity

Theorem 3 (CF Additivity Theorem): *Assume that* $\mathbf{CF_1} = (N_1, \vec{LS_1}, SS_1)$, *and* $\mathbf{CF_2} = (N_2, \vec{LS_2}, SS_2)$ *are the CF vectors of two disjoint clusters, where disjoint means a data point can not belong to more than one cluster at the same time. Then the CF vector of the cluster that is formed by merging the two disjoint clusters, is:*

$$\mathbf{CF_1} + \mathbf{CF_2} = (N_1 + N_2, \vec{LS_1} + \vec{LS_2}, SS_1 + SS_2) \tag{3.13}$$

The proof consists of straightforward algebra, and hence is omitted.

3.1 CF-tree

A height-balanced tree structure is used to organize the CF entries of subclusters into levels based on their containment relationship for efficient insertions.

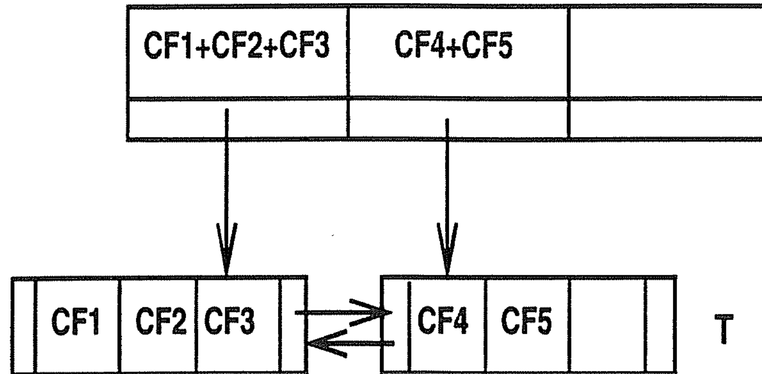


Figure 2: A CF-tree Example

3.1.1 CF-tree Definition

A **CF-tree** is a height-balanced tree with two parameters: branching factor (B for nonleaf node and L for leaf node) and threshold T. It has two types of nodes: nonleaf node and leaf node. A nonleaf node contains at most B entries of the form $[CF_i, child_i]$, where $i = 1, 2, ..., B$, "$child_i$" is a pointer to its i-th child node, and CF_i is the CF of the subcluster represented by its i-th child. So a nonleaf node represents a cluster made up of all the subclusters represented by its entries. A leaf node contains at most L entries, each is an entry of the form $[CF_i]$, where $i = 1, 2, ..., L$, and CF_i is the CF of its i-th subcluster. In addition, each leaf node has two pointers, "*prev*" and "*next*" which are used to chain all leaf nodes together for efficient scans. So A leaf node also represents a cluster made up of all the subclusters represented by its CF entries. But all CF entries in a leaf node must satisfy a *threshold requirement* with respect to a threshold value T. In this thesis, we have decided to define the threshold requirement as: *the diameter of each leaf CF entry has to be less than T.* Whereas in general, there is no reason that we can not define the threshold requirement in other ways to make the CF-tree more versatile.

Figure 2 shows a sample CF-tree with height of 2, $B = L = 3$. The diameters of leaf entry CF1 through CF5 must all satisfy the threshold value T.

With the above CF-tree definition, the tree size will be a function of the threshold value T. The larger T is, the smaller the tree size should be. We require a node (nonleaf

or leaf) to fit in a page of size P for efficient access reason. Once the dimension d of the vector space is given, the sizes of leaf and nonleaf entries are known, then B and L are determined by the page size P. So P can be varied for performance tuning.

Such a CF-tree will be built dynamically as new data objects (points or CF entries) are inserted. It is used to guide a new insertion into the correct subcluster (leaf entry) for clustering purposes just the same as a B+-tree is used to guide a new insertion into the correct position for sorting purposes. But CF-tree is a very compact representation of the dataset because each entry in a leaf node is not a single data point but a subcluster (which absorbs as many data points as the threshold requirement allows)

3.1.2 Insertion Algorithm

We now present the algorithm for inserting an entry 'Ent' into a CF-tree. Here 'Ent' is represented by its CF, and it can be either a single data point, or a subcluster of data points depending on whether a new data point is inserted or an existing leaf entry is re-inserted. The insertion algorithm proceeds in the steps as shown below:

1. *Identifying the appropriate leaf node:* Starting from the root, it recursively descends the CF-tree by choosing the **closest** child node according to a chosen distance metric: $D0$,$D1$,$D2$,$D3$ or $D4$ as defined in Chapter 2 and computed from CF entries on each level of the CF-tree.

2. *Modifying the leaf node:* When it reaches a leaf node, it finds the **closest** leaf entry, say L_i, and then tests whether L_i can **absorb** "Ent" without violating the threshold requirement. That is, the cluster merged with "Ent" and L_i must still satisfy the threshold requirement. Note that the CF entry of the new cluster can be computed from the CF entries for L_i and "Ent". If so, the CF entry for L_i is updated to reflect this. If not, a new entry for "Ent" is added to the leaf node. If there is space on the leaf node for this new entry to **fit in**, we are done, otherwise we must **split** the leaf node. Node splitting is done by choosing the **farthest** pair of leaf entries as seeds, and redistributing the remaining leaf entries based on the

closest criteria.

3. *Modifying the path from the leaf node to the root:* After inserting "Ent" into a leaf node, we must update each nonleaf entry on the path from the root leading to the leaf node. In the absence of a split, this simply involves adding CF entries to reflect the addition of "Ent". A leaf node split requires us to insert a new nonleaf entry into the parent node, to describe the newly created leaf node. If the parent has space for this entry to **fit in**, then at all higher levels, we only need to update the CF entries to reflect the addition of "Ent". In general, however, we may have to **split** the parent as well, and so on up to the root. If the root is split, the tree height will increase by one and a new root will be created.

4. *A Merging refinement when the split stops:* Splits are caused by the page size, which is independent of the clustering properties of the data. In the presence of skewed data input order, this can affect the clustering quality, and also reduce space utilization. A simple additional merging step often helps ameliorate these problems: Suppose that there is a leaf node split, and the propagation of this split stops at some nonleaf node N_j, i.e., N_j can accommodate the additional entry resulting from the split. We now scan node N_j to find the **closest** pair of entries but not the pair corresponding to the split, then we try to merge them and the corresponding two child nodes. If there are more entries in the two child nodes than one page can hold, we resplit the merging result again. During the resplitting, in case one of the seed attracts enough merged entries to fill a page, we just put the rest entries with the other seed. In summary, if the merged entries fit in a single page, we free a node space for later use, create one more entry space in node N_j, thereby increasing space utilization and postponing future splits; otherwise we improve the distribution of entries in the closest two children to reduce the effect of data input order.

The pseudo-code of the insertion algorithm is given in Appendix B.

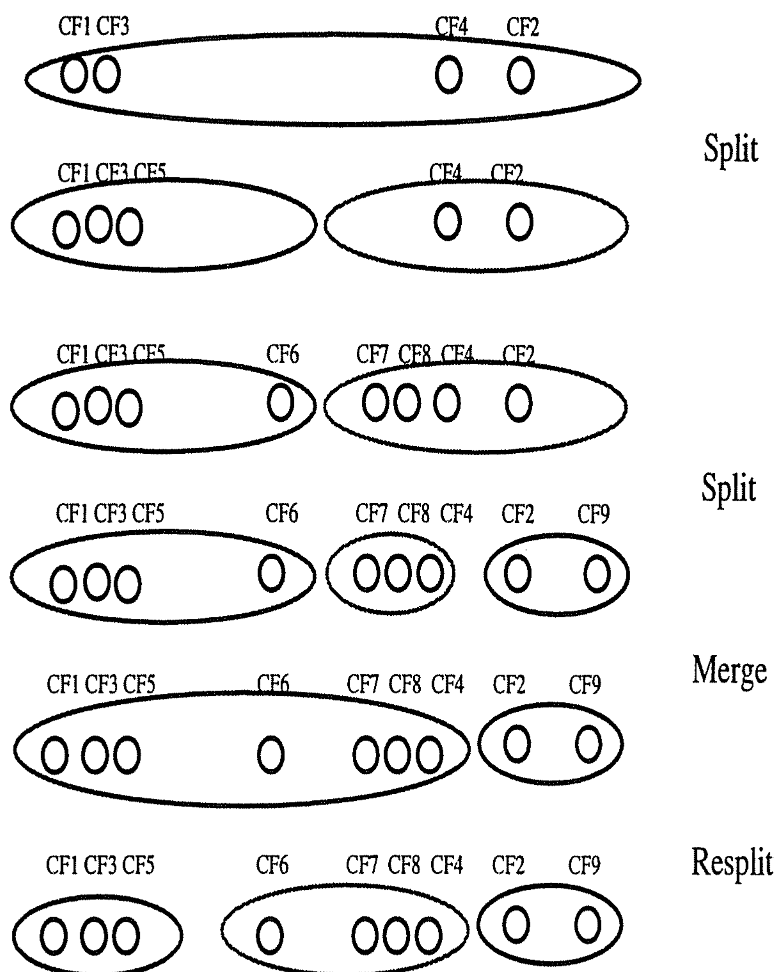


Figure 3: Effect of Split, Merge and Resplit

Figure 3 gives some intuition of the dynamic adjusting effects of the split, merge, and perhaps resplit mechanisms. Imagine the large ovals as leaf nodes (or nonleaf entries) and the small ovals as leaf entries. Assume each leaf node can only hold at most four leaf entries, and the leaf entries are created in the order as labeled: CF1 through CF9. Following is the steps of how the leaf nodes and entries are created:

1. As data is inserted, CF1 to CF4 are created in the same leaf node.
2. With more insertion, CF5 is created which causes the single leaf node to split into two leaf nodes.
3. With more insertion, CF6 to CF8 are created in the two leaf nodes.
4. With more insertion, CF9 is created which causes the right side leaf node to split into two leaf nodes.
5. The closest pair but not just split pair of leaf nodes (or nonleaf entries) are merged.
6. The merging causes overflow and needs to be resplit immediately.

From the above steps, one can see that the split, merge and perhaps resplit mechanisms work together to dynamically adjust the CF-tree to reduce its sensitivity to the data input order.

3.1.3 Anomalies

Since each node can only hold a limited number of entries due to its size, it does not always correspond to a natural cluster. Occasionally, two subclusters that should have been in one cluster are split across nodes. Depending upon the order of data input and the degree of skew, it is also possible that two subclusters that should not be in one cluster are kept in the same node. These infrequent but undesirable anomalies caused by node size limit (anomaly1) will be addressed with a global clustering algorithm discussed in Section 4.3.4.

Another undesirable artifact is that if the same data point is inserted twice, but at different times, the two copies might be entered into two distinct leaf entries. Or, in another word, occasionally with a skewed input order, a point might enter a leaf entry that it should not have entered (anomaly2). This problem will also be addressed with a refining algorithm discussed in Section 4.3.5.

3.1.4 Rebuilding Algorithm

Following we discuss how to compact (or rebuild) the CF-tree by increasing the threshold if the CF-tree size limit is exceeded as more and more data points are inserted.

Assume t_i is a **CF-tree** of threshold T_i. Its height is h, and its size (number of nodes) is S_i. Given $T_{i+1} \geq T_i$, we want to use all the leaf entries of t_i to rebuild a **CF-tree**, t_{i+1}, of threshold T_{i+1} such that the size of t_{i+1} should not be larger than S_i. Following is the rebuilding algorithm as well as the consequent reducibility theorem.

Assume within each node of CF-tree t_i, the entries are labeled contiguously from 0 to $n_k - 1$, where n_k is the number of entries in that node, then a **path** from an entry in the root (level 1) to a leaf node (level h) can be uniquely represented by $(i_1, i_2, ..., i_{h-1})$, where $i_j, j = 1, ..., h-1$ is the label of the j-th level entry on that path. So naturally, path $(i_1^{(1)}, i_2^{(1)}, ..., i_{h-1}^{(1)})$ is **before (or $<$)** path $(i_1^{(2)}, i_2^{(2)}, ..., i_{h-1}^{(2)})$ if $i_1^{(1)} = i_1^{(2)}$, ..., $i_{j-1}^{(1)} = i_{j-1}^{(2)}$, and $i_j^{(1)} < i_j^{(2)} (0 \leq j \leq h-1)$. It is obvious that each leaf node corresponds to a path uniquely, and we will just use path and leaf node interchangeably from now on.

The idea of the rebuilding algorithm is illustrated in Figure 4. With the natural path order defined above, it scans and frees the old CF-tree path by path, and at the same time, creates the new CF-tree path by path. The new CF-tree starts with NULL, and "OldCurrentPath" starts with the leftmost path in the old CF-tree. For the "OldCurrentPath", the algorithm proceeds as the steps shown below:

1. *Create the corresponding "NewCurrentPath" in the new CF-tree*: nodes are added to the new CF-tree exactly the same as in the old CF-tree, so that there is no chance that the new CF-tree ever becomes larger than the old CF-tree.

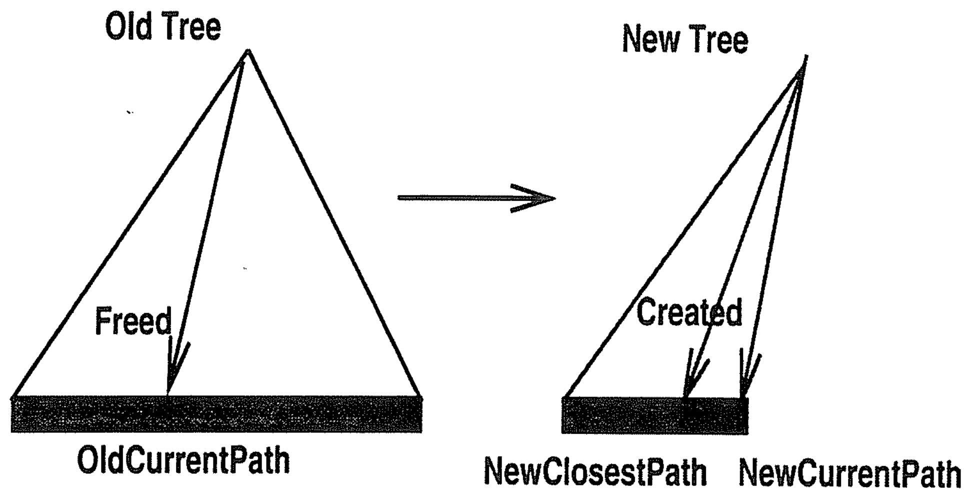


Figure 4: Rebuilding CF-tree

2. *Insert leaf entries in "OldCurrentPath" to the new CF-tree*: with the new threshold, each leaf entry in "OldCurrentPath" is tested against the new CF-tree to see if it can either be **absorbed** by an existing leaf entry, or **fit in** as a new leaf entry without splitting, in the "NewClosestPath" that is found top-down with the **closest** criteria in the new CF-tree. If yes and "NewClosestPath" is **before** "NewCurrentPath", then it is inserted to "NewClosestPath", and the space in "NewCurrentPath" is left available for later use; otherwise it is inserted to "NewCurrentPath" without creating any new node.

3. *Free space in "OldCurrentPath" and "NewCurrentPath"*: Once all leaf entries in "OldCurrentPath" are processed, the un-needed nodes along "OldCurrentPath" can be freed. It is also likely that some nodes along "NewCurrentPath" are empty because leaf entries that originally correspond to this path are now "pushed forward". In this case the empty nodes can be freed too.

4. *"OldCurrentPath" is set to the next path in the old CF-tree if there still exists one, and repeat the above steps.*

From the rebuilding steps, all old leaf entries can be re-inserted, but the new CF-tree can never become larger than the old CF-tree. Since only nodes corresponding to

"OldCurrentPath" and "NewCurrentPath" need to exist simultaneously, the maximal extra space needed for the CF-tree transformation is h (height of the old CF-tree) pages. So by increasing the threshold value T, we can rebuild a smaller CF-tree with a very limited amount of extra memory.

The pseudo-code of the rebuilding algorithm is given in Appendix C. Following is the reducibility theorem that follows from the rebuilding algorithm immediately.

3.1.5 Reducibility

Theorem 4 Reducibility Theorem: *Assume we rebuild CF-tree t_{i+1} of threshold T_{i+1} from CF-tree t_i of threshold T_i by the above algorithm, and let S_i and S_{i+1} be the sizes of t_i and t_{i+1} respectively. If $T_{i+1} \geq T_i$, then $S_{i+1} \leq S_i$, and the transformation from t_i to t_{i+1} needs at most h extra pages of memory, where h is the height of t_i.*

The proof follows directly from the rebuilding algorithm, and hence is omitted. With the rebuilding algorithm as well as the relevant reducibility theorem, it is guaranteed that once we run out of memory, by increasing the threshold, we can always compact the CF-tree to make space for accepting more insertions of new data points.

3.1.6 CF-tree versus Multi-Dimensional Histogram

CF-tree can be thought as a dynamic way of grouping data points into buckets (leaf entries). With the current threshold requirement, CF-tree is similar to equi-width multi-dimensional histogram except that it has the following advantages:

1. It is organized as a balanced hierarchy, so the insertion cost is always bounded.

2. It allocates a "bucket" for a region only when there is indeed data points falling inside that region. So the number of "buckets" grows only if the data distribution requires.

3. It stores more information for each "bucket" (count, linear sum, and square sum) which allows the bucket to evolve automatically in terms of location and spreading.

4. It grows linearly instead of exponentially with the dimension.

3.1.7 Generalizing CF-tree

The definition of CF-tree can be generalized in several directions to make it a more informative and/or more flexible data summarization. Following are a few instances:

1. In the current CF definition, the square sum is a scalar value which collapses the square sums on all dimensions. It can be generalized as a vector which maintains all individual square sums on all dimensions to better describe the data distributions.

2. The current threshold requirement is a limit set for radii or diameters of leaf entries. It, by no means, is the only way to control the CF-tree size. Other kinds of threshold requirements (e.g., limit set for density of leaf entries) and related insertion and rebuilding algorithms should be studied in the future.

3. The current threshold requirement is global for all leaf entries. Whereas localized threshold requirements and nested CF-tree structures should be explored in the future.

4. The current CF-tree only works for *metric* attributes, how to generalize it to make it work for *nonmetric* attributes is also a good topic for the future.

Chapter 4

Data Clustering

The previous chapter illustrates that the CF-tree structure provides an in-memory summarization of a large dataset. It is especially convenient for data clustering analysis. In this chapter, we will focus on the subject of data clustering: previous work and their problems, and how it is approached in BIRCH by using the CF-tree.

4.1 Previous Work

Data clustering has been studied in the statistics [12, 13, 41, 44], machine learning [9, 22, 23, 25, 40, 52], and database [45, 16, 17] communities with different methods and different emphases.

4.1.1 Probability-Based

Previous data clustering work in machine learning is usually referred to as unsupervised conceptual learning [9, 22, 25, 40]. They concentrate on *incremental* (i.e., accept instances one at a time, and do not extensively reprocess previously encountered instances while incorporating the new one.) *concept (or cluster)* formation by means of *top-down "sorting"* each new instance through a *hierarchy* whose nodes are formed gradually to represent concepts. They are usually *probability-based* approaches, that is, (1) they use probabilistic measurements (e.g., *category utility* as discussed in[22, 25]) for making decisions; and (2) they represent concepts (or clusters) with probabilistic descriptions.

For example, COBWEB[22] proceeds this way: to insert a new instance into the hierarchy, it starts from the root, then it has four choices at each level of the hierarchy while it descends: one for recursively incorporating the instance into an existing node,

another for creating a new node for the instance, a third for merging two nodes to host the instance and a final one for splitting out a node to host the instance. The choice that results in the highest *category utility* score will be selected. Its explicit merging and splitting operations are desirable means for an incremental algorithm to recover from earlier nonrepresentative instances. However it has the following limitations.

- It is targeted for handling discrete attributes and the *category utility* measurement used is very expensive to support. To compute the *category utility*, in each node it stores a *discrete probability distribution* for *each individual attribute*. That means that it has to assume the probability distributions on separate attributes are statistically independent, and it has to store a probability for every possible attribute value. So it ignores the correlations among attributes, and makes updating and storing the concept very expensive, especially if the attributes have a large number of values. It works only for discrete attributes, and for a continuous attribute, one has to divide the attribute into ranges, or "discretize" in advance.

- All instances ever encountered are retained as terminal nodes in the hierarchy. For very large datasets, physically, storing and manipulating such a large hierarchy is infeasible; and fundamentally it has been shown that this kind of large hierarchy tends to "overfit" the data[25]. A related problem is that this hierarchy is not width-balanced or height-balanced. So in case of skewed input data, this may cause the performance to degrade dramatically.

For another example, CLASSIT[25] is very similar to COBWEB except for the following four major aspects:

1. It only deals with *continuous* (or real-valued) attributes (instead of discrete attributes in COBWEB).

2. It stores a *continuous normal distribution* (i.e., mean and standard deviation) for each individual attribute in a node (instead of a discrete probability distribution in COBWEB).

3. As it classifies a new instance, it can halt at some higher-level node if the instance is "similar enough" to the node. (instead of always descending to a terminal node as in COBWEB).

4. It modifies the category utility measurement to be in integral format for continuous attributes (instead of in sum format for discrete attributes as in COBWEB).

4.1.2 Distance-Based

Most data clustering algorithms in statistics are distance-based approaches, that is,

1. they assume that the distance measurement between any two instances (or data points) exists, and can be used for making decisions; and

2. they represent clusters by some kind of centers.

There are two categories of distance-based clustering algorithms [35]: *Partitioning Clustering* and *Hierarchical Clustering* algorithms. Partitioning Clustering (PC)[12, 35] starts with an initial partition, then tries all possible moving or swapping of data points from one group to another iteratively to optimize the objective measurement function. Each cluster is represented either by the *centroid* of the cluster (KMEANS algorithms), or by one object centrally located in the cluster (KMEDOIDS algorithms). It guarantees to converge to a local minimum, but the quality of the local minimum is very sensitive to the initial partition, and the worst case time complexity is exponential. Hierarchical Clustering (HC)[12, 44] does not try to find the "best" clusters, instead it keeps merging (agglomerative algorithms) the closest pair or splitting (divisive algorithms) the farthest pair of objects to form the desired number of clusters. With a reasonable distance measurement, the best time complexity of a practical HC algorithm is $O(N^2)$.

In summary, all distance-based clustering algorithms assume that all data points are given in advance and can be stored in memory and scanned frequently (non-incremental). They totally or partially ignore the fact that not all data points in the dataset are equally important with respect to the clustering purpose, and that data points which are close

and dense can be considered collectively instead of individually. They are *global* or *semi-global* methods at the granularity of data points. That is, for each clustering decision, they inspect all data points or all currently existing clusters equally no matter how close or far away they are, and they use global measurements, which require scanning all data points or all currently existing clusters to compute. Hence none of them has linear time scalability with stable quality.

Data clustering has been recognized as a useful spatial data mining method recently. [45] presents CLARANS, which is a KMEDOIDS algorithm but with randomized partial search strategy, and proposes that CLARANS out-performs the traditional KMEDOIDS algorithms. The clustering process in CLARANS is formalized as searching a graph in which each node is a K-partition represented by K medoids, and two nodes are neighbors if they only differ by one medoid. CLARANS starts with a randomly selected node. For the current node, it checks at most the *maxneighbor* number of neighbors randomly, and if a better neighbor is found, it moves to the neighbor and continues; otherwise it records the current node as a *local minimum*, and restarts with a new randomly selected node to search for another *local minimum*. CLARANS stops after the *numlocal* number of the so-called *local minima* have been found , and returns the best of them. CLARANS suffers from the same drawbacks as the KMEDOIDS method with respect to efficiency. In addition, it may not find a real local minimum due to the random searching trimming controlled by *maxneighbor*.

R-tree [27] (or later R^*-tree [8]) is a popular dynamic multi-dimensional spatial index structure that existed in database community for more than a decade. It is a height-balanced tree with index records in its nodes containing (1) pointers to data objects (for leaf) and pointers to child nodes (for nonleaf), and (2) MBR's (Minimum Bounding Rectangles) summarizing all MBR's (for nonleaf) or data objects (for leaf) underneath. It is built dynamically with insertions and deletions intermixed. Based on the spatial locality in R^*-tree (a variation of R-tree), [16] and [17] propose focusing techniques to improve CLARANS's ability to deal with very large datasets that may reside on disks by (1) clustering a sample of the dataset that is drawn from each R^*-tree data page; and

(2) focusing on relevant data points for distance and quality updates. Their experiments show that the time is improved but with a small loss of quality.

4.2 Contributions and Limitations of BIRCH

In this thesis, CF-tree, as we have discussed its details in Chapter 3, is strongly influenced by the dynamic nature of R-tree in terms of top-down "sorting" a new insertion through the tree, splitting an overflow node, bottom-up propagating the node splitting to balance the tree. It is also influenced by the incremental and hierarchical themes of of COBWEB as well as the way of using splitting and merging to alleviate the potential sensitivity to the data input ordering.

However there are many aspects that distinguish BIRCH from others. First, all previous work do not recognize that the problem must be viewed in terms of how to work with a limited amount of resources (e.g., memory and running time) to do the clustering as accurately as possible. So the most important contribution of BIRCH is the *formulation* of the clustering problem in a realistic way that is appropriate for very large datasets by making the time and memory constraints explicit.

Another contribution is that BIRCH exploits the observation that the data space is usually not uniformly occupied, and hence not every data point is equally important for clustering purposes. So BIRCH treats a dense region of points (or a subclusters) collectively by storing a compact *summarization* (clustering feature as discussed in Chapter 3). This way, BIRCH reduces the problem of clustering the original data points into a simpler and smaller one of clustering the clustered subclusters. So "clustering" here actually has two levels of meanings: first summarizing the data with subclusters obtained from clustering, and second clustering the subclusters instead of the original data points.

The introduction of *Clustering Feature* (CF), *CF-tree* and related algorithms, theorems and proofs illustrates that the summarizations (or clustering features) for subclusters identified by BIRCH reflect the natural *closeness* of data, allow for the computation of either distance-based measurements (as defined in Chapter 2) that are currently used

in BIRCH, or probability-based measurements (e.g., mean, standard deviation, and category utility as used in CLASSIT) that we might to explore in the near future. They can be maintained consistently and incrementally in the CF-tree structure. They contain more details about the dataset than the MBR's of a R-tree or R^*-tree do.

Compared with the distance-based algorithms, BIRCH is *incremental* in the sense that each clustering decision is made without scanning all data points or all currently existing clusters. If we omit the optional Phase 4 (Section 4.3.5), BIRCH is an incremental method that does not require the whole dataset in advance, and only scans the dataset once. Compared with the probability-based algorithms, BIRCH tries to make the best use of the available resources to derive the finest possible subclusters (to ensure accuracy) while minimizing I/O costs (to ensure efficiency) by organizing the clustering and reducing process as an in-memory balanced tree structure. Most importantly, BIRCH does not have to assume that the probability distributions on individual attributes are statistical independent.

At this stage, one major limitation of BIRCH is that it can only handle *metric* attributes (pretty similar to the kind of attributes that KMEANS and CLASSIT can handle).

4.3 BIRCH Clustering Algorithm

4.3.1 Overview

Figure 5 presents the overview of the clustering algorithm in BIRCH. In this section, we concentrate on describing the role of each phase and the relationships between the phases. The details of each phase will be discussed in several sub-sections.

The BIRCH clustering algorithm consists of four phases: (1) Loading, (2) Optional Condensing, (3) Global Clustering, and (4) Optional Refining. The main task of Phase 1 is to load the clustering information into memory: That is, to scan and insert all data points to build an initial in-memory CF-tree with a given amount of memory. With every

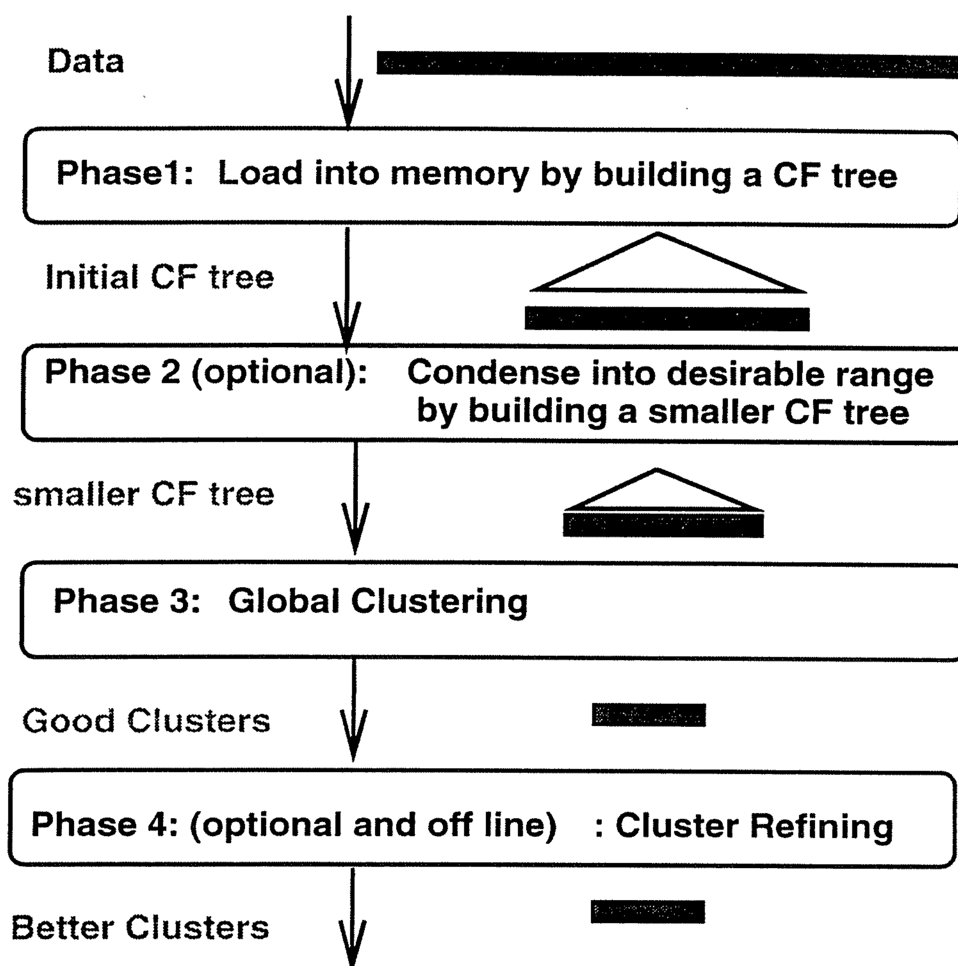


Figure 5: Overview of BIRCH Clustering Algorithm

data point checked, and crowded data points grouped as fine subclusters, and sparse data points removed as outliers optionally, this phase tries to create an in-memory summary of the dataset, as accurate as the memory allows. Subsequent clustering computations of later phases will be:

1. fast because (a) no I/O operations are needed, and (b) the problem of clustering the original data is reduced to a smaller problem of clustering the subclusters in the leaf entries of a CF-tree;
2. still accurate because (a) a lot of outliers are eliminated, and (b) the remaining data is reflected with the finest granularity that can be achieved given the available amount of memory;
3. less order sensitive because the the leaf entries of the CF-tree form an input order containing more data locality compared with the arbitrary original data input order.

Once all the clustering information is loaded into the in-memory CF-tree, we can use an existing global or semi-global algorithm in Phase 3 to cluster all the leaf entries across the boundaries of different leaf nodes to overcome the undesirable anomaly due to the artificial node size (anomaly1 as discussed in Section 3.1.3), which causes the CF-tree nodes to be unfaithful to the actual clusters in the data. We observe that existing clustering algorithms , such as HC, KMEANS and CLARANS, working with a set of data points can be readily adapted to work with a set of subclusters, each described by its CF entry. Several points are worth noting for the adaptation:

1. We can use any of the algorithms available in the literature (e.g., *KMEANS*, *CLARANS*, *HC* etc).
2. Whatever the algorithm, it should be modified to utilize the information in the **CF** entries of the CF-tree.

3. This global or semi-global clustering, in conjunction with the high locality present in the leaf entries, further decreases the sensitivity of the final clustering quality to the data input order.

The details of how to adapt existing algorithms to work for CF entries will be discussed in Section 4.3.4.

Phase 2 is an optional phase. With experimentation, we have observed that the global or semi-global clustering methods that we adapt in Phase 3 have different input size ranges within which they perform well in terms of both speed and quality. For example, if we choose to adapt CLARANS in Phase 3, we know that CLARANS performs pretty well for a set of less than 5000 data objects because within that range, frequent data scanning is still acceptable and getting stuck at a very bad local minimum is very unlikely. So potentially there is a gap between the size of Phase 1 results and the best performance range of the Phase 3 algorithm we select. Phase 2 serves as a cushion between Phase 1 and Phase 3 and bridges this gap: So in Phase 2 we scan the leaf entries in the initial CF-tree to rebuild a smaller CF-tree, while removing more outliers optionally and grouping more crowded subclusters into larger ones.

After Phase 3, we obtain a set of clusters that captures the major distribution pattern in the data. However minor and localized inaccuracies might exist because of (1) the rare misplacement anomaly that is common to all incremental algorithm due to skewed input order (anomaly2 as mentioned in Section 3.1.3), and (2) the fact that Phase 3 is applied on a summary of the data, instead of the data itself, with some level of granularity. So Phase 4 is used to correct these minor and localized inaccuracies and refine the clusters further. It is optional and can be run off-line because it entails the cost of additional passes over the data. Note that up to this point, the original data has only been scanned once, although the CF-tree and outlier information may have been scanned multiple times.

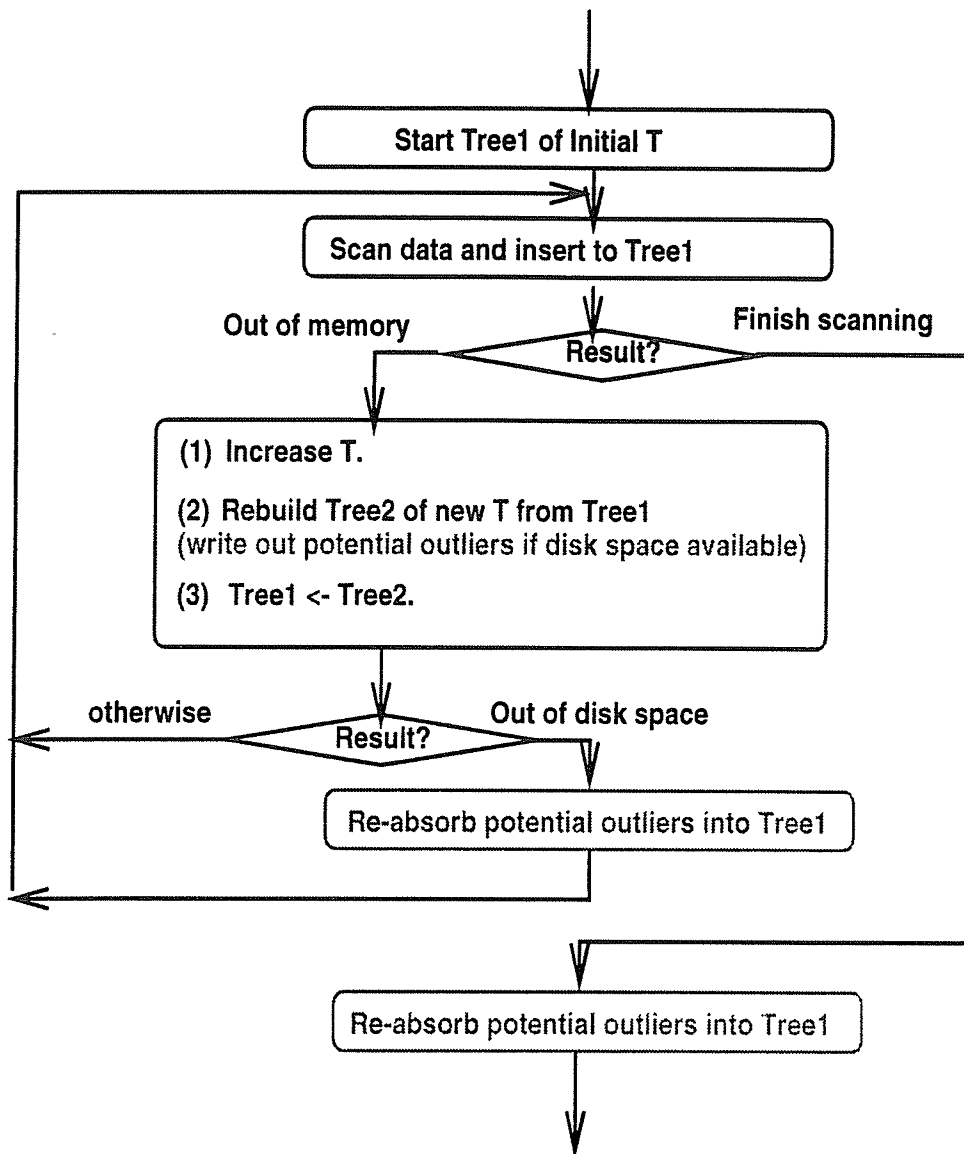


Figure 6: Flow Chart of Phase 1

4.3.2 Phase 1

Figure 6 shows the details of Phase 1. Assume that we have M bytes of memory available for building the CF-tree, and R bytes of disk space available for processing the outliers ($R = 0$ means that outlier handling option is off). Phase 1 starts with a CF-tree of a small initial threshold value, say 0, scans the data, and inserts data points into the CF-tree using the insertion algorithm described in Section 3.1.2. If it runs out of memory before it finishes scanning the data, then it increases the threshold value, rebuilds a new CF-tree of the new threshold value from the old CF-tree using the rebuilding algorithm described in Section 3.1.4. Based on the reducibility theorem associated with the rebuilding algorithm, we know that:

1. in general, by increasing the threshold, we can build a smaller CF-tree;
2. if a CF-tree uses up all available M bytes of memory, with a very limited number of reserved pages, we can transform the old CF-tree into a smaller new CF-tree of higher threshold.

During the rebuilding process, as we will discuss later, some of the old leaf entries are not re-inserted into the new CF-tree, but written out to disk temporarily as potential outliers. After all the old leaf entries have been re-inserted (or written out to disk), the scanning of the data (and insertion into the new CF-tree) is resumed from the point at which it was interrupted.

Outlier-Handling Option

Optionally, we can use the R bytes of disk space for handling *outliers*, which are leaf entries of low density that are judged to be unimportant with respect to the overall clustering pattern. (As a special case, we may have no disk pages available, i.e., $R = 0$. This is handled by not considering any leaf entry to be an outlier in Phase 1.). When we rebuild the CF-tree by re-inserting the old leaf entries, the size of the new tree is actually reduced in two ways. First, we increase the threshold value, thereby allowing

each leaf entry to "absorb" more points. Second, we treat some leaf entries as potential outliers and write them out to disk. An old leaf entry is considered to be a potential outlier if it has "far fewer" data points than the average. The number of data points in a leaf entry is known from the CF for this leaf entry. The average over all leaf entries in the tree can be calculated by maintaining the total number of data points and the total number of leaf entries in the CF-tree as we insert to the CF-tree. "Far fewer", is of course some heuristics, for example, fewer than a quartile of the average number of data points per leaf entry.

The potential outlier entries must be checked after all the data has been scanned to verify that they are indeed outliers — an increase in the threshold value or a change in the distribution due to the new data read after a potential outlier is written to disk could well mean that the potential outlier entry no longer qualifies as an outlier. Ideally, we would like to process all outliers in one pass after scanning all the input data. However, it is possible that we run out of disk space for potential outliers while re-building CF-tree t_2 from t_1, and there is still some data to be scanned. In this case, we free disk space by scanning the potential outlier entries on disk and re-absorbing them into the CF-tree. This way, the potential outliers written out before the current rebuilding pass might well be absorbed into the current CF-tree, because the threshold value has increased and/or new data has come in. This lazy, periodical attempt to free disk space by re-absorbing potential outliers absorbs all entries that can be absorbed into the current CF-tree without causing the CF-tree to grow in size. So at the end of Phase 1, if a potential outlier is not absorbed, it is very likely a real outlier. We can define heuristic outlier re-absorbing condition dynamically in terms of changes of T and changes of the amount of data scanned to avoid frequent re-absorbs of them.

Note that the entire cycle — insufficient memory triggering a rebuilding of the CF-tree, insufficient disk space triggering a re-absorbing of outliers, etc. — could be repeated several times before the dataset is fully scanned. This effort must be considered in addition to the cost of scanning the data in order to assess the cost of Phase 1 accurately.

Threshold Heuristics

A careful reader must notice that a good choice of the threshold value can greatly reduce the number of rebuilds. First for the initial threshold value T_0, since it is increased dynamically, we can adjust for its being too low. But if the initial T_0 is too high, we will obtain a less detailed CF-tree than is feasible with the available memory. So T_0 should be set conservatively. BIRCH sets it to 0, the default value; but a knowledgeable user could change this.

Then suppose that T_i turns out to be too small, and we subsequently run out of memory after N_i data points have been scanned, and C_i leaf entries have been formed (each satisfying the threshold requirement with respect to T_i). Based on the portion of the data that we have scanned and the CF-tree that we have built up so far, we need to estimate the next threshold value T_{i+1}. Currently, we have used the following two heuristic approaches:

Heuristics based on Regression

1. We try to choose T_{i+1} so that $N_{i+1} = \min(2N_i, N)$. That is, if the number of points N in the dataset is greater than $2N_i$, we choose T_{i+1} so that we can absorb only N_i additional data points before we run out of space again. In general, the number of points in the dataset may not be known ($N = \infty$ in the equation). Even if it is, we choose to estimate tree size in proportion to the data we have seen thus far. For a large dataset, this may cause additional re-builds, but successive estimates can be made with increasing knowledge of the dataset.

2. Intuitively, we want to increase threshold based on some measure of *volume*, since each leaf entry can be thought of as occupying a volume in the multidimensional space, and the threshold is a limit on the diameter of this volume.

   There are two distinct notions of volume that we use in estimating the threshold. The first is *average volume*, which is defined as $V_a = r^d$ where r is the average radius of the root cluster in the CF-tree, and d is the dimensionality of the space. Intuitively, this is a measure of the space occupied by the portion of the data seen

thus far (the "footprint" of seen data). A second notion of volume *packed volume*, which is defined as $V_p = C_i * T_i^d$, where C_i is the number of leaf entries and T_i^d is the maximal volume of a leaf entry. Intuitively, this is a measure of the actual volume occupied by the leaf clusters. Since C_i is essentially the same whenever we run out of memory (since we work with a fixed amount of memory), we can approximate V_p by T_i^d.

We make the assumption that r grows with the number of data points N_i. By maintaining a record of r and the number of points N_i, we can estimate r_{i+1} using least squares linear regression. We define the *expansion factor* $f = Max(1.0, \frac{r_{i+1}}{r_i})$, and use it as a heuristic measure of how the data footprint is growing. The use of *Max* is motivated by our observation that for most large datasets, the observed footprint becomes a constant quite quickly (unless the input order is skewed). Similarly, by making the assumption that V_p grows linearly with N_i, we estimate T_{i+1} using least squares linear regression.

3. We traverse a path from the root to a leaf in the CFtree, always going to the child with the most points in a "greedy" attempt to find the most crowded leaf node. We calculate the distance (D_{min}) between the closest two leaf entries on this leaf node. If we want to build a more condensed tree, it is reasonable to expect that we should at least increase the threshold value to D_{min}, so that these two entries can be merged.

4. We multiplied the T_{i+1} value obtained through linear regression with the expansion factor f, and adjusted it using D_{min} as follows: $T_{i+1} = Max(D_{min}, f * T_{i+1})$. To ensure that the threshold value grows monotonically, in the very unlikely case that T_{i+1} obtained thus is less than T_i then we choose $T_{i+1} = T_i * (\frac{N_{i+1}}{N_i})^{\frac{1}{d}}$. (This is equivalent to assuming that all data points are uniformly distributed in a d-dimensional sphere, and is really just a crude approximation; however, it is rarely called for.)

Heuristic based on Memory Utilization

The above heuristic approach tries to predict the correct threshold value for accommodating future unknown incoming data based on linear regressions of historical samples. The correctness of the estimated threshold value, and hence the memory utilization as well as the final clustering quality will heavily depend on both the data distribution pattern and the data input pattern. Whereas in reality, without any prior knowledge of the data, (1) neither the distribution pattern nor the input pattern is known, and (2) the input pattern is extremely hard to formalize even if it is known. So the linear regression approach may produce inaccurate estimation: too small threshold causes too many rebuilds, whereas too large threshold causes bad clustering quality. We are therefore investigating other heuristics than the linear regression prediction.

Another heuristic approach under investigation is based on the memory utilization. In this approach, instead of trying to predict for the future incoming data, we concentrate on the present CF-tree, which is a summary of the data seen thus far, to try to keep memory utilization above a constant level. The problem to solve is thus formalized as: if the current CF-tree occupies all the memory, then how to increase the threshold so that the new CF-tree, which is rebuilt from the current CF-tree, occupies approximately half of the memory. This way,

- the other half of memory will be left for accommodating the future incoming data, and no matter how the future incoming data is distributed or is input, the memory utilization is always maintained approximately above 50%; and
- only the data distribution information of the seen data, which is stored in the current CF-tree, is relevant and needed in the threshold estimation.

To accomplish the above goals, when the current CF-tree uses up all the memory, we increase the threshold value to be the average of the distances between all the "nearest pairs" of leaf entries. So in average, approximately two leaf entries will be merged into one. Then in order to calculate the distance of each "nearest pair" of leaf entries efficiently, we search only within the same leaf node locally, instead of searching all the leaf entries globally because with the CF-tree insertion algorithm, it is very likely that

the nearest neighbor of a leaf entry is in the same leaf node as that leaf entry is. This approach usually does not over-estimate the threshold value, and is hence more stable. However more sophisticated solutions of the threshold estimation problem are needed in the future.

Split-Delaying Option

Besides the outlier-handling option, BIRCH provides another split-delaying option to make the CF-tree concentrate on the dense regions and the threshold not grow too high. We observe that when we run out of main memory, it may well be the case that several additional data points can be absorbed in or fitted in the current CF-tree, without changing the threshold. However, some of the data points that we read may require us to split a leaf node in the CF-tree, since they can neither be absorbed by nor fit in any existing leaf node with the current threshold value. A simple idea is to write such data points to disk (in a manner similar to how potential outliers are written), and to proceed reading the dataset until we run out of disk space available for holding this kind of data points as well. At this point, we must change the threshold value and rebuild the CF-tree, again reading and absorbing the delayed data points that we wrote out to disk earlier (just like outliers are handled). The advantage of this approach is that in general, many more data points are absorbed by or fit in the CF-tree before we have to increase the threshold and rebuild the CF-tree. With luck, we may not have to rebuild at all! Even otherwise, we will have seen much more of the data before we have to rebuild, and this is often worth the added cost of writing out some data points (those that cannot be absorbed by or fit in) and reading them back again.

4.3.3 Phase2

Figure 7 shows the details of Phase 2. Just as Phase 1 scans all data points and builds a CF-tree in memory, Phase 2 scans all leaf entries from Phase 1 and builds a smaller CF-tree whose total number of leaf entries is under the desired range. There are several

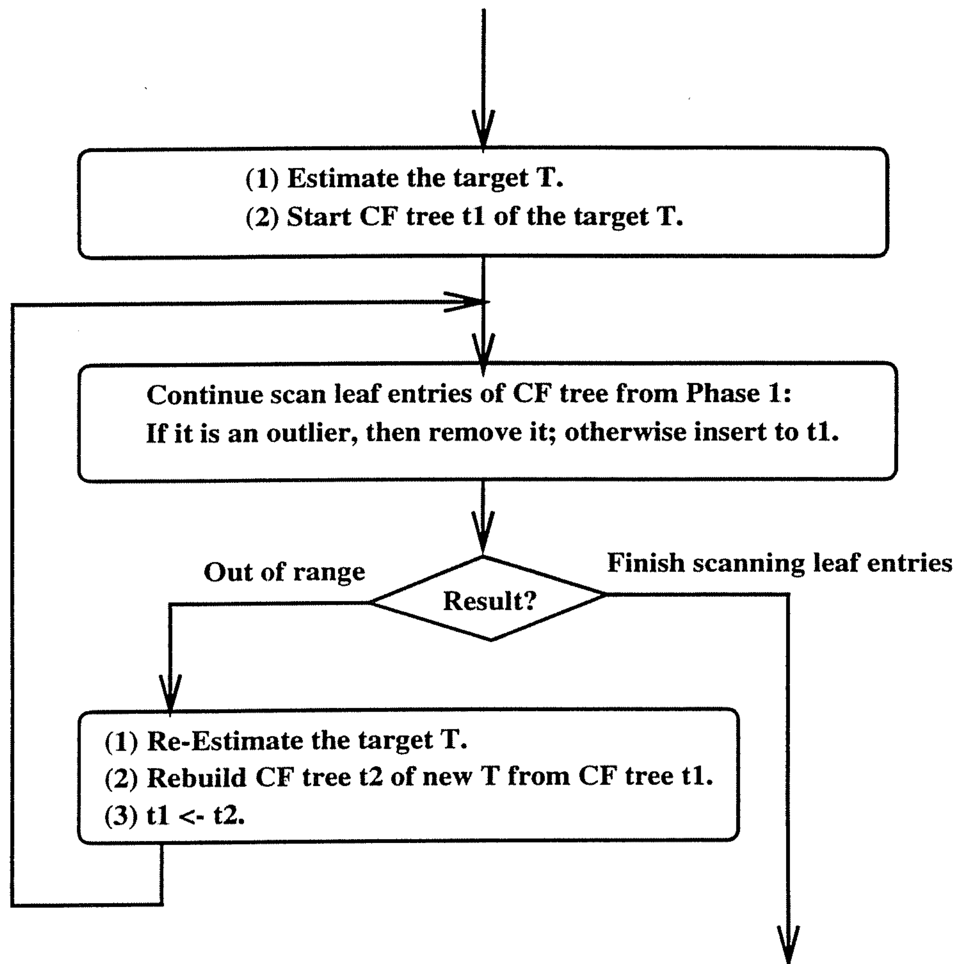


Figure 7: Flow Chart of Phase 2

unique aspects worth mentioning:

- Although in the Phase 2 flow chart, it shows the possibility of the new CF-tree in Phase 2 may run out of the desired range before all the leaf entries in the old CF-tree from Phase 1 are scanned. In that case, it has to re-estimate the target threshold and rebuild the new CF-tree just as Phase 1 did. However in reality, with all data seen and summarized in the CF-tree as well as the heuristics we have discussed, the target threshold is usually estimated very accurately. So the re-estimation and rebuilding part is seldom called for.

- Note that this additional phase further eliminates outliers — some entries that were originally entered into a leaf node may now be detected to be outliers for sure because we have seen the whole dataset.

- This phase also yields a CF-tree that is even less sensitive to the original data input order than that of Phase 1, since the leaf entries inserted are ordered by Phase 1 with very good clustering locality.

4.3.4 Phase 3

The major task of Phase 3 is to adapt the existing global or semi-global clustering methods for clustering subclusters, or CF entries instead of individual data points. With the CF entries, there are several ways to do the adaptation:

1. naively, by calculating the centroid as the representative of a CF entry, we can treat each CF entry as a single point and use an existing algorithm without further modification;

2. to be a little more sophisticated, we can treat a CF entry of n data points as its centroid repeating n times and modify an existing algorithm slightly to take the counting information into account;

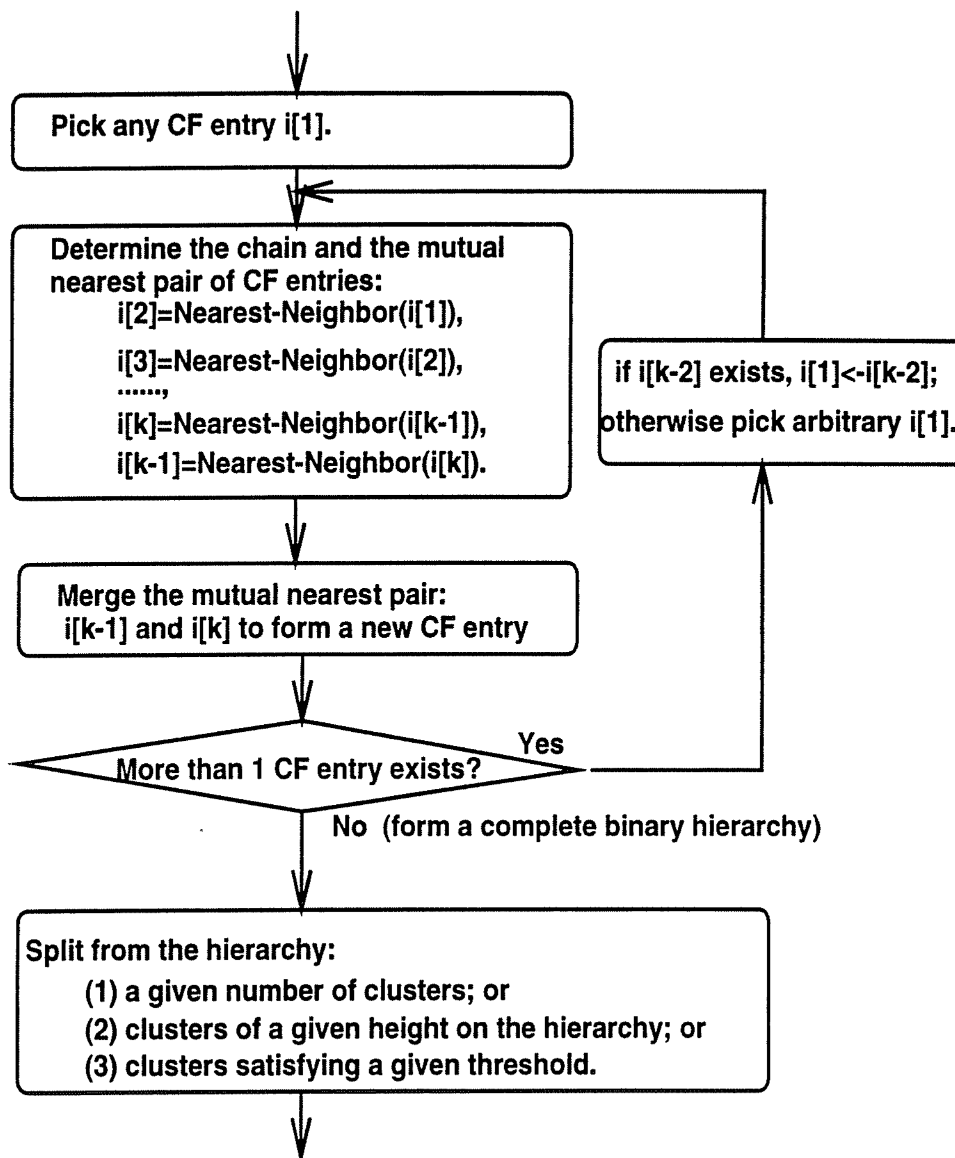


Figure 8: Flow Chart of Phase 3

3. to be general and accurate, we can apply an existing algorithm directly to the CF entries because the information in the CF entries is usually sufficient for calculating most distance and quality metrics without making any approximate assumptions.

We have adapted four algorithms for this phase:

- *HC1*: This is a $O(m^3)$ (where m is the number of CF entries) agglomerative *HC* algorithm [12] that supports all of our distance definitions: $D0$, $D1$, $D2$, $D3$ and $D4$ (as defined in Chapter 2) accurately, and allows the user to find the clusters by specifying the number of clusters, K, or the cluster diameter (or radius) threshold T; or the height of clusters on the formed hierarchy.

- *HC2*: This is an improved $O(m^2)$ agglomerative *HC* algorithm [44]. It is similar to HC1 except that it has a better complexity of $O(m^2)$, and only supports our distance definition $D2$ and $D4$ accurately.

- *CLARANS1*: The original *CLARANS* [45] is applied to the centroids of the subclusters obtained from the previous phase directly. The user has to specify the number of clusters K because *CLARANS* requires to know K in advance.

- *CLARANS2*: *CLARANS* is modified so that it can be applied to the CF entries instead of just their centroids. That is, the distance (D0, D1, D2, D3 or D4) and quality (Q1, Q2, Q3 or Q4) metrics are used in the clustering process.

Figure 8 shows the flow chart of the HC2. It consists of 2 phases: merging to form a hierarchy and splitting from the hierarchy. First it starts from an arbitrary CF entry, determines the chain of the nearest neighbor until it reaches the mutual nearest pair of CF entries. Then it merges the mutual nearest pair of CF entries into a new CF entry, and repeats either from the tail of the chain, or from a new arbitrary CF entry if the tail is empty. All the merging steps are maintained as a binary hierarchy. This merging phase will stop when there is only one CF entry left, which is the root of the binary hierarchy. Second it splits from the binary hierarchy (1) a given number of clusters; or (2) clusters

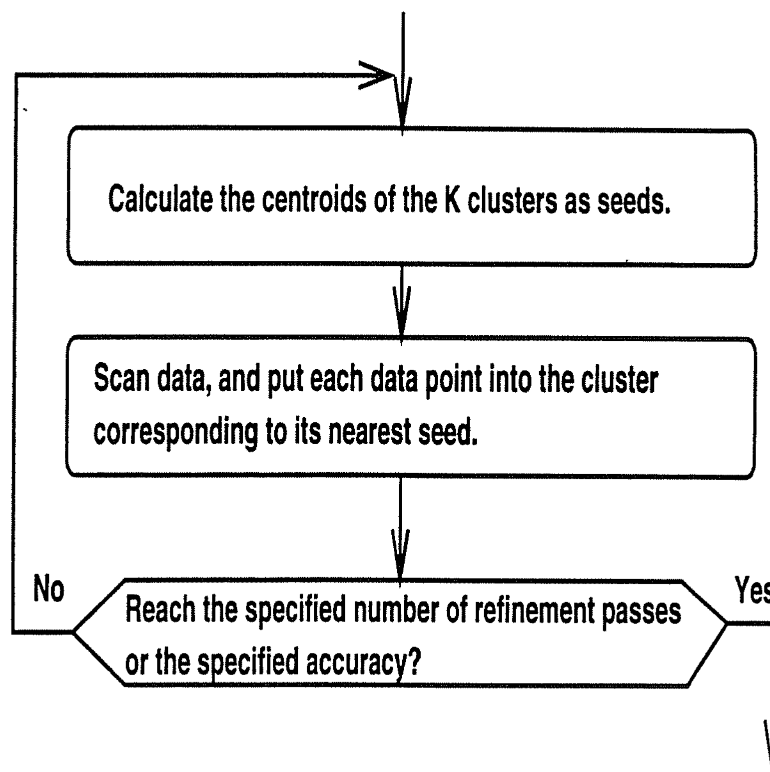


Figure 9: Flow Chart of Phase 4

of a given height on the hierarchy; or (2) clusters satisfying a given radius/diameter threshold value.

4.3.5 Phase 4

Figure 9 shows the refining algorithm of Phase 4. It uses the centroids of the clusters produced by Phase 3 as seeds, and redistributes the data points to its closest seed to obtain a set of new clusters. Not only does this allow points belonging to a cluster to migrate accordingly, but also it ensures that all copies of a given data point will go to the same cluster. Phase 4 can be extended with additional passes if desired by the user, and it has been proved to converge to a minimum [24]. As optional bonuses, during this pass,

- each data point can be labeled with the cluster that it belongs to, if we wish to identify the data points in each cluster;

- the original dataset can be filtered into subsets according to the obtained clusters;
- Phase 4 also provides us with the option of discarding outliers. That is, a point which is too far from its closest seed can be treated as an outlier and not included in the result. "Too far" again is heuristics, such as farther than twice of the cluster radius.

4.3.6 Memory Management

We have observed the following facts: The amount of memory needed for BIRCH to find a good clustering from a given dataset is determined not by the dataset size, but by the data distribution. On the other hand, the amount of memory available to BIRCH is determined in the computing system. So it is very likely that the memory needed and the memory available do not match.

If the memory available is less than the memory needed, then BIRCH can trade running time for memory. Specifically, in Phases 1 through 3, it tries to use all the available memory to generate the subclusters that is as fine as the memory allows, but in Phase 4, by refining the clustering a few more passes, it can compensate for the inaccuracies caused by the coarseness due to insufficient memory in Phases 1.

If the memory available is more than the memory needed, then BIRCH can cluster the given dataset on multiple combinations of attributes concurrently while sharing the same scan of the dataset. So the total available memory will be divided and allocated to the clustering process of each combination of attributes accordingly. This gives the user the chance of exploring the same dataset from multiple perspectives concurrently if the resources allow.

Chapter 5

Performance of BIRCH Data Clustering

In this chapter, we first analyze the CPU and I/O costs of BIRCH, then we present the experimental performance evaluation of BIRCH (KMEANS, and CLARANS) on synthetic datasets for testing its feasibility.

5.1 Analysis

First we analyze the cpu cost of Phase 1. Given the memory is M bytes and each page is P bytes, the maximal size of the tree is $\frac{M}{P}$. To insert a point, we need to follow a path from root to leaf, touching about $1 + \log_B \frac{M}{P}$ nodes. At each node we must examine B entries, looking for the "closest" one; the cost per entry is proportional to the dimension d. So the cost for inserting all data points is $O(d * N * B(1 + \log_B \frac{M}{P}))$. In case we must rebuild the tree, let $C * d$ be the CF entry size where C is a constant mapping the dimension into the CF entry size. There are at most $\frac{M}{C*d}$ leaf entries to re-insert, so the cost of re-inserting leaf entries is $O(d * \frac{M}{C*d} * B(1 + \log_B \frac{M}{P}))$. The number of times we have to re-build the tree depends upon our threshold heuristics. Currently, it is about $\log_2 \frac{N}{N_0}$, where the value 2 arises from the fact that either we never estimate further than twice of the current size in the regression approach, or we always bring the tree size down to about half size in the memory utilization approach, and N_0 is the number of data points loaded into memory with threshold T_0. So the total cpu cost of Phase 1 is $O(d*N*B(1+\log_B \frac{M}{P})+\log_2 \frac{N}{N_0}*d*\frac{M}{C*d}*B(1+\log_B \frac{M}{P}))$. Since B equals $\frac{P}{C*d}$, the total cpu cost of Phase 1 can be rewritten as $O(N*\frac{P}{C}(1+\log_{\frac{P}{C*d}} \frac{M}{P})+\log_2 \frac{N}{N_0}*\frac{M}{C*d}*\frac{P}{C}(1+\log_{\frac{P}{C*d}} \frac{M}{P}))$

The analysis of Phase 2 cpu cost is similar, and hence omitted.

As for I/O, we scan the data once in Phase 1 and not at all in Phase 2. With the outlier-handling and split-delaying options on, there is some cost associated with writing out outlier entries to disk and reading them back during a rebuilt. Considering that the amount of disk available for outlier-handling and split-delaying is not too much, and that there are about $\log_2 \frac{N}{N_0}$ rebuilds, the I/O cost of Phase 1 is not significantly different from the cost of reading in just the original dataset.

There is no I/O in Phase 3. Since the input to Phase 3 is bounded, the cpu cost of Phase 3 is therefore bounded by a constant that depends upon the maximum input size range and the global algorithm chosen for this phase. Based on the above analysis — which is actually rather pessimistic for B, number of leaf entries, and the tree size in the light of our experimental results — the cost of Phases 1, 2 and 3 should scale up linearly with N.

Phase 4 scans the dataset again and puts each data point into the proper cluster; the time taken is proportional to $N * K$. However using the newest "nearest neighbor" techniques, it is improved [31] to be almost linear with N. The idea is that for each of the N data point, instead of looking all K cluster centers to find the nearest one, it only looks those cluster centers that are **around** the data point. This improvement performs very well and brings the time complexity $O(N * K)$ down to be almost linear with respect to N. But, the linear slope is sensitive to the distribution patterns of datasets as well as the the number of clusters K.

5.2 Synthetic Dataset Generator

To study the sensitivity of BIRCH to the characteristics of a wide range of input datasets, we have used a collection of synthetic datasets generated by a generator that we have developed. The data generation is controlled by a set of parameters that are summarized in Table 1.

Table 1: Data Generation Parameters and Their Values or Ranges Experimented

Parameter	Values or Ranges
Dimension d	2 .. 50
Pattern	grid, sine, random
Number of clusters K	4 .. 256
n_l (Lower n)	0 .. 2500
n_h (Higher n)	50 .. 2500
r_l (Lower r)	0 .. $\sqrt{50}$
r_h (Higher r)	$\sqrt{2}$ .. $\sqrt{50}$
Distance multiplier k_g	4 (grid only)
Number of cycles n_c	4 (sine only)
Noise rate r_n (%)	0 .. 10
Input order o	randomized, ordered

Each dataset consists of K clusters of d-dimensional data points. A cluster is characterized by the number of data points in it (n), its radius(r), and its center(c). n is in the range of $[n_l,n_h]$, and r is in the range of $[r_l,r_h]$. Note that when $n_l = n_h$ the number of points is fixed, and when $r_l = r_h$ the radius is fixed. Once placed, the clusters cover a range of values in each dimension. We refer to these ranges as the "overview" of the dataset.

The location of the center of each cluster is determined by the *pattern* parameter. Three patterns — *grid*, *sine*, and *random* — are currently supported by the generator. When the *grid* pattern is used, the cluster centers are placed on a 2-dimensional $\sqrt{K} \times \sqrt{K}$ grid. The distance between the centers of neighboring clusters on the same row/column is controlled by k_g, and is set to $k_g \frac{(r_l+r_h)}{2}$. This leads to an overview of $[0,\sqrt{K} k_g \frac{r_l+r_h}{2}]$ on both dimensions of the grid. The *sine* pattern places the cluster centers on a 2-dimensional curve of sine function. The K clusters are divided into n_c groups, each of which is placed on a different cycle of the sine function. The x location of the center of cluster i is $2\pi i$ whereas the y location is $\frac{K}{n_c} * sine(2\pi i/(\frac{K}{n_c}))$. The overview of a sine dataset is therefore $[0,2\pi K]$ and $[-\frac{K}{n_c},+\frac{K}{n_c}]$ on the x and y directions of the sine curve respectively. The *random* pattern places the cluster centers randomly. The overview of the dataset is $[0,K]$ on both dimensions since the the x and y locations of

the centers are both randomly distributed within the range $[0,K]$.

Once the characteristics of each cluster are determined, the data points for the cluster are generated according to a d-dimensional independent normal distribution whose mean is the center c, and whose variance in each dimension is $\frac{r^2}{d}$. Note that due to the properties of the normal distribution, the maximum distance between a point in the cluster and the center is unbounded. In other words, a point may be arbitrarily far from its belonging cluster. So a data point that belongs to cluster A may be closer to the center of cluster B than to the center of A, and we refer to such points as "outsiders".

In addition to the clustered data points, noise in the form of data points uniformly distributed throughout the overview of the dataset can be added to the dataset. The parameter r_n controls the percentage of data points in the dataset that are considered noise.

The placement of the data points in the dataset is controlled by the order parameter o. When the randomized option is used, the data points of all clusters and the noise are randomized throughout the entire dataset. Whereas when the ordered option is selected, the data points of a cluster are placed together, the clusters are placed in the order they are generated, and the noise is placed at the end.

5.3 Parameters and Default Setting

BIRCH is capable of working under various parameter settings. Table 2 lists the parameters of *BIRCH*, their effecting scopes and their default values. Unless specified explicitly otherwise, an experiments is conducted under this default setting.

M was selected to be about 5% of the dataset size in the base workload used in our experiments. Since disk space (R) is just used for outliers, we assume that $R < M$ and set $R = 20\%$ of M. The experiments on the effects of the 5 distance metrics in the first 3 phases (see Section 5.7) indicate that (1) using $D3$ in Phases 1 and 2 results in a much higher ending threshold, and hence produces clusters of poorer quality; (2) however, there is no distinctive performance difference among the others. So we decided

Table 2: BIRCH Parameters and Their Default Values

Scope	Parameter	Default Value
Global	Memory (M)	5% of dataset size
	Disk (R)	20% of M
	Distance def.	D2
	Quality def.	$\bar{D}$
	Threshold def.	threshold for D
Phase1,2	Initial threshold	0.0
	Threshold Heuristics	memory utilization
	Split-delaying	on
	Page size (P)	1024 bytes
	Outlier-handling	off
Phase3	Input range	1000
	Algorithm	Adapted HC2
Phase4	Refinement pass	1
	Discard-outlier	off

to choose $D2$ as default. Following the statistics tradition, we choose "weighted average diameter" Q2 (denoted as $\bar{D}$) as quality measurement. The smaller $\bar{D}$ is, the better the quality is. The threshold is defined as the threshold for cluster diameters as default. In Phase 1, the initial threshold is default to 0. Based on a study of how page size affects performance (see Section 5.7), we selected $P = 1024$. The split-delaying option is on for building more compact CF-trees. The outlier-handling option is off just for simplicity. We use the memory utilization based heuristics to increase the threshold. In Phase 3, most global algorithms can handle a few thousand objects well. So we set the default input range as 1000. We have chosen the adapted HC2 algorithm to use here. We decided to let Phase 4 refine the clusters only once with its discard-outlier option off, so that all data points will be counted in the quality measurement for fair comparisons with other algorithms such as KMEANS and CLARANS.

5.4 Base Workload Performance

The first set of experiments was to evaluate the ability of BIRCH to cluster large datasets of various patterns and with different input orders. All the times are presented in *seconds*

Table 3: Datasets Used as Base Workload

DS	Generator Setting	D_{int}
1	$d = 2, grid, K = 100, n_l = n_h = 1000,$ $r_l = r_h = \sqrt{2}, k_g = 4, r_n = 0\%, o = randomized$	2.00
2	$d = 2, sine, K = 100, n_l = n_h = 1000,$ $r_l = r_h = \sqrt{2}, n_c = 4, r_n = 0\%, o = randomized$	2.00
3	$d = 2, random, K = 100, n_l = 0, n_h = 2000,$ $r_l = 0, r_h = 4, r_n = r_n = 0\%, o = randomized$	4.18

in this paper. Three 2-dimensional synthetic datasets, one for each pattern, were used. The 2-dimensional datasets were chosen as the base workload because they were easy to visualize. But there is no obstacles for BIRCH to work on high dimensions (We will process higher (2 to 50) dimensional data in the scalability section and 16-dimensional vectors in the image compression application).

Table 3 presents the data generator settings for the base workload. The weighted average diameters of the intended clusters [1] $\bar{D}_{int}$ are also included in the table as a rough measurement of the quality of the original intended clusters. Figure 28 through 30 visualize the three base workload datasets by scatter plots. Figure 31 through 33 visualize the intended clusters of the three base workload datasets by plotting an intended cluster as a circle whose center is the intended centroid, radius is the intended cluster radius, and label is the intended number of points in the cluster. Three additional datasets - DS1o, DS2o and DS3o - which correspond to DS1, DS2 and DS3, respectively except that the parameter o of the generator is set to *ordered* are used to test the order sensitivity of BIRCH.

The BIRCH clusters of DS1 and DS1o are presented in the same way in Figure 34 and 37. We observe that for both input orders, the *BIRCH* clusters are very similar to the intended clusters in terms of location, number of points, and radii. For DS1, the maximal and average distance between the centroids of an intended cluster and its corresponding *BIRCH* cluster are 0.19 and 0.08 respectively. The number of points in

[1] From now on, we refer to the clusters generated by the data generator as the "intended clusters" whereas the clusters identified by BIRCH as "BIRCH clusters", identified by CLARANS as "CLARANS clusters" and identified by KMEANS as "KMEANS clusters".

a BIRCH cluster is no more than 5% different from the corresponding intended cluster. The radii of the *BIRCH* clusters (ranging from 1.26 to 1.39 with an average of 1.32) are close to, or even better than due to the correction of "outsiders", those of the intended clusters (1.41). For DS1o, the maximal and average distance between the centroids of an intended cluster and its corresponding *BIRCH* cluster are 0.15 and 0.06 respectively. The number of points in a BIRCH cluster is no more than 5% different from that of the corresponding intended cluster. The radii of the BIRCH clusters (ranging from 1.25 to 1.41 with an average of 1.32) are close to, and even better than due to the correction of "outsiders", those of the intended clusters (1.41). Similar conclusions can be reached by analyzing the visual presentations of BIRCH clusters on DS2, DS2o, as shown in Figure 35 and 38.

So the quality of BIRCH clusters is close to, and even better than the $\bar{D}_{int}$ of the intended clusters. This is because BIRCH assigns the "outsiders" of an intended cluster to a proper BIRCH cluster. As mentioned earlier, $\bar{D}_{int}$ is only a rough measurement of the quality of the intended clusters. To figure the actual quality ($\bar{D}_{act}$) of the intended clusters, one can start with the intended centers as seeds, scan the data and refine the clusters until an optimal partition is reached. The result is that for DS1, its $\bar{D}_{act}$ is 1.87; and for DS2, its $\bar{D}_{act}$ is 1.99. Comparing them with the quality of BIRCH clusters, one can notice that the clusters identified by BIRCH are almost the optimal clusters one can find assuming the cluster centers are known.

As summarized in Table 4 for all three base workload datasets and two different input orders,

- it took BIRCH less than 15 seconds (on a DEC pentium-pro workstation running Solaris) to cluster 100,000 2-dimensional data points of each dataset, which includes 2 scans of the dataset (about 1.16 seconds for each scan of the ASCII file from disk);
- the quality of BIRCH clusters was very close to the $\bar{D}_{act}$ of the dataset;
- the different patterns of datasets had almost no impact on the clustering time;

Table 4: BIRCH Performance on Base Workload

DS	Time	$\bar{D}$	#Scan	DS	Time	$\bar{D}$	#Scan
1	11.5	1.87	2	1o	13.6	1.87	2
2	10.7	1.99	2	2o	12.1	1.99	2
3	11.4	3.95	2	3o	12.2	3.99	2

Table 5: CLARANS Performance on Base Workload

DS	Time	$\bar{D}$	#Scan	DS	Time	$\bar{D}$	#Scan
1	932	2.10	3307	1o	794	2.11	2854
2	758	2.63	2661	2o	816	2.31	2933
3	835	3.39	2959	3o	934	3.28	3369

Table 6: KMEANS Performance on Base Workload

DS	Time	$\bar{D}$	#Scan	DS	Time	$\bar{D}$	#Scan
1	43.9	2.09	289	1o	33.8	1.97	197
2	13.2	4.43	51	2o	12.7	4.20	29
3	32.9	3.66	187	3o	36.0	4.35	241

- the different order of the data points had almost no impact on the performance (time and quality) of BIRCH.

5.5 Other Methods on Base Workload

In this experiment we try to compare the performance of BIRCH, CLARANS, KMEANS and HC on the base workload. However since HC needs $O(N^2/2)$ memory (about 40 gigabytes for a base workload dataset) for storing the distance matrix, it hangs forever in our system (with 64 megabytes of memory) with the frequent memory swapping. Following we concentrate on comparisons with CLARANS and KMEANS.

- First, for CLARANS and KMEANS we assume that the memory is enough for holding the whole dataset as well as some other $O(N)$ size assisting data structures. So they need much more memory than BIRCH does. (Clearly, this assumption greatly favors these two algorithms in terms of running time comparison!)

- Second, the CLARANS implementation is provided by Ng. In order for CLARANS to stop after an acceptable running time, we set its *maxneighbor* value to be the larger of 50 (instead of 250) and 1.25% of K(N-K), but no more than 100 (newly enforced upper limit recommended by Ng). Its *numlocal* value is still 2. We have implemented the KMEANS algorithm provided in [32] in C and the initial seeds for KMEANS are selected randomly.

- Third, we have observed that the performances of CLARANS and KMEANS are extremely sensitive to the random number generator used. A bad random number generator, such as the UNIX "rand()" used in the original code of CLARANS, can generate random numbers that are were not really random but relevant to the data order, and hence make CLARANS and KMEANS's performances extremely unstable with the different input orders[57]. So to avoid this problem, we have replaced "rand()" with a more elaborate random number generator.

Figure 40 and 46 visualize the CLARANS and KMEANS clusters for DS1. Comparing them with the intended clusters for DS1 we can observe that:

- The pattern of the locations of the cluster centers is distorted.

- The number of data points in a CLARANS or KMEANS cluster can be as many as 40% different from that in the corresponding intended cluster.

- The radii of CLARANS clusters varies largely from 1.15 to 1.94 with an average of 1.44 (larger than those of the intended clusters, 1.41). The radii of KMEANS clusters varies largely from 0.99 to 2.02 with an average of 1.38 (larger than those of BIRCH clusters, 1.32).

Figure 43 and 49 visualize the CLARANS and KMEANS clusters for DS1o. Comparing them with the CLARANS and KMEANS clusters for DS1 we can observe that changing the input order causes CLARANS and KMEANS's results to change:

- The pattern of the cluster center locations found for DS1o is still distorted, but is quite different from that found for DS1.

- The numbers of data points in clusters found for DS1o are spatially distributed very differently from those found for DS1.
- The cluster radii found for DS1o are spatially distributed very differently from those found for DS1.

Similar behavior can be observed in the visualization of CLARANS KMEANS clusters for DS2 and DS2o, as shown in Figure 41, 44, 47 and 50.

Tables 5 and 6 summarize the performances of CLARANS and KMEANS. For all three base workload datasets and two different input orders,

1. They scan the dataset frequently. In spite of the favorable condition that all data are loaded into memory, only the first scan is from ASCII file on disk, and the remaining scans are in memory, CLARANS and KMEANS were still slower than BIRCH and their running times were sensitive to the patterns of the datasets.
2. Even though they frequently scan the data to refine the clustering to reach an optimal solution, the $\bar{D}$ values for the CLARANS and KMEANS clusters were usually larger than those for the BIRCH clusters. That implies that the qualities of CLARANS and KMEANS clusters are worse than those of BIRCH clusters because they can get stuck at a local optimal partition.
3. The results for DS1o, DS2o, and DS3o illustrate that when the data points were input in different orders, the running time and clustering quality of CLARANS and KMEANS will change whereas BIRCH is very stable.

In conclusion, for the base workload, BIRCH uses much less memory, but runs much faster and generate more accurate, pattern-insensitive, and order-insensitive results compared with CLARANS and KMEANS.

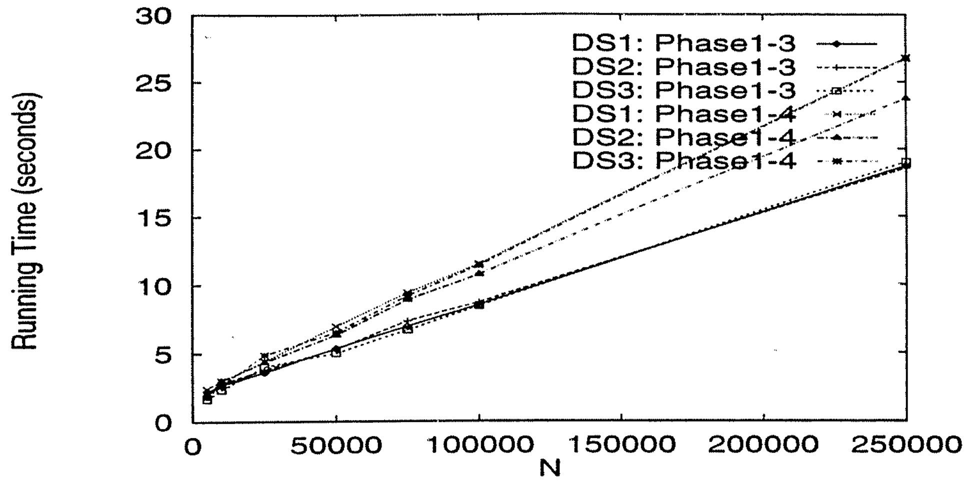


Figure 10: Time Scalability with respect to Increasing Number of Points per Cluster (n)

Figure 11: Quality Stability with respect to Increasing Number of Points per Cluster (n)

DS	$n \in [n_l..n_h]$	N	D/D_{int}
1	50..50	5000	1.87/1.99
	100..100	10000	1.89/2.01
	250..250	25000	1.92/1.99
	500..500	50000	1.87/1.98
	750..750	75000	1.86/2.00
	1000..1000	100000	1.88/2.00
	2500..2500	250000	1.87/2.00
2	50,50	5000	1.98/1.98
	100..100	10000	2.00/2.00
	250..250	25000	2.00/2.00
	500..500	50000	2.00/2.00
	750..750	75000	1.99/1.99
	1000..1000	100000	1.99/2.00
	2500..2500	250000	1.99/1.99
3	0..100	5000	4.08/4.42
	0..200	10000	4.30/4.78
	0..500	25000	4.21/4.65
	0..1000	50000	4.00/4.27
	0..1500	75000	3.74/4.22
	0..2000	100000	3.95/4.18
	0..5000	250000	4.23/4.52

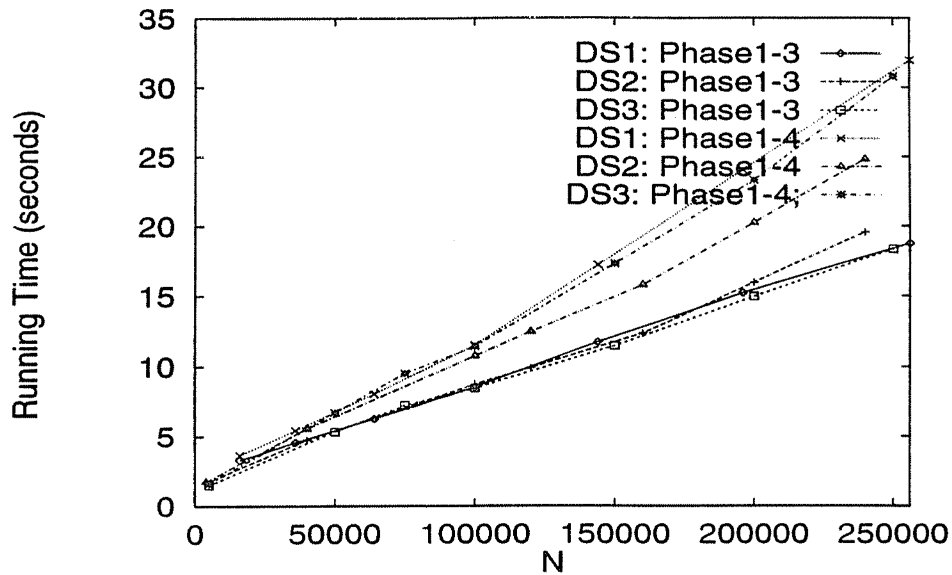


Figure 12: Time Scalability with respect to Increasing Number of Clusters (K)

Figure 13: Quality Stability with respect to Increasing Number of Clusters (K)

DS	K	N	$\bar{D}/\bar{D}_{int}$
1	16	16000	2.32/2.00
	36	36000	1.98/2.00
	64	64000	1.93/1.99
	100	100000	1.87/2.00
	144	144000	1.87/1.99
	196	196000	1.87/2.00
	256	256000	1.87/2.00
2	4	4000	1.98/1.99
	40	40000	1.99/1.99
	100	100000	1.99/2.00
	120	120000	2.00/2.00
	160	160000	2.00/2.00
	200	200000	1.99/1.99
	240	240000	1.99/1.99
3	5	5000	5.57/6.43
	50	50000	4.10/4.52
	75	75000	4.04/4.76
	100	100000	3.95/4.18
	150	150000	4.21/4.26
	200	200000	5.22/4.49
	250	250000	5.52/4.48

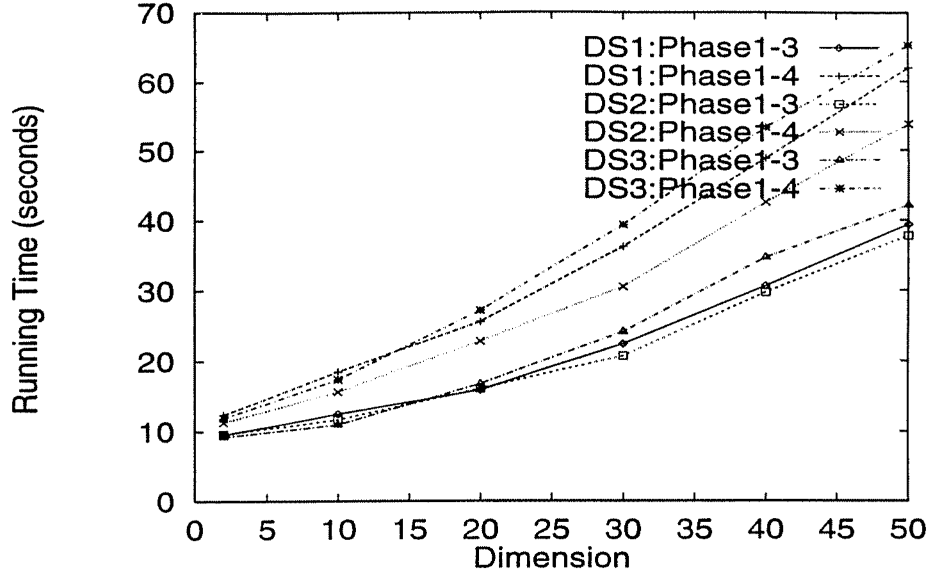


Figure 14: Time Scalability with respect to Increasing Dimension (d)

Figure 15: Quality Stability with respect to Increasing Dimension (d)

DS	d	D/D_{int}
1	2	1.87/1.99
	10	4.68/4.46
	20	6.52/6.31
	30	7.87/7.74
	40	8.97/8.93
	50	10.08/9.99
2	2	1.99/1.99
	10	4.46/4.46
	20	6.31/6.31
	30	7.74/7.74
	40	8.93/8.93
	50	10.02/9.99
3	2	3.95/4.28
	10	11.96/9.62
	20	17.11/13.62
	30	20.76/16.70
	40	25.34/19.28
	50	26.64/21.56

5.6 Scalability and Stability

Three distinct ways of increasing the dataset size are experimented in order to test the time scalability and quality stability of *BIRCH*.

5.6.1 Increasing the Number of Points per Cluster (n)

For each of DS1, DS2 and DS3, we create a range of datasets by keeping the generator settings the same except for changing n_l and n_h (lower and higher bounds for n) to increase the n, and hence increase the total number of data points, N. Since N does not grow too far from that of the base workload (within the same magnitude), we decided to use the same amount of memory for this scaling experiment as we have used for the base workload. This can help us understand that for each pattern, given a fixed amount of memory, if increasing n, how large dataset BIRCH can cluster while maintaining stable qualities. Based on the performance analysis 5.1, in this case, with M, P, d, K fixed, and only N growing, the running time should scale up linearly with N. Following are the experiment results.

With all three patterns of datasets, their running time for the first 3 phases, as well as for all 4 phases are plotted against the dataset size N in Figure 10. One can observe that for all three patterns of data:

1. The running times of the first 3 phases grows linearly with respect to N similarly for all three patterns.

2. The running times of all 4 phases grows linearly with respect to N similarly for all three patterns.

While the running times are shown to be linearly scalable and insensitive to the dataset patterns, Table 11 provides the corresponding quality values of the intended clusters ($\bar{D}_{int}$) and of BIRCH clusters ($\bar{D}$) as n, and N increase for all three patterns of datasets. It is shown from the table that: With the same amount of memory, for a wide range of n, and N, the quality of BIRCH clusters (indicated by $\bar{D}$) are consistently close

to (or better than due to the correction of "outsiders") that of the intended clusters (indicated by $\bar{D}_{int}$).

5.6.2 Increasing the Number of Clusters (K)

For each of DS1, DS2 and DS3, we create a range of datasets by keeping the generator settings the same except for increasing the number of clusters, K, to increase the total number of data points, N. Again, since K does not grow too far from that of the base workload (within the same magnitude), we decided to use the same amount of memory for this scaling experiment as we have used for the base workload. This can help us understand that for each pattern, given a fixed amount of memory, if increasing K, how large dataset BIRCH can cluster while maintaining stable qualities.

With all three patterns of datasets, their running time for the first 3 phases, as well as for all 4 phases are plotted against the dataset size N in Figure 12. The running times for the first 3 phases are again confirmed to grow linearly with respect to N similarly for all three patterns. The running times for all 4 phases are almost linear with N, but have slightly different slopes for the three different patterns of datasets. More specifically, for the grid pattern the slope is the largest, and for the sine pattern, the slope is the smallest. This is due to the fact that K and N are growing at the same time, and the complexity of Phase 4 is $O(K * N)$ (not strictly linear with N) in the worst case. We have tried to improve Phase 4 refining algorithm to be almost linear with N using the "nearest neighbor" techniques proposed in [31] to. However, the linear slope is sensitive to the distribution patterns of datasets as well as K as we have mentioned earlier. In our case, for the grid pattern, since there are usually more cluster centers **around** a given data point, it will check more centers around the data point to find the nearest one; whereas for the sine pattern, since there are usually less cluster centers **around** a given data point, it will check less centers to find the nearest one; random pattern is hence in the middle.

As for the quality stability, we can reach the similar conclusion from Table 13. That is: With the same amount of memory, for a wide range of K, and N, the quality of

BIRCH clusters (indicated by $\bar{D}$) are consistently close to (or better than due to the correction of "outsiders") that of the intended clusters (indicated by $\bar{D}_{int}$).

5.6.3 Increasing the Dimension (d)

For each of DS1, DS2 and DS3, we create a range of datasets by keeping the generator settings the same except for changing the dimension (d) from 2 to 50 to change the dataset size. In this experiment, the amount of memory used for each dataset is scaled up based on the dataset size (5% of the dataset). With all three patterns of datasets, their running times for the first 3 phases, as well as for all 4 phases are plotted against the dimension in Figure 14. We observe that the time deviates slightly from linear as the dimension as well as the corresponding memory increase. This is caused by the following fact: with M_0 (a base amount of memory, and the memory corresponding to a given d-dimensional dataset is scaled up as $M_0 * d$), N and P constant, as the dimension increases, the time complexity will scale up with $1+\log_{\frac{P}{C*d}} \frac{M_0*d}{P}$ according to the analysis study in Section 5.1. That is, as the dimension increases, first the amount of memory increases, and causes the CF-tree size to increase; second the branching factor decreases, and causes the CF-tree height to increase. So for a larger d, incorporating a new data point goes through more levels on a larger CF-tree, and hence needs more time. The interesting thing is that by tuning P, one can make the curves sublinear or superlinear, and in this case, with P=1024 bytes, the curves are slightly superlinear.

As for the quality stability, we can reach the similar conclusion from Table 15. That is: With the same amount of memory, for a wide range of d, the quality of BIRCH clusters (indicated by $\bar{D}$) are consistently close to (or even better than sometimes due to the correction of "outsiders") that of the intended clusters (indicated by $\bar{D}_{int}$).

5.7 Sensitivity to Parameters

I have decided to include some earlier experiments that address the sensitivity of the performance of BIRCH to the value of some of its parameters in this section. In the

Table 7: Sensitivity to Initial Threshold

T_0	DS1		DS2		DS3	
	Time	D	Time	D	Time	D
0.0	47.14	1.87	47.5	1.99	49.52	3.39
0.001	46.39	1.87	47.75	1.99	49.83	3.38
0.01	46.75	1.87	45.77	1.99	49.75	3.69
0.1	47.34	1.87	46.14	1.99	48.47	3.59
1.0	44.75	1.87	43.98	1.99	45.77	3.53

experiment results we will use the notation of a superscript in a pair of parentheses to indicate the phase(s) for which a measured quantity is associated. For example, $Time^{(1-3)}$ stands for the time spent in Phase 1 through Phase 3 and $\bar{D}^{(3)}$ is the quality at the end of Phase 3. Again all times presented are in *seconds.*

Threshold

Table 7 shows how the performance of the performance of *BIRCH* is affected by the value of the initial threshold, T_0. From the results presented in the table we conclude that:

1. BIRCH's performance(time and quality) is stable as long as the initial threshold is not excessively high.

2. The conservative default $T_0 = 0.0$ works well with a little extra running time.

3. If a user does have a good T_0 to provide, then she/he can be rewarded by saving up to 10% of the processing time.

Based on this experiment, we have chosen to set $T_0 = 0$ as default in BIRCH.

Branching Factor and Page Size

B is an important parameter affecting the performance of Phase 1. If we only look the analysis of the time complexity in Phase 1, assuming d, N, N_0 M and C are all given, from the formula $O(d * N * B(1 + \log_B \frac{M}{P}) + \log_2 \frac{N}{N_0} * d * \frac{M}{C*d} * B(1 + \log_B \frac{M}{P}))$, the best

value for B is $e = 2.718$ (i.e., 2 or 3). However, the analysis of time complexity did not take into account a very important phenomenon we have observed: with the same threshold value, the same amount of data and the same data input order, the smaller value of B we choose, the larger CF-tree we will get. This happens because a smaller B value means we have less information at each level to guide where a newly inserted point belongs in the CF-tree. So a data point that could have been absorbed by an existing leaf entry, if directed to the appropriate leaf, could well go to the wrong leaf node and cause a new leaf entry to be generated. So a smaller B value tends to cause more rebuilds, eventually requires a higher threshold value and generates less entries at the end of Phase 1, hence affects the clustering efficiency and quality. Since B is determined by P, Table 8 shows how P affects Phase 1 as well as the final clustering quality for the base workload. It suggests that in Phase 1, smaller (larger) P tend to decrease (increase) running time (as shown by $Time^{(1)}$), require higher (lower) ending threshold (as shown by $T^{(1)}$), produce less (more) but "coarser (finer)" leaf entries (as shown by $Entry^{(1)}$), and hence degrade (improve) the clustering quality (as shown by $\bar{D}^{(3)}$). However with the refinement in Phase 4, the experiments suggest that from $P = 256$ to $P = 4096$, although the $\bar{D}^{(3)}$'s are different, the final qualities after the refinement are almost the same (as shown by $\bar{D}$)[2]. Based on this experiment, we have chosen to set $P = 1024$ as default in BIRCH.

Outlier Options

In Phase 1, a leaf entry is considered to be an outlier if it contains "too few" data points (Here we choose *less than 25% of the average data points per leaf entry*). In Phase 4, we provide the option to discard outliers as noise. We observed that for many common distributions (e.g., uniform, normal) more than 90% of the points of a cluster fall within the range of twice radius. Therefore we consider a point as noise if it falls outside that range.

[2]Some $\bar{D}^{(3)}$'s are smaller than the corresponding $\bar{D}$ because there are some points taken as outliers and not counted in the quality at the end of Phase 3.

Table 8: Sensitivity to Page Size

Dataset	P	$Time^{(1)}$	$Entry^{(1)}$	$T^{(1)}$	$\bar{D}^{(3)}$	$\bar{D}$
DS1	64	19.86	445	2.17	2.30	1.87
	256	17.32	1171	1.36	2.03	1.87
	1024	27.24	2065	0.90	1.97	1.87
	4096	53.43	1980	0.77	1.96	1.87
DS2	64	19.55	436	2.16	2.05	1.99
	256	17.81	681	1.55	2.00	1.99
	1024	28.05	1400	1.26	1.96	1.99
	4096	59.35	1568	1.23	1.96	1.99
DS3	64	22.36	665	4.29	4.81	3.96
	256	17.97	1483	2.64	3.65	3.36
	1024	29.45	1966	1.78	3.74	3.39
	4096	59.66	1612	1.72	3.50	3.35

Table 9: Effects of Outlier Options

Dataset	Outlier Handling Options			
	On		Off	
	Time	D	Time	D
DS4	48.02	1.92	48.63	1.94
DS5	47.14	2.05	51.51	8.44
DS6	47.41	4.41	49.68	5.06

In this section, we use three additional datasets DS4, DS5 and DS6 which are the same as DS1, DS2 and DS3 respectively, except that each of them contains 10% noise. We run *BIRCH* on the three "noisy" datasets with the outlier options all on or all off. Table 9 presents the timing results and Figures 52 through 57 visualize the corresponding *BIRCH* clusters. With outlier options all on, *BIRCH* is not slower but faster and at the same time $\bar{D}$ is smaller. Whether the smaller $\bar{D}$ is due to better clustering or less data points is unclear from the table. However, the clustering results visualized in the figures are a clear display that with the outlier options all on, *BIRCH* is more noise-robust and obtains clusters of better quality.

Table 10: Effects of Phase 3 Algorithms

Global Algorithm	DS1		DS2		DS3	
	$Time^{(3)}$	D	$Time^{(3)}$	D	$Time^{(3)}$	D
HC1	55.29	1.87	39.79	1.99	47.21	3.39
HC2	2.27	1.87	1.84	1.99	1.98	3.39
CLARANS1	59.15	1.99	69.35	2.15	52.79	3.73
CLARANS2	27.43	2.04	22.22	2.48	47.34	3.14

Phase 3 Algorithms

Table 10 shows the time spent in Phase 3 and the final quality with the four different global algorithms. It shows:

- *HC1* and *HC2* result in exactly the same quality when distance metrics D2 and D4 are used, but the latter runs much faster.
- *CLARANS1* is slower than *CLARANS2* , and its quality is not necessarily better.
- *HC1* and *HC2* have stabler quality than *CLARANS1* and *CLARANS2*. The reason is that in our case where objects fed to Phase 3 for global clustering are subclusters (represented as CF entries), and when they are further grouped to form clusters, they are hardly centrally located in the clusters. So the idea of looking for medoid objects (or centrally located subclusters) will tend to distort the clustering.

Based on this experiment, we have chosen HC2 as default in BIRCH Phase 3.

Memory Size

In Table 11, *BIRCH* is applied to the base workload with memory size varying from 20 kbytes to 80 kbytes. It suggests that in Phase 1, as memory size (or the maximal tree size) increases,

1. $Time^{(1)}$ increases because the time spent in transforming an old CF-tree into a new CF-tree increases, but only slightly because this transformation is done in memory;

Table 11: Effects of Memory Size

Dataset	Memory(kbytes)	$Time^{(1)}$	$Entry^{(1)}$	$T^{(1)}$	$\bar{D}^{(3)}$	$\bar{D}$
DS1	20	21.91	334	1.89	2.23	1.87
	40	24.39	1029	1.29	2.04	1.87
	60	26.22	1535	1.02	1.98	1.87
	80	27.24	2065	0.90	1.97	1.87
DS2	20	20.16	114	2.59	2.26	1.99
	40	24.34	963	1.84	2.01	1.99
	60	27.13	1338	1.52	1.98	1.99
	80	28.05	1400	1.26	1.96	1.99
DS3	20	21.3	369	4.63	5.15	3.83
	40	25.4	588	3.06	4.56	3.93
	60	26.31	1245	2.30	3.81	3.58
	80	29.45	1966	1.78	3.74	3.39

2. more (as $Entry^{(1)}$ shows), but finer (as $T^{(1)}$ shows) subclusters are generated for the next phase, and hence results in better quality (as $\bar{D}^{(3)}$ shows);

3. however, by comparing $\bar{D}^{(3)}$ and $\bar{D}$, one can see that the inaccuracy caused by insufficient memory can be compensated to some extent with Phase 4 refinements, or in another word, to achieve the same quality, *BIRCH* provides the tradeoff between time and memory.

Distance Metrics

Suppose that in Phase 1 and Phase 2 we use Di (i=0,1,2,3,4), and in Phase 3 we use Dj (j=2,4 only due to the fact HC2 only supports D2 and D4). Table 12 summarizes the performances for the 10 different distance combinations on the base workload datasets. It is shown from the experiments that D3 is a bad choice for Phase 1 and Phase 2. By tracing the BIRCH execution, we observed that using D3 in Phase 1 and 2 tended to need more re-builds and results in higher threshold value to finish, which then degraded the efficiency and the quality. Other than that, there is no distinctive performance

difference among the other distance combinations. [3]

Table 12: Effects of Distance Metrics

Di Dj	DS1		DS2		DS3	
	Time	Quality	Time	Quality	Time	Quality
D0 D2	13.27	1.91	10.95	1.99	12.68	4.00
D0 D4	13.72	1.87	11.34	1.99	12.91	3.35
D1 D2	17.64	1.91	14.77	1.99	22.31	3.92
D1 D4	18.67	1.87	15.37	1.99	23.16	3.24
D2 D2	12.94	1.94	10.14	1.99	12.15	4.09
D2 D4	13.73	1.87	10.32	1.99	12.71	3.31
D3 D2	18.36	2.35	10.55	4.26	15.86	3.89
D3 D4	18.96	2.28	10.62	4.41	16.25	3.82
D4 D2	14.56	1.87	13.24	1.99	16.95	3.95
D4 D4	14.64	1.87	13.54	1.99	17.63	3.27

[3] This table of data is generated with the newest version of birch whose data reading module has been greatly improved recently by Mr. Kent Wenger. In this version, reading an ASCII file of 100000 2-dimensional tuples needs only 2.02 seconds

Chapter 6

BIRCH Applications

In this chapter, we intend to show (1) how a clustering system like BIRCH can be used to help solving real world problems; and (2) how BIRCH, CLARANS and KMEANS perform on the real datasets.

6.1 Pixel Classification Tool

6.1.1 Motivation

The first application is motivated by the MVI (Multiband Vegetation Imager) technique developed in [36, 37]. The MVI is the combination of a charge-coupled device (CCD) camera, a filter exchange mechanism, and laptop computer used to capture rapid, successive images of plant canopies in two wavelength bands. The purpose of using two wavelength bands is to allow for identification of different canopy components such as sunlit and shaded leaf area, sunlit and shaded branch area, clouds, and blue sky based upon the camera's resolution and the varying spectral properties that canopy components have in the two different wavelength bands being used.

If the pixels in these images taken by the MVI can be classified into the above categories, then the MVI can be used as a technique to help quantify the canopy structure. Canopy structure influences many biophysical processes in forests, and virtually all models that attempt to quantify plant-environment interactions, such as effects of climate change on boreal-forest carbon budgets, require the kind of information that is provided by the MVI. So classifying pixels in the MVI image will be important to many fields of research including ecology, forestry, meteorology, and other agricultural sciences.

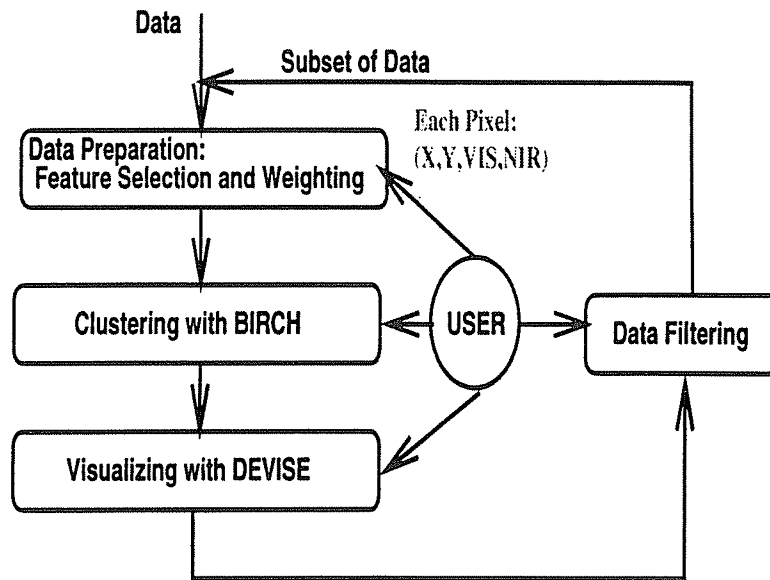


Figure 16: Interactive and Iterative Pixel Classification Tool

6.1.2 Pixel Classification Tool

The main use of BIRCH is to help classify pixels in these images, such as separate foliage in a MVI image from sky and clouds in the background which aids in the analysis of the forest canopy. Motivated by the MVI images, but not just restricted to them, a general interactive and iterative pixel classification tool for multi-band images is designed with BIRCH and DEVISE (A Data Exploration via VISualization Environment) [10] integrated in the way as shown in Figure 16. It includes four major steps:

1. Image pixels are read and prepared for clustering, that is, interesting features are selected, and then weighted and/or shifted in order to use BIRCH to classify them based on clustering.

2. *BIRCH* is applied for clustering the pixels in the space of the selected, and then weighted and/or shifted features.

3. The relevant results such as the obtained clusters as well as their corresponding pixels in the original images are visualized with matching colors in two linked windows by DEVISE for the user to (1) look for "patterns", or (2) evaluate the

"qualities" of the classification visually.

4. Then with the visual feedback, the user may decide to (1) explore other feature selection and/or weighting strategies for better classification results; or (2) filter out a subset of the pixels which corresponds to some clusters for further clustering and visualizing.

It has a "exploring" flavor because the above four major steps are iterated, and during each iteration, the user can adjust the parameters of each step interactively instead of just using the default settings to explore the data for hidden clusters from different aspects.

The history of exploration should be maintained automatically for the user to access conveniently. That is, the user should be able to specify things such as what part of the exploration are (not) worth to keep, and where to proceed or backtrack during the exploration. A directed acyclic graph (DAG) is proposed for tracking this kind of information. This DAG has two types of nodes: D-node and S-node. A D-node corresponds to a specific *dataset* and a S-node corresponds to clustering with a specific parameter *setting.* Interleavingly, only a D-node or several D-nodes can point to S-nodes, and only a S-node can point to a D-node or several D-nodes. A D-node pointing to A S-node corresponds to clustering a specific dataset with a specific parameter setting, whereas several D-nodes pointing to a S-node corresponds to merging those datasets first, and then clustering with a specific parameter setting. A S-node branching to point to several D-nodes corresponds to the generated subsets of data based on clustering under that parameter setting. All S-nodes are labeled based on the order that they are created or explored. Figure 17 shows an example of the DAG. The root node is a D-node and it corresponds to the original dataset Dataset0. It branches to point to three S-nodes which corresponds to clustering Dataset0 with three different parameter settings in the order as labeled by 1,2,3. For the first parameter setting S1, the clustering generates three clusters, and hence three corresponding subsets of data. The first and third subsets are merged and clustered again with a new parameter setting as labeled

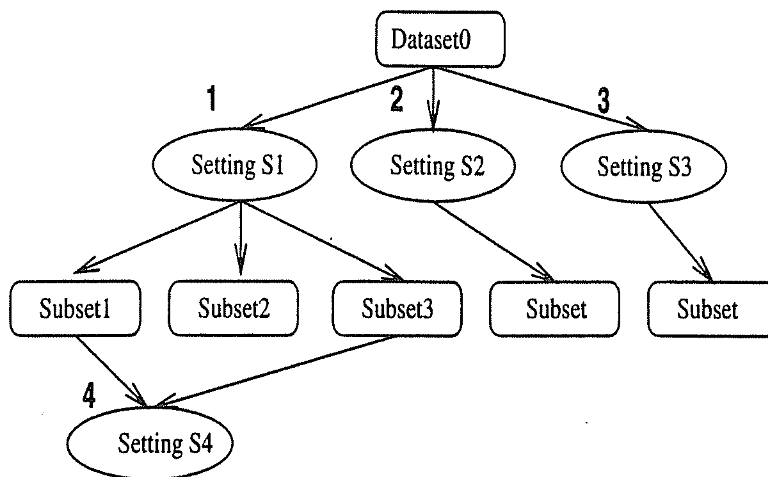


Figure 17: An Example of DAG Used to Track History

by 4.

6.1.3 Example of Using the Tool

Following is an example of using this tool to help separate pixels in a MVI image. Figure 18 is a MVI image which contains two similar images of the trees with the sky as the background, taken in two different bands. The top one is taken in the near-infrared band (NIR image), and the bottom one is taken in the visible wavelength band (VIS image). Each image contains 512x1024 (i.e., 524288) pixels, and each pixel can be represented a tuple with schema (x, y, nir, vis), where x and y are the coordinates of the pixel, and nir and vis are the corresponding brightness values in the NIR image and the VIS image respectively. The edges of each image are "cut" (i.e., ignoring a few rows near the top and the bottom edges and a few columns near the left and the right edges) to avoid the influence of noises along edges. So the actual images used for clustering contains 490x990 (i.e., 485100) pixels.

We start the first iteration with the number of clusters being 2 in the hope of finding the two clusters corresponding to the trees and the sky background. It is easy to notice

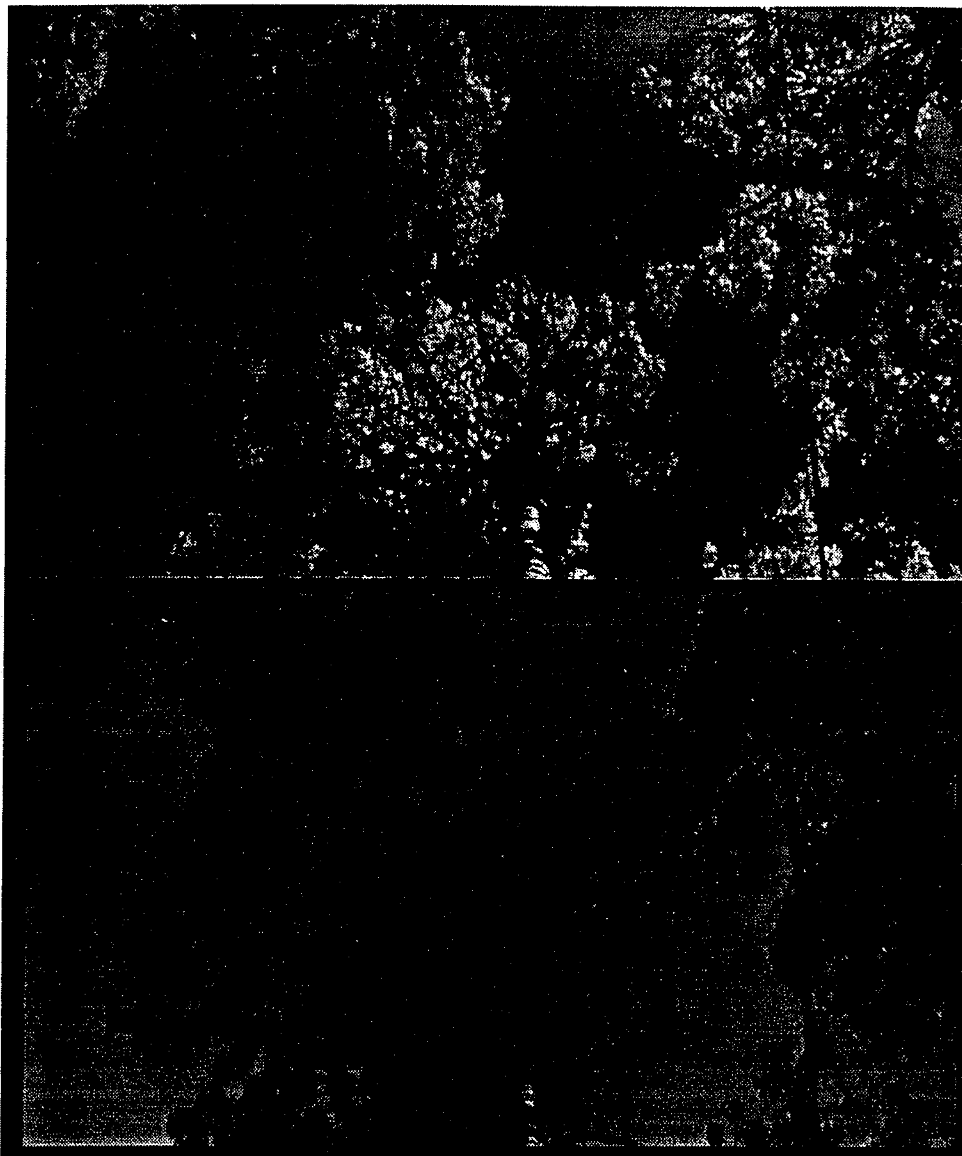


Figure 18: An Example of MVI image

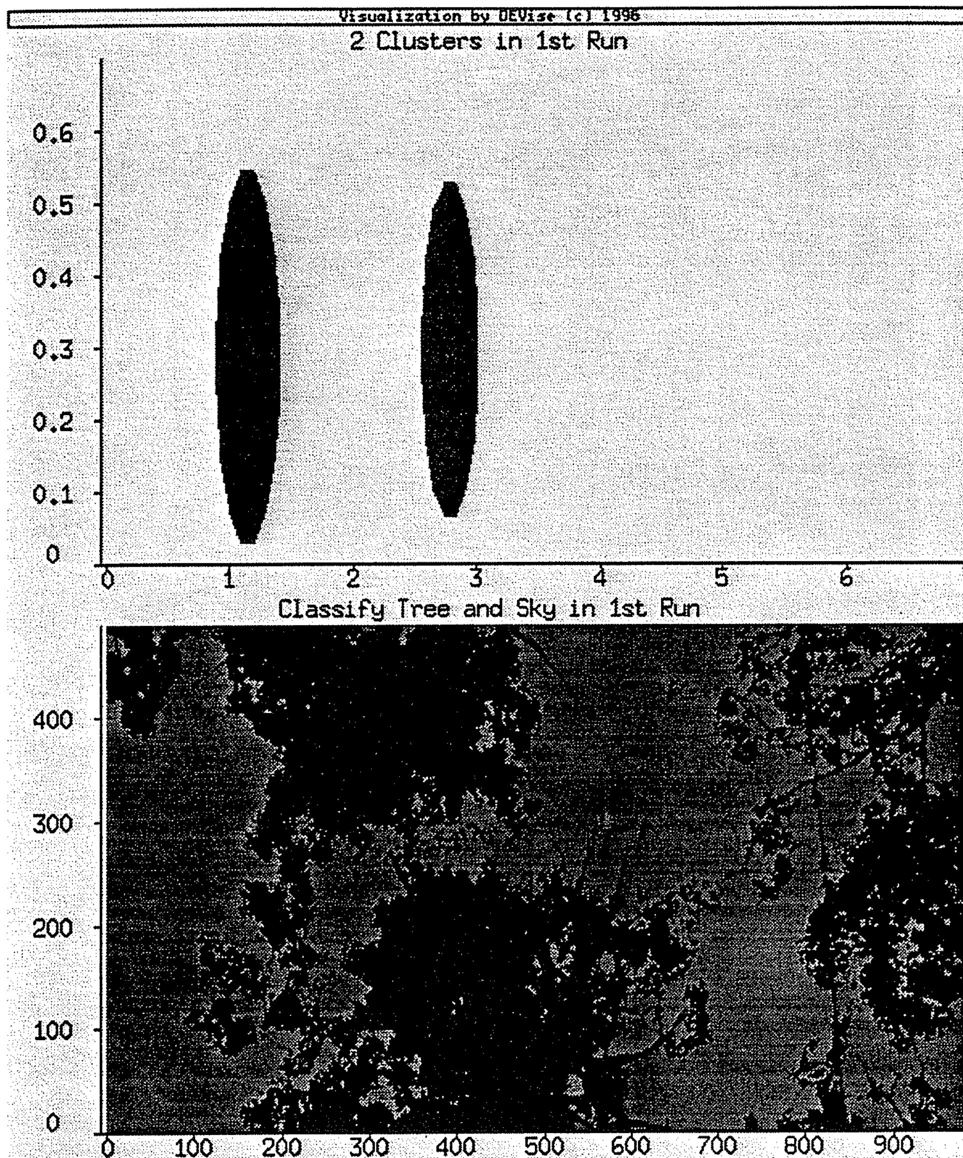


Figure 19: 1st Run: Separate Trees and Sky

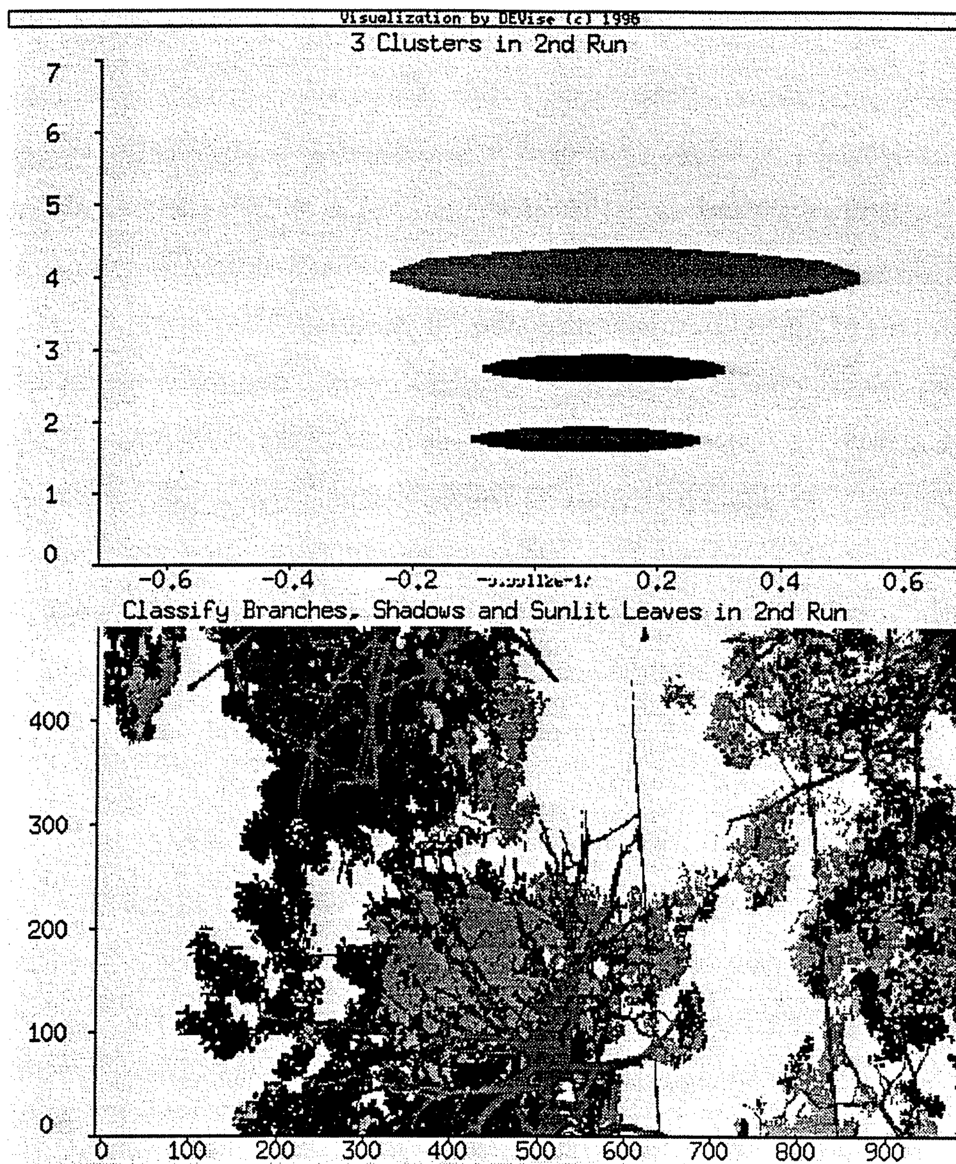


Figure 20: 2nd Run: Separate Branches, Shadows and Sunlit Leaves

that the trees and the sky are better differentiated in the VIS image than in the NIR image. So the weight assigned to *vis* is 10 times more than the weight assigned to *nir*. Then *BIRCH* is invoked under the default settings to do the clustering which takes a total of 50 seconds (including two scans of the VIS and NIR values form ASCII file on disk in Phase 1 and Phase 4 and each scan takes about 4.5 seconds). Figure 19 is the DEVISE visualization of the clusters (top: where x-axis is for weighted *vis* value, y-axis is for weighted *nir* value, each cluster is plotted as a circle with the centroid as the center and the standard deviation as the radius) as well as their corresponding parts of image (bottom: where x-axis is for the x coordinates of pixels and y-axis is for the y coordinates of pixels) obtained after the 1st iteration.

Visually, by comparing with the original images, one can see that the two clusters form a satisfactory classification of the trees and the sky. Whereas in reality, in general, it may takes a few more iterations for the user to identify a good set of weights. However the important thing is that once a set of good weights are identified from one MVI image, they can be used for classifying a lot of other MVI images taken under the same condition (e.g., the same kind of tree, the same weather condition, the same time in a day), and the pixel classification task becomes automatic massive processing without much human interference.

In the second iteration, the part of the data that corresponds to the trees is filtered out for further clustering. The number of cluster is set as 3 in the hope of finding three clusters which correspond to branches, shadows and sunlit leaves. One can observe from the original images that the branches, shadows and sunlit leaves are easier to tell apart from the NIR image than from the VIS image. So we weight *nir* 10 times heavier than *vis* this time. This time, with the same amount of memory, but a smaller set of data (263401 tuples versus 485100 tuples), the clustering can be done in a much finer granularity which results in better quality. It takes BIRCH a total of 29.08 seconds (including two scans of the subset of VIS and NIR values from ASCII file on disk). Figure 20 shows the clusters as well as their corresponding parts of image resulted from the second iteration.

Again visually, one can see that the three clusters are a satisfactory classification of the branches, shadows and sunlit leaves. However, in general, it may takes the usr a few more iterations to decide on the weights. Once the weights are decided, they can be used for classifying tree pixels into branches, shadows and sunlit leaves for other MVI images taken under the same condition.

6.1.4 User Evaluation

Following are the feedback we obtained from the soil scientists in the department of soil science at University of Wisconsin-Madison , who have used our tool in their canopy structure research[36, 37].

Before our tool, they use the 2-dimensional histogram to split the pixels into two classes: sky and foliage. It simply plots VIS pixel values along the Y-axis, and NIR pixel values along the X axis. It shows clusters of pixels according to their relative intensities in the image. The problem with this approach is that there are many transition (mixed) pixels from the sky to the foliage in each image. Thus, it is not always easy to decide where the division of the sky and foliage class should be drawn on the histogram. BIRCH allows for a less subjective method to split these classes because clustering is searching for an optimized partition.

Besides, our tool has the ability to weight each band's contribution in the canopy, and then the combination of BIRCH and DEVISE allows for visualing the effects of weights in the pixel classification. This is very important because in some images one has to weight the VIS dimension more than the NIR dimension since leaf pixels can become confused with sky pixels under partly-cloudy sky conditions in the near infrared wavelength band; or to weight the NIR dimension more than the VIS dimension in other situations. So the user can explore the correct weight setting through the visualization of the clustering results in our tool.

Following are the quotes of some of their comments.

"BIRCH was also able to find very small gaps in the canopy which might otherwise be lost in the use of the 2-D histogram approach."

"Another plus in using BIRCH is that if the right number of clusters are solved for, we are also able to pull out the branch structure in the forest canopy, which is almost impossible to do with the 2-D histogram. Given this additional information, there is no doubt that BIRCH is a truly valuable tool in our scheme for image processing."

"The time involved using BIRCH is also a bonus. Using 2-D histograms requires many processing steps (algorithms) which involve numerous iterations to judge the classification that is made, and whether it is a reasonable solution for a particular image. It may take 1 hour of processing per image pair using the 2-D histogram approach. Given that we analyze 100's of images, the processing time is incredible. BIRCH is able to take the raw data in an MVI image pair, and solve for 3-4 clusters in less than 5 minutes on a HP-9000 series (735) workstation."

"What is also unique to BIRCH is that its treatment of image data is such that we are able to write other algorithms that easily read the data that is processed after clustering, which is needed in most cases to get as much information as possible from the image. Clustering of the images is just the FIRST STEP!!! There is much more data processing that is needed once the clustering takes place, but having an algorithm such as BIRCH which performs the clustering in minimal time is a very important time-saving, reliable process."

"We use BIRCH as a beginning step to our image processing. We do much more with the images after they are clustered. Our processing doesn't end with BIRCH. The science in each image comes out as we move past the clustering. However, without the initial clustering using BIRCH, we would be unable to do ANYTHING to process the images as they are. Thus BIRCH is an important first step for us!"

6.1.5 BIRCH,CLARANS and KMEANS on Pixel Classification

These MVI image pairs are real datasets that we can use to compare BIRCH, CLARANS and KMEANS. In Table 13, two different MVI image pairs are used. Image1 (see Figure

Table 13: BIRCH,CLARANS,KMEANS on Pixel Classification

Method	Image1: N=485100,K=2			Image2: N=485100,K=5		
	Time	D	#Scan	Time	D	#Scan
BIRCH	50	0.5085	2	53	0.6269	2
CLARANS	368	0.5061	330	642	0.6292	523
KMEANS	8.3	0.5015	5	30	0.6090	48

18) is the pair that we have used in the previous example, where we try to separate 485100 pixels into 2 categories: (1) tree and (2) background. Image2 (see Figure 21) is another image pair, where the background is partly cloudy sky, and we try to separate 485100 pixels into 5 categories: (1) very bright part of sky, (2) ordinary part of sky, (3) clouds, (4) sunlit leaves (5) tree branches and shadows on the trees. For CLARANS and KMEANS, again, the running time is measured with all pixels are read into memory and then scanned inside memory. Whereas for BIRCH, the memory used is only 5% percent of the dataset size, and the dataset is scanned from disk (Each scan of the dataset from disk took about 4.5 seconds). From the table one can see that:

1. For both MVI image pairs, BIRCH, CLARANS and KMEANS have almost the same quality. The qualities of CLARANS and KMEANS are slightly better than that of BIRCH. To explain this, one should know that (1) CLARANS and KMEANS are "hill-climbing" methods and they stop after they reach some local optimal clustering; (2) in this application, N is not too big, K and d are very small, and the pixel distribution in terms of VIS and NIR values is very simple in modality, so the "hill" that they climb tends to be very simple without many bad local optimal solutions prevailing, and CLARANS and KMEANS can usually reach a pretty good clustering in the "hill". BIRCH stops after scanning the data only twice, but it reaches a clustering almost as good as those optimal ones found by CLARANS and KMEANS.

2. For CLARANS and KMEANS, their running times and their numbers of scans on the dataset are very sensitive to the two different MVI image pairs used. Whereas

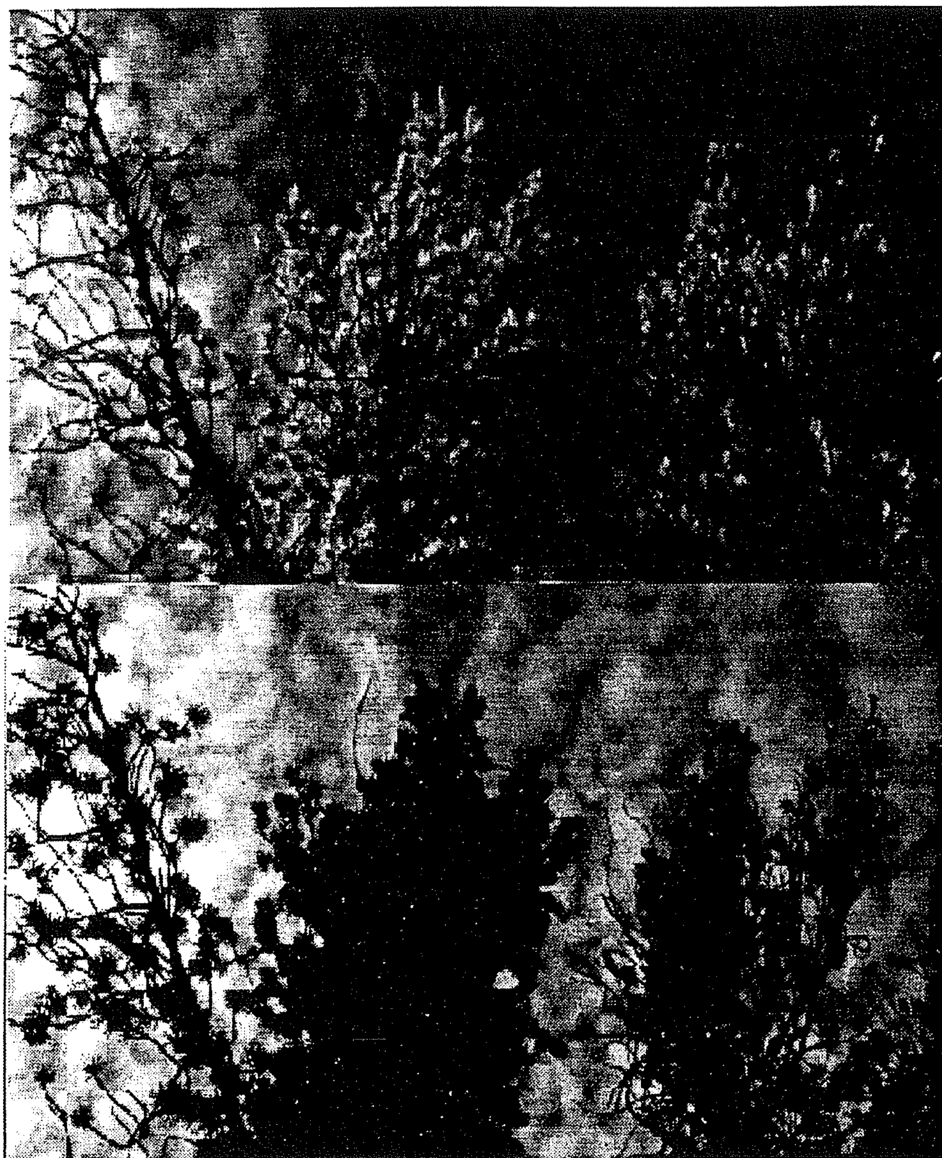


Figure 21: Another Example of MVI image

the running times and the numbers of scans on the dataset of BIRCH are stable for the two different MVI image pairs.

3. It would be hard for CLARANS and KMEANS to scale up for larger and obscurer images because when N and K becomes larger, (1) the "hill" they climb tends to be more complex with plentiful local optimal solutions, which in turn makes CLARANS and KMEANS easier to get stuck in a bad local optimal clustering; and (2) it is impossible to store all pixels in memory, and considering the time for each scan, CLARANS and KMEANS can be slow.

6.2 Codebook Generation in Image Compression

6.2.1 Motivation

The second application is motivated by digital image compression technology. Digital image compression [38] is the technology of reducing image data to save storage space and transmission bandwidth. *Vector quantization* [24] is a widely used image compression/decompression technique which operates on blocks of pixels instead of pixels for better efficiency.

In vector quantization[24], the original image is first decomposed into small rectangle blocks, and each block is represented as a vector. Then each vector is **encoded** with the *codebook*, i.e., finding its *nearest codeword* from the *codebook*, and later is **decoded** with the same *codebook*, i.e., using its *nearest codeword* in the *codebook* as its value. So given the training vectors (derived from the training image) and the desired codebook size (i.e., number of codewords), the main problem of vector quantization is how to generate the codebook.

6.2.2 BIRCH,CLARANS and LBG on Image Compression

A commonly used codebook generating algorithm is the LBG algorithm [39]. LBG is a KMEANS algorithm but with two specific modifications that must be made to be

suitable for image compression. One is to try to avoid getting stuck at a bad local optimum by starting with an initial codebook of size 1 (instead of the desired size), and proceeding by refining and splitting the codebook iteratively. Another is that if empty cells in the codebook are found during the refinement, it fills the empty cells via codebook splitting. Following is a brief description of the LBG algorithm.

1. It uses the GLA algorithm [24] (KMEANS with empty cell filling strategy) to find the "optimal" codebook of current size.

2. If it reaches the desired codebook size, then stop; otherwise it doubles the current codebook size, perturbs and splits the current "optimal" codebook and goes to Step 1.

The above LBG algorithm invokes many extra scans of the training vectors during the codebook optimizations at all the codebook size levels before reaching the desired codebook size level, and yet it still can not completely escape locally optimal solutions.

With BIRCH clustering the training vectors (usually of high dimensions, such as 16), we know that from the first 3 phases, with a single scan of the training vectors, the clusters obtained generally capture the major vector distribution patterns and only have minor inaccuracies. So in Phase 3, if we set the number of clusters directly as the desired codebook size, and use the centroids of the obtained clusters as the codebook, then in Phase 4, we can use GLA to further optimize the codebook and save a lot of scans of the training vectors. So in contrast with LBG,

- the initial codebook from the first 3 phases of BIRCH is not likely to lead to a bad locally optimal codebook;

- Using BIRCH to generate the codebook will involve much less scans of the training vectors.

Also we intend to show how BIRCH, CLARANS and LBG perform on high dimensional (d=16) real datasets in this application.

We use two very different images: *Lena* and *Baboon* (each with 512x512 pixels) as examples to compare BIRCH, CLARANS and LBG's performances in terms of the number of the running time, scans on the dataset, distortion and entropy [38, 49]. First the training vectors are derived by blocking each image into 4x4 blocks (d=16), and the desired codebook size is set to 256. Distortion is defined as the sum of squares of Euclidean distances from all training vectors to their nearest codewords. It is a widely used quality measurement, and smaller distortion values imply better qualities. Entropy is the average number of bits needed for encoding a training vector, so lower entropy also means better compression.

Table 14 summarizes the performance results. The *Time* is the total time of generating the initial codebook of the desired size (256) and then using GLA algorithm to refine the codebook. The *#Scans* means the total scans of the dataset in order to reach the final codebook. *Distortion* and *Entropy* are obtained by compressing the image using the final codebook. One can see that for both images,

1. using BIRCH to generate the initial codebook is consistently better than using using LBG in terms of the running time, scans of data, distortion and entropy;

2. the final codebook obtained through CLARANS is slightly better than that obtained through BIRCH in terms of distortion only, whereas is significantly worse in terms of the other three aspects.

Readers can perceive the compressed images in Figure 22 through 27 for easy visual quality comparisons.

Table 14: BIRCH, CLARANS and LBG on Image Compression

Method	Lena			
	Time	#Scans	Distortion	Entropy
BIRCH	15	9	712	6.70
CLARANS	2222	8146	693	7.43
LBG	19	46	719	7.38
Method	**Baboon**			
	Time	#Scan	Distortion	Entropy
BIRCH	17	8	5097	7.15
CLARANS	2111	7651	5070	7.70
LBG	31	42	5155	7.73

Figure 22: Lena Compressed with BIRCH Codebook

Figure 25: Baboon Compressed with BIRCH Codebook

Figure 23: Lena Compressed with CLARANS Codebook

Figure 26: Baboon Compressed with CLARANS Codebook

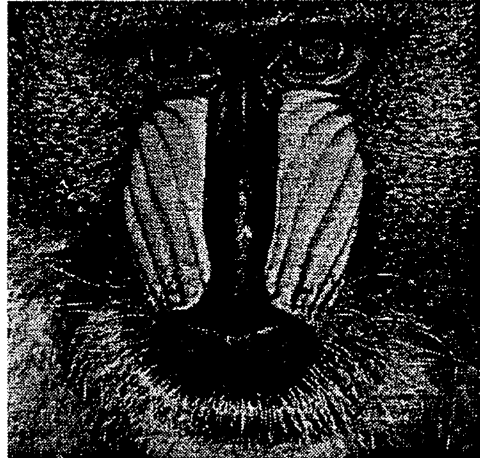
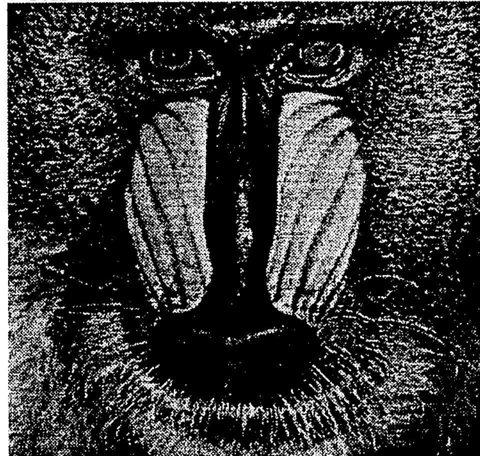


Figure 24: Lena Compressed with LBG Codebook

Figure 27: Baboon Compressed with LBG Codebook



Chapter 7

Implementation Issues

7.1 Code Components

BIRCH is implemented in C++. There are four code components in BIRCH: (1) utility, (2) data clustering, (3) density estimation, and (4) data input and preparation.

7.1.1 Utility

This component contains all the utility code needed in BIRCH, but not just restricted to the use in BIRCH. It is further decomposed into the following modules:

- vector module: provides d-dimensional vector and vector operations such as addition, subtraction, dot product, distances, input and output.
- rectangle (or MBR[27]) module: provides d-dimensional rectangle and rectangle operations such as addition, intersection, overlap, containment, input and output.
- timing module: provides convenient timing facilities for performance evaluations.
- random number module: generates random numbers based on a given distribution.
- sample and regression module: takes samples and performs regressions on the samples.

7.1.2 Data Clustering

This component is the core of BIRCH. It contains the following modules:

- CF entry module: provides d-dimensional CF entry and operations such as addition, distances, radius/diameter, etc.
- CF-tree module: provides node (leaf or nonleaf) interface and relevant insertion, splitting and rebuilding algorithms.
- Status module: manages status information and heuristics for a CF-tree.
- Phase 1 and 2 module: manages split buffering, outlier queuing and CF-tree building.
- Phase 3: adapts existing global or semi-global clustering algorithms such as HC and CLARANS.
- Phase 4: provides clustering refinement algorithm and different output options.

7.1.3 Density Estimation

This component is the preliminary work for using BIRCH for density estimation. More experiments are needed to compare its performance with other related methods.

7.1.4 Data Input and Preparation

This component is able to read data from different media and/or with different formats. Then for each tuple it reads, it extracts the specified attribute projections for data clustering and density estimation. This part of code was provided by Mr. R. Kent Wenger.

7.2 BIRCH Execution

BIRCH is executed by entering the following command: *birch para scheme proj data* where *birch* is the executable and *para*, *scheme* and *proj* are the input files explained as below.

7.2.1 Input of BIRCH

BIRCH takes three files as its input, they are:

1. *para*: parameter file that contains the parameter setting for running BIRCH. Another file *para.config* contains a detailed explanation of how to set the parameter file. In the parameter file, the user can specify, for example, the following information:

   (a) Do data clustering or density estimation?

   (b) Amount of memory for building CF-tree, amount of memory for buffering splits, amount of disk space for queuing outliers.

   (c) Then for each projection of attributes:

   - Whether and how to weight and/or shift the attribute values?
   - Page size?
   - Which distance measurement is to be used?
   - Which threshold definition is to be used?
   - Which quality measurement is to be used?
   - Whether the outlier options are to be used?
   - Whether the buffering option is to be used?

2. *data*: data file that contains the data.

3. *schema*: scheme file that describes the scheme of the data including data format (binary or ascii), comment signs and separators used in the data file, names and types of attributes.

4. *proj*: projection file that specifies the desired attribute projections for doing data clustering or density estimation concurrently.

7.2.2 Output of BIRCH

1. *para+scheme+proj+data-log*: log file that logs execution information such as running time and quality of each phase.

2. *para+scheme+proj+data-i-cluster*: cluster file for attribute projection i that contains clusters resulted from Phase 3 for attribute projection i. Each cluster in the file is represented as the form of CF entry in terms of that attribute projection.

3. *para+scheme+proj+data-i-refcluster*: refined cluster file for attribute projection i that contains clusters resulted from Phase 4. Each cluster in the file is represented as the form of CF entry.

4. *para+scheme+proj+data-i-label*: label file for attribute projection i that is generated optionally only when -DLABEL compilation option is used. In this file, each record_id is matched with its belonging cluster_id according to the clustering based on attribute projection i.

5. *para+scheme+proj+data-i-dat-j*: filtered data file for cluster j under attribute projection i that is generated optionally only when -DFILTER compilation option is used. In this file, the data corresponding to the cluster j of clustering under attribute projection i is stored.

6. *para+scheme+proj+data-i-summary*: summary file for attribute projection i that is generated only when -DSUMMARY compilation option is used. In this file, each cluster has a summary in the form of CF entry in terms of all the numeric attributes.

Chapter 8

Conclusions and Future Research

8.1 Data Clustering

BIRCH provides an efficient data clustering algorithm for very large datasets. It makes a large data clustering problem tractable by concentrating on a compact in-memory summary of the dataset. This way the I/O cost is minimized and the problem is reduced to the range where most existing algorithms can be adapted to perform well. It utilizes measurements that capture the natural closeness of data and at the same time can be stored and updated accurately and incrementally in a height-balanced hierarchy. BIRCH works with datasets of any size and memory of any size, and it tries to scan the data only once.

Experimentally, BIRCH is shown to perform consistently well on different patterns of datasets with different input orders. It is significantly superior to *CLARANS* and *KMEANS* in terms of speed, quality, stability and scalability on both synthetic and real datasets.

Proper parameter setting and further generalization are two important topics to explore in the future. We will look into:

1. more sophisticated heuristic method for increasing the threshold dynamically,
2. other threshold requirements and related insertion, rebuilding algorithms,
3. confidence measurements.

We will also study how to make use of the clustering information obtained from BIRCH to help solve existing problems such as storage optimization, data partition and index construction.

8.2 Density Estimation

One continuing work in BIRCH is multi-dimensional density estimation. For simplicity, we will use 1-dimensional data and equations for illustration of density estimation, however it can be generalized to any dimensions in BIRCH easily.

Density estimation, as discussed here, is the construction of an estimate of the *density function* from observed data. Informally, a density function $f(x)$ gives a natural description of the distribution of a random variable x, and allows the associated probability function $P(x)$ to be computed using the integral $\int_{-\infty}^{x} f(t)dt$. It is a natural way to explore and present data. It can give valuable insight into such features as skew and multimodality in the data, and thus provoke further analysis by other means. Data clustering and density estimation are two related but slightly different topics. On one hand, compared with data clustering, density estimation is a more general way of exploring and presenting the data because the *modes* (or "bumps") in the density estimate actually correspond to the clusters in the data. On the other hand, data clustering searches for an *optimized partition* of the dataset whereas density estimation does not address the optimal partition issue.

8.2.1 Previous Work

Prior work in density estimation[55, 15, 56], has been in statistics. There are mainly two categories of approaches to density estimation: *parametric and nonparametric.* The parametric approach assumes that the underlying density function $f(x)$ belongs to some parametric family of distributions, such as the normal or gamma families. Then samples are used to determine the parameters of the assumed function forms. The main drawback of the parametric approach is that its estimation accuracy depends heavily on this assumption, which might not be true in reality sometimes. On the contrast, the nonparametric approach does not assume any pre-specified functional forms for the underlying density function $f(x)$, and it is sometimes referred to as "let the data speak for themselves". So it is generally a more useful preliminary data analysis method in

reality.

Histogram

The oldest and most well-known nonparametric method for density estimation is the *histogram* method [55, 15, 56]. It assumes that the data range is known, then it cuts the range into disjoint and contiguous intervals, or bins, of equal or different widths, then within each bin, uniformity is assumed. So given N data points, the histogram estimation of $f(x)$ is defined as:

$$\hat{f}(x) = \frac{1}{N} \times \frac{number\ of\ X_i\ in\ the\ same\ bin\ as\ x}{width\ of\ the\ bin\ containing\ x} \tag{8.14}$$

The main drawbacks of the histogram method are:

- Curse of dimensionality: the number of bins will grow exponentially as the dimension increases.
- Some prior knowledge of the data range must be known in advance in order to allocate the bins effectively, so it can not be an incremental method.
- The discontinuity of $\hat{f}(x)$ makes the derivatives or other smoothing metrics unavailable, and hence inconvenient for further more advanced analysis.
- It is difficult to allocate bins, and the estimation accuracy depends not only on the bin widths but also on the bin locations.

Kernel Method

Apart from the histogram method, the *kernel* method [55, 15, 56] is the most mathematically-studied and commonly-used nonparametric density and probability estimation method. Given N data points, the kernel estimation of $f(x)$ is defined as:

$$\hat{f}(x) = \frac{1}{N} \times \sum_{i=1}^{N} \frac{1}{h} K(\frac{x - X_i}{h}) \tag{8.15}$$

where the **kernel function** $K(x)$ is generally a symmetric, bounded density function, for instance, the standard normal density function, and the h is called **smoothing parameter**. Imagine it intuitively, a "bump" is placed on each data point, and the sum of all "bumps" reflects the overall distribution of all data points. The kernel function $K(x)$ determines the shape of each bump while the smoothing parameter h determines the width, or affecting scope, of each bump.

Then some preferable properties will follow immediately from the above kernel estimation definition:

- No prior knowledge about the data is needed and the estimation is totally data-driven.
- The kernel estimation $\hat{f}(x)$ itself is a density function which inherits all the continuity, differentiability and integrability properties of the selected kernel function $K(x)$, and this allows for further more advanced data analysis.
- $K(x)$ and h are the two factors affecting the accuracy. However it has been shown[56] that the choice of the kernel function $K(x)$ is not very crucial, and the estimation accuracy is primarily affected by the smoothing parameter h.

However if the *kernel* method is applied to very large datasets, it has its own problems:

- It is very time/space expensive because for each data point, it keeps a "bump", and there will be N distinct terms, or "bumps" in the representation of $\hat{f}(x)$. So first, it needs $O(N)$ space to store $\hat{f}(x)$, and then for calculating $\hat{f}(x)$ and/or $\hat{P}(x)$ at a specific value x, it needs to scan all N terms.
- Another problem is the difficulty and complexity of finding the proper h, so that given N data points, $\hat{f}(x)$ can reflect $f(x)$ very well.

8.2.2 CF-kernel Method

With CF-tree, a new *CF-kernel* method [59] is proposed that integrates the advantages of the CF-tree[57, 58] and the *kernel* method[55, 15, 56], and at the same time overcomes the efficiency difficulty met by the *kernel* method. The idea is that first we summarize the dataset into an in-memory CF-tree [57, 58], second instead of placing a **kernel function** on each single data point, we place a **CF-kernel function** on each subcluster (or leaf entry of the CF-tree). This way the efficiency is definitely improved.

So CF-tree is used as a dynamic and incremental way to *bin* data. However, it differs from the existing *binned* kernel methods [56] in that:

- it does not require the data range in advance for allocating bins, instead, it allocate "bins" dynamically and incrementally according to the data distribution and the available memory, the number of bins does not grow with dimension but with data expansion;
- more information is stored for each bin (CF, which allows to construct an density function to approximate the data distribution inside a bin);

The following questions remain about the *CF-kernel* function:

1. How to define the **CF-kernel function** so that the *kernel* estimation and the *CF-kernel* estimation will be as close as the available memory allows?
2. How easy is it to compute the **CF-kernel function**?

Following is our preliminary results about the *CF-kernel* method with respect to the above two questions. First, theoretically, assume that the N_i data points in subcluster i are drawn independently from an empirically-known distribution whose density function is $g_i(x)$ ($g_i(x)$ can be approximated by using the information stored in the CF entry), if we define the *CF-kernel function* placed on subcluster i as $\int K_h(x-t)g_i(t)dt$, then we can prove that at any x, the expected value of the difference between the *kernel* estimation and the *CF-kernel* estimation is 0; and the variance of the difference between the *kernel*

estimation and the *CF-kernel* estimation will converge to 0 as the memory increases. Experimentally, we show that the variance converges to 0 almost as fast as exponential. Second, if $g_i(x)$ is chosen to be normal or uniform distribution, then the *CF-kernel function* can be symbolically derived to avoid computing the integral numerically. So the *CF-kernel function* can be computed efficiently as long as the programming language supports $exp(x)$ (exponential function) and $erf(x)$ (error function) in their math library.

However, this work is very preliminary. Further experiments or even modifications are needed to (1) compare it with methods other than the *kernel* method such as sampling; and (2) apply it to real applications such as query optimization.

Appendix A

Proof of CF Representativity Theorem

First, $\vec{X0}$, R, D, $D0$, $D1$, $D2$, $D3$ and $D4$ can be calculated from the CF vectors of the corresponding clusters as shown by their definitions and the following formulae.

$$\begin{aligned} \textit{centroid } \vec{X0} &= \frac{\sum_{i=1}^{N} \vec{X_i}}{N} \\ &= \frac{\vec{LS}}{N} \end{aligned}$$

$$\begin{aligned} \textit{radius } R &= \left(\frac{\sum_{i=1}^{N} (\vec{X_i} - \vec{X0})^2}{N}\right)^{\frac{1}{2}} \\ &= \left(\frac{SS}{N} - \left(\frac{\vec{LS}}{N}\right)^2\right)^{\frac{1}{2}} \end{aligned}$$

$$\begin{aligned} \textit{diameter } D &= \left(\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (\vec{X_i} - \vec{X_j})^2}{N(N-1)}\right)^{\frac{1}{2}} \\ &= \left(\frac{2N*SS - 2*\vec{LS}^2}{N(N-1)}\right)^{\frac{1}{2}} \end{aligned}$$

$$\begin{aligned} \textit{centroid Euclidean distance } D0 &= ((\vec{X0}_1 - \vec{X0}_2)^2)^{\frac{1}{2}} \\ &= \left(\left(\frac{\vec{LS_1}}{N_1} - \frac{\vec{LS_2}}{N_2}\right)^2\right)^{\frac{1}{2}} \end{aligned}$$

$$\begin{aligned} \textit{centroid Manhattan distance } D1 &= |\vec{X0}_1 - \vec{X0}_2| \\ &= \left|\frac{\vec{LS_1}}{N_1} - \frac{\vec{LS_2}}{N_2}\right| \end{aligned}$$

$$\text{average intercluster distance } D2 = \left(\frac{\sum_{i=1}^{N_1}\sum_{j=N_1+1}^{N_1+N_2}(\vec{X_i}-\vec{X_j})^2}{N_1N_2}\right)^{\frac{1}{2}}$$
$$= \left(\frac{N_1 * SS_2 + N_2 * SS_1 - 2 * \vec{LS_1}\,\vec{LS_2}}{N_1N_2}\right)^{\frac{1}{2}}$$

$$\text{average intracluster distance } D3 = \left(\frac{\sum_{i=1}^{N_1+N_2}\sum_{j=1}^{N_1+N_2}(\vec{X_i}-\vec{X_j})^2}{(N_1+N_2)(N_1+N_2-1)}\right)^{\frac{1}{2}}$$
$$= \left(\frac{2(N_1+N_2)*(SS_1+SS_2)-2*(\vec{LS_1}+\vec{LS_2})^2}{(N_1+N_2)(N_1+N_2-1)}\right)^{\frac{1}{2}}$$

$$\text{variance increase distance } D4 = \Big[\sum_{k=1}^{N_1+N_2}\Big(\vec{X_k}-\frac{\sum_{l=1}^{N_1+N_2}\vec{X_l}}{N_1+N_2}\Big)^2$$
$$-\sum_{i=1}^{N_1}\Big(\vec{X_i}-\frac{\sum_{l=1}^{N_1}\vec{X_l}}{N_1}\Big)^2$$
$$-\sum_{j=N_1+1}^{N_1+N_2}\Big(\vec{X_j}-\frac{\sum_{l=N_1+1}^{N_1+N_2}\vec{X_l}}{N_2}\Big)^2\Big]^{\frac{1}{2}}$$
$$= \Big[\frac{SS_1+SS_2}{N_1+N_2}-\Big(\frac{\vec{LS_1}+\vec{LS_2}}{N_1+N_2}\Big)^2$$
$$+\frac{SS_1}{N_1}-\Big(\frac{\vec{LS_1}}{N_1}\Big)^2$$
$$+\frac{SS_2}{N_2}-\Big(\frac{\vec{LS_2}}{N_2}\Big)^2\Big]^{\frac{1}{2}}$$

Since the R and D of each cluster can be calculated from the CF vectors, and $Q1$, $Q2$, $Q3$ and $Q4$ are defined in terms of R's and D's of clusters, they can be calculated from the CF vectors too, which is straightforward from their definitions.

Appendix B

CF-tree Insertion Algorithm

```
1   Node* Insert_Into_CF-tree (Node **Root,Node *CurNode,Entry Ent,Float T) {
2   Node *NewNode; Entry NewEnt;
3   if (CurNode is Nonleaf Node) {
4       Ci = Closest_Child(CurNode,Ent);
5       NewNode=Insert_Into_CF-tree(Root,Ci,Ent,T);
6       if (NewNode==NULL) {
7           Update_CF(CurNode,Ci,Ent);
8           return NULL;
9           }
10      else {
11          Update_CF(CurNode,Ci,Ent);
12          NewEnt=Make_Entry_From_Node(NewNode);
13          NewNode=Insert_To_Nonleaf_Node_Might_Split(CurNode,NewEnt);
14          if (NewNode==NULL) {
15              Merge_Closest_But_Not_Split_Pair_Might_Resplit(CurNode);
16              return NULL;
17              }
18          else {
19              if (CurNode==*Root) {
20                  *Root=Create_New_Root(CurNode,NewNode);
21                  return NULL;
22                  }
23              else return NewNode;
```

```
24                  }
25              }
26          }
27      else { /* CurNode is Leaf Node */
28          Li = Closest_Entry(CurNode,Ent);
29          If (Absorb(Li,Ent) Satisfies T) {
30              Absorb(Li,Ent);
31              return NULL;
32              }
33          else {
34              NewNode = Insert_To_Leaf_Node_Might_Split(CurNode,Ent);
35              if (NewNode==NULL) return NULL;
36              else return NewNode;
37              }
38          }
39  }
```

Appendix C

CF-tree Rebuilding Algorithm

```
1   void Re-build_CF_Tree(t_i,t_{i+1},T_i,T_{i+1}) {
2   t_{i+1}=NULL;
3   CurrentPath=Path_Of_Tree(t_i,(0,...,0));
4   while (CurrentPath exists) {
5       Attach_Nodes_To_NewTree_By_CurrentPath(t_{i+1},CurrentPath);
6       foreach leaf entry on CurrentPath of OldTree, say CurrentEntry, do {
7           Status=Can_Fit_In_ClosestPath(t_{i+1},T_{i+1},ClosestPath,CurrentEntry);
8           if (Status==YES && ClosestPath<CurrentPath)
9               Fit_In_Path(t_{i+1},T_{i+1},ClosestPath,CurrentEntry);
10          else Fit_In_Path(t_{i+1},T_{i+1},CurrentPath,CurrentEntry);
11          }
12      Free_CurrentPath(t_i,CurrentPath);
13      CurrentPath=NextPath_Of_Tree(t_i,CurrentPath);
14      }
15  t_{i+1}=Free_Empty_Nodes(t_{i+1});
16  }
```

Appendix D

Base Workload

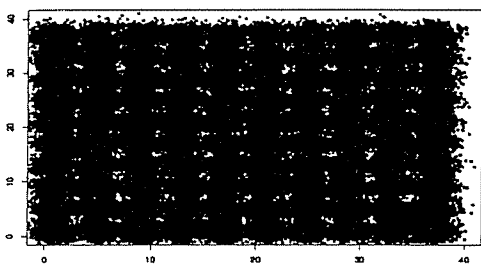


Figure 28: Data of DS1

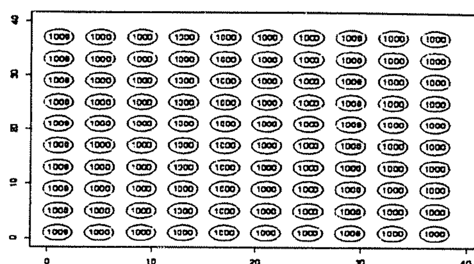


Figure 31: Intended Clusters of DS1

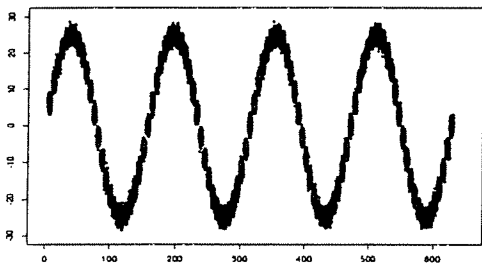


Figure 29: Data of DS2

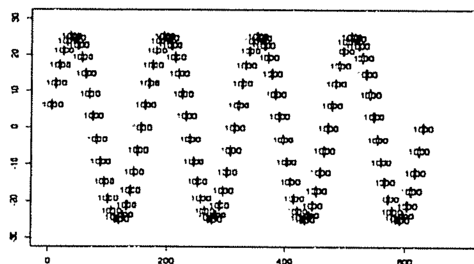


Figure 32: Intended Clusters of DS2

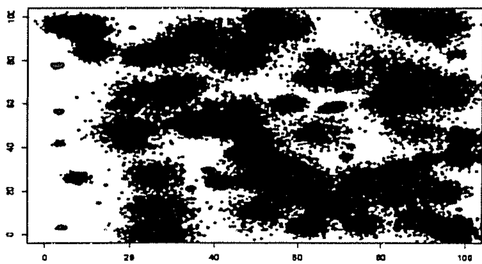


Figure 30: Data of DS3

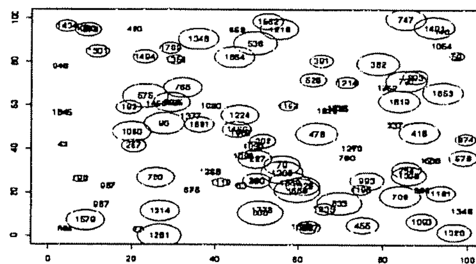


Figure 33: Intended Clusters of DS3

Appendix E

BIRCH on Base Workload

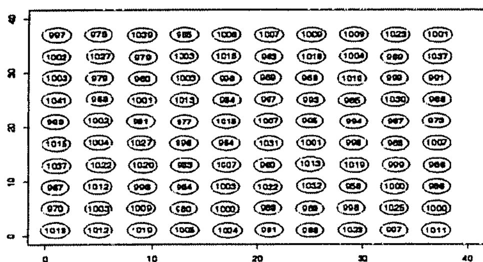


Figure 34: BIRCH Clusters of DS1

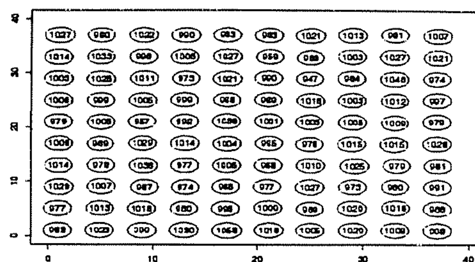


Figure 37: BIRCH Clusters of DS1o

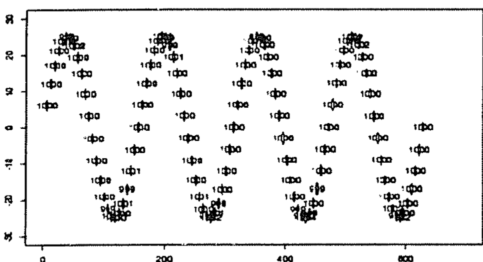


Figure 35: BIRCH Clusters of DS2

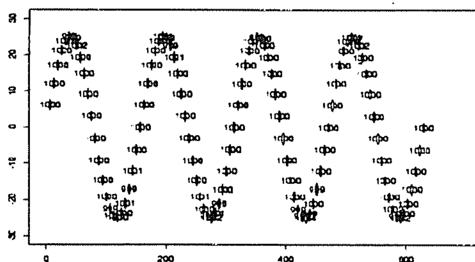


Figure 38: BIRCH Clusters of DS2o

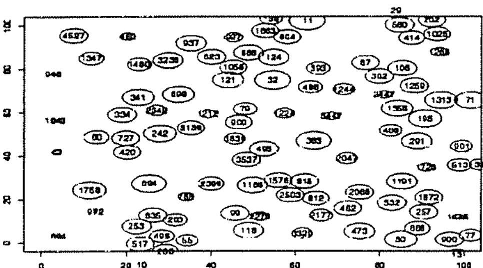


Figure 36: BIRCH Clusters of DS3

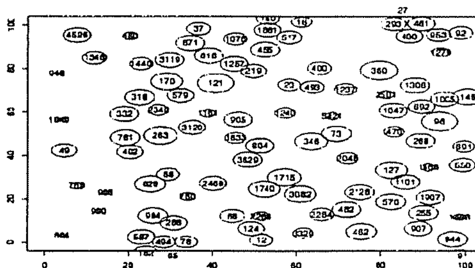


Figure 39: BIRCH Clusters of DS3o

Appendix F

CLARANS on Base Workload

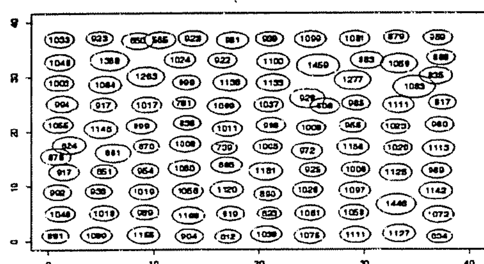


Figure 40: CLARANS Clusters of DS1

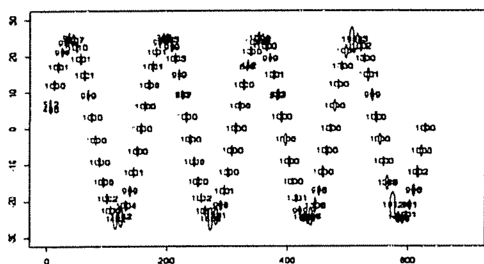


Figure 41: CLARANS Clusters of DS2

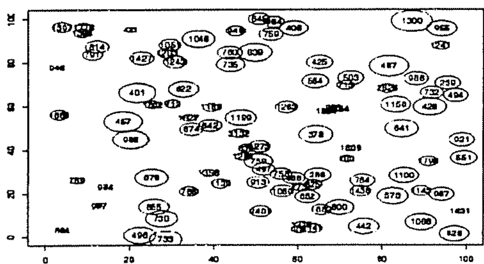


Figure 42: CLARANS Clusters of DS3

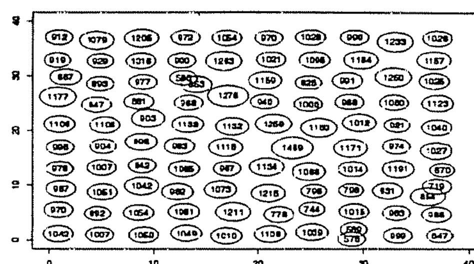


Figure 43: CLARANS Clusters of DS1o

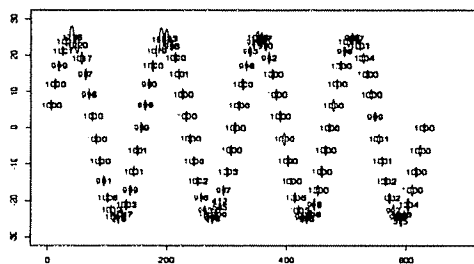


Figure 44: CLARANS Clusters of DS2o

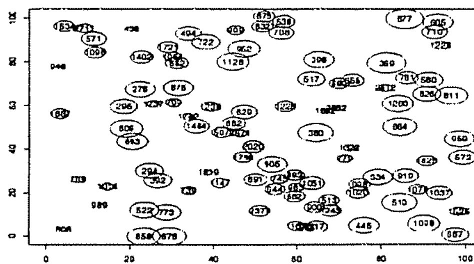


Figure 45: CLARANS Clusters of DS3o

Appendix G

KMEANS on Base Workload

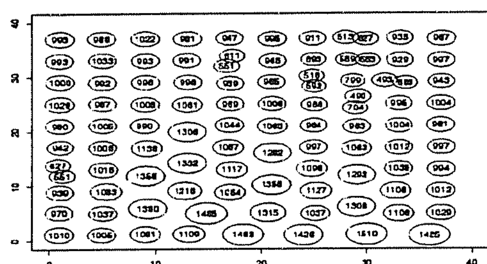


Figure 46: KMEANS Clusters of DS1

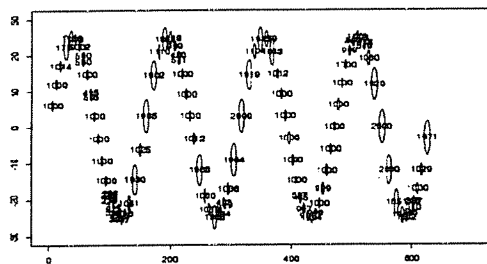


Figure 47: KMEANS Clusters of DS2

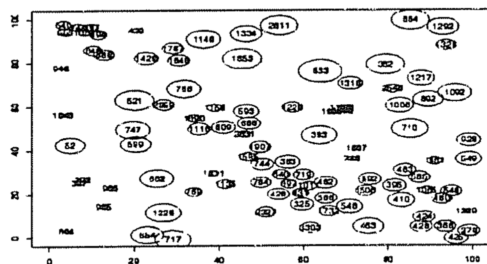


Figure 48: KMEANS Clusters of DS3

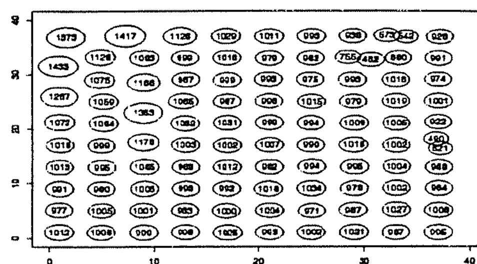


Figure 49: KMEANS Clusters of DS1o

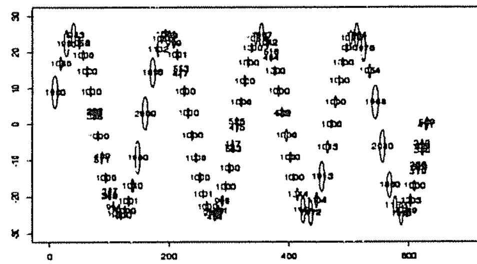


Figure 50: KMEANS Clusters of DS2o

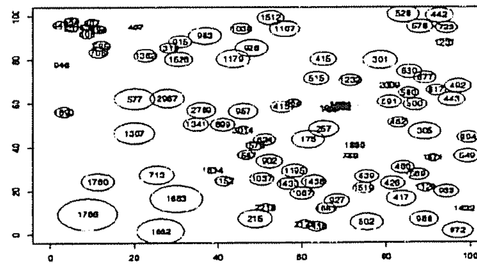


Figure 51: KMEANS Clusters of DS3o

Appendix H

Effects of Outlier Options

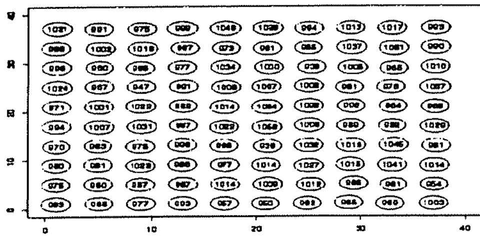


Figure 52: BIRCH Clusters of DS4 with Outlier Options On

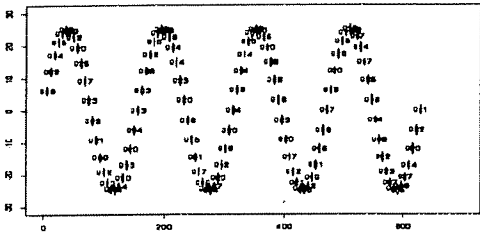


Figure 53: BIRCH Clusters of DS5 with Outlier Options On

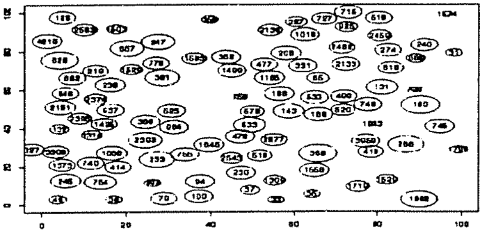


Figure 54: BIRCH Clusters of DS6 with Outlier Options On

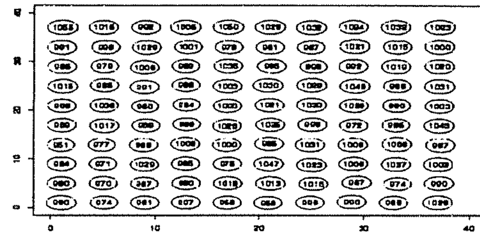


Figure 55: BIRCH Clusters of DS4 with Outlier Options Off

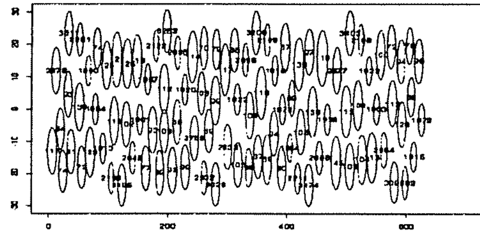


Figure 56: BIRCH Clusters of DS5 with Outlier Options Off

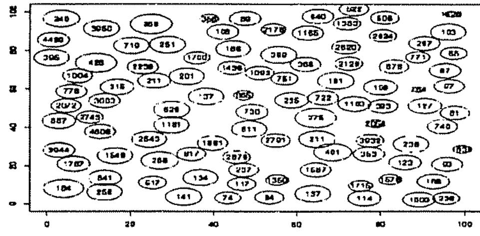


Figure 57: BIRCH Clusters of DS6 with Outlier Options Off

Bibliography

[1] Arie Segev, and Abhirup Chatterjee, *Supporting Statistics In Extensible Databases: A Case Study*, Scientific and Statistical Database Management, 1994.

[2] Rakesh Agrawal, Christos Faloutsos, and Arun Swami, *Efficient Similarity Search in Sequence Databases*, Proc.of the Fourth International Conf. on Foundations of Data Organization and Algorithms, Chicago, U.S.A., Oct. 1993.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami, *Database Mining: A Performance Perspective*, IEEE Transactions on Knowledge and Data Engineering, 5(6), 1993, Special Issue on Learning and Discovery in Knowledge-Based Databases.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami, *Mining Association Rules between Sets of Items in Large Databases*, Proc. of the ACM SIGMOD Conf. on Management of Data, Washington, D.C., May 1993.

[5] Rakesh Agrawal, and Ramakrishnan Srikant, *Mining Sequential Patterns*, Proc. of the 11th IEEE Int'l Conf. on Data Engr., Taipei, Taiwan, Mar. 1995.

[6] R.A.Becker, J.M.Chambers and A. R. Wilks, *The New S Language: A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole Advanced Books and Software, 1988.

[7] J. Buhmann, and T.Hofmann, *A Maximum Entropy Approach to Pairwise Data Clustering*, Proc. of Int'l Conf. on Pattern Recognition, Hebrew Jerusalem, IEEE Computer Society Press, 1994.

[8] Norbert. Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles*, Proc. of ACM SIGMOD Int. Conf. on Management of Data, 322-331,1990.

[9] Peter Cheeseman, James Kelly, Matthew Self, et al., *AutoClass : A Bayesian Classification System*, Proc. of the 5th Int'l Conf. on Machine Learning, Morgan Kaufman, Jun. 1988.

[10] Michael Cheng, Miron Livny, and Raghu Ramakrishnan, *Visual Analysis of Stream Data*, Proc. of IS&T/SPIE Conf. on Visual Data Exploration and Analysis, San Jose, CA, Feb. 1995.

[11] T. Dean, J. Allen, and Y.Aloimonos, *Artificial Intelligence: Theory and Practice*, Benjamin/Cummings Pub. 1995.

[12] Richard Duda, and Peter E. Hart, *Pattern Classification and Scene Analysis*, Wiley, 1973.

[13] R. Dubes, and A.K. Jain, *Clustering Methodologies in Exploratory Data Analysis* Advances in Computers, Edited by M.C. Yovits, Vol. 19, Academic Press, New York, 1980.

[14] Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, N.J., 1974.

[15] Luc Devroye, *A Course in Density Estimation*, Birkhäuser Boston, 1987.

[16] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu, *A Database Interface for Clustering in Large Spatial Databases*, Proc. of 1st Int'l Conf. on Knowledge Discovery and Data Mining, 1995.

[17] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu, *Knowledge Discovery in Large Spatial Databases: Focusing Techniques for Efficient Class Identification*, Proc. of 4th Int'l Symposium on Large Spatial Databases, Portland, Maine, U.S.A., 1995.

[18] Christos Faloutsos, and Ibrahim Kamel, *Beyond Uniformity and Independence: Analysis of R=trees Using the Concept of Fractal Dimension* PODS of 1994.

[19] M.C. Ferris, and O.L. Mangasarian, *Linear Programming with MATLAB*, preliminary version of textbook.

[20] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos, *Fast Subsequence Matching in Time-Series Databases*, Proc. of ACM SIGMOD Conf., 1994.

[21] E. A. Feigenbaum, and H. Simon, *EPAM-like models of recognition and learning*, Cognitive Science, vol. 8, 1984, 305-336.

[22] Douglas H. Fisher, *Knowledge Acquisition via Incremental Conceptual Clustering*, Machine Learning, 2(2), 1987

[23] Douglas H. Fisher, *Iterative Optimization and Simplification of Hierarchical Clusterings*, Technical Report CS-95-01, Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235.

[24] A. Gersho and R. Gray, *Vector quantization and signal compression*, Boston, Ma.: Kluwer Academic Publishers, 1992.

[25] John H. Gennari, Pat Langley, and Douglas Fisher, *Models of Incremental Concept Formation*, Artificial Intelligence, vol. 40, 1989, 11-61.

[26] A.D.Gordon, *Classification* Chapman and Hall, 1981.

[27] A. Guttman, *R-trees: a dynamic index structure for spatial searching*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984.

[28] Jiawei Han, Yandong Cai, and Nick Cercone, *Knowledge Discovery in Databases : An Attribute Oriented Approach*, Proc. of the VLDB Conf., Vancouver, Canada, 1992.

[29] Maurice Houtsma and Arun Swami, *Set-oriented Mining of Association Rules*, Research Report RJ 9567, IBM Almaden Research Center, San Jose, 1993.

[30] Harold M. Hastings, and George Sugihara, *Fractals: A User's Guide for the Natural Sciences*, Oxford Science Publications, 1993.

[31] C. Huang, Q. Bi, G. Stiles, R. Harris, *Fast Full Search Equivalent Encoding Algorithms for Image Compression Using Vector Quantization*, IEEE Trans. on Image Processing, vol. 1, no. 3, July, 1992.

[32] J. A. Hartigan, and M. A. Wong, *A K-Means Clustering Algorithm*, Appl. Statist., vol. 28, no. 1, 1979.

[33] D.J.Hand, *Discrimination and Classification*, John Wiley & Sons, 1981.

[34] D.J.Hand, *Kernel Discriminant Analysis*, Research Studies Press, A Division of John Wiley & Sons Ltd., 1982.

[35] Leonard Kaufman, and Peter J. Rousseeuw, *Finding Groups in Data - An Introduction to Cluster Analysis*, Wiley Series in Probability and Mathematical Statistics, 1990.

[36] C.J. Kucharik, and J.M. Norman, *Measuring Canopy Architecture with a Multiband Vegetation Imager (MVI)* Proc. of the 22nd conf. on Agricultural and Forest Meteorology, Ame rican Meteorological Society annual meeting, Atlanta, GA, Jan 28-Feb 2, 1996.

[37] C.J. Kucharik, J.M. Norman, L.M. Murdock, and S.T. Gower, *Characterizing Canopy non-randomness with a Multiband Vegetation Imager (MVI)*, Submitted to Journal of Geophysical Research, to appear in the Boreal Ecosystem-Atmosphere Study (BOREAS) special issue.

[38] Weidong Kou, *Digital Image Compression Algorithms and Standards*, Kluwer Academic Publishers, 1995.

[39] Yoseph Linde, Andre's Buzo, and Robert M. Gray, *An Algorithm for Vector Quantizer Design*, IEEE trans. comm., Vol. COM-28, No.1, Jan. 1980.

[40] Michael Lebowitz, *Experiments with Incremental Concept Formation : UNIMEM*, Machine Learning, 1987.

[41] R.C.T.Lee, *Clustering analysis and its applications*, Advances in Information Systems Science, Edited by J.T.Toum, Vol. 8, pp. 169-292, Plenum Press, New York, 1981.

[42] Stuart P. Lloyd, *Least Squares Quantization in PCM*, IEEE trans. information theory, Vol. IT-28, No.2, Mar. 1982.

[43] Kathleen McKusick and Kevin Thompson, *COBWEB/3 : A Portable Implementation*, NASA Ames Research Center, Artificial Intelligence Research Branch, TR FIA-90-6-18-2, June, 1990.

[44] F. Murtagh, *A Survey of Recent Advances in Hierarchical Clustering Algorithms*, The Computer Journal 1983.

[45] Raymond T. Ng and Jiawei Han, *Efficient and Effective Clustering Methods for Spatial Data Mining*, Proc. of VLDB 1994.

[46] Clark F. Olson, *Parallel Algorithms for Hierarchical Clustering*, Technical Report, Computer Science Division, Univ. of California at Berkeley, Dec.,1993.

[47] Franco P. Preparata, and Michael Ian Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1990.

[48] Gregory Piatetsky-Shapiro, and William J. Frawley, editors, *Knowledge Discovery in Databases*, AAAI/MIT Press, 1991.

[49] Majid Rabbani, and Paul W. Jones, *Digital Image Compression Techniques*, SPIE Optical Engineering Press, 1991.

[50] Rawlings, John O., *Applied regression analysis : a research tool*, Wadsworth & Brooks/Cole Advanced Books & Software, 1988.

[51] Arie Segev, and Abhirup Chatterjee, *Supporting Statistics In Extensible Databases: A Case Study*, Scientific and Statistical Database Management, 1994.

[52] Jude W. Shavlik, and Thomas G. Dietterich, editors, *Readings in Machine Learning*, Morgan Kaufmann, 1990.

[53] Praveen Seshadri, Miron Livny and Raghu Ramakrishnan, *SEQ: A Model for Sequence Database*, Preprint, 1994.

[54] David W. Scott, and George R. Terrell, *Biased and Unbiased Cross-Validation in Density Estimation*, Journal of the American Statistical Association, Vol. 82, No. 400, p1131-1146, Dec., 1987.

[55] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, London ; New York : Chapman and Hall, 1986.

[56] M.P. Wand, and M.C. Jones, *Kernel Smoothing*, Chapman and Hall, 1995.

[57] Tian Zhang, Raghu Ramakrishnan, and Miron Livny, *BIRCH: An Efficient Data Clustering Method for Very Large Databases*, Proc. of ACM SIGMOD Int'l Conf. on Management of Data, p103-114, June 1996, Montreal, Canada.

[58] Tian Zhang, Raghu Ramakrishnan, and Miron Livny, *Interactive Classification of Very Large Datasets with BIRCH*, Proc. of Workshop on Research Issues on Data Mining and Knowledge Discovery (in cooperation with ACM-SIGMOD'96), June 196, Montreal, Canada.

[59] Miron Livny, Raghu Ramakrishnan, and Tian Zhang, *Fast Density and Probability Estimations Using CF-Kernel Method for Very Large Databases*, Technical Report, July, 1996.