Clustering Large Datasets in Arbitrary Metric Spaces

Venkatesh Ganti* Raghu Ramakrishnan Johannes Gehrke†
Computer Sciences Department, University of Wisconsin-Madison
Allison Powell‡ James French§
Department of Computer Science, University of Virginia, Charlottesville

Abstract

Clustering partitions a collection of objects into groups called clusters, such that similar objects fall into the same group. Similarity between objects is defined by a distance function satisfying the triangle inequality; this distance function along with the collection of objects describes a distance space. In a distance space, the only operation possible on data objects is the computation of distance between them. All scalable algorithms in the literature assume a special type of distance space, namely a k-dimensional vector space, which allows vector operations on objects.

We present two scalable algorithms designed for clustering very large datasets in distance spaces. Our first algorithm BUBBLE is, to our knowledge, the first scalable clustering algorithm for data in a distance space. Our second algorithm BUBBLE-FM improves upon BUBBLE by reducing the number of calls to the distance function, which may be computationally very expensive. Both algorithms make only a single scan over the database while producing high clustering quality. In a detailed experimental evaluation, we study both algorithms in terms of scalability and quality of clustering. We also show results of applying the algorithms to a real-life dataset.

1. Introduction

Data clustering is an important data mining problem [1, 8, 9, 10, 12, 17, 21, 26]. The goal of clustering is to partition a collection of objects into groups, called *clusters*, such that "similar" objects fall into the same group. Similarity between objects is captured by a distance function.

In this paper, we consider the problem of clustering large datasets in a *distance space* in which the only operation possible on data objects is the computation of a distance function that satisfies the triangle inequality. In contrast, objects in a *coordinate space* can be represented as vectors. The vector representation allows various vector operations, e.g., addition and subtraction of vectors, to form condensed representations of clusters and to reduce the time and space requirements of the clustering problem [4, 26]. These operations are not possible in a distance space thus making the problem much harder.[1]

The distance function associated with a distance space can be computationally very expensive [5], and may dominate the overall resource requirements. For example, consider the domain of strings where the distance between two strings is the *edit distance*.[2] Computing the edit distance between two strings of lengths m and n requires $O(mn)$ comparisons between characters. In contrast, computing the Euclidean distance between two n-dimensional vectors in a coordinate space requires just $O(n)$ operations. Most algorithms in the literature have paid little attention to this particular issue when devising clustering algorithms for data in a distance space.

In this work, we first abstract out the essential features of the BIRCH clustering algorithm [26] into the *BIRCH** framework for scalable clustering algorithms. We then instantiate BIRCH* resulting in two new scalable clustering algorithms for distance spaces: BUBBLE and BUBBLE-FM.

The remainder of the paper is organized as follows. In Section 2, we discuss related work on clustering and some of our initial approaches. In Section 3, we present the BIRCH* framework for fast, scalable, incremental clustering algorithms. In Sections 4 and 5, we instantiate the framework for data in a distance space resulting in our algorithms BUBBLE and BUBBLE-FM. Section 6 evaluates the performance of BUBBLE and BUBBLE-FM on synthetic datasets. We discuss an application of BUBBLE-FM

*The first three authors were supported by Grant 2053 from the IBM corporation.

†Supported by an IBM Corporate Fellowship

‡Supported by NASA GSRP NGT5-50062.

§This work supported in part by DARPA contract N66001-97-C-8542.

[1] A distance space is also referred to as an *arbitrary metric space*. We use the term *distance space* to emphasize that only distance computations are possible between objects. We call an n-dimensional space a *coordinate space* to emphasize that vector operations like centroid computation, sum, and difference of vectors are possible.

[2] The edit distance between two strings is the number of simple edit operations required to transform one string into the other.

to a real-life dataset in Section 7 and conclude in Section 8.

We assume that the reader is familiar with the definitions of the following standard terms: metric space, L_p norm of a vector, radius, and centroid of a set of points in a coordinate space. (See the full paper [16] for the definitions.)

2. Related Work and Initial Approaches

In this section, we discuss related work on clustering, and three important issues that arise when clustering data in a distance space vis-a-vis clustering data in a coordinate space.

Data clustering has been extensively studied in the Statistics [20], Machine Learning [12, 13], and Pattern Recognition literature [6, 7]. These algorithms assume that all the data fits into main memory, and typically have running times super-linear in the size of the dataset. Therefore, they do not scale to large databases.

Recently, clustering has received attention as an important data mining problem [8, 9, 10, 17, 21, 26]. CLARANS [21] is a medoid-based method which is more efficient than earlier medoid-based algorithms [18], but has two drawbacks: it assumes that all objects fit in main memory, and the result is very sensitive to the input order [26]. Techniques to improve CLARANS's ability to deal with disk-resident datasets by focussing only on relevant parts of the database using R^*-trees were also proposed [9, 10]. But these techniques depend on R^*-trees which can only index vectors in a coordinate space. DBSCAN [8] uses a density-based notion of clusters to discover clusters of arbitrary shapes. Since DBSCAN relies on the R^*-Tree for speed and scalability in its nearest neighbor search queries, it cannot cluster data in a distance space. BIRCH [26] was designed to cluster large datasets of n-dimensional vectors using a limited amount of main memory. But the algorithm relies heavily on vector operations, which are defined only in coordinate spaces. CURE [17] is a sampling-based hierarchical clustering algorithm that is able to discover clusters of arbitrary shapes. However, it relies on vector operations and therefore cannot cluster data in a distance space.

Three important issues arise when clustering data in a distance space versus data in a coordinate space. First, the concept of a centroid is not defined. Second, the distance function could potentially be computationally very expensive as discussed in Section 1. Third, the domain-specific nature of clustering applications places requirements that are tough to be met by just one algorithm.

Many clustering algorithms [4, 17, 26] rely on vector operations, e.g., the calculation of the centroid, to represent clusters and to improve computation time. Such algorithms cannot cluster data in a distance space. Thus one approach is to map all objects into a k-dimensional coordinate space while preserving distances between pairs of objects and then cluster the resulting vectors.

Multidimensional scaling (MDS) is a technique for distance-preserving transformations [25]. The input to a MDS method is a set S_{in} of N objects, a distance function d, and an integer k; the output is a set S_{out} of N k-dimensional image vectors in a k-dimensional coordinate space (also called the *image space*), one image vector for each object, such that the distance between any two objects is equal (or very close) to the distance between their respective image vectors. MDS algorithms do not scale to large datasets for two reasons. First, they assume that all objects fit in main memory. Second, most MDS algorithms proposed in the literature compute distances between all possible pairs of input objects as a first step thus having complexity at least $O(N^2)$ [19]. Recently, Lin et al. developed a scalable MDS method called FastMap [11]. FastMap preserves distances approximately in the image space while requiring only a fixed number of scans over the data. Therefore, one possible approach for clustering data in a distance space is to map all N objects into a coordinate space using FastMap, and then cluster the resultant vectors using a scalable clustering algorithm for data in a coordinate space. We call this approach the *Map-First* option and empirically evaluate it in Section 6.2. Our experiments show that the quality of clustering thus obtained is not good.

Applications of clustering are domain-specific and we believe that a single algorithm will not serve all requirements. A pre-clustering phase, to obtain a data-dependent summarization of large amounts of data into sub-clusters, was shown to be very effective in making more complex data analysis feasible [4, 24, 26]. Therefore, we take the approach of developing a *pre-clustering* algorithm that returns condensed representations of sub-clusters. A domain-specific clustering method can further analyze the sub-clusters output by our algorithm.

3. BIRCH*

In this section, we present the *BIRCH** framework which generalizes the notion of a cluster feature (CF) and a CF-tree, the two building blocks of the BIRCH algorithm [26]. In the BIRCH* family of algorithms, objects are read from the database sequentially and inserted into incrementally evolving clusters which are represented by *generalized cluster features (CF*s)*. A new object read from the database is inserted into the closest cluster, an operation, which potentially requires an examination of all existing CF*s. Therefore BIRCH* organizes all clusters in an in-memory index, a height-balanced tree, called a *CF*-tree*. For a new object, the search for an appropriate cluster now requires time logarithmic in the number of clusters as opposed to a linear scan.

In the remainder of this section, we abstractly state the components of the BIRCH* framework. Instantiations of these components generate concrete clustering algorithms.

3.1. Generalized Cluster Feature

Any clustering algorithm needs a representation for the clusters detected in the data. The naive representation uses all objects in a cluster. However, since a cluster corresponds to a dense region of objects, the set of objects can be treated collectively through a summarized representation. We will call such a condensed, summarized representation of a cluster its *generalized cluster feature (CF*)*.

Since the entire dataset usually does not fit in main memory, we cannot examine all objects simultaneously to compute CF*s of clusters. Therefore, we incrementally evolve clusters and their CF*s, i.e., objects are scanned sequentially and the set of clusters is updated to assimilate new objects. Intuitively, at any stage, the next object is inserted into the cluster "closest" to it as long as the insertion does not deteriorate the "quality" of the cluster. (Both concepts are explained later.) The CF* is then updated to reflect the insertion. Since objects in a cluster are not kept in main memory, CF*s should meet the following requirements.

- Incremental updatability whenever a new object is inserted into the cluster.
- Sufficiency to compute distances between clusters, and quality metrics (like radius) of a cluster.

CF*s are efficient for two reasons. First, they occupy much less space than the naive representation. Second, calculation of inter-cluster and intra-cluster measurements using the CF*s is much faster than calculations involving all objects in clusters.

3.2. CF*-Tree

In this section, we describe the structure and functionality of a CF*-tree.

A CF*-tree is a height-balanced tree structure similar to the R^*-tree [3]. The number of nodes in the CF*-tree is bounded by a pre-specified number M. Nodes in a CF*-tree are classified into *leaf* and *non-leaf* nodes according to their position in the tree. Each non-leaf node contains at most B entries of the form (CF_i^*,$child_i$), $i \in \{1, \ldots, B\}$, where $child_i$ is a pointer to the i^{th} child node, and CF$_i^*$ is the CF* of the set of objects summarized by the sub-tree rooted at the i^{th} child. A leaf node contains at most B entries, each of the form [CF$_i^*$], $i \in \{1, \ldots, B\}$; each leaf entry is the CF* of a cluster. Each cluster at the leaf level satisfies a *threshold requirement* T, which controls its tightness or quality.

The purpose of the CF*-tree is to direct a new object O to the cluster closest to it. The functionality of non-leaf entries and leaf entries in the CF*-tree is different: non-leaf entries exist to "guide" new objects to appropriate leaf clusters, whereas leaf entries represent the dynamically evolving clusters. For a new object O, at each non-leaf node on the downward path, the non-leaf entry "closest" to O is selected to traverse downwards. Intuitively, directing O to the child node of the closest non-leaf entry is similar to identifying the most promising region and zooming into it for a more thorough examination. The downward traversal continues till O reaches a leaf node. When O reaches a leaf node L, it is inserted into the cluster C in L closest to O if the threshold requirement T is not violated due to the insertion. Otherwise, O forms a new cluster in L. If L does not have enough space for the new cluster, it is split into two leaf nodes and the entries in L redistributed: the set of leaf entries in L is divided into two groups such that each group consists of "similar" entries. A new entry for the new leaf node is created at its parent. In general, all nodes on the path from the root to L may split. We omit the details of the insertion of an object into the CF*-tree because it is similar to that of BIRCH [26].

During the data scan, existing clusters are updated and new clusters are formed. The number of nodes in the CF*-tree may increase beyond M before the data scan is complete due to the formation of many new clusters. Then it is necessary to reduce the space occupied by the CF*-tree which can be done by reducing the number of clusters it maintains. The reduction in the number of clusters is achieved by merging close clusters to form bigger clusters. BIRCH* merges clusters by increasing the threshold value T associated with the leaf clusters and re-inserting them into a new tree. The re-insertion of a leaf cluster into the new tree merely inserts its CF*; all objects in leaf clusters are treated collectively. Thus a new, smaller CF*-tree is built. After all the old leaf entries have been inserted into the new tree, the data scan resumes from the point of interruption.

Note that the CF*-tree insertion algorithm requires *distance* measures between the "inserted entries" and node entries to select the closest entry at each level. Since insertions are of two types: insertion of a single object, and that of a leaf cluster, the BIRCH* framework requires distance measures to be instantiated between a CF* and an object, and between two CF*s (or clusters).

In summary, CF*s, their incremental maintenance, the distance measures, and the threshold requirement are the components of the BIRCH* framework, which have to be instantiated to derive a concrete clustering algorithm.

4. BUBBLE

In this section, we instantiate BIRCH* for data in a distance space resulting in our first algorithm called BUBBLE. Recall that CF*s at leaf and non-leaf nodes differ in their functionality. The former incrementally maintain information about the output clusters, whereas the latter are used to direct new objects to appropriate leaf clusters. Sections 4.1 and 4.2 describe the information in a CF* (and then incremental maintenance) at the leaf and non-leaf levels.

4.1. CF*s at the leaf level

4.1.1 Summary statistics at the leaf level

For each cluster discovered by the algorithm, we return the following information (which is used in further processing): the number of objects in the cluster, a centrally located object in it and its radius. Since a distance space, in general, does not support creation of new objects using operations on a set of objects, we assign an actual object in the cluster as the cluster center. We define the *clustroid* $\hat{\mathcal{O}}$ of a set of objects $\mathcal{O}$ which is the generalization of the centroid to a distance space.[3] We now introduce the RowSum of an object O with respect to a set of objects $\mathcal{O}$, and the concept of an *image space* $\mathrm{IS}(\mathcal{O})$ of a set of objects $\mathcal{O}$ in a distance space. Informally, the image space of a set of objects is a coordinate space containing an *image vector* for each object such that the distance between any two image vectors is the same as the distance between the corresponding objects.

In the remainder of this section, we use $(\mathcal{S}, d)$ to denote a distance space where $\mathcal{S}$ is the domain of all possible objects and $d : \mathcal{S} \times \mathcal{S} \mapsto \mathcal{R}$ is a distance function.

Definition 4.1 Let $\mathcal{O} = \{O_1, \ldots, O_n\}$ be a set of objects in a distance space $(\mathcal{S}, d)$. The RowSum of an object $O \in \mathcal{O}$ is defined as $\mathtt{RowSum}(O) \stackrel{\text{def}}{=} \sum_{j=1}^{n} d^2(O, O_j)$. The *clustroid* $\hat{\mathcal{O}}$ is defined as the object $\hat{\mathcal{O}} \in \mathcal{O}$ such that $\forall O \in \mathcal{O} : \mathtt{RowSum}(\hat{\mathcal{O}}) \leq \mathtt{RowSum}(O)$.

Definition 4.2 Let $\mathcal{O} = \{O_1, \ldots, O_n\}$ be a set of objects in a distance space $(\mathcal{S}, d)$. Let $f : \mathcal{O} \mapsto \mathcal{R}^k$ be a function. We call f an *$\mathcal{R}^k$-distance-preserving transformation* if $\forall i, j \in \{1, \ldots, n\} : d(O_i, O_j) = ||f(O_i) - f(O_j)||$ where $||X - Y||$ is the Euclidean distance between X and Y in $\mathcal{R}^k$. We call $\mathcal{R}^k$ the *image space* of $\mathcal{O}$ under f (denoted $\mathrm{IS}_f(\mathcal{O})$). For an object $O \in \mathcal{O}$, we call $f(O)$ the *image vector* of O under f. We define $f(\mathcal{O}) \stackrel{\text{def}}{=} \{f(O_1), \ldots, f(O_n)\}$.

The existence of a distance-preserving transformation is guaranteed by the following lemma.

Lemma 4.1 [19] Let $\mathcal{O}$ be a set of objects in a distance space $(\mathcal{S}, d)$. Then there exists a positive integer k ($k < |\mathcal{O}|$) and a function $f : \mathcal{O} \mapsto \mathcal{R}^k$ such that f is an $\mathcal{R}^k$-distance-preserving transformation.

For example, three objects x, y, z with the inter-object distance distribution $[d(x, y) = 3, d(y, z) = 4, d(z, x) = 5]$ can be mapped to vectors $(0, 0), (3, 0), (0, 4)$ in the 2-dimensional Euclidean space. This is one of many possible mappings.

The following lemma shows that under any $\mathcal{R}^k$-distance-preserving transformation f, the clustroid of $\mathcal{O}$ is the object $O \in \mathcal{O}$ whose image vector $f(O)$ is closest to the centroid of the set of image vectors $f(\mathcal{O})$. Thus, the clustroid is the generalization of the centroid to distance spaces. Following the generalization of the centroid, we generalize the definitions of the radius of a cluster, and the distance between clusters to distance spaces.

Lemma 4.2 Let $\mathcal{O} = \{O_1, \ldots, O_n\}$ be a set of objects in a distance space $(\mathcal{S}, d)$ with clustroid $\hat{\mathcal{O}}$ and let $f : \mathcal{O} \mapsto \mathcal{R}^k$ be a $\mathcal{R}^k$-distance-preserving transformation. Let $\overline{\mathcal{O}}$ be the centroid of $f(\mathcal{O})$. Then the following holds:

$$\forall O \in \mathcal{O} : ||f(\hat{\mathcal{O}}) - \overline{\mathcal{O}}|| \leq ||f(O) - \overline{\mathcal{O}}||$$

Definition 4.3 Let $\mathcal{O} = \{O_1, \ldots, O_n\}$ be a set of objects in a distance space $(\mathcal{S}, d)$ with clustroid $\hat{\mathcal{O}}$. The *radius* $r(\mathcal{O})$ of $\mathcal{O}$ is defined as $r(\mathcal{O}) \stackrel{\text{def}}{=} \sqrt{\frac{\sum_{i=1}^{n} d^2(O_i, \hat{\mathcal{O}})}{n}}$.

Definition 4.4 We define two different inter-cluster distance metrics between cluster features. Let $\mathcal{O}_1$ and $\mathcal{O}_2$ be two clusters consisting of objects $\{O_{11}, \ldots, O_{1n_1}\}$ and $\{O_{21}, \ldots, O_{2n_2}\}$. Let their clustroids be $\hat{\mathcal{O}}_1$ and $\hat{\mathcal{O}}_2$ respectively. We define the *clustroid distance* D_0 as $D_0(\mathcal{O}_1, \mathcal{O}_2) \stackrel{\text{def}}{=} d(\hat{\mathcal{O}}_1, \hat{\mathcal{O}}_2)$ and the *average inter-cluster distance* D_2 as $D_2(\mathcal{O}_1, \mathcal{O}_2) \stackrel{\text{def}}{=} (\frac{\sum_{i=1}^{n_1} \sum_{j=1}^{n_2} d^2(O_{1i}, O_{2j})}{n_1 n_2})^{\frac{1}{2}}$.

Both BUBBLE and BUBBLE-FM use D_0 as the distance metric between leaf level clusters, and as the threshold requirement T, i.e., a new object O_{new} is inserted into a cluster $\mathcal{O}$ with clustroid $\hat{\mathcal{O}}$ only if $D_0(\mathcal{O}, \{O_{new}\}) = d(\hat{\mathcal{O}}, O_{new}) \leq T$. (The use for D_2 is explained later.)

4.1.2 Incremental maintenance of leaf-level CF*s

In this section, we describe the incremental maintenance of CF*s at the leaf levels of the CF*-tree. Since the sets of objects we are concerned with in this section are clusters, we use $\mathcal{C}$ (instead of $\mathcal{O}$) to denote a set of objects.

The incremental maintenance of the number of objects in a cluster $\mathcal{C}$ is trivial. So we concentrate next on the incremental maintenance of the clustroid $\hat{\mathcal{C}}$. Recall that for a cluster $\mathcal{C}$, $\hat{\mathcal{C}}$ is the object in $\mathcal{C}$ with the minimum RowSum value. As long as we are able to keep all the objects of $\mathcal{C}$ in main memory, we can maintain $\hat{\mathcal{C}}$ incrementally under insertions by updating the RowSum values of all objects $O \in \mathcal{C}$ and then selecting the object with minimum RowSum value as the clustroid. But this strategy requires all objects in $\mathcal{C}$ in main memory, which is not a viable option for large datasets. Since exact maintenance is not possible, we develop a heuristic strategy which works well in

[3] The *medoid* O_k of a set of objects $\mathcal{O}$ is sometimes used as a cluster center [18]. It is defined as the object $O^m \in \mathcal{O}$ that minimizes the average dissimilarity to all objects in $\mathcal{O}$ (i.e., $\sum_{i=1}^{n} d(O_i, O)$ is minimum when $O = O^m$). But, it is not possible to motivate the heuristic maintenance—*a la* clustroid—of the medoid. However, we expect similar heuristics to work even for the medoid.

practice while significantly reducing main memory requirements. We classify insertions in clusters into two types, *Type I* and *Type II*, and motivate heuristics for each type of insertion. A Type I insertion is the insertion of a single object or, equivalently, a cluster containing only one object. Each object in the dataset causes a Type I insertion when it is read from the data file, making it the most common type of insertion. A Type II insertion is the insertion of a cluster containing more than one object. Type II insertions occur only when the CF*-tree is being rebuilt. (See Section 3.)

Type I Insertions: In our heuristics, we make the following approximation: under any distance-preserving transformation f into a coordinate space, the image vector of the clustroid is the centroid of the set of all image vectors, i.e., $f(\hat{\mathcal{C}}) = \overline{f(\mathcal{C})}$. From Lemma 4.2, we know that this is the *best* possible approximation. In addition to the approximation, our heuristic is motivated by the following two observations.

Observation 1: Consider the insertion of a new object O_{new} into a cluster $\mathcal{C} = \{O_1, \ldots, O_n\}$ and assume that only a subset $R \subset \mathcal{C}$ is kept in main memory. Let $f : \mathcal{C} \cup \{O_{new}\} \mapsto \mathcal{R}^k$ be a $\mathcal{R}^k$-distance-preserving transformation from $\mathcal{C} \cup \{O_{new}\}$ into $\mathcal{R}^k$, and let $\overline{f(\mathcal{C})} = \frac{\sum_{i=1}^n f(O_i)}{n}$ be the centroid of $f(\mathcal{C})$. Then, $\texttt{RowSum}(O_{new})$

$$
\begin{aligned}
&= \sum_{j=1}^{n} d^2(O_{new}, O_j) = \sum_{j=1}^{n} (f(O_{new}) - f(O_j))^2 \\
&= \sum_{j=1}^{n} (f(O_j) - \overline{f(\mathcal{C})})^2 + n(\overline{f(\mathcal{C})} - f(O_{new}))^2 \\
&\approx nr^2(\mathcal{C}) + nd^2(\hat{\mathcal{C}}, O_{new})
\end{aligned}
$$

Thus, we can calculate $\texttt{RowSum}(O_{new})$ approximately using only $\hat{\mathcal{C}}$ and significantly reduce the main memory requirements.

Observation 2: Let $\mathcal{C} = \{O_1, \ldots, O_n\}$ be a leaf cluster in the CF*-tree and O_{new} an object which is inserted into $\mathcal{C}$. Let $\hat{\mathcal{C}}$ and $\hat{\mathcal{C}}^*$ be the clustroids of $\mathcal{C}$ and $\mathcal{C} \cup \{O_{new}\}$ respectively. Let D_0 be the distance metric between leaf clusters and T the threshold requirement of the CF*-tree under D_0. Then

$$d(\hat{\mathcal{C}}, \hat{\mathcal{C}}^*) \le \epsilon, \text{ where } \epsilon = \frac{T}{(n+1)}.$$

An implication of Observation 2 is that as long as we keep a set $R \subset \mathcal{C}$ of objects consisting of all objects in $\mathcal{C}$ within a distance of ϵ from $\hat{\mathcal{C}}$, we know that $\hat{\mathcal{C}}^* \in R$. However, when the clustroid changes due to the insertion of O_{new}, we have to update R to consist of all objects within ϵ of $\hat{\mathcal{C}}^*$. Since we cannot assume that all objects in the dataset fit in main memory, we have to retrieve objects in $\mathcal{C}$ from the disk. Repeated retrieval of objects from the disk, whenever a clustroid changes, is expensive. Fortunately (from Observation 2), if n is large then the new set of objects within ϵ of $\hat{\mathcal{C}}^*$ is almost the same as the old set R because $\hat{\mathcal{C}}^*$ is very close to $\hat{\mathcal{C}}$.

Observations 1 and 2 motivate the following heuristic maintenance of the clustroid. As long as $|\mathcal{C}|$ is small (smaller than a constant p), we keep all the objects of $\mathcal{C}$ in main memory and compute the new clustroid exactly. If $|\mathcal{C}|$ is large (larger than p), we invoke Observation 2 and maintain a subset of $\mathcal{C}$ of size p. These p objects have the lowest $\texttt{RowSum}$ values in $\mathcal{C}$ and hence are closest to $\hat{\mathcal{C}}$. If the $\texttt{RowSum}$ value of O_{new} is less than the highest of the p values, say that of O_p, then O_{new} replaces O_p in R. Our experiments confirm that this heuristic works very well in practice.

Type II Insertions: Let $\mathcal{C}_1 = \{ O_{11}, \ldots, O_{1n_1} \}$ and $\mathcal{C}_2 = \{ O_{21}, \ldots, O_{2n_2} \}$ be two clusters being merged. Let f be a distance-preserving transformation of $\mathcal{C}_1 \cup \mathcal{C}_2$ into $\mathcal{R}^k$. Let X_{ij} be the image vector in IS($\mathcal{C}_1 \cup \mathcal{C}_2$) of object O_{ij} under f. Let $\overline{X}_1$ ($\overline{X}_2$) be the centroid of $\{X_{1i}, \ldots, X_{1n_1}\}$ ($\{X_{2i}, \ldots, X_{2n_2}\}$). Let $\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2$ be their clustroids, and $r(\mathcal{C}_1), r(\mathcal{C}_2)$ be their radii. Let $\hat{\mathcal{C}}^*$ be the clustroid of $\mathcal{C}_1 \cup \mathcal{C}_2$.

The new centroid $\overline{X}$ of $f(\mathcal{C}_1 \cup \mathcal{C}_2)$ lies on the line joining $\overline{X_1}$ and $\overline{X_2}$; its exact location on the line depends on the values of n_1 and n_2. Since D_0 is used as the threshold requirement for insertions, the distance between $\overline{X}$ and $\overline{X}_1$ is bounded as shown below:

$$(\overline{X} - \overline{X}_1)^2 = \frac{n_2^2(\overline{X_1} - \overline{X_2})^2}{(n_1+n_2)^2} \approx \frac{n_2^2 d^2(\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2)}{(n_1+n_2)^2} < \frac{n_2^2 T}{(n_1+n_2)^2}$$

The following two assumptions motivate the heuristic maintenance of the clustroid under Type II insertions.

(i) $\mathcal{C}_1$ and $\mathcal{C}_2$ are non-overlapping but very close to each other. Since $\mathcal{C}_1$ and $\mathcal{C}_2$ are being merged, the threshold criterion is satisfied implying that $\mathcal{C}_1$ and $\mathcal{C}_2$ are close to each other. We expect the two clusters to be almost non-overlapping because they were two distinct clusters in the old CF*-tree.

(ii) $n_1 \approx n_2$. Due to lack of any prior information about the clusters, we assume that the objects are uniformly distributed in the merged cluster. Therefore, the values of n_1 and n_2 are close to each other in Type II insertions.

For these two reasons, we expect the new clustroid $\hat{\mathcal{C}}^*$ to be midway between $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$, which corresponds to the periphery of either cluster. Therefore we maintain a few objects (p in number) on the periphery of each cluster in its CF*. Because they are the farthest objects from the clustroid, they have the highest $\texttt{RowSum}$ values in their respective clusters.

Thus we overall maintain $2 \cdot p$ objects for each leaf cluster $\mathcal{C}$, which we call the *representative objects* of $\mathcal{C}$; the value $2p$ is called the *representation number* of $\mathcal{C}$. Storing the representative objects enables the approximate incremental maintenance of the clustroid. The incremental maintenance of the radius of $\mathcal{C}$ is similar to that of `RowSum` values; details are given in the full paper [16].

Summarizing, we maintain the following information in the CF* of a leaf cluster $\mathcal{C}$: (i) the number of objects in $\mathcal{C}$, (ii) the clustroid of $\mathcal{C}$, (iii) $2 \cdot p$ representative objects, (iv) the `RowSum` values of the representative objects, and (v) the radius of the cluster. All these statistics are incrementally maintainable—as described above—as the cluster evolves.

4.2. CF*s at Non-leaf Level

In this section, we instantiate the cluster features at non-leaf levels of the BIRCH* framework and describe their incremental maintenance.

4.2.1 Sample Objects

In the BIRCH* framework, the functionality of a CF* at a non-leaf entry is to guide a new object to the sub-tree which contains its prospective cluster. Therefore, the cluster feature of the i^{th} non-leaf entry NL$_i$ of a non-leaf node NL summarizes the distribution of all clusters in the subtree rooted at NL$_i$. In Algorithm BUBBLE, this summary, the CF*, is represented by a set of objects; we call these objects the *sample objects* S(NL$_i$) of NL$_i$ and the union of all sample objects at all the entries the sample objects S(NL) of NL.

We now describe the procedure for selecting the sample objects. Let $child_1, \ldots,$child$_k$ be the child nodes at NL with $n_1, \ldots, n_k$ entries respectively. Let S(NL$_i$) denote the set of sample objects collected from $child_i$ and associated with NL$_i$. S(NL) is the union of sample objects at all entries of NL. The number of sample objects to be collected at any non-leaf node is upper bounded by a constant called the *sample size (SS)*. The number $|$S(NL$_i$)$|$ contributed by $child_i$ is $MAX(\left\lfloor \frac{n_i * SS}{\sum_{j=1}^{k} n_j} \right\rfloor, 1)$. The restriction that each child node have at least one representative in S(NL) is placed so that the distribution of the sample objects is representative of all its children, and is also necessary to define distance measures between a newly inserted object and a non-leaf cluster. If $child_i$ is a leaf node, then the sample objects S(NL$_i$) are randomly picked from all the clustroids of the leaf clusters at $child_i$. Otherwise, they are randomly picked from $child_i$'s sample objects S($child_i$).

4.2.2 Updates to Sample Objects

The CF*-tree evolves gradually as new objects are inserted into it. The accuracy of the summary distribution captured by sample objects at a non-leaf entry depends on how recently the sample objects were gathered. The periodicity of updates to these samples, and when these updates are actually triggered, affects the currency of the samples. Each time we update the sample objects we incur a certain cost. Thus we have to strike a balance between the cost of updating the sample objects and their currency.

Because a split at $child_i$ of NL causes redistribution of its entries between $child_i$ and the new node $child_{k+1}$, we have to update samples S(NL$_i$) and S(NL$_{k+1}$) at entries NL$_i$ and NL$_{k+1}$ of the parent (we actually create samples for the new entry NL$_{k+1}$). However, to reflect changes in the distributions at all children nodes we update the sample objects at all entries of NL whenever one of its children splits.

4.2.3 Distance measures at non-leaf levels

Let O_{new} be a new object inserted into the CF*-tree. The *distance* between O_{new} and NL$_i$ is defined to be $D_2(\{O_{new}\},$S(NL$_i$)). Since $D_2(\{O_{new}\}, \emptyset)$ is meaningless, we ensure that each non-leaf entry has at least one sample object from its child during the selection of sample objects. Let L$_i$ represent the i^{th} leaf entry of a leaf node L. The distance between $\mathcal{C}$ and L$_i$ is defined to be the clustroid distance $D_0(\mathcal{C}$,L$_i)$.

The instantiation of distance measures completes the instantiation of BIRCH* deriving BUBBLE. We omit the the cost analysis of BUBBLE because it is similar to that of BIRCH.

5. BUBBLE-FM

While inserting a new object O_{new}, BUBBLE computes distances between O_{new} and all the sample objects at each non-leaf node on its downward path from the root to a leaf node. The distance function d may be computationally very expensive (e.g., the edit distance on strings). We address this issue in our second algorithm BUBBLE-FM—which improves upon BUBBLE by reducing the number of invocations of d—using FastMap [11]. We first give a brief overview of FastMap and then describe BUBBLE-FM.

5.1. Overview of FastMap

Given a set $\mathcal{O}$ of N objects, a distance function d, and an integer k, FastMap quickly (in time linear in N) computes N vectors (called *image vectors*), one for each object, in a k-dimensional Euclidean image space such that the distance between two image vectors is close to the distance between the corresponding two objects. Thus, FastMap is an "approximate" $\mathcal{R}^k$-distance-preserving transformation. Each of the k axes is defined by the line joining two objects.[4] The $2k$ objects are called *pivot objects*. The space defined by the k axes is the *fastmapped image space* IS$_{fm}(\mathcal{O})$ of $\mathcal{O}$. The number of calls to d made by FastMap to map N objects is $3Nkc$, where c is a parameter (typically set to 1 or 2).

An important feature of FastMap that we use in BUBBLE-FM is its fast *incremental mapping* ability. Given

[4] See Lin et. al. for details [11].

a new object O_{new}, FastMap projects it onto the k coordinate axes of IS$_{fm}(\mathcal{O})$ to compute a k-dimensional vector for O_{new} in IS$_{fm}(\mathcal{O})$ with just $2k$ calls to d. Distance between O_{new} and any object $O \in \mathcal{O}$ can now be measured through the Euclidean distance between their image vectors.

5.2. Description of BUBBLE-FM

BUBBLE-FM differs from BUBBLE only in its usage of sample objects at a non-leaf node. In BUBBLE-FM, we first map—using FastMap—the set of all sample objects at a non-leaf node into an "approximate" image space. We then use the image space to measure distances between an incoming object and the CF*s. Since CF*s at non-leaf entries function merely as guides to appropriate children nodes, an approximate image space is sufficient. We now describe the construction of the image space and its usage in detail.

Consider a non-leaf node NL. Whenever S(NL) is updated, we use FastMap to map S(NL) into a k-dimensional coordinate space IS$_{fm}(NL)$; k is called the *image dimensionality* of NL. FastMap returns a vector for each object in S(NL). The centroid of the image vectors of S(NL$_i$) is then used as the centroid of the cluster represented by NL$_i$ while defining distance metrics.

Let fm:S(NL)$\mapsto$IS$_{fm}(NL)$ be the distance preserving transformation associated with FastMap that maps each sample object $s \in$ S(NL) to a k-dimensional vector $fm(s) \in$ IS$_{fm}(NL)$. Let $\overline{S(NL_i)}$ be the centroid of the set of image vectors of S(NL$_i$), i.e., $\overline{S(NL_i)} = \frac{\sum_{s\in S(NL_i)} fm(s)}{|S(NL_i)|}$.
The non-leaf CF* in BUBBLE-FM consists of (1) S(NL$_i$) and (2) $\overline{S(NL_i)}$. In addition, we maintain the image vectors of the $2k$ pivot objects returned by FastMap.

The $2k$ pivot objects define the axes of the k-dimensional image space constructed by FastMap. Let O_{new} be a new object. Using FastMap, we incrementally map O_{new} to $V_{new} \in$ IS$_{fm}(NL)$. We define the distance between O_{new} and NL$_i$ to be the Euclidean distance between V_{new} and $\overline{S(NL_i)}$. Formally,

$$D(O_{new}, S(NL_i)) \stackrel{\text{def}}{=} ||V_{new} - \overline{S(NL_i)}||$$

Similarly, the distance between two non-leaf entries NL$_i$ and NL$_j$ is defined to be $||\overline{S(NL_i)} - \overline{S(NL_j)}||$. Whenever |S(NL)| $\leq 2k$, BUBBLE-FM measures distances at NL in the distance space, as in BUBBLE.

5.2.1 An alternative at the leaf level

We do not use FastMap at the leaf levels of the CF*-tree for the following reasons.

1. Suppose FastMap were used at the leaf levels also. The approximate image space constructed by FastMap does not accurately reflect the relative distances between clustroids; the inaccuracy causes erroneous insertions of objects into clusters deteriorating the clustering quality. Similar errors at non-leaf levels merely cause new entries to be redirected to wrong leaf nodes where they will form new clusters. Therefore, the impact of these errors is on the maintenance costs of the CF*-tree, but not on the clustering quality, and hence are not so severe.

2. If IS$_{fm}(L)$ has to be maintained accurately under new insertions then it should be reconstructed whenever any clustroid in the leaf node L changes. In this case, the overhead of repeatedly invoking FastMap offsets the gains due to measuring distances in IS$_{fm}(L)$.

5.2.2 Image dimensionality and other parameters

The image dimensionalities of non-leaf nodes can be different because the sample objects at each non-leaf node are mapped into independent image spaces. The problem of finding the right dimensionality of the image space has been studied well [19]. We set the image dimensionalities of all non-leaf nodes to the same value; any technique used to find the right image dimensionality can be incorporated easily into the mapping algorithm.

Our experience with BUBBLE and BUBBLE-FM on several datasets showed that the results are not very sensitive to small deviations in the values of the parameters: the representation number and the sample size. We found that a value of 10 for the representation number works well for several datasets including those used for the experimental study in Section 6. An appropriate value for the sample size depends on the branching factor BF of the CF*-tree. We observed that a value of $5 \times BF$ works well in practice.

6. Performance Evaluation

In this section, we evaluate BUBBLE and BUBBLE-FM on synthetic datasets. Our studies show that BUBBLE and BUBBLE-FM are scalable high quality clustering algorithms.[5]

6.1. Datasets and Evaluation Methodology

To compare with the Map-First option, we use two datasets DS1 and DS2. Both DS1 and DS2 have 100000 2-dimensional points distributed in 100 clusters [26]. However, the cluster centers in DS1 are uniformly distributed on a 2-dimensional grid; in DS2, the cluster centers are distributed on a sine wave. These two datasets are also used to visually observe the clusters produced by BUBBLE and BUBBLE-FM.

We also generated k-dimensional datasets as described by Agrawal et al. [1]. The k-dimensional box $[0, 10]^k$ is divided into 2^k cells by halving the range $[0, 10]$ over each dimension. A cluster center is randomly placed in each of K cells chosen randomly from the 2^k cells, where K is the number of clusters in the dataset. In each cluster, $\frac{N}{K}$ points are distributed uniformly within a radius randomly picked from $[0.5, 1.0]$. A dataset containing N k-dimensional points and K clusters is denoted DSkd.Kc.N. Even though these datasets consist of k-dimensional vectors we *do not* exploit the operations specific to coordinate spaces, and treat the vectors in the dataset merely as objects. The distance between any two objects is returned by the Euclidean distance function.

We now describe the evaluation methodology. The clustroids of the sub-clusters returned by BUBBLE and BUBBLE-FM are further clustered using a *hierarchical clustering* algorithm [20] to obtain the required number of clusters. To minimize the effect of hierarchical clustering on the final results, the amount of memory allocated to the algorithm was adjusted so that the number of sub-clusters returned by BUBBLE or BUBBLE-FM is very close (not exceeding the actual number of clusters by more than 5%) to the actual number of clusters in the synthetic dataset. Whenever the final cluster is formed by merging sub-clusters, the clustroid of the final cluster is the centroid of the clustroids of sub-clusters merged. Other parameters to the algorithm, the *sample size (SS), the branching factor (B), and the representation number (*$2 \cdot p$*)* are fixed at 75, 15, and 10 respectively (unless otherwise stated) as they were found to result in good clustering quality. The image dimensionality for BUBBLE-FM is set to be equal to the dimensionality of the data. The dataset D is scanned a second time to associate each object $O \in D$ with a cluster whose representative object is closest to O.

We introduce some notation before describing the evaluation metrics. Let $A_1, \ldots, A_K$ be the actual clusters in the dataset and $C_1, \ldots, C_K$ be the set of clusters discovered by BUBBLE or BUBBLE-FM. Let $\overline{A_i}$ ($\overline{C_i}$) be the centroid of cluster A_i (C_i). Let $\hat{C}_i$ be the clustroid of C_i. Let $n(C)$ denote the number of points in the cluster C. We use the following metrics, some of which are traditionally used in the Statistics and the Pattern Recognition communities [6, 7], to evaluate the clustering quality and speed.

- The *distortion* ($\sum_{j=1}^{K} \sum_{X \in C_j} (X - \overline{C}_j)^2$) of a set of clusters indicates the tightness of the clusters.
- The *clustroid quality* ($CQ = \frac{||\overline{A_i} - \hat{C}_j||}{K}$) is the average distance between the actual centroid of a cluster $\overline{A_i}$ and the clustroid $\hat{C}_j$ that is closest to $\overline{A_i}$.
- The *number of calls to d (NCD)* and the time taken by the algorithm indicate the cost of the algorithm. NCD is useful to extrapolate the performance for computationally expensive distance functions.

6.2. Comparison with the Map-First option

We mapped DS1, DS2, and DS20d.50c.100K into an appropriate k-dimensional space (k = 2 for DS1, DS2, and 20 for DS20d.50c.100K) using FastMap, and then used BIRCH to cluster the resulting k-dimensional vectors.

The clustroids of clusters obtained from BUBBLE and BUBBLE-FM on DS2 are shown in Figures 1 and 2 respectively, and the centroids of clusters obtained from BIRCH are shown in Figure 3. From the distortion values (Table 1), we see that the quality of clusters obtained by BUBBLE or BUBBLE-FM is clearly better than the Map-First option.

Dataset	Map-First	BUBBLE	BUBBLE-FM
DS1	195146	129798	122544
DS2	1147830	125093	125094
DS20d.50c.100K	$2.214 * 10^6$	21127.5	21127.5

Table 1. Comparison with the Map-First option

6.3. Quality of Clustering

In this section, we use the dataset DS20d.50c.100K. To place the results in the proper perspective, we mention that the average distance between the centroid of each cluster $\overline{A_i}$ and an *actual point* in the dataset closest to $\overline{A_i}$ is 0.212. Hence the clustroid quality (CQ) cannot be less than 0.212. From Table 2, we observe that the *CQ* values are close to the

Algorithm	CQ	Actual Distortion	Computed Distortion
BUBBLE	0.289	21127.4	21127.5
BUBBLE-FM	0.294	21127.4	21127.5

Table 2. Clustering Quality

[5] The quality of the result from BIRCH was shown to be independent of the input order [26]. Since, BUBBLE and BUBBLE-FM are instantiations of the BIRCH* framework which is abstracted out from BIRCH, we do not present more results on order-independence here.

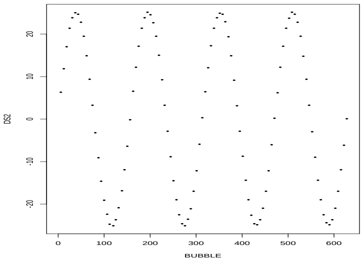

Figure 1. DS2: BUBBLE

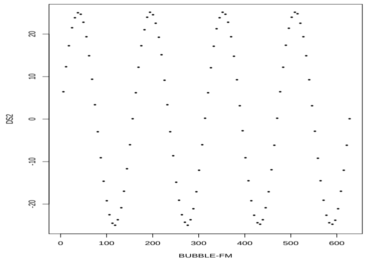

Figure 2. DS2: BUBBLE-FM

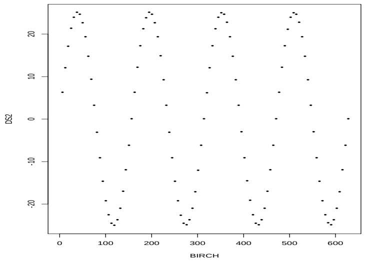

Figure 3. DS2: BIRCH

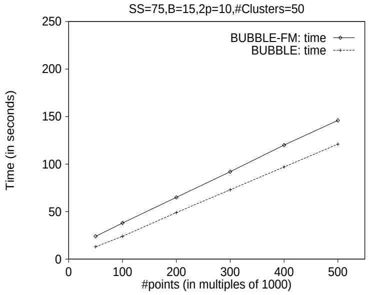

Figure 4. Time vs #points

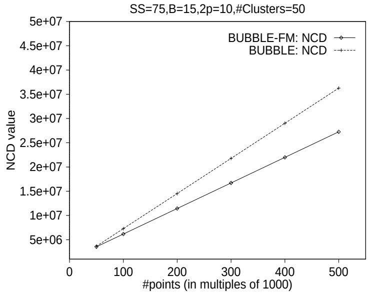

Figure 5. NCD vs #points

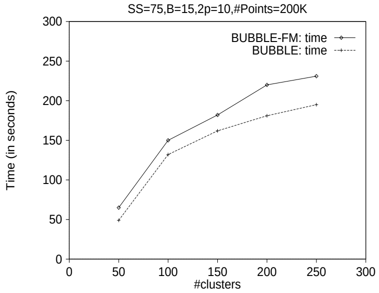

Figure 6. Time vs #clusters

minimum possible value (0.212), and the distortion values match almost exactly. Also, we observed that all the points except a few (less than 5) were placed in the appropriate clusters.

6.4. Scalability

To study scalability characteristics with respect to the number of points in the dataset, we fixed the number of clusters at 50 and varied the number of data points from 50000 to 500000 (i.e., DS20d.50c.*).

Figures 4 and 5 plot the time and NCD values for BUBBLE and BUBBLE-FM as the number of points is increased. We make the following observations. (i) Both algorithms scale linearly with the number of points, which is as expected. (ii) BUBBLE consistently outperforms BUBBLE-FM. This is due to the overhead of FastMap in BUBBLE-FM. (The distance function in the fastmapped space as well as the original space is the Euclidean distance function.) However, the constant difference between their running times suggests that the overhead due to the use of FastMap remains constant even though the number of points increases. The difference is constant because the overhead due to FastMap is incurred only when the nodes in the CF*-tree split. Once the distribution of clusters is captured the nodes do not split that often any more. (iii) As expected, BUBBLE-FM has smaller NCD values. Since the overhead due to the use of FastMap remains constant, as the number of points is increased the difference between the NCD values increases.

To study scalability with respect to the number of clusters, we varied the number of clusters between 50 and 250 while keeping the number of points constant at 200000. The results are shown in Figure 6. The plot of time versus number of clusters is almost linear.[6]

7. Data Cleaning Application

When different bibliographic databases are integrated, different conventions for recording bibliographic items such as author names and affiliations cause problems. Users familiar with one set of conventions will expect their usual forms to retrieve relevant information from the entire collection when searching. Therefore, a necessary part of the integration is the creation of a joint *authority file* [2, 15] in which classes of equivalent strings are maintained. These equivalent classes can be assigned a canonical form. The process of reconciling variant string forms ultimately requires domain knowledge and inevitably a human in the loop, but it can be significantly speeded up by first achieving a rough clustering using a metric such as the edit distance. Grouping closely related entries into initial clusters that act as *representative strings* has two benefits: (1) Early aggregation acts as a “sorting” step that lets us use more aggressive strategies in later stages with less risk of erroneously separating closely related strings. (2) If an error is made in the placement of a representative, only that representative need be moved to a new location. Also, even the small reduction in the data size is valuable, given the cost of the subsequent detailed analysis involving a domain expert.[7]

Applying edit distance techniques to obtain such a “first

[6] NCD versus number of clusters is in the full paper [16].

[7] Examples and more details are given in the full paper.

pass" clustering is quite expensive, however, and we therefore applied BUBBLE-FM to this problem. We view this application as a form of data cleaning because a large number of closely related strings differing from each other by omissions, additions, and transposition of characters and words, are placed together in a single cluster. Moreover, it is preparatory to more detailed domain specific analysis involving a domain expert. We compared BUBBLE-FM with some other clustering approaches [14, 15], which use *relative edit distance (RED)*. Our results are very promising and indicate that BUBBLE-FM achieves high quality in much less time.

We used BUBBLE-FM on a real-life dataset RDS of about 150,000 strings (representing 13,884 different variants) to determine the behavior of BUBBLE-FM. Table 3 shows our results on the dataset RDS. A string is said to be *misplaced* if it is placed in the wrong cluster. Since we know the exact set of clusters, we can count the number of misplaced strings. We first note that BUBBLE-FM is much faster than RED. Moreover, more than 50% of the time is spent in the second phase where each string in the dataset is associated with a cluster. Second, parameters in BUBBLE-FM can be set according to the tolerance on *misclassification error*. If the tolerance is low then BUBBLE-FM returns a much larger number of clusters than RED but the misclassification is much lower too. If the tolerance is high, then it returns a lower number of clusters with higher misclassification error.

Algorithm	# of clusters	# of misplaced strings	Time (in hrs)
RED (run 1)	10161	69	45
BUBBLE-FM (run 1)	10078	897	7.5
BUBBLE-FM (run 2)	12385	20	7

Table 3. Results on the dataset RDS

8. Conclusions

In this paper, we studied the problem of clustering large datasets in arbitrary metric spaces. The main contributions of this paper are:

1. We introduced the BIRCH* framework for fast scalable incremental pre-clustering algorithms and instantiated BUBBLE and BUBBLE-FM for clustering data in a distance space.
2. We introduced the concept of image space to generalize the definitions of summary statistics like centroid, radius to distance spaces.
3. We showed how to reduce the number of calls to an expensive distance function by using FastMap without deteriorating the clustering quality.

Acknowledgements: We thank Tian Zhang for helping us with the BIRCH code base. We also thank Christos Faloutsos and David Lin for providing us the code for FastMap.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining. In *SIGMOD*, 1998.

[2] L. Auld. Authority Control: An Eighty-Year Review. *Library Resources & Technical Services*, 26:319–330, 1982.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[4] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *KDD*, 1998.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *VLDB*, 1997.

[6] R. Dubes and A. Jain. *Clustering methodologies in exploratory data analysis, Advances in Computers.* Academic Press, New York, 1980.

[7] R. Duda and P. Hart. *Pattern Classification and Scene analysis*. Wiley, 1973.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1995.

[9] M. Ester, H.-P. Kriegel, and X. Xu. A database interface for clustering in large spatial databases. *KDD*, 1995.

[10] M. Ester, H.-P. Kriegel, and X. Xu. Focussing techniques for efficient class indentification. *Proc. of the 4th Intl. Sym. of Large Spatial Databases*, 1995.

[11] C. Faloutsos and K.-I. Lin. Fastmap: A fast algorithm for indexing, datamining and visualization of traditional and multimedia databases. *SIGMOD*, 1995.

[12] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2), 1987.

[13] D. H. Fisher. Iterative optimization and simplification of hierarchical clusterings. Technical report, Department of Computer Science, Vanderbilt University, TN 37235, 1995.

[14] J. C. French, A. L. Powell, and E. Schulman. Applications of Approximate Word Matching in Information Retrieval. In *CIKM*, 1997.

[15] J. C. French, A. L. Powell, E. Schulman, and J. L. Pfaltz. Automating the Construction of Authority Files in Digital Libraries: A Case Study. In C. Peters and C. Thanos, editors, *First European Conf. on Research and Advanced Technology for Digital Libraries*, volume 1324 of *Lecture Notes in Computer Science*, pages 55–71, 1997. Springer-Verlag.

[16] V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, and J. French. Clustering large datasets in arbitrary metric spaces. Technical report, University of Wisconsin-Madison, 1998.

[17] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *SIGMOD*, 1998.

[18] L. Kaufmann and P. Rousseuw. *Finding Groups in Data - An Introduction to Cluster Analysis*. Wiley series in Probability and Mathematical Statistics, 1990.

[19] J. Kruskal and M. Wish. Multidimensional scaling. *Sage University Paper*, 1978.

[20] F. Murtagh. A survey of recent hierarchical clustering algorithms. *The Computer Journal*, 1983.

[21] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *VLDB*, 1994.

[22] R. Shepard. The analysis of proximities: Multidimensional scaling with an unknown distance, i and ii. *Psychometrika*, pages 125–140, 1962.

[23] W. Torgerson. Multidimensional scaling:i. theory and method. *Psychometrika*, 17:401–419, 1952.

[24] M. Wong. A hybrid clustering method for identifying high-density clusters. *J. of Amer. Stat. Assocn.*, 77(380):841–847, 1982.

[25] F. Young. *Multidimensional scaling: history, theory, and application s*. Lawrence Erlbaum associates, Hillsdale, New Jersey, 1987.

[26] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for large databases. *SIGMOD*, 1996.