BIRCH: An Efficient Data Clustering Method for Very Large Databases

Tian Zhang
Computer Sciences Dept.
Univ. of Wisconsin-Madison
zhang@cs.wisc.edu

Raghu Ramakrishnan
Computer Sciences Dept.
Univ. of Wisconsin-Madison
raghu@cs.wisc.edu

Miron Livny*
Computer Sciences Dept.
Univ. of Wisconsin-Madison
miron@cs.wisc.edu

Abstract

Finding useful patterns in large datasets has attracted considerable interest recently, and one of the most widely studied problems in this area is the identification of *clusters*, or densely populated regions, in a multi-dimensional dataset. Prior work does not adequately address the problem of large datasets and minimization of I/O costs.

This paper presents a data clustering method named *BIRCH* (Balanced Iterative Reducing and Clustering using Hierarchies), and demonstrates that it is especially suitable for very large databases. *BIRCH* incrementally and dynamically clusters incoming multi-dimensional metric data points to try to produce the best quality clustering with the available resources (i.e., available memory and time constraints). *BIRCH* can typically find a good clustering with a single scan of the data, and improve the quality further with a few additional scans. *BIRCH* is also the first clustering algorithm proposed in the database area to handle "noise" (data points that are not part of the underlying pattern) effectively.

We evaluate *BIRCH*'s time/space efficiency, data input order sensitivity, and clustering quality through several experiments. We also present a performance comparisons of *BIRCH* versus *CLARANS*, a clustering method proposed recently for large datasets, and show that *BIRCH* is consistently superior.

1 Introduction

In this paper, we examine *data clustering*, which is a particular kind of data mining problem. Given a large set of multi-dimensional data points, the data space is usually not uniformly occupied. *Data clustering* identifies the sparse and the crowded places, and hence discovers the overall distribution patterns of the dataset. Besides, the derived clusters can be visualized more efficiently and effectively than the original dataset[Lee81, DJ80].

*This research has been supported by NSF Grant IRI-9057562 and NASA Grant 144-EC78.

Generally, there are two types of attributes involved in the data to be clustered: *metric* and *nonmetric*[1]. In this paper, we consider metric attributes, as in most of the Statistics literature, where the clustering problem is formalized as follows: *Given the desired number of clusters K and a dataset of N points, and a distance-based measurement function (e.g., the weighted total/average distance between pairs of points in clusters), we are asked to find a partition of the dataset that minimizes the value of the measurement function.* This is a *nonconvex discrete* [KR90] optimization problem. Due to an abundance of local minima, there is typically no way to find a global minimal solution without trying all possible partitions.

We adopt the problem definition used in Statistics, but with an additional, database-oriented constraint: *The amount of memory available is limited (typically, much smaller than the data set size) and we want to minimize the time required for I/O.* A related point is that it is desirable to be able to take into account the amount of *time* that a user is willing to wait for the results of the clustering algorithm.

We present a clustering method named *BIRCH* and demonstrate that it is especially suitable for very large databases. Its I/O cost is linear in the size of the dataset: 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 evaluating *BIRCH*'s time/space efficiency, data input order sensitivity, and clustering quality, and comparing with other existing algorithms through experiments, we argue that *BIRCH* is the best available clustering method for very large databases. *BIRCH*'s architecture also offers opportunities for parallelism, and for interactive or dynamic performance tuning based on knowledge about the dataset, gained over the course of the execution. Finally, *BIRCH* is the first clustering al-

[1]Informally, a *metric* attribute is an attribute whose values satisfy the requirements of *Euclidian* space, i.e., self identity (for any X, $X = X$) and triangular inequality (there exists a distance definition such that for any X_1,X_2,X_3, $d(X_1, X_2) + d(X_2, X_3) \geq d(X_1, X_3)$).

gorithm proposed in the database area that addresses *outliers* (intuitively, data points that should be regarded as "noise") and proposes a plausible solution.

1.1 Outline of Paper

The rest of the paper is organized as follows. Sec. 2 surveys related work and summarizes *BIRCH*'s contributions. Sec. 3 presents some background material. Sec. 4 introduces the concepts of clustering feature (CF) and CF tree, which are central to *BIRCH*. The details of *BIRCH* algorithm is described in Sec. 5, and a preliminary performance study of *BIRCH* is presented in Sec. 6. Finally our conclusions and directions for future research are presented in Sec. 7.

2 Summary of Relevant Research

Data clustering has been studied in the Statistics [DH73, DJ80, Lee81, Mur83], Machine Learning [CKS88, Fis87, Fis95, Leb87] and Database [NH94, EKX95a, EKX95b] communities with different methods and different emphases. Previous approaches, probability-based (like most approaches in Machine Learning) or distance-based (like most work in Statistics) , do not adequately consider the case that the dataset can be too large to fit in main memory. In particular, they do not recognize that the problem must be viewed in terms of how to work with a limited resources (e.g., memory that is typically, much smaller than the size of the dataset) to do the clustering as accurately as possible while keeping the I/O costs low.

Probability-based approaches: They typically [Fis87, CKS88] make the assumption that probability distributions on separate attributes are statistically independent of each other. In reality, this is far from true. Correlation between attributes exists, and sometimes this kind of correlation is exactly what we are looking for. The probability representations of clusters make updating and storing the clusters very expensive, especially if the attributes have a large number of values because their complexities are dependent not only on the number of attributes, but also on the number of values for each attribute. A related problem is that often (e.g., [Fis87]), the probability-based tree that is built to identify clusters is not height-balanced. For skewed input data, this may cause the performance to degrade dramatically.

Distance-based approaches: They assume that all data points are given in advance and can be scanned frequently. 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 should 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. Hence none of them have linear time scalability with stable quality.

For example, using *exhaustive enumeration (EE)*, there are approximately $K^N/K!$ [DH73] ways of partitioning a set of N data points into K subsets. So in practice, though it can find the global minimum, it is infeasible except when N and K are extremely small. *Iterative optimization (IO)* [DH73, KR90] starts with an initial partition, then tries all possible moving or swapping of data points from one group to another to see if such a moving or swapping improves the value of the measurement function. It can find a local minimum, but the quality of the local minimum is very sensitive to the initially selected partition, and the worst case time complexity is still exponential. *Hierarchical clustering (HC)* [DH73, KR90, Mur83] does not try to find "best" clusters, but keeps merging the closest pair (or splitting the farthest pair) of objects to form clusters. With a reasonable distance measurement, the best time complexity of a practical *HC* algorithm is $O(N^2)$. So it is still unable to scale well with large N.

Clustering has been recognized as a useful spatial data mining method recently. [NH94] presents *CLARANS* that is based on randomized search, and proposes that *CLARANS* outperforms traditional clustering algorithms in Statistics. In *CLARANS*, a cluster is represented by its *medoid*, or the most centrally located data point in the cluster. The clustering process is formalized as searching a graph in which each node is a K-partition represented by a set of 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 these.

CLARANS suffers from the same drawbacks as the above *IO* method wrt. efficiency. In addition, it may not find a real local minimum due to the searching trimming controlled by *maxneighbor*. Later [EKX95a] and [EKX95b] propose focusing techniques (based on R^*-trees) to improve *CLARANS*'s ability to deal with data objects 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 with a small loss of quality.

2.1 Contributions of *BIRCH*

An important contribution is our formulation of the clustering problem in a way that is appropriate for very large datasets, by making the time and memory constraints explicit. In addition, *BIRCH* has the following advantages over previous distance-based approaches.

- *BIRCH* is *local* (as opposed to global) in that each clustering decision is made without scanning all data points or all currently existing clusters. It uses measurements that reflect the natural *closeness* of points, and at the same time, can be incrementally maintained during the clustering process.
- *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. A dense region of points is treated collectively as a single *cluster*. Points in sparse regions are treated as *outliers* and removed optionally.
- *BIRCH* makes full use of available memory to derive the finest possible subclusters (to ensure accuracy) while minimizing I/O costs (to ensure efficiency). The clustering and reducing process is organized and characterized by the use of an in-memory, height-balanced and highly-occupied tree structure. Due to these features, its running time is linearly scalable.
- If we omit the optional Phase 4 5, *BIRCH* is an incremental method that does not require the whole dataset in advance, and only scans the dataset once.

3 Background

Assume that readers are familiar with the terminology of vector spaces, we begin by defining centroid, radius and diameter for a 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:

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

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

$$D = \left(\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (\vec{X_i} - \vec{X_j})^2}{N(N-1)}\right)^{\frac{1}{2}} \tag{3}$$

R is the average distance from member points to the centroid. D is the average pairwise distance within a cluster. They are two alternative measures of the tightness of the cluster around the centroid. 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 Euclidian distance** $D0$ and **centroid Manhattan distance** $D1$ of the two clusters are defined as:

$$D0 = ((\vec{X0_1} - \vec{X0_2})^2)^{\frac{1}{2}} \tag{4}$$

$$D1 = |\vec{X0_1} - \vec{X0_2}| = \sum_{i=1}^{d} |\vec{X0_1}^{(i)} - \vec{X0_2}^{(i)}| \tag{5}$$

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:

$$D2 = \left(\frac{\sum_{i=1}^{N_1} \sum_{j=N_1+1}^{N_1+N_2} (\vec{X_i} - \vec{X_j})^2}{N_1 N_2}\right)^{\frac{1}{2}} \tag{6}$$

$$D3 = \left(\frac{\sum_{i=1}^{N_1+N_2} \sum_{j=1}^{N_1+N_2} (\vec{X_i} - \vec{X_j})^2}{(N_1+N_2)(N_1+N_2-1)}\right)^{\frac{1}{2}} \tag{7}$$

$$D4 = \sum_{k=1}^{N_1+N_2} \left(\vec{X_k} - \frac{\sum_{l=1}^{N_1+N_2} \vec{X_l}}{N_1+N_2}\right)^2$$

$$- \sum_{i=1}^{N_1} \left(\vec{X_i} - \frac{\sum_{l=1}^{N_1} \vec{X_l}}{N_1}\right)^2 - \sum_{j=N_1+1}^{N_1+N_2} \left(\vec{X_j} - \frac{\sum_{l=N_1+1}^{N_1+N_2} \vec{X_l}}{N_2}\right)^2 \tag{8}$$

$D3$ is actually 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. Users can optionally preprocess data by weighting or shifting along different dimensions without affecting the relative placement.

4 Clustering Feature and CF Tree

The concepts of **Clustering Feature** and **CF** tree are at the core of *BIRCH*'s incremental clustering. A **Clustering Feature** is a triple summarizing the information that we maintain about a cluster.

Definition 4.1 Given N d-dimensional data points in a cluster: $\{\vec{X_i}\}$ where $i = 1, 2, ..., N$, the **Clustering Feature (CF)** vector 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., $\sum_{i=1}^{N} \vec{X_i}^2$. □

Theorem 4.1 *(***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. Then the* **CF** *vector of the cluster that is formed by merging the two disjoint clusters, is:*

$$\mathbf{CF_1} + \mathbf{CF_2} = (N_1 + N_2, \vec{LS_1} + \vec{LS_2}, SS_1 + SS_2) \tag{9}$$

The proof consists of straightforward algebra. []

From the **CF** definition and additivity theorem, we know that the **CF** vectors of clusters can be stored and calculated incrementally and accurately as clusters are merged. It is also easy to prove that given the **CF** vectors of clusters, the corresponding $\vec{X0}$, R, D, $D0$, $D1$, $D2$, $D3$ and $D4$, as well as the usual quality metrics (such as weighted total/average diameter of clusters) can all be calculated easily.

One can think of a cluster as a set of data points, but only the **CF** vector stored as summary. This **CF** summary is not only efficient because it stores much less than all the data points in the cluster, but also accurate because it is sufficient for calculating all the measurements that we need for making clustering decisions in *BIRCH*.

4.1 CF Tree

A **CF** tree is a height-balanced tree with two parameters: branching factor B and threshold T. Each nonleaf node contains at most B entries of the form $[\mathbf{CF}_i, child_i]$, where $i = 1, 2, ..., B$, "$child_i$" is a pointer to its i-th child node, and $\mathbf{CF}_i$ is the **CF** of the subcluster represented by this 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 of the form $[\mathbf{CF}_i]$, where $i = 1, 2, ..., L$. In addition, each leaf node has two pointers, "*prev*" and "*next*" which are used to chain all leaf nodes together for efficient scans. A leaf node also represents a cluster made up of all the subclusters represented by its entries. But all entries in a leaf node must satisfy a *threshold requirement*, with respect to a threshold value T: *the diameter (or radius) has to be less than T*.

The tree size is a function of T. The larger T is, the smaller the tree is. We require a node to fit in a page of size P. Once the dimension d of the data space is given, the sizes of leaf and nonleaf entries are known, then B and L are determined by P. So P can be varied for performance tuning.

Such a **CF** tree will be built dynamically as new data objects are inserted. It is used to guide a new insertion into the correct subcluster for clustering purposes just the same as a B+-tree is used to guide a new insertion into the correct position for sorting purposes. The **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 many data points with diameter (or radius) under a specific threshold T).

4.2 Insertion into a CF Tree

We now present the algorithm for inserting an entry into a **CF** tree. Given entry "Ent", it proceeds as below:

1. *Identifying the appropriate leaf:* 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 Sec. 3.
2. *Modifying the leaf:* 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 condition[2]. If so, the **CF** vector for L_i is updated to reflect this. If not, a new entry for "Ent" is added to the leaf. If there is space on the leaf for this new entry, we are done, otherwise we must *split* the leaf node. Node splitting is done by choosing the **farthest** pair of entries as seeds, and redistributing the remaining entries based on the **closest** criteria.
3. *Modifying the path to the leaf:* After inserting "Ent" into a leaf, we must update the **CF** information for each nonleaf entry on the path to the leaf. In the absence of a split, this simply involves adding **CF** vectors to reflect the addition of "Ent". A leaf split requires us to insert a new nonleaf entry into the parent node, to describe the newly created leaf. If the parent has space for this entry, at all higher levels, we only need to update the **CF** vectors 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 increases by one.
4. *A Merging Refinement:* 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 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 two **closest** entries. If they are not the pair corresponding to the split, 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 split 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 on 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.

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 page size are remedied with a global (or semi-global) algorithm that arranges leaf entries across nodes (Phase 3 discussed in Sec. 5). Another undesirable artifact is that if the same data point is inserted twice, but at different times, the two copies might be entered into 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. This problem can be addressed with further refinement passes over the data (Phase 4 discussed in Sec. 5).

5 The BIRCH Clustering Algorithm

Fig. 1 presents the overview of *BIRCH*. The main task of Phase 1 is to scan all data and build an initial in-memory **CF** tree using the given amount of memory

[2] That is, the cluster merged with "Ent" and L_i must satisfy the threshold condition. Note that the **CF** vector of the new cluster can be computed from the **CF** vectors for L_i and "Ent".

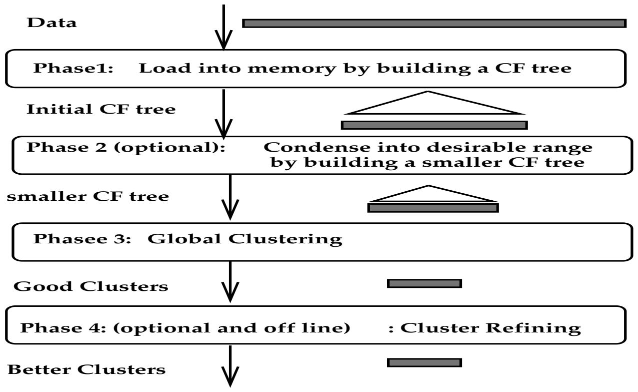

Figure 1: *BIRCH Overview*

and recycling space on disk. This **CF** tree tries to reflect the clustering information of the dataset as fine as possible under the memory limit. With crowded data points grouped as fine subclusters, and sparse data points removed as outliers, this phase creates a in-memory summary of the data. The details of Phase 1 will be discussed in Sec. 5.1. After Phase 1, subsequent computations in 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;
2. 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 memory;
3. less order sensitive because the leaf entries of the initial tree form an input order containing better data locality compared with the arbitrary original data input order.

Phase 2 is optional. We have observed that the existing global or semi-global clustering methods applied in Phase 3 have different input size ranges within which they perform well in terms of both speed and quality. So potentially there is a gap between the size of Phase 1 results and the input range of Phase 3. Phase 2 serves as a cushion and bridges this gap: Similar to Phase 1, it scans the leaf entries in the initial **CF** tree to rebuild a smaller **CF** tree, while removing more outliers and grouping crowded subclusters into larger ones.

The undesirable effect of the skewed input order, and splitting triggered by page size (Sec. 4.2) causes us to be unfaithful to the actual clustering patterns in the data. This is remedied in Phase 3 by using a global or semi-global algorithm to cluster all leaf entries. We observe that existing clustering algorithms for a set of data points can be readily adapted to work with a set of subclusters, each described by its **CF** vector. For example, with the **CF** vectors known, (1) naively, by calculating the centroid as the representative of a subcluster, we can treat each subcluster as a single point and use an existing algorithm without modification; (2) or to be a little more sophisticated, we can treat a subcluster of n data points as its centroid repeating n times and modify an existing algorithm slightly to take the counting information into account; (3) or to be general and accurate, we can apply an existing algorithm directly to the subclusters because the information in their **CF** vectors is usually sufficient for calculating most distance and quality metrics.

In this paper, we adapted an agglomerative hierarchical clustering algorithm by applying it directly to the subclusters represented by their **CF** vectors. It uses the accurate distance metric $D2$ or $D4$, which can be calculated from the **CF** vectors, during the whole clustering, and has a complexity of $O(N^2)$. It also provides the flexibility of allowing the user to specify either the desired number of clusters, or the desired diameter (or radius) threshold for clusters.

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 the rare misplacement problem mentioned in Sec. 4.2, and the fact that Phase 3 is applied on a coarse summary of the data. Phase 4 is optional and entails the cost of additional passes over the data to correct those inaccuracies and refine the clusters further. Note that up to this point, the original data has only been scanned once, although the tree and outlier information may have been scanned multiple times.

Phase 4 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, but also it ensures that all copies of a given data point 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 [GG92]. As a bonus, 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. 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.

5.1 Phase 1 Revisited

Fig. 2 shows the details of Phase 1. It starts with an initial threshold value, scans the data, and inserts points into the tree. If it runs out of memory before it finishes scanning the data, it increases the threshold value, rebuilds a new, *smaller* **CF** tree, by re-inserting the leaf entries of the old tree. After the old leaf entries have been re-inserted, the scanning of the data (and insertion into the new tree) is resumed from the point at which it was interrupted.

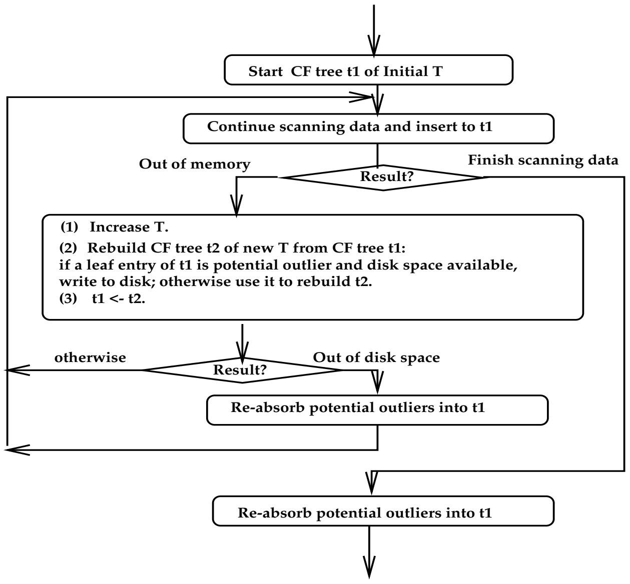

Figure 2: *Control Flow of Phase 1*

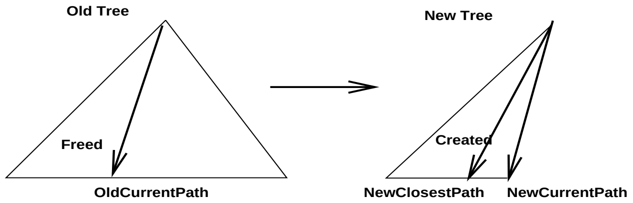

Figure 3: *Rebuilding CF Tree*

5.1.1 Reducibility

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 a leaf node corresponds to a path uniquely, and we will use path and leaf node interchangeably from now on.

The algorithm is illustrated in Fig. 3. With the natural path order defined above, it scans and frees the old tree path by path, and at the same time, creates the new tree path by path. The new tree starts with NULL, and "OldCurrentPath" starts with the leftmost path in the old tree. For "OldCurrentPath", the algorithm proceeds as below:

1. *Create the corresponding "NewCurrentPath" in the new tree*: nodes are added to the new tree exactly the same as in the old tree, so that there is no chance that the new tree ever becomes larger than the old tree.
2. *Insert leaf entries in "OldCurrentPath" to the new tree*: with the new threshold, each leaf entry in "OldCurrentPath" is tested against the new tree to see if it can fit [3] in the "NewClosestPath" that is found top-down with the **closest** criteria in the new 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 tree if there exists one, and repeat the above steps.*

From the rebuilding steps, old leaf entries are re-inserted, but the new tree can never become larger than the old tree. Since only nodes corresponding to "OldCurrentPath" and "NewCurrentPath" need to exist simultaneously, the maximal extra space needed for the tree transformation is h pages. So by increasing the threshold, we can rebuild a smaller **CF** tree with a limited extra memory.

Theorem 5.1 (**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.*

5.1.2 Threshold Values

A good choice of threshold value can greatly reduce the number of rebuilds. Since the initial threshold value T_0 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 zero by default; a knowledgeable user could change this.

[3] Either absorbed by an existing leaf entry, or created as a new leaf entry without splitting.

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 condition wrt. T_i). Based on the portion of the data that we have scanned and the tree that we have built up so far, we need to estimate the next threshold value T_{i+1}. This estimation is a difficult problem, and a full solution is beyond the scope of this paper. Currently, we use the following heuristic approach:

1. We try to choose T_{i+1} so that $N_{i+1} = \text{Min}(2N_i, N)$. That is, whether N is known, we choose to estimate T_{i+1} at most in proportion to the data we have seen thus far.
2. Intuitively, we want to increase threshold based on some measure of *volume*. There are two distinct notions of volume that we use in estimating 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 **CF** tree, 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 entries on this leaf. 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.)

5.1.3 Outlier-Handling Option

Optionally, we can use R bytes of disk space for handling *outliers*, which are leaf entries of low density that are judged to be unimportant wrt. the overall clustering pattern. When we rebuild the **CF** tree by re-inserting the old leaf entries, the size of the new tree is 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. "Far fewer", is of course another heuristics.

Periodically, the disk space may run out, and the potential outliers are scanned to see if they can be re-absorbed into the current tree without causing the tree to grow in size. — An increase in the threshold value or a change in the distribution due to the new data read after a potential outlier is written out could well mean that the potential outlier no longer qualifies as an outlier. When all data has been scanned, the potential outliers left in the disk space must be scanned to verify if they are indeed outliers. If a potential outlier can not be absorbed at this last chance, it is very likely a real outlier and can be removed.

Note that the entire cycle — insufficient memory triggering a rebuilding of the 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.

5.1.4 Delay-Split Option

When we run out of main memory, it may well be the case that still more data points can fit in the current **CF** tree, without changing the threshold. However, some of the data points that we read may require us to split a node in the **CF** tree, A simple idea is to write such data points to disk (in a manner similar to how outliers are written), and to proceed reading the data until we run out of disk space as well. The advantage of this approach is that in general, more data points can fit in the tree before we have to rebuild.

6 Performance Studies

We present a complexity analysis, and then discuss the experiments that we have conducted on *BIRCH* (and *CLARANS*) using synthetic as well as real datasets.

6.1 Analysis

First we analyze the cpu cost of Phase 1. 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"; 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 ES be the **CF** entry size. There are at most $\frac{M}{ES}$ leaf entries to re-insert, so the cost of re-inserting leaf entries is $O(d * \frac{M}{ES} * B(1 + \log_B \frac{M}{P}))$. 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 we never estimate farther than twice of the current size, 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 $O(d*N*B(1+\log_B \frac{M}{P})+\log_2 \frac{N}{N_0}*d*\frac{M}{ES}*B(1+\log_B \frac{M}{P}))$. 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 delay-split 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 delay-split) is not more than M, and that there are about $\log_2 \frac{N}{N_0}$ re-builds, the I/O cost of Phase 1 is not significantly different from the cost of reading in the dataset. Based on the above analysis — which is actually rather pessimistic, in the light of our experimental results — the cost of Phases 1 and 2 should scale linearly with N.

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 and the global algorithm chosen for this phase. Phase 4 scans the dataset again and puts each data point into the proper cluster; the time taken is proportional to $N * K$. (However with the newest "nearest neighbor" techniques, it can be improved [GG92] to be almost linear wrt. N.)

6.2 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.

Each dataset consists of K clusters of 2-d 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 $[r_l, r_h]$[4]. Once placed, the clusters cover a range of values in each dimension. We refer to these ranges as the "overview" of the dataset.

Parameter	Values or Ranges
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 .. $\sqrt{2}$
r_h (Higher r)	$\sqrt{2}$ .. $\sqrt{32}$
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

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

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 $\sqrt{K} \times \sqrt{K}$ 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. The *sine* pattern places the cluster centers on a 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 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 2-d independent normal distribution whose mean is the center c, and whose variance in each dimension is $\frac{r^2}{2}$. 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

[4] Note that when $n_l = n_h$ the number of points is fixed and when $r_l = r_h$ the radius is fixed.

Scope	Parameter	Default Value
Global	Memory (M)	80x1024 bytes
	Disk (R)	20% M
	Distance def.	D2
	Quality def.	$(\bar{D})$
	Threshold def.	threshold for D
Phase1	Initial threshold	0.0
	Delay-split	on
	Page size (P)	1024 bytes
	Outlier-handling	on
	Outlier def.	Leaf entry which contains < 25% of the average number of points per leaf entry
Phase3	Input range	1000
	Algorithm	Adapted HC
Phase4	Refinement pass	1
	Discard-outlier	off
	Outlier def.	Data point whose Euclidian distance to the closest seed is larger than twice of the radius of that cluster

Table 2: *BIRCH Parameters and Their Default Values*

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.

6.3 Parameters and Default Setting

BIRCH is capable of working under various 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 80 kbytes which is 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[ZRL95] 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 to choose $D2$ as default. Following Statistics tradition, we choose "weighted average diameter" (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 diameter as default.

In Phase 1, the initial threshold is default to 0. Based on a study of how page size affects performance[ZRL95], we selected $P = 1024$. The delay-split option is on so that given a threshold, the **CF** tree accepts more data points and reaches a higher capacity. The outlier-handling option is on so that *BIRCH* can remove outliers and concentrate on the dense places with the given amount of resources. For simplicity, we treat a leaf entry of which the number of data points is less than a quarter of the average as an outlier.

In Phase 3, most global algorithms can handle 1000 objects quite well. So we default the input range as 1000. We have chosen the adapted *HC* 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.

6.4 Base Workload Performance

The first set of experiments was to evaluate the ability of BIRCH to cluster various large datasets. All the times are presented in *second* in this paper. Three synthetic datasets, one for each pattern, were used. Table 3 presents the generator settings for them. The weighted average diameters of the actual clusters[5] , $\bar{D}_{act}$ are also included in the table.

Fig. 6 visualizes the actual clusters of DS1 by plotting a cluster as a circle whose center is the centroid, radius is the cluster radius, and label is the number of points in the cluster. The *BIRCH* clusters of DS1 are presented in Fig. 7. We observe that the *BIRCH* clusters are very similar to the actual clusters in terms of location, number of points, and radii. The maximal and average difference between the centroids of an actual cluster and its corresponding *BIRCH* cluster are 0.17 and 0.07 respectively. The number of points in a *BIRCH* cluster is no more than 4% different from the corresponding actual cluster. The radii of the *BIRCH* clusters (ranging from 1.25 to 1.40 with an average of 1.32) are close to, those of the actual clusters (1.41). Note that all the *BIRCH* radii are smaller than the actual radii. This is because *BIRCH* assigns the "outsiders" of an actual clusters to a proper *BIRCH* cluster. Similar conclusions can be reached by analyzing the visual presentations of DS2 and DS3 (but omitted here due to the lack of space).

As summarized in Table 4, it took *BIRCH* less than 50 seconds (on an HP 9000/720 workstation) to cluster 100,000 data points of each dataset. The pattern of the dataset had almost no impact on the clustering time. Table 4 also presents the performance results for 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*. As demonstrated in Table 4, changing the order of the data points had almost no impact on the performance of *BIRCH*.

6.5 Sensitivity to Parameters

We studied the sensitivity of *BIRCH*'s performance to the change of the values of some parameters. Due to the lack of space, here we can only present some major conclusions (for details, see [ZRL95]).

[5] From now on, we refer to the clusters generated by the generator as the "actual clusters" whereas the clusters identified by *BIRCH* as "*BIRCH* clusters".

Dataset	Generator Setting	D_{act}
DS1	grid, $K = 100, n_l = n_h = 1000, r_l = r_h = \sqrt{2}, k_g = 4, r_n = 0\%, o = randomized$	2.00
DS2	sine, $K = 100, n_l = n_h = 1000, r_l = r_h = \sqrt{2}, n_c = 4, r_n = 0\%, o = randomized$	2.00
DS3	random, $K = 100, n_l = 0, n_h = 2000, r_l = 0, r_h = 4, r_n = r_n = 0\%, o = randomized$	4.18

Table 3: *Datasets Used as Base Workload*

Initial threshold: (1) *BIRCH*'s performance is stable as long as the initial threshold is not excessively high wrt. the dataset. (2) $T_0 = 0.0$ works well with a little extra running time. (3) If a user does know a good T_0, then she/he can be rewarded by saving up to 10% of the time.

Page Size P: In Phase 1, smaller (larger) P tends to decrease (increase) the running time, requires higher (lower) ending threshold, produces less (more) but "coarser (finer)" leaf entries, and hence degrades (improves) the quality. However with the refinement in Phase 4, the experiments suggest that from $P = 256$ to 4096 , although the qualities at the end of Phase 3 are different, the final qualities after the refinement are almost the same.

Outlier Options: *BIRCH* was tested on "noisy" datasets with all the outlier options *on*, and *off*. The results show that with all the outlier options on, *BIRCH* is not slower but faster, and at the same time, its quality is much better.

Memory Size: In Phase 1, as memory size (or the maximal tree size) increases, the running time increases because of processing a larger tree per rebuilt, but only slightly because it is done in memory; (2) more but finer subclusters are generated to feed the next phase, and hence results in better quality; (3) the inaccuracy caused by insufficient memory can be compensated to some extent by Phase 4 refinements. In another word, *BIRCH* can tradeoff between memory and time to achieve similar final quality.

6.6 Time Scalability

Two distinct ways of increasing the dataset size are used to test the scalability of *BIRCH*.

Increasing the Number of Points per Cluster: 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 to change n, and hence N. The running time for the first 3 phases, as well as for all 4 phases are plotted against the dataset size N in Fig. 4. Both of them are shown to grow linearly wrt. N consistently for all three patterns.

Increasing the Number of Clusters: For each of DS1, DS2 and DS3, we create a range of datasets by keeping the generator settings the same except for changing K to change N. The running time for the first 3 phases, as well as for all 4 phases are plotted against the dataset size N in Fig. 5. Since both N and K are growing, and Phase 4's complexity is now $O(K*N)$ (can be improved to be almost linear in the future), the total time is not exactly linear wrt. N. However the running time for the first 3 phases is again confirmed to grow linearly wrt. N consistently for all three patterns.

Dataset	Time	$\bar{D}$	Dataset	Time	$\bar{D}$
DS1	47.1	1.87	DS1o	47.4	1.87
DS2	47.5	1.99	DS2o	46.4	1.99
DS3	49.5	3.39	DS3o	48.4	3.26

Table 4: *BIRCH Performance on Base Workload wrt. Time, $\bar{D}$ and Input Order*

Dataset	Time	$\bar{D}$	Dataset	Time	$\bar{D}$
DS1	839.5	2.11	DS1o	1525.7	10.75
DS2	777.5	2.56	DS2o	1405.8	179.23
DS3	1520.2	3.36	DS3o	2390.5	6.93

Table 5: *CLARANS Performance on Base Workload wrt. Time, $\bar{D}$ and Input Order*

6.7 Comparisons of BIRCH and CLARANS

In this experiment we compare the performance of *CLARANS* and *BIRCH* on the base workload. First *CLARANS* assumes that the memory is enough for holding the whole dataset, so it needs much more memory than *BIRCH* does. 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. Fig. 8 visualizes the *CLARANS* clusters for DS1. Comparing them with the actual clusters for DS1 we can observe that: (1) The pattern of the location of the cluster centers is distorted. (2) The number of data points in a *CLARANS* cluster can be as many as 57% different from the number in the actual cluster. (3) The radii of *CLARANS* clusters varies largely from 1.15 to 1.94 with an average of 1.44 (larger than those of the actual clusters, 1.41). Similar behaviors can be observed the visualization of *CLARANS* clusters for DS2 and DS3 (but omitted here due to the lack of space).

Table 5 summarizes the performance of *CLARANS*. For all three datasets of the base workload, (1) *CLARANS* is at least 15 times slower than *BIRCH*, and is sensitive to the pattern of the dataset. (2) The $\bar{D}$ value for the *CLARANS* clusters is much larger than that for the *BIRCH* clusters. (3) The results for DS1o, DS2o, and DS3o show that when the data points are ordered, the time and quality of *CLARANS* degrade dramatically. In conclusion, for the base workload, *BIRCH* uses much less memory, but is faster, more accurate, and less order-sensitive compared with *CLARANS*.

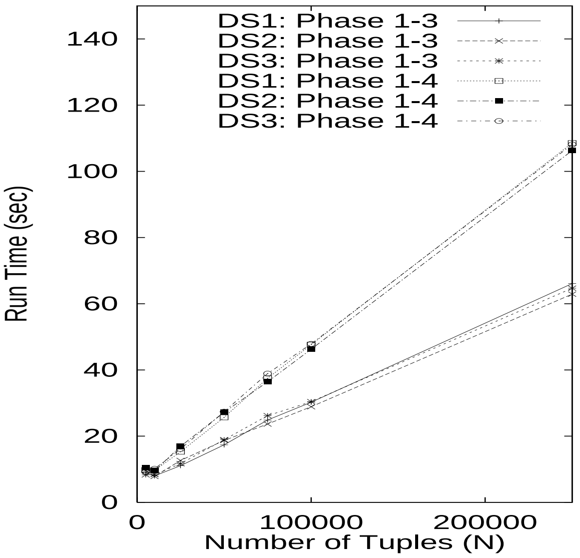

Figure 4: *Scalability wrt. Increasing n_l, n_h*

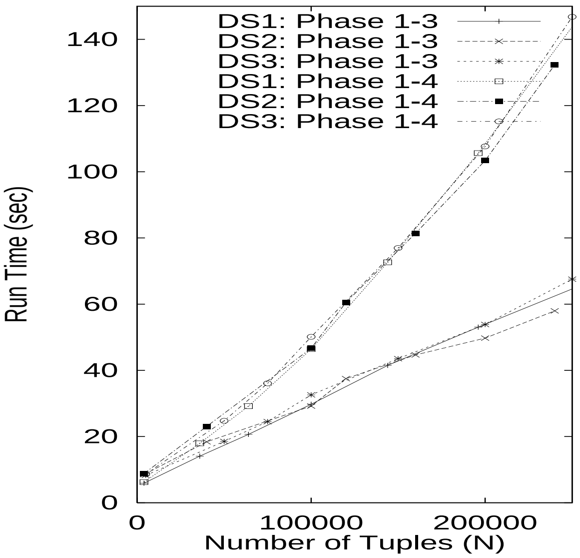

Figure 5: *Scalability wrt. Increasing K*

6.8 Application to Real Datasets

BIRCH has been used for filtering real images. Fig. 9 are two similar images of trees with a partly cloudy sky as the background, taken in two different wavelengths. The top one is in near-infrared band (NIR), and the bottom one is in visible wavelength band (VIS). Each image contains 512x1024 pixels, and each pixel actually has a pair of brightness values corresponding to NIR and VIS. Soil scientists receive hundreds of such image pairs and try to first filter the trees from the background, and then filter the trees into sunlit leaves, shadows and branches for statistical analysis.

We applied *BIRCH* to the (NIR,VIS) value pairs for all pixels in an image (512x1024 2-d tuples) by using 400 kbytes of memory (about 5% of the dataset size) and 80 kbytes of disk space (about 20% of the memory size), and weighting NIR and VIS values equally. We obtained 5 clusters that correspond to (1) very bright part of sky, (2) ordinary part of sky, (3) clouds, (4) sunlit leaves (5) tree branches and shadows on the trees. This step took 284 seconds.

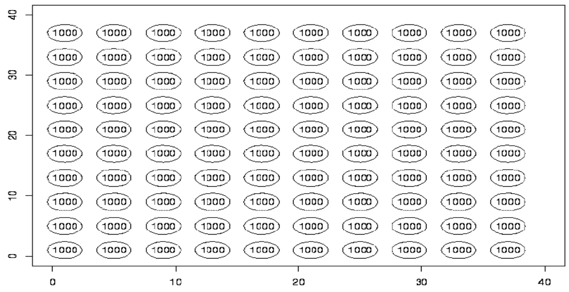

Figure 6: *Actual Clusters of DS1*

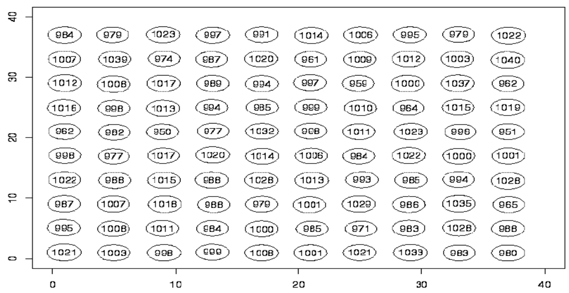

Figure 7: *BIRCH Clusters of DS1*

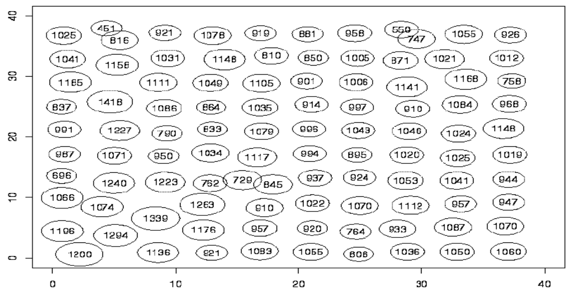

Figure 8: *CLARANS Clusters of DS1*

However the branches and shadows were too similar to be distinguished from each other, although we could separate them from the other cluster categories. So we pulled out the part of the data corresponding to (5) (146707 2-d tuples) and used *BIRCH* again. But this time, (1) NIR was weighted 10 times heavier than VIS because we observed that branches and shadows were easier to tell apart from the NIR image than from the VIS image; (2) *BIRCH* ended with a finer threshold because it processed a smaller dataset with the same amount of memory. The two clusters corresponding to branches and shadows were obtained with 71 seconds. Fig. 10 shows the parts of image that correspond to

Figure 9: *The images taken in NIR and VIS*

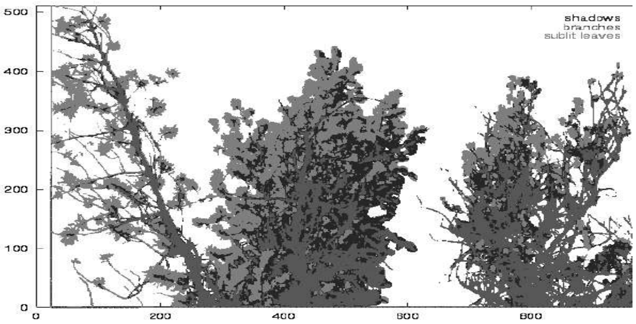

Figure 10: *The sunlit leaves, branches and shadows*

sunlit leaves, tree branches and shadows on the trees, obtained by clustering using *BIRCH*. Visually, we can see that it is a satisfactory filtering of the original image according to the user's intention.

7 Summary and Future Research

BIRCH is a clustering method for very large datasets. It makes a large clustering problem tractable by concentrating on densely occupied portions, and using a compact summary. It utilizes measurements that capture the natural closeness of data. These measurements can be stored and updated incrementally in a height-balanced tree. *BIRCH* can work with any given amount of memory, and the I/O complexity is a little more than one scan of data. Experimentally, *BIRCH* is shown to perform very well on several large datasets, and is significantly superior to *CLARANS* in terms of quality, speed and order-sensitivity.

Proper parameter setting is important to *BIRCH*'s efficiency. In the near future, we will concentrate on studying (1) more reasonable ways of increasing the threshold dynamically, (2) the dynamic adjustment of outlier criteria, (3) more accurate quality measurements, and (4) data parameters that are good indicators of how well *BIRCH* is likely to perform. We will explore *BIRCH*'s architecture for opportunities of parallel executions as well as interactive learnings. As an incremental algorithm, *BIRCH* will be able to read data directly from a tape drive, or from network by matching its clustering speed with the data reading speed. We will also study how to make use of the clustering information obtained to help solve problems such as storage or query optimization, and data compression.

References

[CKS88] 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.

[DH73] Richard Duda, and Peter E. Hart, *Pattern Classification and Scene Analysis*, Wiley, 1973.

[DJ80] 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.

[EKX95a] 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.

[EKX95b] 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.

[Fis87] Douglas H. Fisher, *Knowledge Acquisition via Incremental Conceptual Clustering*, Machine Learning, 2(2), 1987

[Fis95] Douglas H. Fisher, *Iterative Optimization and Simplification of Hierarchical Clusterings*, Technical Report CS-95-01, Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235.

[GG92] A. Gersho and R. Gray, *Vector quantization and signal compression*, Boston, Ma.: Kluwer Academic Publishers, 1992.

[KR90] Leonard Kaufman, and Peter J. Rousseeuw, *Finding Groups in Data - An Introduction to Cluster Analysis*, Wiley Series in Probability and Mathematical Statistics, 1990.

[Leb87] Michael Lebowitz, *Experiments with Incremental Concept Formation : UNIMEM*, Machine Learning, 1987.

[Lee81] 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.

[Mur83] F. Murtagh, *A Survey of Recent Advances in Hierarchical Clustering Algorithms*, The Computer Journal, 1983.

[NH94] Raymond T. Ng and Jiawei Han, *Efficient and Effective Clustering Methods for Spatial Data Mining*, Proc. of VLDB, 1994.

[Ols93] Clark F. Olson, *Parallel Algorithms for Hierarchical Clustering*, Technical Report, Computer Science Division, Univ. of California at Berkeley, Dec.,1993.

[ZRL95] Tian Zhang, Raghu Ramakrishnan, and Miron Livny, *BIRCH: An Efficient Data Clustering Method for Very Large Databases*, Technical Report, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1995.