Clustering Data Streams: Theory and Practice

Sudipto Guha[*] Adam Meyerson [†] Nina Mishra [‡] Rajeev Motwani [§]
Liadan O'Callaghan [¶]

January 14, 2003

Abstract

The data stream model has recently attracted attention for its applicability to numerous types of data, including telephone records, web documents and clickstreams. For analysis of such data, the ability to process the data in a single pass, or a small number of passes, while using little memory, is crucial. We describe such a streaming algorithm that effectively clusters large data streams. We also provide empirical evidence of the algorithm's performance on synthetic and real data streams.

1 Introduction

A data stream is an ordered sequence of points $x_1, \ldots, x_n$ that must be accessed in order and that can be read only once or a small number of times. Each reading of the sequence is called a *linear scan* or a *pass*. The stream model is motivated by emerging applications involving massive data sets; for example, customer click streams, telephone records, large sets of web pages, multimedia data, financial transactions,

[*]Corresponding author. Department of Computer Science, University of Pennsylvania, Philadelphia, PA 19104. This work was done while the author was a student at Stanford University supported by an IBM Research Fellowship and NSF Grant IIS-9811904. Email: `sudipto@cis.upenn.edu`

[†]Currently at Department of Computer Science, Carnegie Mellon University and would be at Department of Computer Science, UCLA, starting Fall 2003. This work was done while the author was at Department of Computer Science, Stanford University, Palo Alto, CA 94305 an his research was supported by ARO DAAG-55-97-1-0221. Email: `awm@cs.stanford.edu`

[‡]Hewlett Packard Laboratories and Stanford University, Department of Computer Science. Research supported in part by NSF Grant EIA-0137761. Email: `nmishra@hpl.hp.com` and `nmishra@theory.stanford.edu`

[§]Department of Computer Science, Stanford University, Palo Alto, CA 94305. Research supported by NSF Grant IIS-9811904. Email: `rajeev@cs.stanford.edu.`

[¶]Department of Computer Science, Stanford University, Palo Alto, CA 94305. Research supported in part by NSF Graduate Fellowship and NSF Grant IIS-9811904. Email: `loc@cs.stanford.edu`

and observational science data are better modeled as data streams. These data sets are far too large to fit in main memory and are typically stored in secondary storage devices. Linear scans are the only cost-effective access method; random access is prohibitively expensive. Some data sets, such as router packet statistics, meteorological data, and sensor network data, are transient and need not be realized on disk; the data must be processed as they are produced, and discarded in favor of summaries whenever possible. As the size of such data sets far exceeds the amount of space (main memory) available to an algorithm, it is not possible for a data stream algorithm to "remember" too much of the data scanned in the past. This scarcity of space necessitates the design of a novel kind of algorithm that stores only a *summary* of past data, leaving enough memory for the processing of future data. Each scan of a large set on a slow device is expensive, and so the criteria by which the performance of a data stream algorithm is judged include the number of linear scans in addition to the usual ones (running time and memory usage). In the case of "transient" streams, only one scan is possible.

The data stream and online or incremental models are similar in that they both require decisions to be made before all the data are available. The models are not identical, however. Whereas an online algorithm can have access to the first i data points (and its i previous decisions) when reacting to the $(i+1)$th point, the amount of memory available to a data stream algorithm is bounded by a function (usually a sub-linear function) of the input size. Furthermore, unlike an online algorithm, a data stream algorithm may not be required to take an irrevocable action after the arrival of each point; it may be allowed to take action after a group of points arrives. Still, the models are very similar, and, in particular, a sublinear-space online algorithm is a data stream algorithm as well.

The definition of the streaming model, inclusive of multiple passes, was first characterized in this form by Henzinger *et al.* [38], although the work of Munro and Patterson [64] and of Flajolet and Martin [23] predates this definition. The interest in the model started from the results of Alon *et al.* [4], who proved upper and lower bounds for the memory requirements of one-pass algorithms computing statistics over data streams.

Clustering, a useful and ubiquitous tool in data analysis, is, in broad strokes, the problem of finding a partition of a data set so that, under some definition of "similarity," similar items are in the same part of the partition and different items are in different parts. The particular definition of clustering that is the focus of this paper is the k–Median objective, that of identifying k centers so that the sum of distances from each point to its nearest center is minimized.

We study the k–Median problem in the stream context and provide a streaming algorithm with theoretical performance guarantees. We begin by giving an algorithm that requires small space, and then later address the issue of clustering in one pass. We give a simple algorithm based on divide-and-conquer that achieves a constant-factor approximation in small space. Elements of the algorithm and its analysis form the basis for the constant-factor algorithm given subsequently. This algorithm runs in time $O(n^{1+\epsilon})$, uses $O(n^{\epsilon})$ memory and makes a single pass over the data. Next, using randomization we show how to reduce the the running time to $\tilde{O}(nk)$ without requiring more than a single pass [1].

We also provide a new clustering algorithm that is used by our streaming method. The algorithm is based on a facility location algorithm (defined in Section 4.1) that might produce more than k centers. We show how the facility location algorithm can be modified to produce exactly k clusters and thereby solve the k–Median problem. Since the running time of the resulting algorithm is expensive – $O(n^2)$ for n points – we offer some innovative techniques to speed up the algorithm. We give a sound way of quickly initializing the facility location algorithm with a reasonably good set of centers. We also show, under some assumptions, that we can restrict the choice of potential centers and yet find a good clustering.

We performed an extensive series of experiments comparing our enhanced facility location algorithm against the commonly-used k–Means algorithm. The results uncover an interesting trade-off between the cluster quality and the running time. Our facility location-based algorithm produces solutions of near-optimum quality, with smaller variance over multiple runs as compared to k–Means which produced solutions of inferior quality with higher variance over multiple runs. However, our algorithm took more time to find its better answers.

In our experiments with real data, we found that our stream technique has clustering quality comparable to the clustering quality of running the algorithm on all the data at once. We also compared our streaming algorithm with BIRCH and found a similar tradeoff between cluster quality and running time. Even though we retain a tiny fraction of the amount of information that BIRCH retains, our algorithms took somewhat longer to run, but produced solutions of higher quality.

2 Related Work

Streaming Algorithms: A rich body of fundamental research has emerged in the data stream model of computation. Problems that can be solved in small space when the data is a stream include: frequency estimation [36, 12], norm estimation [4, 22, 44], order statistics [64, 57, 56, 30, 28], synopsis structures [26], time indexed data [25, 32, 17, 8], signal reconstructions [24, 18, 1, 33, 27, 29, 72].

[1]The notation $\tilde{O}(nk)$ is same as the O–notation except it hides polylogarithmic terms as opposed to constants. For example $O(n \log^2 n)$ is $\tilde{O}(n)$.

Theoretical Analysis of Clustering: For most natural clustering objective functions, the optimization problems turn out to be NP hard. Therefore, most theoretical work is on the design of approximation algorithms: algorithms that guarantee a solution whose objective function value is within a fixed factor of the value of the optimal solution.

The k–Center and k–Median problems are among the most studied clustering problems, since, for both problems, a solution simply consists of k members of the data set, where k, the desired number of clusters, is given as input. The k members are chosen as cluster centers, each data point is assumed to be "assigned" to cluster center nearest it, and its "assignment" distance is its distance to this nearest center. Thus, for the purpose of this paper, the solution to the clustering problem is exactly k centers.

The k–Center objective function is to minimize the largest assignment distance whereas in k–Median the *sum* of assignment distances is to be minimized. The k–Center measure is obviously sensitive to outliers. The k–Median objective is less sensitive to noise. The Facility Location problem is the Lagrangian relaxation of the k–Median problem, where the number of centers is unrestricted but there is an additional cost for each center included in the solution. There is abundant literature on these, books [45, 62, 55], provable algorithms [41, 49, 54, 53, 71, 31, 16, 15, 6, 52, 47, 14, 59, 7, 46], the running time of provable clustering heuristics [21, 10, 42, 34, 73, 60], and special metric spaces [6, 52, 43, 68].

The k–Median problem is also relevant in the context of the well-known EM and k–Means heuristics. EM (expectation maximization) iteratively refines a clustering by trying to maximize the probability that the current clustering corresponds to a model. If the clusters are modeled as Gaussian spheres enforcing a partition, the heuristic becomes k–Means. In k–Means, k centers are chosen initially, and, in each iteration, each center is replaced by the geometric mean of the cluster corresponding to it. This is the k–Median objective function defined over real spaces in which assignment costs (distances) are replaced by their squares.

Charikar *et al.* [13] gave a constant-factor, single-pass k–Center algorithm using $O(nk \log k)$ time and $O(k)$ space. For k-Median, we give a constant-factor, single-pass approximation in time $\tilde{O}(nk)$ and sublinear n^{α} space for constant $\alpha > 0$. Notice that in a single pass the data points cannot be labeled with the clusters they belong to. These algorithms output the cluster centers only.

More involved definitions of clustering based on other graph theoretic notions exist; cliques [9], cuts [74], conductance [48]. [18, 69, 2] consider clustering defined by projections onto subspaces.

Existing Large Scale Clustering Algorithms: Clustering is very well studied in the applied literature, and the following is by no means an exhaustive study of related work. k–Means is a widely used heuristic, but, since the mean of a cluster is not always defined, alternate medoid-based algorithms have been developed. For example, k–Medoids [50] selects k initial centers and repeatedly replaces an existing center with a random chosen point, if doing so improves the sum of squared assignment distances.[2]. CLARA [50] proposed sampling to reduce the number of exchanges considered, since choosing a new medoid among all the remaining points is time-consuming; CLARANS [66] draws a fresh sample of feasible centers before each calculation of improvement. We will later see a slightly differing non-repeated sampling approach. k-medoid approaches, including PAM, CLARA, and CLARANS, are known not to be scalable and thus are inappropriate for stream analysis.

Other partitioning methods include that of Bradley *et al*. [11], and its subsequent improvement by Farnstorm *et al*. [20], which repeatedly takes k weighted centers (initially chosen randomly with weight 1) and as much data as can fit in main memory, and computes a k-clustering. The new k centers so obtained are then weighted by the number of points assigned, the data in memory is discarded and the process repeats on the remaining data. A key difference between this approach and ours is that their algorithm places higher significance on points later in the data set; we make no such assumptions. Further, these approaches are not known to outperform the popular BIRCH algorithm.

A hierarchical clustering is a sequence of nested partitions. An *agglomerative* algorithm for hierarchical clustering starts by placing each input data point in its own cluster, and then repeatedly merges the closest pair of clusters until the number of clusters reduces to k. Many hierarchical agglomerative clustering (HAC) heuristics exist. Under the celebrated SLINK heuristic, the distance between clusters A and B is defined by the closest pair of points $a \in A$, $b \in B$. Another hierarchical technique is CURE [35] which represents a cluster by multiple points that are initially well-scattered in the cluster and then shrunk towards the cluster center by a certain fraction. Depending on the values of the CURE parameters, the algorithm can fall anywhere along a spectrum from SLINK (HAC) to k–Medoid. Both HAC and CURE are designed to discover clusters of arbitrary shape and thus do not necessarily optimize the k–Median objective. Hierarchical algorithms, including BIRCH [76] are known to suffer from the problem that hierarchical merge or split operations are irrevocable [37].

The stream clustering algorithm we present is somewhat similar to CURE. The algorithm given here constructs a hierarchical clustering. However if we consider the dendogram, our algorithm operates at the same time on all layers of the tree and maintains a front. Our algorithm is similar to CURE in that both apply a partitioning approach and cluster data bottom up, but, whereas CURE is geared towards robustness and clustering arbitrary shapes, the algorithm presented here is designed to produce a provably good clustering.

[2] In Section 4.3, we will see a pivoting scheme which at every step will not preserve the number of medoids but eventually guarantee that the final solution is close to the best possible one.

Other known approaches such as DBSCAN [19], OPTICS [5] and DENCLUE [39], STING [75], CLIQUE [3], Wave-Cluster [70], and OPTIGRID [40], are not designed to optimize the k–Median objective.

3 A Provable Stream Clustering Framework

3.1 Clustering in Small Space

Data stream algorithms must not have large space requirements, and so our first goal will be to show that clustering can be carried out in small space (n^ϵ for n data points, and $0 < \epsilon < 1$), without being concerned with the number of passes. Subsequently we will develop a one-pass algorithm. We first investigate algorithms that examine the data in a piecemeal fashion. In particular, we study the performance of a divide-and-conquer algorithm, called Small-Space, that divides the data into pieces, clusters each of these pieces, and then again clusters the centers obtained (where each center is weighted by the number of points assigned to it). We show that this piecemeal approach is good, in that if we had a constant-factor approximation algorithm, running it in divide-and-conquer fashion would still yield a (slightly worse) constant-factor approximation. We then propose another algorithm (Smaller-Space) that is similar to the piecemeal approach except that instead of reclustering only once, it repeatedly reclusters weighted centers. For this algorithm, we prove that if we recluster a constant number of times, a constant-factor approximation is still obtained, although, as expected, the constant factor worsens with each successive reclustering.

3.1.1 Simple Divide-and-Conquer and Separability Theorems

For simplicity we start with the version of the algorithm that reclusters only once.

Algorithm Small-Space(S)

1. Divide S into l disjoint pieces $\chi_1, \ldots, \chi_l$.
2. For each i, find $O(k)$ centers in χ_i. Assign each point in χ_i to its closest center.
3. Let χ' be the $O(lk)$ centers obtained in (2), where each center c is weighted by the number of points assigned to it.
4. Cluster χ' to find k centers.

We are interested in clustering in small space, l will be set so that both S and χ' fit in main memory. If S is very large, no such l may exist – we will address this issue later.

Definition 1 (The k–median Problem) *Given an instance (S,k) of k–Median, i.e., an integer k and a set S of n points with metric $d(\cdot,\cdot)$, the k–*Median cost *(or simply the* cost*) of a set of medians $C_1,\ldots,C_k$ is $f(S,C_1,\ldots,C_k) = \sum_{x\in S}\min_{1\le i\le k} d(x,C_i)$. That is, the cost of a solution is the sum of assignment distances. Define cost(S,Q) to be the smallest possible cost if the medians are required to belong to the set Q. The optimization problem is to find cost(S,S) for the discrete case and cost(S,R^d) for the Euclidean case.*

Before analyzing algorithm Small-Space, we describe the relationship between the discrete and continuous clustering problem. The following is folklore (the proof can be found in [34]):

Theorem 1 *Given an instance (S,k) of k–Median cost$(S,S) \le 2$cost(S,Q) for any Q.*[3]

The following separability theorem sets the stage for a divide-and-conquer algorithm. This theorem carries over to other clustering metrics such as the sum of squared distances.

Theorem 2 *Consider an arbitrary partition of a set S of n points into $\chi_1,\ldots,\chi_\ell$. Then $\sum_{i=1}^{\ell}$cost$(\chi_i,\chi_i) \le 2$cost(S,S).* [4]

Proof: ¿From Theorem 1, cost$(\chi_i,\chi_i) \le 2$cost(χ_i,S). Summing over i the result follows. □

Next we show that the new instance, where all the points i that have median i' shift their weight to the point i' (i.e., the weighted $O(lk)$ centers S' in Step 2 of Algorithm Small-Space), has a good feasible clustering solution. Assigning a point i' of weight w to a median at distance d will cost wd; that is, assignment distances are multiplied by weights in the objective function. Notice that the set of points in the new instance is much smaller and may not even contain the optimum medians for S.

Theorem 3 *If $C = \sum_{i=1}^{\ell}$cost(χ_i,χ_i) and C^* =cost$(S,S) = \sum_{i=1}^{\ell}$cost$(\chi_i,S)$ then there exists a solution of cost at most $2(C+C^*)$ to the new weighted instance χ'.* [4]

Proof: For all $1\le i\le \ell$, let $C_{i,1},\ldots,C_{i,k}\in\chi_i$ be the medians that achieve the minimum cost(χ_i,χ_i). Let the medians that achieve the minimum cost(S,S) be $C_1^*,\ldots,C_k^*$.

For $x\in\chi_i$, let $c(x)$ denote the closest of $C_{i,1},\ldots,C_{i,k}$ to x, and let $C^*(x)$ denote the closest of $C_1^*,\ldots,C_k^*$ to x. Also let $w_{i,j}$ be the number of members x of χ_i for which $c(x) = C_{i,j}$ (that is, $w_{i,j}$ is the weight in χ' of $C_{i,j}$). For each $C_{i,j}$, there is a member of $C_1^*,\ldots,C_k^*$ within a distance of $\min_{c(x)=C_{i,j}}(d(x,c(x)) + d(x,C^*(x)))$ by the triangle inequality. Therefore, $f(\chi',C_1^*,\ldots,C_k^*) \le$

[3]The factor 2 is not present in the familiar case where $S\subset Q = R^d$.

[4]Again the factor 2 is not present in the case that the data are points in R^d and the medians can be anywhere in R^d.

$\sum_{C_{i,j} \in \chi'} w_{i,j} \min_{c(x)=C_{i,j}} (d(x, c(x)) + d(x, C^*(x)))$, which is at most $\sum_{x \in S} (d(x, c(x)) + d(x, C^*(x))) \leq C + C^*$. Thus cost$(\chi', S) \leq C + C^*$ and by Theorem 1 there is a solution for χ' of cost at most $2(C + C^*)$. □

We now show that if we run a bicriteria (a, b)-approximation algorithm (where at most ak medians are output with cost at most b times the optimum k–Median solution) in Step 2 of Algorithm Small-Space and we run a c-approximation algorithm in Step 4, then the resulting approximation by Small-Space can be suitably bounded. Note that Theorems 2 and 3 still apply if ak medians (where $a > 1$ is a constant) are found for each χ_i.

Theorem 4 *The algorithm Small-Space has an approximation factor of* $2c(1 + 2b) + 2b$.[5]

Proof: Let the optimal k-median solution be of cost C^*. Theorem 2 implies that the sum of the costs of the optimal solutions to the instances $\chi_1, \ldots, \chi_\ell$ is at most $2C^*$. Since a b-approximate solution is found for each χ_i, the cost C of the solution at the end of the first stage is at most $2bC^*$. By Theorem 3, the new instance χ' must admit a solution of cost at most $2(C + C^*)$; a c-approximation algorithm is guaranteed to find a solution of cost $2c(1 + 2b)C^*$ or less. The sum of the two bounds gives a bound on the cost of the final medians; the theorem follows. □

The black-box nature of this algorithm allows us to devise divide-and-conquer algorithms.

3.1.2 Divide-and-Conquer Strategy

We generalize Small-Space so that the algorithm recursively calls itself on a successively smaller set of weighted centers.

Algorithm Smaller-Space(S,i)

1. If $i < 1$ then Halt.
2. Divide S into l disjoint pieces $\chi_1, \ldots, \chi_l$.
3. For each $h \in \{1, \ldots, l\}$, find $O(k)$ centers in χ_h. Assign each point in χ_h to its closest center.
4. Let χ' be the $O(lk)$ centers obtained in (2), where each center c is weighted by the number of points assigned to it.
5. Call Algorithm Smaller-Space$(\chi', i - 1)$.

[5]The approximation factor is $c(b + 1) + b$ when the points fall in R^d and the medians can be placed anywhere in R^d.

We can claim the following.

Theorem 5 *For constant i, Algorithm Smaller-Space(S, i) gives a constant-factor approximation to the k–Median problem.*

Proof: Assume that the approximation factor for the jth level is A_j. From Theorem 2 we know that the cost of the solution of the first level is $2b$ times optimal. From Theorem 4 we get that the approximation factor A_j would satisfy a simple recurrence,

$$A_j = 2A_{j-1}(2b + 1) + 2b$$

The solution of the recurrence is $c \cdot (2(2b + 1))^j$. This is $O(1)$ given j is a constant. $\square$

Since the intermediate medians in χ' must be stored in memory, the number of subsets l that we partition S into is limited. In particular, $lk \leq M$ and $(n/l) \leq M$ since each partition must also be resident. Such an l may not always exist.

In the next section we will see a way to get around this problem. We will implement the hierarchical scheme more cleverly and obtain a clustering algorithm for the streaming model.

3.2 Clustering in The Data Stream Model

Under the Data Stream Model, computation takes place within bounded space M and the data can only be accessed via linear scans (i.e., a data point can be seen only once in a scan, and points must be viewed in order). In this section we will modify the multi-level algorithm to operate on data streams. We will present a one-pass, $O(1)$-approximation in this model assuming that the bounded memory M is not too small, more specifically n^ϵ where n denotes the size of the stream.

We will maintain a forest of assignments. We will complete this to k trees, and all the nodes in a tree will be assigned to the median denoted by the root of the tree. First we will show how to solve the problem of storing intermediate medians. Next we will inspect the space requirements and running time.

Data Stream Algorithm We will modify our multi-level algorithm slightly:

1. Input the first m points; use a bicriterion algorithm to reduce these to $O(k)$ (say $2k$) points. As usual, the weight of each intermediate median is the number of points assigned to it in the bicriterion clustering. (Assume m is a multiple of $2k$.) This requires $O(f(m))$ space, which for a primal dual algorithm can be $O(m^2)$. We will see a $O(mk)$-space algorithm later.

2. Repeat the above till we have seen $m^2/(2k)$ of the original data points. At this point we have m intermediate medians.
3. Cluster these m first-level medians into $2k$ second-level medians and proceed.
4. In general, maintain at most m level-i medians, and, on seeing m, generate $2k$ level-$i+1$ medians, with the weight of a new median as the sum of the weights of the intermediate medians assigned to it.
5. After seeing all the original data points (or to have a clustering of the points seen so far) we cluster all the intermediate medians into k final medians.

Note that this algorithm is identical to the multi-level algorithm described before.

The number of levels required by this algorithm is at most $O(\log(n/m)/\log(m/k))$. If we have $k \ll m$ and $m = O(n^\epsilon)$ for some constant $\epsilon < 1$, we have an $O(1)$-approximation. We will have $m = \sqrt{M}$ where M is the memory size (ignoring factors due to maintaining intermediate medians of different levels). We argued that the number of levels would be a constant when $m = n^\epsilon$ and hence when $M = n^{2\epsilon}$ for some $\epsilon < 1/2$.

Linear Space Clustering The approximation quality which we can prove (and intuitively the actual quality of clustering obtained on an instance) will depend heavily on the number of levels we have. From this perspective it is profitable to use a space-efficient algorithm.

We can use the local search algorithm in [14] to provide a bicriterion approximation in *space linear in* m, the number of points clustered at a time. The advantage of this algorithm is that it maintains only an assignment and therefore uses linear space. However the complication is that for this algorithm to achieve a bounded bicriterion approximation, we need to set a "cost" to each median used, so that we penalize if many more than k medians are used. The algorithm solves a facility location problem after setting the cost of each median to be used. However this can be done by guessing this cost in powers of $(1+\gamma)$ for some $0 < \gamma < 1/6$ and choosing the best solution with at most $2k$ medians. In the last step, to get k medians we use a two step process to reduce the number of medians to $2k$ and then use [47, 14] to reduce to k. This allows us to cluster with $m = M$ points at a time provided $k^2 \leq M$.

The Running Time The running time of this clustering is dominated by the contribution from the first level. The local search algorithm is quadratic and the total running time is $O(n^{1+\epsilon})$ where $M = n^\epsilon$. We argued before, however, that ϵ will not be very small and hence the approximation quality which we can prove will remain small. The theorem follows,

Theorem 6 *We can solve the k–Median problem on a data stream with time $O(n^{1+\epsilon})$ and space $\Theta(n^{\epsilon})$ up to a factor $2^{O(\frac{1}{\epsilon})}$.*

3.3 Clustering Data Streams in $\tilde{O}(nk)$ Time

To achieve scalability we would like a $\tilde{O}(nk)$ algorithm over an algorithm super linear in n. Let us recall the algorithm we have developed so far. We have $k^2 \ll M$, and we are applying an alternate implementation of a multi-level algorithm. We are clustering $m = O(M)$ (assuming $M = O(n^{\epsilon})$ for constant $\epsilon > 0$) points and storing $2k$ medians to "compress" the description of these data points. We use the local search-based algorithm in [14]. We keep repeating this procedure till we see m of these descriptors or intermediate medians and compress them further into $2k$. Finally, when we are required to output a clustering, we compress all the intermediate medians (over all the levels there will be at most $O(M)$ of them) and get $O(k)$ penultimate medians which we cluster into exactly k using the primal dual algorithm as in [47, 14].

Subquadratic time Clustering: We will use the results in [42] on metric space algorithms that are subquadratic. The algorithm as defined will consist of two passes and will have constant probability of success. For high probability results, the algorithm will make $O(\log n)$ passes. As stated, the algorithm will only work if the original data points are *unweighted*. Consider the following algorithm:

1. Draw a sample of size $s = \sqrt{nk}$.
2. Find k medians from these s points using the primal dual algorithm in [47].
3. Assign each of the n original points to its closest median.
4. Collect the n/s points with the largest assignment distance.
5. Find k medians from among these n/s points.
6. We have at this point $2k$ medians.

Theorem 7 *[42] The above algorithm gives an $O(1)$ approximation with $2k$ medians with constant probability.*

The above algorithm provides a constant-factor approximation for the k–Median problem (using $2k$ medians) with constant probability. Repeat the above experiment $O(\log n)$ times for high probability. We will not run this algorithm by itself, but as a substep in our algorithm. The algorithm requires $\tilde{O}(nk)$ time and space. Using this algorithm with the local search tradeoff results in [14] reduces the space requirement to

$O(\sqrt{nk})$. Alternate sampling-based results exist for the k–Median measure that do extend to the weighted case [63] but the sample sizes depend on the diameter of the space.

Extension to the Weighted Case: We need this sampling-based algorithm to work on weighted input. It is necessary to draw a random sample based on the weights of the points; otherwise the medians with respect to the sample do not convey much information.

The simple idea of sampling points with respect to their weights does not help. In the first step we may only eliminate k points.

We suggest the following "scaling". We can round the weights to the nearest power of $(1+\epsilon)$ for $\epsilon > 0$. In each group we can ignore the weight and lose a $(1+\epsilon)$ factor. Since we have an $\tilde{O}(nk)$ algorithm, summing over all groups, the running time is still $\tilde{O}(nk)$. The correct way to implement this is to compute the exponent values of the weights and use only those groups which exist, otherwise the running time will depend on the largest weight.

3.3.1 The Full Algorithm

We will use this sampling-based scheme to develop a one-pass and $O(nk)$-time algorithm that requires only $O(n^\epsilon)$ space.

- Input the first $O(M/k)$ points, and use the randomized algorithm above to cluster this to $2k$ intermediate median points.
- Use a local search algorithm to cluster $O(M)$ intermediate medians of level i to $2k$ medians of level $i+1$.
- Use the primal dual algorithm [47] to cluster the final $O(M)$ medians to k medians.

Notice that the algorithm remains one pass, since the $O(\log n)$ iterations of the randomized subalgorithm just add to the running time. Thus, over the first phase, the contribution to the running time is $\tilde{O}(nk)$. Over the next level, we have $\frac{nk}{M}$ points, and if we cluster $O(M)$ of these at a time taking $O(M^2)$ time, the total time for the second phase is $O(nk)$ again. The contribution from the rest of the levels decreases geometrically, so the running time is $\tilde{O}(nk)$. As shown in the previous sections, the number of levels in this algorithm is $O(\log_{\frac{M}{k}} n)$, and so we have a constant-factor approximation for $k \ll M = \Theta(n^\epsilon)$ for some small ϵ. Thus,

Theorem 8 *The k–Median problem has a constant-factor approximation algorithm running in time $O(nk \log n)$, in one pass over the data set, using n^ϵ memory, for small k.*

3.4 Lower Bounds

In this section we explore whether our algorithms could be speeded up further and whether randomization is needed. For the former, note that we have a clustering algorithm that requires time $\tilde{O}(nk)$ and a natural question is could we have done better? We'll show that we couldn't have done much better since a deterministic lower bound for k–Median is $\Omega(nk)$. Thus, modulo randomization, our time bounds pretty much match the lower bound. For the latter, we show one way to get rid of randomization that yields a single pass, small memory k–Median algorithm that is a poly-$\log n$ approximation. Thus we do also have a deterministic algorithm, but with more loss of clustering quality.

We now show that any constant-factor deterministic approximation algorithm requires $\Omega(nk)$ time. We measure the running time by the number of times the algorithm queries the distance function.

We consider a restricted family of sets of points where there exists a k-clustering with the property that the distance between any pair of points in the same cluster is 0 and the distance between any pair of points in different clusters is 1. Since the optimum k-clustering has value 0 (where the *value* is the distance from points to nearest centers), any algorithm that doesn't discover the optimum k-clustering does not find a constant-factor approximation.

Note that the above problem is equivalent to the following Graph k-Partition Problem: Given a graph G which is a complete k-partite graph for some k, find the k-partition of the vertices of G into independent sets. The equivalence can be easily realized as follows: The set of points $\{s_1, \ldots, s_n\}$ to be clustered naturally translates to the set of vertices $\{v_1, \ldots, v_n\}$ and there is an edge between v_i, v_j iff dist$(s_i, s_j) > 0$. Observe that a constant-factor k-clustering can be computed with t queries to the distance function iff a graph k-partition can be computed with t queries to the adjacency matrix of G.

Kavraki, Latombe, Motwani, and Raghavan [51] show that any deterministic algorithm that finds a Graph k-Partition requires $\Omega(nk)$ queries to the adjacency matrix of G. This result establishes a deterministic lower bound for k–Median.

Theorem 9 *A deterministic k–Median algorithm must make $\Omega(nk)$ queries to the distance function to achieve a constant-factor approximation.*

Note that the above theorem only applies to the deterministic case. A lower bound that applies to randomized algorithms was later proven in [60, 73]. These proofs use Yao's MinMax principle and construct a distribution over which the best deterministic algorithm takes $\Omega(nk)$ time. The following is proved in [60].

Theorem 10 *Any k–Median algorithm must make $\Omega(nk)$ queries to the distance function to achieve a constant factor approximation.*

3.5 The Framework in Retrospect

An important aspect of the previous sections is the framework that emerges from the sequence of algorithms. First choose a linear time algorithm that performs well on static data. Repeatedly compose this favored algorithm in layers – each subsequent layer inputs the (weighted) cluster centers from the previous layer, and outputs $O(k)$ clusters. The final layer ensures that only k clusters remain.

In essence we prove here theoretically that an algorithm that performs well on static chunks of data, can be made to operate on a stream, preserving reasonable performance guarantees. We chose algorithms with proven bounds, to help us quantify the notion of preserving performance guarantees; but the underlying intuition carries over to any good linear time clustering algorithm in suitable domains.

4 The Issue of k: Facility Location or k-Median

In this section we will consider an interesting issue in clustering: the parameter k, the number of clusters. The parameter is required to define a suitable optimization objective and frequently used to denote an upper bound on the number of possible clusters an user wishes to consider.

However the parameter k is a target, and *need not be held fixed in the intermediate stages of the algorithm.* In fact we will use this flexibility to reduce the quadratic running time of the local search algorithm in the second step of the algorithm in Section 3.3.1. The local search algorithm that we devise will settle on a number of centers larger than k and then will reduce the number of centers to exactly k if required.

The notion of ability to relax the parameter k in the intermediate steps of the algorithm provide an interesting contrast to k-Means which (as defined commonly) does not relax the number of clusters. And in fact we believe that some of the issues in stability of the k-Means algorithms are related to this fact and we discuss them further in our experimental evaluation in Section 5.

4.1 A LSEARCH Algorithm

In this section we will speed up local search by relaxing the number of clusters in the intermediate steps and achieve exactly k clusters in the final step. We must use at least k medians in the intermediate steps, since the best solution with $k-1$ medians can be much more expensive than the best k median solution, and we are interested in guarantees. At the same time we cannot use too many since we want to save space. Since we have flexibility in k, we can develop a new local search based algorithm.

Definition 2 (The Facility Location Problem) *We are given a set N of n data points in a metric space, a distance function $d : N \times N \to \Re^+$, and a parameter z. For any choice of $C = \{c_1, c_2, \dots, c_k\} \subset N$ of k*

cluster centers, define a partition of N into k clusters $N_1, N_2, \ldots, N_k$ such that N_i contains all points in N that are closer to c_i than to any other center. The goal is to select a value of k and a set of centers C so as to minimize the facility clustering (FC) cost function:

$$FC(N, C) = z|C| + \sum_{i=1}^{k} \sum_{x \in N_i} d(x, c_i).$$

We start with an initial solution and then refine it by making local improvements. Throughout this section, we will refer to the "cost" of a set of medians to mean the FC cost from the facility location definition. The significant difference from k-means and other such iterative heuristic would be that instead of trying to preserve a good clustering with exactly k medians or representation points, we would use a few extra points (twice as many) but would be able to provide provable guarantees. As we mentioned before Lagrangian Relaxation techniques provide a powerful tool for combinatorial optimization problems and the facility location minimization is a Lagrangian relaxation of the k-median problem.

4.1.1 A Throwback on Facility Location

We begin by describing a simple algorithm of Charikar and Guha [14], referred to as CG, for solving the facility location problem on a set N of n points in a metric space with metric (relaxed metric) $d(\cdot, \cdot)$, when the facility cost is z.

Assume that we have a feasible solution to facility location on N given $d(\cdot, \cdot)$ and z. That is, we have some set $I \subseteq N$ of currently open facilities, and an assignment for each point in N to some (not necessarily the closest) open facility. For every $x \in N$ we define *gain* of x to be the cost we would save (or further expend) if we were to open a facility at x (if one does not already exist), and then perform all possible advantageous reassignments and facility closings, subject to the following two constraints: first, that points cannot be reassigned except to x, and second, that a facility can be closed only if its members are first reassigned to x. The *gain* of x can be easily computed in $O(|NI|)$ time.

Algorithm CG(data set N, facility cost z)

1. Obtain an initial solution (I, f) ($I \subseteq N$ of facilities, f an assignment function) that gives a n-approximation to facility location on N with facility cost z.

2. Repeat $\Omega(\log n)$ times:

   - Randomly order N.

- For each x in this random order: calculate $gain(x)$, and if $gain(x) > 0$, add a facility there and perform the allowed reassignments and closures.

Let C^* denote the cost of the optimal k-Median solution. Setting $z = C^*/(\gamma k)$ gives a $(1 + \frac{2}{\gamma})$ approximation using $(1+\gamma)k$ medians [14]. This of course is infeasible if we do not know C^*, what [14] suggests is to try all possible t such that

$$\min\{d(x,y) \mid d(x,y) > 0\} \leq (1+\epsilon)^t \leq \sum_y d(x,y) \text{ for some x}$$

The value on the left denotes the smallest non-zero solution possible and hence a lower bound for C^*. The value on the right indicates one feasible solution for $k = 1$, when x (some arbitrary point) is chosen as a median; thus serves as an upper bound for C^*. The arguments carry over for a relaxed metric as well. See [14] for details.

4.2 Our New Algorithm

The above algorithm does not directly solve k–Median but could be used as a subroutine to a k–Median algorithm, as follows. We first set an initial range for the facility cost z (between 0 and an easy-to-calculate upper bound); we then perform a binary search within this range to find a value of z that gives us the desired number k of facilities; for each value of z that we try, we call Algorithm CG to get a solution.

Binary Search Two questions spring to mind: first, will such a binary search technique work, and second, will such an algorithm, be sufficiently fast and accurate ?

Notice to get a 3-approximation using $2k$ medians we just have to find the optimum C^* value, and set $z = C^*/k$. Since we do not know C^* we can try to find it by binary search; which is the same as binary search of z. This search for the parameter z or C^* calls the CG algorithm repeatedly for a setting and checks if the solution contains exactly k medians or the range of values for z is very small. But the CG algorithm's running time $\Theta(n^2 \log n)$ is expensive for large data streams. Therefore, we describe a new local search algorithm that relies on the correctness of the above algorithm but avoids the quadratic running time by taking advantage of the structure of local search.

4.3 Finding a Good Initial Solution

On each iteration of step 2 above, we expect the total solution cost to decrease by some constant fraction of the way to the best achievable cost [14]; if our initial solution is a constant-factor approximation rather than an n-approximation (as used by Charikar and Guha), we can (provably) reduce our number of iterations from $\Theta(\log n)$ to $\Theta(1)$. We will therefore use the following algorithm for our initial solution:

Algorithm InitialSolution(data set N, facility cost z)

```
1. Reorder data points randomly
2. Create a cluster center at the first point
3. For every point after the first,
    • Let $d$ = distance from the current point to the nearest existing cluster center
    • With probability $d/z$ create a new cluster center at the current point; otherwise add the current
      point to the best current cluster
```

This algorithm runs in time proportional to n times the number of facilities it opens and obtains an expected 8-approximation to optimum [61].

4.4 Sampling to Obtain Feasible Centers

Next we present a theorem that will motivate a new way of looking at local search. It is stated and proved in terms of the actual k–Median problem, but holds, with slightly different constants, for minimization for other measures such as SSQ, the sum of squares of the distances to the median.

Assume the points $c_1, \ldots, c_k$ constitute an optimal solution to the k–Median problem for the data set N, that C_i is the set of points in N assigned to c_i, and that r_i is the average distance from a point in C_i to c_i for $1 \leq i \leq k$. Assume also that, for $1 \leq i \leq k$, $|C_i|/|N| \geq p$. Let $0 < \delta < 1$ be a constant and let S be a set of $m = \frac{8}{p} \log \frac{2k}{\delta}$ points drawn independently and uniformly at random from N.

Theorem 11 *There is a constant α such that with high probability, the optimum k–Median solution in which medians are constrained to be from S has cost at most α times the cost of the optimum unconstrained k–Median solution (where medians can be arbitrary points in N).*

Proof: If $|S| = m = \frac{8}{p} \log \frac{2k}{\delta}$ then $\forall i$, $Pr\{|S \cap C_i| < mp/2\} < \frac{\delta}{2k}$, by Chernoff bounds. Then $Pr\{\exists i : |S \cap C_i| < mp/2\} < \frac{\delta}{2}$. Given that $|S \cap C_i| \geq mp/2$, the probability that no point from S is within distance $2r_i$ of the optimum center c_i is at most $\frac{1}{2}^{mp/2} \leq \frac{1}{2}^{\log \frac{2k}{\delta}} = \frac{\delta}{2k}$ by Markov's Inequality. So $Pr\{\exists c_j : \forall x \in S,\ d(x, c_i) > 2r_i\} \leq \frac{\delta}{2}$. If, for each cluster C_i, our sample contains a point x_i within $2r_i$ of c_i, the cost of the median set $\{x_1, \ldots, x_k\}$ is no more than 3 times the cost of the optimal k–Median solution (by the triangle inequality, each assignment distance would at most triple). □

In some sense we are assuming that the smallest cluster is not too small. If, for example, the smallest cluster contains just one point, i.e., $p = 1/|N|$, then clearly no point can be overlooked as a feasible center. We view such a small subset of points as outliers, not clusters. Hence we assume that outliers have been removed. Therefore if instead of evaluating $gain()$ for every point x we only evaluate it on a randomly chosen set of $\Theta(\frac{1}{p}\log k)$ points, we are still likely to choose good medians but will finish our computation sooner.

4.5 The Complete Algorithm

We now present the full algorithm with a modification that speeds up the binary search. The observation is if our cost changes very little from one iteration to the next and we are far from k centers, then we have gotten the value of z incorrect.

We first give a Facility Location subroutine that our k–Median algorithm will call; it will take a parameter $\epsilon \in \Re$ that controls convergence. The other parameters will be the data set N of size n, the metric or relaxed metric $d(\cdot,\cdot)$, the facility cost z, and an initial solution (I, a) where $I \subseteq N$ is a set of facilities and $a : N \to I$ is an assignment function.

Algorithm FL(N, $d(\cdot,\cdot)$, z, ϵ, (I,a))

```
1. Begin with $(I, a)$ as the current solution
2. Let $C$ be the cost of the current solution on $N$. Consider the feasible centers in random order, and for
   each feasible center $y$, if $gain(y) > 0$, perform all advantageous closures and reassignments (as per
   $gain$ description), to obtain a new solution $(I', a')$ [$a'$ should assign each point to its closest center in
   $I'$]
3. Let $C'$ be the cost of the new solution; if $C' \leq (1-\epsilon)C$, return to step 2
```

Now we will give our k–Median algorithm for a data set N with distance function d.

Algorithm LSEARCH $(N, d(\cdot,\cdot), k, \epsilon, \epsilon', \epsilon'')$

```
1. $z_{min} \leftarrow 0$
2. $z_{max} \leftarrow \sum_{x \in N} d(x, x_0)$ (for $x_0$ an arbitrary point in $N$)
3. $z \leftarrow (z_{max} + z_{min})/2$
4. $(I, a) \leftarrow$ InitialSolution($N$, $z$).
5. Randomly pick $\Theta(\frac{1}{p}\log k)$ points as feasible medians
```

6. While #medians $\neq k$ and $z_{min} < (1-\epsilon'')z_{max}$:
   - Let (F, g) be the current solution
   - Run $FL(N, d, \epsilon, (F, g))$ to obtain a new solution (F', g')
   - If $k \leq |F'| \leq 2k$, then exit loop
   - If $|F'| > 2k$ then $\{z_{min} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2\}$; else if $|F'| < k$ then $\{z_{max} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2\}$
7. Return our solution (F', g').[6]

The initial value of z_{max} is chosen as a trivial upper bound (sum of assignment costs) on the value of z we will be trying to find. The running time of LSEARCH is $O(nm + nk \log k)$ where m is the number of facilities opened by InitialSolution. Although m depends on the data set but it is usually small, leading to a significant improvement over previous algorithms.

5 Empirical Evaluations

5.1 Empirical Evaluation of LSEARCH

We present the results of experiments comparing the performance of k–Means and LSEARCH . We conducted all experiments on a Sun Ultra 2 with two, 200MHz processors, 256 MB of RAM, and 1.23 GB of swap space,[7] running SunOS 5.6. A detailed description can be found in the conference version[67]; we present the highlights. All the experiments compare the SSQ measure, which is the sum of *squares* of the distances to the medians. This is because the algorithms we compare against use this measure. As mentioned the proof our our algorithms carry over in this setting with weaker theoretical guarantees in this relaxed triangle inequality setting[8].

Small Low-Dimensional Data Sets We generated small data sets, each containing between 1000 and 6400 points of dimension at most four, and ran LSEARCH and k–Means on each. Each of these consists of five to sixteen uniform-density, radius-one spheres of real vectors, with five percent random noise uniform over the space. We generated grid data sets, in which the spheres are centered at regular or nearly regular

[6]To simulate a continuous space, move each cluster center to the center-of-mass for its cluster.

[7]Our processes never used more than one processor or went into swap.

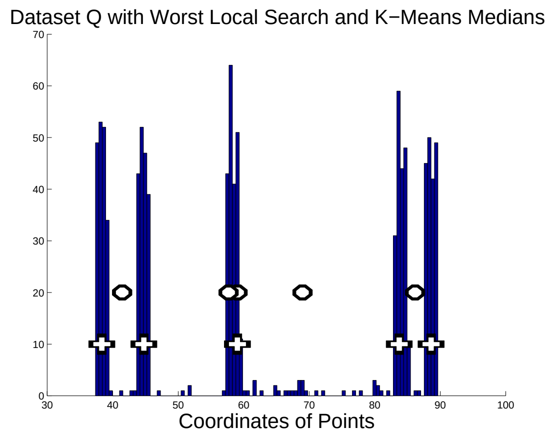
[8]The square of distances satisfies a relaxed triangle inequality. In particular, $d^2(u,v) \leq (d(u,y) + d(y,v))^2 = d(u,y)^2 + 2d(u,y)d(y,v) + d(y,v)^2 \leq 2(d(u,y)^2 + d(y,v)^2)$. The last step follows by noting that $2d(u,y)d(y,v) \leq d(u,y)^2 + d(y,v)^2$ since $2ab \leq a^2 + b^2$ and $(a-b)^2 \geq 0$ for all a, b.

intervals; shifted-center data sets, in which the spheres are centered at positions slightly shifted from those of the grid data sets; and random-center data sets. Because k–Means and LSEARCH are randomized, we ran each algorithm ten times on each data set, recording the mean and variance of SSQ. For each of the small data sets, we stored the minimum SSQ achieved on the data set, and used this value as an upper bound on the best achievable SSQ for this data set.

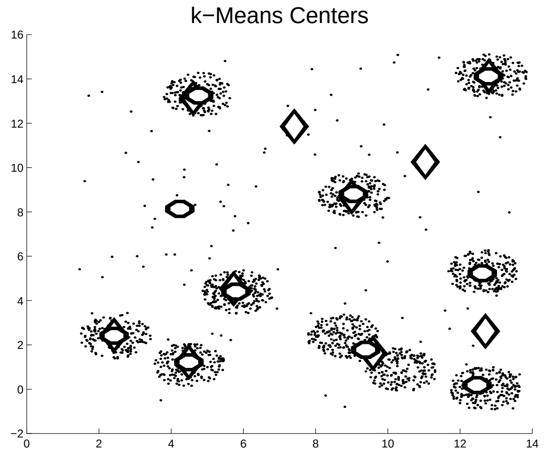
Figure 2(a) and (b) show the k–Means and LSEARCH SSQ values for the grid and shifted-center data sets, normalized by the best-known SSQ for each set. The error bars represent normalized standard deviations in SSQ. In 2(a) and (b), the data set order is such that the centers from which the clusters in the ith grid data set were generated are the same as the centers for the ith *shifted-center* set, except that each shifted center is independently shifted a small distance from the position of the corresponding grid center. Since the slight asymmetry is the only difference between the distributions of the ith grid and shifted-center data sets, the results in these two graphs suggest that this asymmetry accounts for the greater disparity between k–Means and LSEARCH performance on the shifted-center sets. The shifted-center and random-center data sets exhibit much less symmetry than the grid data sets. The gap between k–Means and LSEARCH performance seems to increase with increasing data asymmetry. Figure 2(c) shows the k–Means and LSEARCH SSQ values for the random-center sets, normalized as before by the best-known SSQs.

LSEARCH performed consistently well (i.e., with low variance in SSQ) on all the synthetic data sets. Figure 1(a) shows a one-dimensional, random-center data set, represented as a histogram; the horizontal axis gives the coordinates of the data points, and the height of each bar is the number of points in the data set that fell within the range covered by the width of the bar. Therefore, the five peaks correspond to the 5 high-density regions (clusters), and the remaining area corresponds to low-level, uniform noise. The ovals show the coordinates of the highest-cost medians found by k–Means for this data set (recall that both algorithms were run ten times each), and the "plus" signs give the locations of the highest-cost medians found by LSEARCH . Figure 1(b) shows the best medians (represented as octagons) and worst medians (diamonds) found by k–Means on a two-dimensional data set. 1(c) shows the lowest-cost medians (rectangles) and highest-cost medians ("plus" signs) found by LSEARCH on the same data set. These examples illustrate the general theme that LSEARCH consistently found good medians, even on its worst run, whereas k–Means missed some clusters, found clusters of noise, and often assigned more than one median to a cluster.

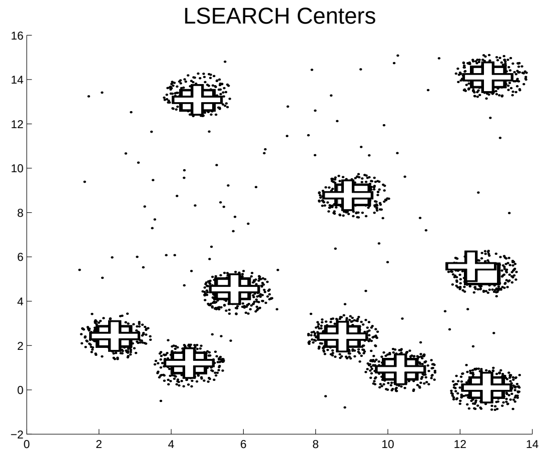
Larger, Low-Dimensional Data Set We also ran both algorithms on a data set distributed by the authors of BIRCH [76], which consists of one hundred, two-dimensional Gaussians in a ten-by-ten grid, with a thousand points each. k–Means ran, on average, for 298 s (with standard deviation 113 s.) LSEARCH



(a) Worst k–Means and LSEARCH solutions for 1-D data set Q

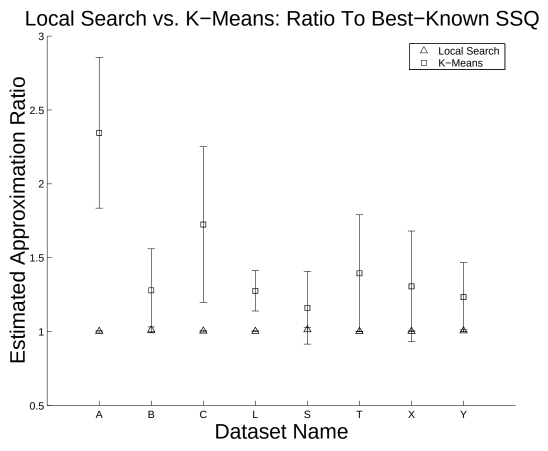


(b) Best and worst k–Means solutions for 2-D data set J

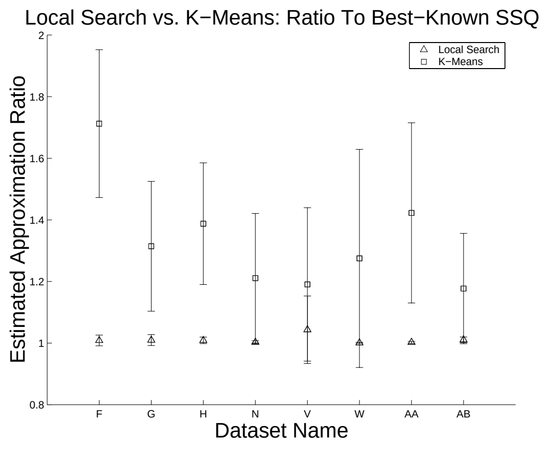


(c) Best and worst LSEARCH solutions for data set J

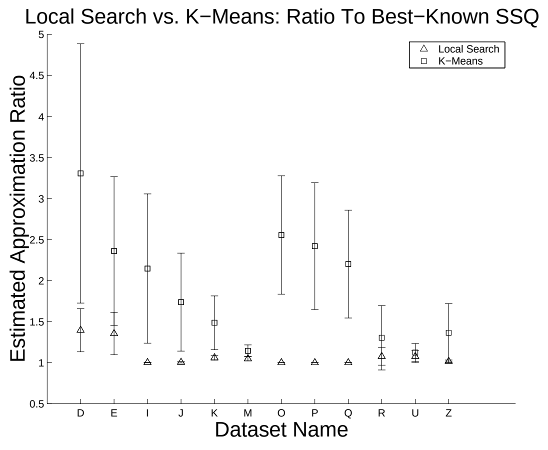
Figure 1: Examples of k–Means and LSEARCH solutions



(a) Grid data sets



(b) Shifted-center data sets



(c) Random-center data sets

Figure 2: k–Means vs. LSEARCH on small, synthetic data sets: SSQ

ran for 1887 s on average (with standard deviation of 836s). The average SSQ of k–Means solutions was 209077, with standard deviation 5711, whereas for LSEARCH the SSQ was 176136 with standard deviation 337.

Small, High-Dimensional Data Sets We ran k–Means and LSEARCH , on 10, 100-to-200-dimensional data sets. All ten have one thousand points and consist of ten uniform-density, randomly-centered, d-dimensional hypercubes with edge length two, and five percent noise. For all data sets, the answer found by k–Means has, on average, four to five times the average cost of the answer found by LSEARCH , which is always very close to the best-known SSQ. On the other hand, k–Means ran three to four times faster. See also [67].

In summary The standard deviations of the k–Means costs are typically orders of magnitude larger than those of LSEARCH , and they are even higher, relative to the best-known cost, than in the low-dimensional data set experiments. This increased unpredictability may indicate that k–Means was more sensitive than LSEARCH to dimensionality.

In terms of running time, LSEARCH is consistently slower, although its running time has very low variance. LSEARCH appears to run approximately 3 times as long as k–Means. As before, if we count only the amount of time it takes each algorithm to find a good answer, LSEARCH is competitive in running time and excels in solution quality.

These results characterize the differences between LSEARCH and k–Means. Both algorithms make decisions based on local information, but LSEARCH uses more global information as well. Because it allows itself to “trade” one or more medians for another median at a different location, it does not tend to get stuck in the local minima that plague k–Means.

5.2 Clustering Streams: Top-Down versus Bottom-Up

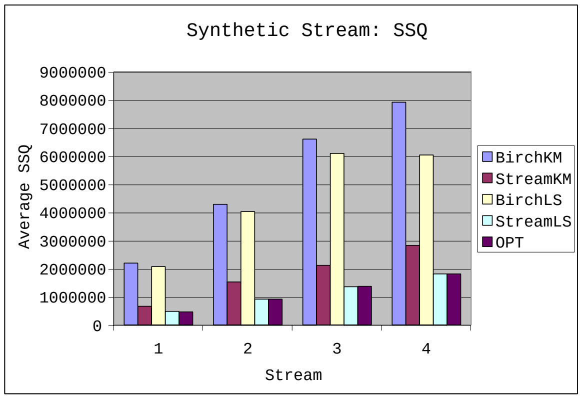
STREAM K-means Theorem 6 only guarantees that the performance of STREAM can be bounded if a constant factor approximation algorithm is run in steps 2 and 4 of STREAM. Despite the fact that k-means has no such guarantees, due to its popularity we experimented with running k-means as the clustering algorithm in Steps 2 and 4. Our experiments compare the performance of STREAM LSEARCH and STREAM K-Means with BIRCH.

BIRCH compresses a large data set into a smaller one via a CFtree (clustering feature tree). Each leaf of this tree captures sufficient statistics (namely the first and second moments) of a subset of points. Internal nodes capture sufficient statistics of the leaves below them. The algorithm for computing a CFtree tree repeatedly inserts points into the leaves, provided that the radius of the set of points associated with a leaf does not exceed a certain threshold. If the threshold is exceeded, a new leaf is created and the tree is appropriately balanced. If the tree does not fit in main memory then a new threshold is used to create a smaller tree. The BIRCH paper [76] details the decision making process.

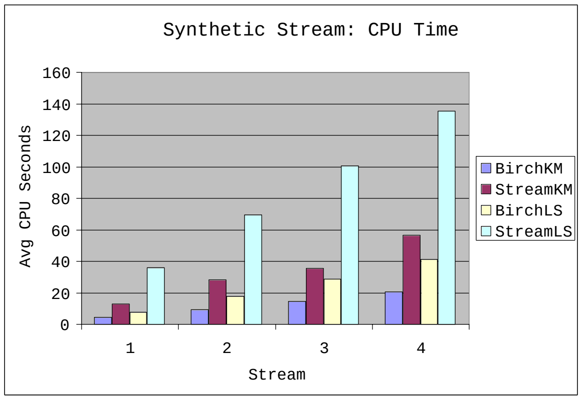
STREAM and BIRCH have a common method of attack: repeated preclustering of the data. However the preclustering of STREAM is bottom-up, where every substep is a clustering process, whereas the preclustering in BIRCH is top-down partitioning. To put the results on equal footing, we gave both algorithms the same amount of space for retaining information about the stream. Hence the results compare SSQ and running time.

Synthetic Data Stream We generated a stream approximately 16MB in size, consisting of 50,000 points in 40-dimensional Euclidean space. The stream was generated similarly to the high-dimensional synthetic data sets, except that the cluster diameters varied by a factor of 9, and the number of points by a factor of 6.33. We divided the point set into four consecutive chunks, each of size 4MB, and calculated an upper bound on the SSQ for each, as in previous experiments, by finding the SSQ for the centers used to generate the set. We ran experiments on each of the four "prefixes" induced by this segmentation into chunks: the prefix consisting of the first chunk alone, the first two chunks, the first three chunks, and the entire stream. As before, we ran LSEARCH and k–Means ten times each on every data set (or CF-tree) on which we tested them. Since BIRCH and HAC are deterministic, this repetition was not necessary when we generated CF-trees or ran HAC. Thus we had four choices depending on the clustering algorithm and the preclustering method. The performance of each algorithm was linear both in error and running time, as expected. In summary,

- The SSQ achieved by STREAM was 2 to 3 times lower than that achieved by the implementation with BIRCH that used the same clustering algorithm. A possible reason was that the BIRCH CF-trees usually had one "mega-center," a point of very high weight, along with a few points of very small weight; thus incorrectly summarizing the stream.
- STREAM ran 2 to 3 times slower than the corresponding implementation using BIRCH. BIRCH preclusters uses a top-down partitioning, while STREAM uses a slower, bottom-up preclustering step. The results demonstrate the effect of more accurate decisions in STREAM regarding storing the summary statistics.



(a) BIRCH vs. STREAM: SSQ



(b) BIRCH vs. STREAM: CPU Time

Figure 3: BIRCH vs. STREAM: Synthetic Data

- STREAM LSEARCH gave nearly optimal quality. This result is unsurprising given the good empirical performance of LSEARCH , discussed in the previous section. The bottom-up approach in STREAM introduces little extra error in the process, and so the combination finds a near-optimal solution.

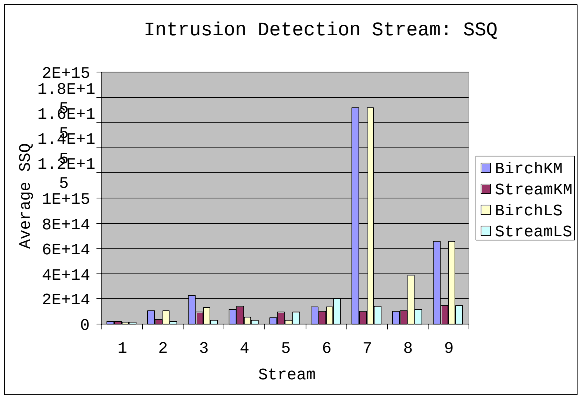
Network Intrusions Clustering, and in particular algorithms that minimize SSQ, are popular techniques for detecting intrusions [65, 58]. Since detecting intrusions the moment they happen is essential to the effort to protect a network from attack, intrusions are a particularly fitting application of streaming. Offline algorithms simply do not offer the immediacy required for successful network protection.

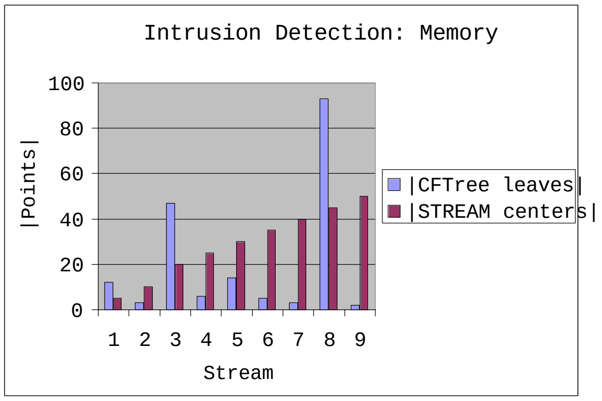
In our experiments we used the KDD-CUP'99[9] intrusion detection data set which consists of two weeks' worth of raw TCP dump data for a local area network, simulating a true Air Force environment with occasional attacks. Features collected for each connection include the duration of the connection, the number of bytes transmitted from source to destination (and vice versa), and the number of failed login attempts. All 34 continuous attributes out of the total 42 available attributes were selected for clustering. One outlier point was removed. The data set was treated as a stream of nine 16-MByte-sized chunks. The data was clustered into five clusters since each point represented one of four types of possible attacks, or non-attack, normal behavior. The four attacks included denial of service, unauthorized access from a remote machine (e.g., password guessing), unauthorized access to root, and probing (e.g., port scanning).

The leftmost chart in Figure 4 compares the SSQ of BIRCH-LS with that of STREAMLS; the middle chart makes the same comparison for BIRCH k–Means and STREAM k–Means. BIRCH's performance on the 7th an 9th chunks can be explained by the number of leaves in BIRCH's CFTree, which appears in the third chart.

Even though STREAM and BIRCH are given the same amount of memory, BIRCH does not fully take advantage of it. BIRCH's CFTree has 3 and 2 leaves respectively even though it was allowed 40 and 50 leaves, respectively. We believe that the source of the problem lies in BIRCH's global decision to increase the radius of points allowed in a leaf when the CFTree size exceeds constraints. For many data sets BIRCH's decision to increase the radius is probably a good one - it certainly reduces the size of the tree. However, this global decision can fuse points from separate clusters into the same CF leaf. Running any clustering algorithm on the fused leaves will yield poor clustering quality; here, the effects are dramatic. In terms of cumulative average running time, shown in Figure 4, BIRCH is faster. STREAM LS varies in its running time due to the creation of a weighted data set (Step 1).

[9]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html





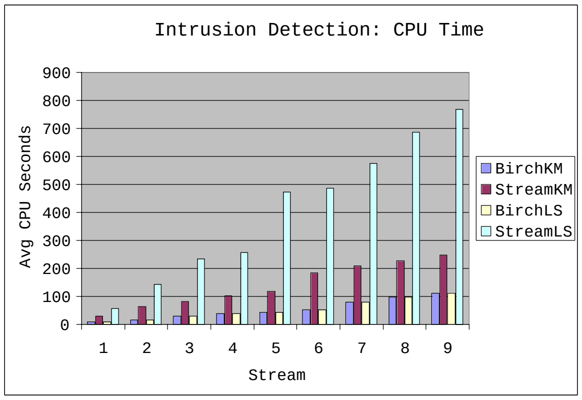


Figure 4: Network Intrusion Data :BIRCH vs. STREAM

Conclusion Overall, the results point to a trade-off between cluster quality and running time. In applications where speed is of the essence, e.g., clustering web search results, BIRCH appears to do a reasonable quick-and-dirty job. In applications like intrusion detection or target marketing where mistakes can be costly our STREAM algorithm exhibits superior SSQ performance.

Acknowledgments We are grateful to Moses Charikar, Umesh Dayal, Aris Gionis, Meichun Hsu, Piotr Indyk, Andy Kacsmar, Dan Oblinger, and Bin Zhang for their support of this work.

References

[1] D. Achlioptas and F. McSherry. Fast computation of low-rank approximations. *Proc. STOC*, pages 611–618, 2001.

[2] P. K. Agarwal and C. Procopiuc. Approximation algorithms for projective clustering. *Proc. SODA*, pages 538–547, 2000.

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

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Proc. STOC*, pages 20–29, 1996.

[5] M. Ankerst, M. Breunig, H. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *Proc. SIGMOD*, 1999.

[6] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for euclidean k -medians and related problems. In *Proc. STOC*, pages 106–113, 1998.

[7] V. Arya, N. Garg, R. Khandekar, K. Munagala, and V. Pandit. Local search heuristic for k-median and facility location problems. In *Proc. STOC*, pages 21–29, 2001.

[8] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. *Proc. SODA*, 2002.

[9] Y. Bartal, M. Charikar, and D. Raz. Approximating min-sum k-clustering in metric spaces. *Proc. STOC*, 2001.

[10] A. Borodin, R. Ostrovsky, and Y. Rabani. Subquadratic approximation algorithms for clustering problems in high dimensional spaces. *Proc. STOC*, 1999.

[11] P.S. Bradley, U.M. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. KDD*, pages 9–15, 1998.

[12] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *Proc.PODS*, pages 268–279, 2000.

[13] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *Proc. STOC*, pages 626–635, 1997.

[14] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. FOCS*, pages 378–388, 1999.

[15] M. Charikar, S. Guha, É. Tardos, and D. B. Shmoys. A constant factor approximation algorithm for the k-median problem. *Proc. STOC*, 1999.

[16] F. Chudak. Improved approximation algorithms for uncapacitated facility location. *Proc. IPCO*, LNCS 1412:180–194, 1998.

[17] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *Proc. SODA*, 2002.

[18] P. Drineas, R. Kannan, A. Frieze, and V. Vinay. Clustering in large graphs and matrices. *Proc. SODA*, 1999.

[19] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases. In *Proc. KDD,* pages 226–231, 1996.

[20] F. Farnstrom, J. Lewis, and C. Elkan. True scalability for clustering algorithms. In *SIGKDD Explorations*, 2000.

[21] T. Feder and D. H. Greene. Optimal algorithms for appropriate clustering. *Proc. STOC*, pages 434–444, 1988.

[22] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. *Proc. FOCS*, 1999.

[23] P. Flajolet and G. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 31:182–209, 1985.

[24] A. Frieze, R. Kannan, and S. Vempala. Fast monte-carlo algorithms for finding low-rank approximations. *Proc. FOCS*, 1998.

[25] V. Ganti, J. Gehrke, and R. Ramakrishnan. DEMON: Mining and monitoring evolving data. *Knowledge and Data Engineering*, 13(1):50–63, 2001.

[26] P. Gibbons and Y. Matias. Synopsis data structures for massive data sets. *Proc. SODA*, pages S909–S910, 1999.

[27] A. Gilbert, S. Guha, P. Indyk, Y. Kotadis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintanance. *Proc. STOC*, 2002.

[28] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. *Proc. VLDB*, 2002.

[29] A. C. Gilbert, S. Guha, P. Indyk, S. Muthukrishnan, and M. J. Strauss. Near-optimal sparse fourier representations via sampling. *Proc. STOC*, 2002.

[30] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. *Proc. SIGMOD*, 2001.

[31] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Proc. SODA*, pages 649–657, 1998.

[32] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. *Proceedings of ICDE*, 2002.

[33] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. STOC*, pages 471–475, 2001.

[34] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. FOCS*, pages 359–366, 2000.

[35] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *Proc. SIGMOD*, pages 73–84, 1998.

[36] P. J. Haas, J. F. Naughton, S. Seahadri, and L. Stokes. Sampling-based esitimation of the number of distinct values of an attribute. In *Proc. VLDB*, pages 311–322, 1995.

[37] J. Han and M. Kimber, editors. *Data Mining: Concepts and Techniques*. Morgan Kaufman, 200.

[38] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on Data Streams. *Digital Equipment Corporation, TR-1998-011*, August 1998.

[39] A. Hinneburg and D. Keim. An efficient approach to clustering large multimedia databases with noise. In *KDD*, 1998.

[40] A. Hinneburg and D. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *Proc. VLDB*, 1999.

[41] D. Hochbaum and D. B. Shmoys. A best possible heuristic for the k-center problem. *Math of Operations Research*, 10(2):180–184, 1985.

[42] P. Indyk. Sublinear time algorithms for metric space problems. In *Proc. STOC*, 1999.

[43] P. Indyk. A sublinear time approximation scheme for clustering in metric spaces. *Proc. FOCS*, pages 154–159, 1999.

[44] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. *Proc. FOCS*, 2000.

[45] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[46] K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problem. *Proc. STOC*, 2002.

[47] K. Jain and V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. *Proc. FOCS*, 1999.

[48] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Proc. FOCS*, pages 367–377, 2000.

[49] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems, part ii: p-media ns. *SIAM Journal on Applied Mathematics*, pages 539–560, 1979.

[50] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data. An Introduction to Cluster Analysis*. Wiley, New York, 1990.

[51] L. E. Kavraki, J. C. Latombe, R. Motwani, and P. Raghavan. Randomized query processing in robot path planning. *Journal of Computer and System Sciences*, 57:50–60, 1998.

[52] S. Kolliopoulos and S. Rao. A nearly linear-time approximation scheme for the euclidean k-median problem. *Proc. 7th ESA*, pages 378–389, 1999.

[53] J. H. Lin and J. S. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44:245–249, 1992.

[54] J. H. Lin and J. S. Vitter. ϵ-approximations with minimum packing constraint violations. *Proc. STOC*, 1992.

[55] O. L. Managasarian. Mathematical programming in data mining. *Data Mining and Knowledge Discovery*, 1997.

[56] G. S. Manku, S. Rajagopalan, and B. Lindsay. Approximate medians and other quantiles in one pass with limited memory. *Proc. SIGMOD*, 1998.

[57] G. S. Manku, S. Rajagopalan, and B. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. *Proc. SIGMOD*, 1999.

[58] D. Marchette. A statistical method for profiling network traffic. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999.

[59] R. Mettu and C. G. Plaxton. The onlike median problem. *Proc. FOCS*, 2000.

[60] R. Mettu and C. G. Plaxton. Optimal time bounds for approximate clustering. *Proc. UAI*, 2002.

[61] A. Meyerson. Online facility location. *Proc. FOCS*, 2001.

[62] P. Mirchandani and R. Francis, editors. *Discrete Location Theory*. John Wiley and Sons, Inc., New York, 1990.

[63] N. Mishra, D. Oblinger, and L. Pitt. Sublinear time approximate clustering. *Proc. SODA*, 2001.

[64] J. Munro and M. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, pages 315–323, 1980.

[65] K. Nauta and F. Lieble. Offline network intrusion detection: Looking for footprints. In *SAS White Paper*, 2000.

[66] R.T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proc. VLDB*, pages 144–155, 1994.

[67] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. *Proceedings of ICDE*, 2002.

[68] R. Ostrovsky and Y. Rabani. Polynomial time approximation schemes for geometric k-clustering. *Proc. FOCS*, 2000.

[69] C. Procopiuc, M. Jones, P. K. Agarwal, and T. M. Murali. A monte carlo algorithm for fast projective clustering. *Proc. SIGMOD*, 2002.

[70] G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A multi-resolution clustering approach for very large spatial databases. In *Proc. VLDB*, pages 428–439, 1998.

[71] D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. *Proc. STOC*, pages 265–274, 1997.

[72] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. *Proc. SIGMOD*, 2002.

[73] M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. *ICALP*, pages 249–260, 2001.

[74] V. Vazirani. *Approximation Algorithms*. Springer Verlag, 2001.

[75] W. Wang, J. Yang, and R. Muntz. Sting: A statistical information grid approach to spatial data mining. In *Proc. VLDB*, 1997.

[76] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proc. SIGMOD*, pages 103–114, 1996.