StreamKM++: A Clustering Algorithm for Data Streams

MARCEL R. ACKERMANN, MARCUS MÄRTENS, CHRISTOPH RAUPACH, and KAMIL SWIERKOT, University of Paderborn
CHRISTIANE LAMMERSEN, Simon Fraser University
CHRISTIAN SOHLER, TU Dortmund

We develop a new *k*-means clustering algorithm for data streams of points from a Euclidean space. We call this algorithm StreamKM++. Our algorithm computes a small weighted sample of the data stream and solves the problem on the sample using the *k*-means++ algorithm of Arthur and Vassilvitskii (SODA '07). To compute the small sample, we propose two new techniques. First, we use an adaptive, nonuniform sampling approach similar to the *k*-means++ seeding procedure to obtain small coresets from the data stream. This construction is rather easy to implement and, unlike other coreset constructions, its running time has only a small dependency on the dimensionality of the data. Second, we propose a new data structure, which we call coreset tree. The use of these coreset trees significantly speeds up the time necessary for the adaptive, nonuniform sampling during our coreset construction.

We compare our algorithm experimentally with two well-known streaming implementations: BIRCH [Zhang et al. 1997] and StreamLS [Guha et al. 2003]. In terms of quality (sum of squared errors), our algorithm is comparable with StreamLS and significantly better than BIRCH (up to a factor of 2). Besides, BIRCH requires significant effort to tune its parameters. In terms of running time, our algorithm is slower than BIRCH. Comparing the running time with StreamLS, it turns out that our algorithm scales much better with increasing number of centers. We conclude that, if the first priority is the quality of the clustering, then our algorithm provides a good alternative to BIRCH and StreamLS, in particular, if the number of cluster centers is large. We also give a theoretical justification of our approach by proving that our sample set is a small coreset in low-dimensional spaces.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*; G.3 [**Probability and Statistics**]: Computing Classification—*Probabilistic algorithms (including Monte Carlo)*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering*; I.5.3 [**Pattern Recognition**]: Clustering—*Algorithms, Similarity measures*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Clustering, coresets, data stream, *k*-means

ACM Reference Format:
Ackermann, M. R., Martens, M., Raupach, C., Swierkot, K., Lammersen, C., and Sohler, C. 2012. StreamKM++: A clustering algorithm for data streams. ACM J. Exp. Algor. 17, 2, Article 2.4 (May 2012), 30 pages.
DOI = 10.1145/2133803.2184450 http://doi.acm.org/10.1145/2133803.2184450

This research was partially supported by DFG, Grants Bl 314/6-1 and So 514/1-2.
Authors' addresses: M. R. Ackermann, M. Märtens, C. Raupach, and K. Swierkot, Department of Computer Science, University of Paderborn, 33095 Paderborn, Germany; C. Lammersen, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, Canada; C. Sohler, Department of Computer Science, TU Dortmund, 44221 Dortmund, Germany.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1084-6654/2012/05-ART2.4 $10.00
DOI 10.1145/2133803.2184450 http://doi.acm.org/10.1145/2133803.2184450

1. INTRODUCTION

Clustering is the problem to partition a given set of objects into subsets called *clusters*, such that objects in the same cluster are similar and objects in different clusters are dissimilar. The goal of clustering is to simplify data by replacing a cluster by one or a few representatives, classify objects into groups of similar objects, or find patterns in the dataset. Often, the datasets that are to be clustered, are very large, so clustering algorithms for very large datasets are basic tools in many different areas including data mining, database systems, data compression, and machine learning. These very large datasets often occur in the form of data streams or are stored on hard disks, where a streaming access is orders of magnitude faster than random access. Therefore, one needs streaming algorithms for clustering very large datasets.

In case of Euclidean k-means clustering, which is considered in this article, the given dataset is a set of points from a Euclidean space, and the goal is to find k cluster centers such that the sum of the squared distances of the points to their nearest cluster center is minimized. It is known that Euclidean k-means clustering is an NP-hard optimization problem [Aloise et al. 2009; Dasgupta 2008]. To deal with this fact, several heuristics have been developed.

One of the most widely used heuristics is Lloyd's algorithm (sometimes also called the *k-means algorithm*) [Lloyd 1982; Forgy 1965; MacQueen 1967]. This algorithm chooses k points from the input as starting centers and then repeatedly applies some local optimizations steps to the current solution, until no more improvement is possible. Concerning the complexity, it has been proven that the worst-case running time of Lloyd's algorithm is exponential in the number of input points even in the plane [Vattani 2009]. However, Arthur et al. [2009] showed that the running time is polynomially bounded in the model of smoothed analysis. Concerning the quality of the computed solutions, it is known that Lloyd's algorithm converges to a local optimum [Selim and Ismail 1984], and no approximation guarantee can be given [Kanungo et al. 2004]. This negative result can be averted in case the input points satisfy some separation condition. In this case, Kumar and Kannan [2010] were able to show that Lloyd's algorithm converges to the true cluster centers provided the starting centers are close enough to the corresponding true centers.

Recently, Arthur and Vassilvitskii [2007] developed the k-MEANS++ algorithm, which is a seeding procedure for Lloyd's k-means algorithm that guarantees a solution with certain quality (without imposing any side condition on the input points) and gives good experimental results.

However, the k-MEANS++ algorithm (as well as Lloyd's algorithm) needs random access to the input data and is not suited for data streams. In this article, we develop a new clustering algorithm for data streams that is based on the idea of the k-MEANS++ seeding procedure.

1.1. Related Work

Clustering data streams is a well-studied problem in both theory and practice. One of the earliest and best known practical clustering algorithms for data streams is BIRCH [Zhang et al. 1996]. BIRCH is a heuristic that computes a preclustering of the data into so-called clustering features and then clusters this preclustering using an agglomerative (bottom-up) clustering algorithm. Another well-known algorithm is STREAMLS [O'Callaghan et al. 2002; Guha et al. 2003], which partitions the input stream into chunks and computes for each chunk a clustering using a local search algorithm [Guha et al. 2000]. STREAMLS is slower than BIRCH but provides a clustering with much better quality (with respect to the sum of squared errors).

In the theory community, a number of streaming algorithms for k-median and k-means clusterings have been developed [Chen 2009; Feldman et al. 2007; Frahling and Sohler 2005; Guha et al. 2000; Har-Peled and Mazumdar 2004]. Many of these algorithms apply a merge-and-reduce technique based on a decomposition technique of Bentley and Saxe [1980] to obtain a small coreset of the data stream, that is, a small weighted point set that approximates the points from the data stream with respect to the k-means clustering problem [Agarwal et al. 2004].

1.2. Our Contribution

We develop a new coreset construction for the Euclidean k-means clustering problem. In general, a coreset is a small weighted point set that approximates the original input point set with respect to a given optimization problem. Our focus is to propose a coreset construction for the Euclidean k-means clustering problem that is suitable for high-dimensional data. Existing constructions based on grid computations [Har-Peled and Mazumdar 2004; Frahling and Sohler 2005] yield coresets of a size that is exponential in the dimension. Since the k-MEANS++ seeding works well for high-dimensional data, a coreset construction based on this approach seems to be more promising.

In order to implement this coreset construction efficiently, we propose a new data structure, which we call the *coreset tree*. This is a treelike data structure that stores points in such a way that we can perform a fast adaptive sampling, which is very similar to the k-MEANS++ seeding. According to our experiments, the seed computed on the coreset tree has essentially the same properties as the original k-MEANS++ seed. The advantage of the coreset tree approach is that the running time is significantly shorter than the running time of the original k-MEANS++ seeding.

Finally, we use a standard streaming technique, called the *merge-and-reduce technique* [Bentley and Saxe 1980; Har-Peled and Mazumdar 2004], to maintain our sample in data streams. After processing the whole input stream, we apply the k-MEANS++ algorithm [Arthur and Vassilvitskii 2007] on the sample to obtain a k-means clustering. We call the resulting streaming algorithm *StreamKM++*.

We compare our algorithm with algorithms BIRCH [Zhang et al. 1996] and STREAMLS [O'Callaghan et al. 2002; Guha et al. 2003], which are both frequently used to cluster data streams. We also compare our approach with the nonstreaming version of algorithm k-MEANS++. It turns out that our algorithm is slower than BIRCH, but it computes significantly better solutions (in terms of sum of squared errors). In addition, to obtain the desired number of clusters, our algorithm does not require the trial-and-error adjustment of parameters as BIRCH does. The quality of the clustering of algorithm STREAMLS is comparable to that of our algorithm, but the running time of STREAMKM++ scales much better with the number of cluster centers. For example, on the dataset *Tower*, our algorithm computes a clustering with $k = 100$ centers in about 3% of the running time of STREAMLS. In comparison with the standard implementation of k-MEANS++, our algorithm runs much faster on larger datasets and computes solutions that are on a par with k-MEANS++. For example, on the dataset *Covertype*, our algorithm computes a clustering with $k = 50$ centers of essentially the same quality as k-MEANS++, but it needs only about 3% of the running time of k-MEANS++.

We back up our strategy with a theoretical analysis of the new coreset construction. We prove that, with high probability, sampling according to the k-MEANS++ seeding procedure gives small coresets, at least in low-dimensional spaces. It should be noted that the k-MEANS++ seeding has also been theoretically investigated in Aggarwal et al. [2009] and Ailon et al. [2009]. Aggarwal et al. [2009] used the k-MEANS++ seeding to compute a small weighted point set such that a constant-factor approximation of the k-means clustering for the original point set can be obtained by clustering the small weighted set, whereas Ailon et al. [2009] used the k-MEANS++ seeding to obtain a streaming algorithm for the k-means clustering problem that guarantees an approximation factor of $\mathcal{O}(c^\alpha \log k)$, where c is some constant, $\alpha \approx \log n / \log M$, n is the number of input points in the stream, and M is the amount of work memory available to the algorithm. However, our result differs from the results given in Aggarwal et al. [2009] and Ailon et al. [2009] and was obtained independently.

2. PRELIMINARIES

2.1. Euclidean *k*-Means Clustering

For any two points $x, y \in \mathbb{R}^d$ and any set of points $C \subset \mathbb{R}^d$, we denote the Euclidean distance between x and y by $\mathrm{D}(x, y) = \|x - y\|_2$, and we define

$$\mathrm{D}(x, C) = \min_{c \in C} \mathrm{D}(x, c).$$

Similarly, for squared Euclidean distances, we define

$$\mathrm{D}^2(x, y) = \|x - y\|_2^2 \quad \text{and} \quad \mathrm{D}^2(x, C) = \min_{c \in C} \mathrm{D}^2(x, c).$$

In the following, let $P \subset \mathbb{R}^d$ be a set of points with size $|P| = n$. The Euclidean k-means clustering problem for P is given as follows.

Definition 2.1 (*Euclidean k-Means Clustering Problem*). For a set $P \subset \mathbb{R}^d$ and $k \in \mathbb{N}$, the Euclidean k-means clustering problem is to find a set of centers $C \subset \mathbb{R}^d$ with $|C| = k$, such that

$$\mathrm{cost}(P, C) = \sum_{x \in P} \mathrm{D}^2(x, C)$$

is minimized. The value $\mathrm{cost}(P, C)$ gives the cost of the k-means clustering for P with center set C.

Analogously, for a weighted set $S \subset \mathbb{R}^d$ with weight function $w : S \to \mathbb{R}_{\geq 0}$ and $k \in \mathbb{N}$, the *weighted Euclidean k-means problem* is to find a set $C \subset \mathbb{R}^d$ with $|C| = k$, such that

$$\mathrm{cost}_w(S, C) = \sum_{y \in S} w(y) \cdot \mathrm{D}^2(y, C)$$

is minimized. The value $\mathrm{cost}_w(S, C)$ gives the cost of the weighted k-means clustering for S with center set C.

We denote the cost of an optimal Euclidean k-means clustering of P by

$$\mathrm{cost}_k^*(P) = \min_{C' \subset \mathbb{R}^d : |C'| = k} \mathrm{cost}(P, C').$$

2.2. Definition of Coresets

An important concept we use is the notion of coresets. Generally speaking, a coreset for a set P is a small (weighted) set, such that for any set of k cluster centers, the (weighted) clustering cost of the coreset is an approximation for the clustering cost of the original set P with small relative error. The advantage of such a coreset is that we can apply any fast approximation algorithm (for the weighted problem) on the usually much smaller coreset to compute an approximate solution for the original set P more efficiently. We use the following formal definition.

Definition 2.2 (*Coreset for k-Means Clustering Problem*). Let $k \in \mathbb{N}$ and ε with $0 < \varepsilon \leq 1$ be a precision parameter. A weighted multiset $S \subset \mathbb{R}^d$ with positive weight function $w : S \to \mathbb{R}_{\geq 0}$ is called (k, ε)-coreset of P for the k-means clustering problem if, for each $C \subset \mathbb{R}^d$ of size $|C| = k$, we have

$$(1 - \varepsilon) \cdot \mathrm{cost}(P, C) \leq \mathrm{cost}_w(S, C) \leq (1 + \varepsilon) \cdot \mathrm{cost}(P, C).$$

2.3. Streaming Model

Our clustering algorithm maintains a small coreset in the data stream model. In this model, the input is a sequence of data items $a_1, \ldots, a_i, \ldots, a_n$ given in some order that we cannot control. It is assumed that the number of data items is too large to fit in the main memory of a computer. Due to this fact, the type of access to the data and the amount of resources to process the data are restricted. More precisely, instead of random access to the data items, which would be very time-consuming, streaming algorithms perform one pass over the input. Such a pass is a sequential scan over the data that reads the items one by one in increasing order of the indices i. Furthermore, streaming algorithms are only allowed to use local memory that is polylogarithmic in the size of the input stream.

In Sections 3 and 4, we develop a coreset construction based on the k-MEANS++ seeding procedure without considering streaming. In Section 5, we show how our streaming algorithm uses this coreset construction as a subroutine in order to maintain a small coreset in the data stream model.

2.4. *k*-Means Clustering Algorithms

In the experiments, we compare STREAMKM++ with two frequently used clustering algorithms for processing data streams, namely with algorithm BIRCH of [Zhang et al. 1996] and with a streaming variant of the local search algorithm [O'Callaghan et al. 2002; Guha et al. 2003], which we refer to as STREAMLS. On smaller datasets, we also compare our algorithm with a classical implementation of Lloyd's k-means algorithm [Lloyd 1982], using initial seeds either uniformly at random (algorithm k-MEANS) or according to the adaptive, nonuniform seeding from Arthur and Vassilvitskii [2007] (algorithm k-MEANS++). In the following, we give a brief overview of these k-means clustering algorithms.

2.4.1. Algorithm k-MEANS. One of the most widely used clustering algorithms is Lloyd's algorithm. This algorithm is sometimes also called the *k-means algorithm* [Lloyd 1982; Forgy 1965; MacQueen 1967]. Lloyd's algorithm is based on two observations.

(1) Given a fixed set of centers, we obtain the best clustering by assigning each point to the nearest center.
(2) Given a cluster, the best center of the cluster is the center of gravity (mean) of its points [Matousek 2000].

Lloyd's algorithm applies these two local optimizations steps repeatedly to the current solution, until no more improvement is possible. See Algorithm 2.1 for a description in pseudocode.

It is known that the algorithm converges to a local optimum [Selim and Ismail 1984], and the quality of the computed solution is sensitive to the choice of starting centers. Kanungo et al. [2004] give a simple example where, for a fixed set of starting centers, Lloyd's algorithm converges to a local minimum that is arbitrarily bad compared to the optimal solution. This example can be extended to the case where the starting centers are chosen by uniform seeding, as shown in Algorithm 2.1.

2.4.2. Algorithm k-MEANS++. Arthur and Vassilvitskii [2007] developed the k-MEANS++ algorithm, which is a seeding procedure for Lloyd's k-means algorithm. This seeding procedure considers the fact that the quality of the solution of Lloyd's k-means algorithm strongly depends on the initial set of centers. In order to achieve a better arrangement, it chooses the initial set of centers adaptively and nonuniformly at random by choosing each point as the next center with probability proportional to its squared distance from the nearest center already chosen. Note that, for a given set of centers,

ALGORITHM 2.1. k-MEANS(P, k)

```
1 choose $k$ initial centers $c_1, \dots, c_k$ uniformly at random from $P$
2 **repeat**
3         partition $P$ into $k$ subsets $P_1, \dots, P_k$, such that $P_i$, $1 \leq i \leq k$,
          contains all points whose nearest center is $c_i$
4         replace the current set of centers by a new set of centers $c_1, \dots, c_k$,
          such that center $c_i$, $1 \leq i \leq k$, is the center of gravity of $P_i$
5   **until** the set of centers has not changed
```

ALGORITHM 2.2. AdaptiveSeeding(P, k)

```
1 choose an initial center $c_1$ uniformly at random from $P$
2 $C \leftarrow \{c_1\}$
3 **for** $i \leftarrow 2$ **to** $k$
4     choose the next center $c_i$ at random from $P$, where the probability
      of each $p \in P$ is given by $\mathrm{D}^2(p, C)/\mathrm{cost}(P, C)$
5     $C \leftarrow C \cup \{c_i\}$
```

the squared distance of a point from its nearest center corresponds to the current contribution of this point to the total k-means clustering cost. The k-MEANS++ seeding procedure is given by Algorithm 2.2. For simplicity of description, we say that Algorithm 2.2 chooses the set C *at random according to* D^2.

By replacing line 1 of Algorithm 2.1 with Algorithm 2.2, Arthur and Vassilvitskii developed a k-means clustering algorithm, which is known as k-MEANS++ algorithm, that works well in practice and guarantees a solution with certain quality. More precisely, they showed the following.

LEMMA 2.3 ([ARTHUR AND VASSILVITSKII 2007]). *Let $C \subseteq P$ be a set of k points chosen at random according to* D^2. *Then, we have*

$$\mathrm{E}[\mathrm{cost}(P, C)] \leq 8\,(2 + \ln k)\,\mathrm{cost}_k^*(P),$$

where $\ln(\cdot)$ *denotes the natural logarithm to the base* $e \approx 2.718$.

2.4.3. Algorithm BIRCH. One of the earliest and best-known practical clustering algorithms for data streams is BIRCH (balanced iterative reducing and clustering using hierarchies) [Zhang et al. 1996]. BIRCH is a heuristic that exploits the observation that the point space is usually not uniformly occupied. It scans the given set of input points once and computes a preclustering by summarizing dense regions of points by their so-called clustering features. Such a clustering feature consists of the number of points in the region, the center of gravity and the sum of squared distances to the origin. Thereby, the problem of clustering the original input point set is reduced to the problem of clustering the set of summaries, which is much smaller than the original point set. The preclustering is then clustered by using an agglomerative (bottom-up) clustering algorithm. Here, the algorithm uses the clustering features to calculate the intracluster distances. BIRCH successively merges the closest pair of clusters until the desired number of clusters is obtained.

To a certain extent, BIRCH uses a kind of coreset construction. However, there is no theoretical analysis of this method known. For more details, refer to Zhang et al. [1996].

2.4.4. Algorithm StreamLS. Another well-known clustering algorithm for data streams is the streaming implementation of algorithm LSearch from [O'Callaghan et al. 2002; Guha et al. 2003], which we refer to as StreamLS. This algorithm partitions the input stream into chunks and computes a k-means clustering solution using a local search algorithm for each chunk [Guha et al. 2000]. Finally, the local search algorithm is applied once more on the union of the solutions for the chunks to obtain a k-means clustering for the whole input stream.

The local search algorithm [Guha et al. 2000] takes advantage of the relationship between the k-means clustering problem and the uniform facility location problem. In the latter problem, the number of facilities (or cluster centers) is unbounded, but we have to pay some cost for each open facility. The goal is to minimize the total cost that is given by the cost for opening the facilities plus the sum of the distances of the input points to their nearest open facility. One observation is now that if the facility cost increases, then the number of facilities of an optimal solution tends to decrease. Hence, to solve the k-means problem, the algorithm from Guha et al. [2000] performs a binary search on the facility cost to find a cost that gives the desired number of cluster centers. During the binary search, each facility location problem is solved by starting with an initial solution that is obtained by a simple nonuniform sampling approach and then refining this solution by making local improvements. More details can be found in O'Callaghan et al. [2002] and Guha et al. [2000, 2003].

3. CORESET CONSTRUCTION

In the following, we will describe our coreset construction for k-means clustering without considering streaming. In Section 5, we show how this coreset construction serves as subroutine for our streaming algorithm.

Our coreset construction is based on the k-means++ seeding procedure from Arthur and Vassilvitskii [2007]. One reason for this design decision was that the k-means++ seeding works well for high-dimensional datasets, which is often required in practice. This nice property does not apply to many other clustering methods, such as the grid-based methods from Har-Peled and Mazumdar [2004] and Frahling and Sohler [2005], for instance.

Let $P \subset \mathbb{R}^d$ be a set of points with size $|P| = n$. For an arbitrary fixed integer m, our coreset construction is as follows (see also Algorithm 3.1). First, we choose a set $S = \{q_1, q_2, \ldots, q_m\}$ of size m at random according to D^2. Let Q_i denote the set of points from P that are closest to q_i (breaking ties arbitrarily). By using the weight function $w : S \to \mathbb{R}_{\geq 0}$ with $w(q_i) = |Q_i|$, we obtain the weighted set S as our coreset. In short, the difference to the k-means++ seeding procedure is that we choose a seed of size m (instead of k) and weighed each sample point in this seed by the number of input points closest to this sample point.

Note that our coreset construction is rather easy to implement, and its running time has a merely linear dependency on the dimension d. Furthermore, empirical evaluation (as given in Section 6) suggests that our construction leads to good coresets even for relatively small choices of m (i.e., say $m = 200k$). Unfortunately, we do not have a formal proof supporting this observation. However, we are able to do a first step by proving that, at least in low-dimensional spaces, our construction indeed leads to small coresets, as is stated in the following theorem.

Theorem 3.1. *Let $k \in \mathbb{N}$, let ε with $0 < \varepsilon \leq 1$ be a precision parameter, and let δ with $0 < \delta < 1$ be an error probability. Given a point set $P \subset \mathbb{R}^d$ of size $|P| = n$ and a size*

ALGORITHM 3.1. AdaptiveCoreset(P, m)

```
1  choose an initial coreset point $q_1$ uniformly at random from $P$
2  $w(q_1) \leftarrow 0$
3  $S \leftarrow \{q_1\}$
4  for $i \leftarrow 2$ to $m$
5      choose $q_i$ at random according to $\mathrm{D}^2$ from $P$
6      $w(q_i) \leftarrow 0$
7      $S \leftarrow S \cup \{q_i\}$
8  for each $p \in P$
9      let $q_i \in S$, $1 \le i \le m$, be the nearest coreset point to $p$
10     $w(q_i) \leftarrow w(q_i) + 1$
```

parameter

$$m = \left(\frac{d}{\delta\varepsilon}\right)^{\mathcal{O}(d)} \cdot k \cdot \log(n) \cdot \log^{d/2}\left(\frac{k\log(n)}{\delta\varepsilon}\right),$$

algorithm AdaptiveCoreset *computes a weighted multiset S with size m that is a $(k, 6\varepsilon)$-coreset of P with probability at least $1 - \delta$. Here, $\log(\cdot)$ denotes the binary logarithm to the base 2.*

Please note that the size bound on the number of coreset points m from Theorem 3.1 is merely a sufficient condition, and that, to the best of our knowledge, there is no reason to assume that this size bound is tight. Hence, in compliance with our experiments, the actual dependency of m on the dimension d may very well be better than is suggested by the theorem.

3.1. Proof of Theorem 3.1

Our proof is based on the following lemma. Intuitively, this lemma states that if we consider an optimal m-clustering of P, with m large enough, then the optimal m-clustering cost is merely a tiny fraction of the optimal k-clustering cost of P. Lemma 3.2 is a consequence of the fact that there exist (k, γ)-coresets of size $m = (d/\gamma)^{\mathcal{O}(d)} k \log(n)$, which has already been proven by Har-Peled and Mazumdar [2004].

Lemma 3.2. *Let $\gamma > 0$ and $m \in \mathbb{N}$. If*

$$m \ge \left(\frac{9d}{\gamma}\right)^{d/2} \cdot k \cdot \lceil \log(n) + 3 \rceil,$$

then we get

$$\mathrm{cost}^*_m(P) \le \gamma \cdot \mathrm{cost}^*_k(P).$$

Proof. Let $C^* = \{c_1, \ldots, c_k\}$ be an optimal solution to the Euclidean k-means clustering problem for P with $|P| = n$, that is, $\mathrm{cost}(P, C^*) = \mathrm{cost}^*_k(P)$. We consider an exponential grid around each center in C^*. The construction of this grid is essentially the same as the one from Har-Peled and Mazumdar [2004]. In detail, the construction is defined as follows.

Let the average cost per point of an optimal solution be denoted by

$$R = \frac{\mathrm{cost}^*_k(P)}{n}.$$

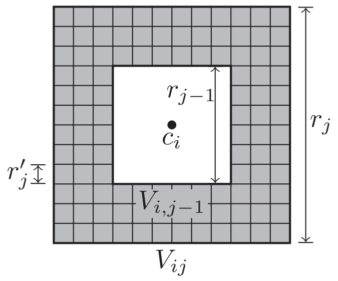

Fig. 1. Illustration of the partition of W_{ij} into small grid cells. The inner square indicates $V_{i,j-1}$, and the outer square indicates V_{ij}. The area W_{ij} is colored in gray.

Furthermore, let $\nu = \lceil \log(n) + 2 \rceil$. For each $j \in \{0, 1, \dots, \nu\}$ and each center $c_i \in C^*$, let V_{ij} be the axis-parallel square centered at c_i with side length

$$r_j = \sqrt{2^j R}.$$

Then, we recursively define $W_{i0} = V_{i0}$ and $W_{ij} = V_{ij} \setminus V_{i,j-1}$ for $j \in \{1, 2, \dots, \nu\}$. It follows that each $p \in P$ is contained within a W_{ij}, since otherwise we would have

$$\mathrm{D}^2(p, C^*) \; > \; \left(\frac{r_\nu}{2}\right)^2 \; = \; \frac{2^{\lceil \log(n)+2 \rceil} R}{4} \; \geq \; nR \; \geq \; \mathrm{cost}^*_k(P),$$

which is a contradiction.

For each i, j individually, we partition W_{ij} into small grid cells with side length

$$r'_j = \sqrt{\frac{\gamma}{9d}} \cdot r_j = \sqrt{\frac{\gamma}{9d} \cdot 2^j R}.$$

This partition is illustrated in Figure 1.

For each grid cell that contains points from P, we select a single point from within the cell as its representative. Let G be the set of all these representatives. Note that there are at most

$$\sum_{c_i \in C^*} \sum_{j=0}^{\nu} \left(\frac{r_j}{r'_j}\right)^d = k \cdot \lceil \log(n) + 3 \rceil \cdot \left(\frac{9d}{\gamma}\right)^{d/2} \leq m$$

grid cells and, hence, $|G| \leq m$.

Let g_p denote the representative of $p \in P$ in G. Then, we have

$$\mathrm{cost}^*_m(P) \; \leq \; \mathrm{cost}^*_{|G|}(P) \; \leq \; \mathrm{cost}(P, G) \; \leq \; \sum_{p \in P} \mathrm{D}^2\left(p, g_p\right). \tag{1}$$

For each point $p \in P$, the distance from its representative g_p is upper bounded by the diagonal of the grid cell that contains p. Thus, for any $p \in W_{i0}$, we have

$$\mathrm{D}^2\left(p, g_p\right) \; \leq \; \left(\sqrt{d} \cdot r'_0\right)^2 \; \leq \; \frac{\gamma R}{9}. \tag{2}$$

Now, let $j \geq 1$ and p be any point in W_{ij} for which j is the smallest index j' such that p is contained in some $W_{i'j'}$. We know that c_i is the center of $V_{i,j-1}$ and p is not contained in $V_{i,j-1}$. It follows that

$$\mathrm{D}^2(p, C^*) \; \geq \; \left(\frac{r_{j-1}}{2}\right)^2 \; \geq \; 2^{j-3} R.$$

Hence, in this case, we get

$$\mathrm{D}^2\left(p, g_p\right) \le \left(\sqrt{d} \cdot r_j'\right)^2 = \frac{\gamma}{9} \cdot 2^j R \le \frac{8\gamma}{9} \cdot \mathrm{D}^2(p, C^*). \tag{3}$$

Due to Equations (1) through (3) and the definition of R, we obtain

$$\begin{aligned}\mathrm{cost}_m^*(P) &\le \sum_{p \in P} \mathrm{D}^2\left(p, g_p\right) \\ &\le \sum_{p \in P}\left(\frac{\gamma R}{9} + \frac{8\gamma}{9} \cdot \mathrm{D}^2(p, C^*)\right) \\ &= n \cdot \frac{\gamma}{9} R + \frac{8\gamma}{9} \sum_{p \in P} \mathrm{D}^2(p, C^*) \\ &= \frac{\gamma}{9} \cdot \mathrm{cost}_k^*(P) + \frac{8\gamma}{9} \cdot \mathrm{cost}_k^*(P) \\ &= \gamma \cdot \mathrm{cost}_k^*(P).\end{aligned}$$

□

Now, let C be an arbitrary set of k centers. For $p \in P$, let q_p denote the element from S closest to p, breaking ties arbitrarily. Then, the difference between the cost of clustering P and the cost of clustering S is at most

$$\begin{aligned}\left|\mathrm{cost}(P, C) - \mathrm{cost}_w(S, C)\right| &= \left|\sum_{p \in P} \mathrm{D}^2(p, C) - \sum_{p \in P} \mathrm{D}^2(q_p, C)\right| \\ &\le \sum_{p \in P} \left|\mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C)\right|.\end{aligned}$$

We partition P into two subsets P_{near} and P_{dist}. Roughly speaking, the set P_{near} contains the points $p \in P$ which are quite close to their coreset point q_p, when this distance is compared to the distance from p toward its nearest center in C. More precisely, for any constant ε with $0 < \varepsilon \le 1$, we define

$$P_{\text{near}} = \left\{p \in P \mid \mathrm{D}(p, q_p) \le \varepsilon\, \mathrm{D}(p, C)\right\}.$$

The set P_{dist} contains all other points from P, that is,

$$P_{\text{dist}} = \left\{p \in P \mid \mathrm{D}(p, q_p) > \varepsilon\, \mathrm{D}(p, C)\right\}.$$

First, in Claim 3.3, we estimate the error of the clustering cost that occurs for any point in P_{near}. Then, in Claim 3.4, we give an estimation of the error for any point in P_{dist}.

CLAIM 3.3. *If $p \in P_{\text{near}}$, then*

$$\left|\mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C)\right| \le 3\varepsilon\, \mathrm{D}^2(p, C).$$

PROOF. For the moment, let us assume that $\mathrm{D}(p, C) \le \mathrm{D}(q_p, C)$. Let c_p denote the element from C closest to p. By triangle inequality and the definition of P_{near}, we have

$$\begin{aligned}\mathrm{D}(q_p, C) &\le \mathrm{D}(q_p, c_p) \\ &\le \mathrm{D}(p, c_p) + \mathrm{D}(p, q_p) \\ &\le (1 + \varepsilon) \cdot \mathrm{D}(p, C).\end{aligned}$$

Hence, for the squared distances, we obtain

$$\begin{aligned} \mathrm{D}^2(q_p, C) &\le (1+\varepsilon)^2 \cdot \mathrm{D}^2(p, C) \\ &\le (1+3\varepsilon) \cdot \mathrm{D}^2(p, C). \end{aligned}$$

Thus, we get

$$\mathrm{D}^2(q_p, C) - \mathrm{D}^2(p, C) \le 3\varepsilon\, \mathrm{D}^2(p, C),$$

which proves the claim in the case $\mathrm{D}(p, C) \le \mathrm{D}(q_p, C)$.

Now, assume that $\mathrm{D}(p, C) > \mathrm{D}(q_p, C)$. Let c_s denote the element from C closest to q_p. Again, by triangle inequality and the definition of P_{near}, we have

$$\begin{aligned} \mathrm{D}(p, C) &\le \mathrm{D}(p, c_s) \\ &\le \mathrm{D}(q_p, c_s) + \mathrm{D}(p, q_p) \\ &\le \mathrm{D}(q_p, C) + \varepsilon\, \mathrm{D}(p, C). \end{aligned}$$

It follows that

$$(1-\varepsilon) \cdot \mathrm{D}(p, C) \le \mathrm{D}(q_p, C).$$

For the squared distances, we obtain

$$\begin{aligned} \mathrm{D}^2(q_p, C) &\ge (1-\varepsilon)^2 \cdot \mathrm{D}^2(p, C) \\ &> (1-2\varepsilon) \cdot \mathrm{D}^2(p, C). \end{aligned}$$

Hence, we get

$$\begin{aligned} \mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C) &\le 2\varepsilon\, \mathrm{D}^2(p, C) \\ &< 3\varepsilon\, \mathrm{D}^2(p, C), \end{aligned}$$

which proves the claim in the case $\mathrm{D}(p, C) > \mathrm{D}(q_p, C)$. □

CLAIM 3.4. *If $p \in P_{\text{dist}}$, then*

$$\left| \mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C) \right| \le \frac{3}{\varepsilon}\, \mathrm{D}^2(p, q_p).$$

PROOF. Let c_p denote the element from C closest to p, and let c_s denote the element from C closest to q_p. By triangle inequality, we have

$$\begin{aligned} \mathrm{D}(p, C) &\le \mathrm{D}(p, c_s) \\ &\le \mathrm{D}(p, q_p) + \mathrm{D}(q_p, c_s) \\ &= \mathrm{D}(p, q_p) + \mathrm{D}(q_p, C). \end{aligned}$$

Similarly, we get

$$\begin{aligned} \mathrm{D}(q_p, C) &\le \mathrm{D}(q_p, c_p) \\ &\le \mathrm{D}(p, q_p) + \mathrm{D}(p, c_p) \\ &= \mathrm{D}(p, q_p) + \mathrm{D}(p, C). \end{aligned}$$

It follows that

$$\left| \mathrm{D}(p, C) - \mathrm{D}(q_p, C) \right| \le \mathrm{D}(p, q_p)$$

and

$$\mathrm{D}(p, C) + \mathrm{D}(q_p, C) \le 2\, \mathrm{D}(p, C) + \mathrm{D}(p, q_p).$$

Since $\mathrm{D}(p, q_p) > \varepsilon\, \mathrm{D}(p, C)$ and $\varepsilon \leq 1$, we get

$$\begin{aligned}\left|\mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C)\right| &= \left|\mathrm{D}(p, C) - \mathrm{D}(q_p, C)\right| \cdot \left(\mathrm{D}(p, C) + \mathrm{D}(q_p, C)\right)\\ &\leq \mathrm{D}(p, q_p) \cdot \left(2\,\mathrm{D}(p, C) + \mathrm{D}(p, q_p)\right)\\ &\leq \left(\frac{2}{\varepsilon} + 1\right) \mathrm{D}^2(p, q_p)\\ &\leq \frac{3}{\varepsilon}\, \mathrm{D}^2(p, q_p).\end{aligned}$$

□

Now, we can show our main result. Due to Claims 3.3 and 3.4, we have

$$\begin{aligned}&\left|\mathrm{cost}(P, C) - \mathrm{cost}_w(S, C)\right|\\ &\leq \sum_{p \in P} \left|\mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C)\right|\\ &\leq \sum_{p \in P_{\mathrm{near}}} \left|\mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C)\right| + \sum_{p \in P_{\mathrm{dist}}} \left|\mathrm{D}^2(p, C) - \mathrm{D}^2(q_p, C)\right|\\ &\leq 3\varepsilon \sum_{p \in P_{\mathrm{near}}} \mathrm{D}^2(p, C) + \frac{3}{\varepsilon} \sum_{p \in P_{\mathrm{dist}}} \mathrm{D}^2(p, q_p)\\ &\leq 3\varepsilon \cdot \mathrm{cost}(P, C) + \frac{3}{\varepsilon} \cdot \mathrm{cost}(P, S).\end{aligned}$$

Due to Lemma 2.3 and Markov's inequality, we obtain

$$\mathrm{cost}(P, S) \leq \frac{8}{\delta}\,(2 + \ln m) \cdot \mathrm{cost}^*_m(P)$$

with probability at least $1 - \delta$. Hence, by using Lemma 3.2 with

$$\gamma = \frac{\varepsilon^2 \delta}{8(2 + \ln(m))},$$

we have

$$\begin{aligned}\mathrm{cost}(P, S) &\leq \frac{8}{\delta}\,(2 + \ln m) \cdot \mathrm{cost}^*_m(P)\\ &\leq \frac{8}{\delta}\,(2 + \ln m) \cdot \gamma \cdot \mathrm{cost}^*_k(P)\\ &\leq \varepsilon^2\, \mathrm{cost}^*_k(P)\\ &\leq \varepsilon^2\, \mathrm{cost}(P, C)\end{aligned}$$

and, thus, $|\mathrm{cost}(P, C) - \mathrm{cost}_w(S, C)| \leq 6\varepsilon \cdot \mathrm{cost}(P, C)$ with probability $1 - \delta$, provided that the coreset size m satisfies the condition

$$m \geq \left(\frac{9d}{\gamma}\right)^{d/2} \cdot k \cdot \lceil \log(n) + 3 \rceil \tag{4}$$

of Lemma 3.2.

Hence, the only thing left to do is to prove that there exists a coreset size

$$m = \left(\frac{d}{\delta\varepsilon}\right)^{\mathcal{O}(d)} \cdot k \cdot \log(n) \cdot \log^{d/2}\left(\frac{k\log(n)}{\delta\varepsilon}\right)$$

that satisfies Inequality (4). Since we can assume that $n \geq 16$ and $m \geq 8$, we get $\lceil \log(n) + 3 \rceil \leq 2\log(n)$ and $2 + \ln(m) \leq 2\log(m)$. It follows that Inequality (4) is satisfied if we have

$$\frac{m}{\log^{d/2}(m)} \geq \underbrace{\frac{2 \cdot 12^d \, d^{d/2} \, k \log n}{\delta^{d/2}\varepsilon^d}}_{=t}. \tag{5}$$

We conclude that Inequality (5) is satisfied for a choice of

$$m = (2d)^{d/2} \cdot t \cdot \log^{d/2}(t),$$

since we have

$$\begin{aligned} \log^{d/2}(m) &= \log^{d/2}\left((2d)^{d/2} \cdot t \cdot \log^{d/2}(t)\right) \\ &\leq \left(\frac{d}{2}\right)^{d/2} \cdot \log^{d/2}(2d \cdot t \cdot \log(t)) \\ &\leq \left(\frac{d}{2}\right)^{d/2} \cdot \log^{d/2}\left(t^4\right) \\ &= (2d)^{d/2} \cdot \log^{d/2}(t). \end{aligned}$$ □

4. THE CORESET TREE

Unfortunately, there is one practical problem concerning the k-MEANS++ seeding procedure. Assume that we have chosen a sample set $S = \{q_1, q_2, \ldots, q_i\}$ from the input set $P \subseteq \mathbb{R}^d$ so far, where $i < m$ and $|P| = n$. In order to compute the probabilities to choose the next sample point q_{i+1}, we need to determine the distance from each point in P to its nearest neighbor in S. Hence, using a standard implementation of such a computation, we require time $\Theta(dnm)$ to obtain all m coreset points, which is too slow for larger values of m. Therefore, we propose a new data structure called *coreset tree*, which speeds up this computation. Note that the resulting coreset construction still does not work in the data stream model. In Section 5, we show how our coreset construction can be transferred to the data stream model.

Roughly speaking, a coreset tree is a hierarchical decomposition of P, where each leaf represents a set of this decomposition. The advantage of the coreset tree is that it allows us to compute subsequent sample points by taking only points from a subset of P into account that is significantly smaller than n. We obtain that if the constructed coreset tree is balanced (i.e., the tree is of depth $\Theta(\log m)$ and each leaf represents roughly the same number of points), we merely need time $\Theta(dn \log m)$ to compute all m coreset points. This intuition is supported by our empirical evaluation on real-world datasets, where we find that the process of sampling according to D^2 is significantly sped up, while the resulting sample set S has essentially the same properties as the original k-MEANS++ seed. Note that we do not know whether the result of Theorem 3.1 can be transferred to the coreset tree construction, which would mean that the seed computed by the coreset tree is a (k, ε)-coreset of P. However, the coreset tree construction is strongly allied to sampling according to D^2, so we are optimistic that a similar result can be obtained here as well.

In the following, we explain the construction of the coreset tree in more detail. A description in pseudocode is given by Algorithm 4.1.

4.1. Definition of the Coreset Tree

A coreset tree T for a point set P is a binary tree that is associated with a hierarchical divisive clustering for P: One starts with a single cluster that contains the whole point set P and successively partitions existing clusters into two subclusters, such that the points in one subcluster are far from the points in the other subcluster. The division step is repeated until the number of clusters corresponds to the desired number of clusters. Associated with this procedure, the coreset tree T has to satisfy the following properties.

—Each node of T is associated with a cluster in the hierarchical divisive clustering.
—The root of T is associated with the single cluster that contains the whole point set P.
—The nodes associated with the two subclusters of a cluster C are the child nodes of the node associated with C.

With each node v of T, we store the following attributes: A point set P_v, a representative point q_v from P_v, and a value weight(v). Here, point set P_v is the cluster associated with node v. Note that this attribute has only to be stored explicitly in the leaf nodes of T, while, for an inner node v, the set P_v is implicitly defined by the union of the point sets of its children. At any point of time, the set of all the points q_ℓ stored at a leaf node ℓ are the points that have been chosen so far to be points of the eventual coreset. Furthermore, for a leaf node ℓ, the attribute weight(ℓ) equals cost(P_ℓ, q_ℓ), which is the sum of squared distances over all points in P_ℓ to q_ℓ. The value weight(v) of an inner node v is defined as the sum of the weights of its children.

4.2. Construction of the Coreset Tree

For sake of simplicity, at any time, we number the leaf nodes of the current coreset tree consecutively starting with 1. At the beginning, T consists of one node, the root, which is given the number 1 and associated with the whole point set P. The attribute q_1 of the root is our first point in S and computed by sampling one point uniformly at random from P. Now, let us assume that our current tree has i leaf nodes $1, 2, \ldots, i$, the corresponding sample points are $q_1, q_2, \ldots, q_i$, and $P_1, P_2, \ldots, P_i$ are the associated clusters. We obtain the next sample point q_{i+1}, new clusters in our hierarchical divisive clustering, and, thus, new nodes in T by performing the following three steps.

(1) Choose a leaf node ℓ at random with a probability proportional to cost(P_ℓ, q_ℓ).
(2) Choose a new sample point, denoted by q_{i+1}, from the subset P_ℓ at random according to D^2. Note that, in this case, sampling according to D^2 means that a point is chosen with a probability proportional to its squared distance to the single representative p_ℓ.
(3) Based on q_ℓ and q_{i+1}, split P_ℓ into two subclusters and create two child nodes of ℓ in T.

Now, we describe the three steps in detail (see Algorithm 4.1 for a description in pseudocode). The first step is implemented as follows. Starting at the root of T, let u be the current inner node. Then, we select randomly a child node of u, where the probability distribution for the child nodes of u is given by their associated weights. More precisely, each child node v of the current node u is chosen with probability weight(v)/weight(u). We continue this selection process until we reach a leaf node. Let ℓ be the selected leaf node, let q_ℓ be the sample point stored at ℓ, and let P_ℓ be the

ALGORITHM 4.1. TREECORESET(P, m)

```
1  choose $q_1$ uniformly at random from $P$
2  $root$ ← node with $q_{root} = q_1$ and weight($root$) = cost($P, q_1$)
3  $S \leftarrow \{q_1\}$
4  for $i \leftarrow 2$ to $m$
5      start at $root$, iteratively select one of the two child nodes at random
       according to their weights, until a leaf $\ell$ is chosen
6      choose $q_i$ according to $D^2$ from $P_\ell$
7      $S \leftarrow S \cup \{q_i\}$
8      create two child nodes $\ell_1$, $\ell_2$ of $\ell$ and update weight($\ell$)
9      propagate update of weight attribute upwards to node $root$
```

ALGORITHM 4.2. INSERTPOINT(p)

```
1   put $p$ into $B_0$
2   if $B_0$ is full
3       create empty bucket $S$
4       move points from $B_0$ to $S$
5       empty $B_0$
6       $i \leftarrow 1$
7       while $B_i$ is not empty
8           create coreset from the union of $B_i$ and $S$
9           store coreset in $S$
10          empty $B_i$
11          $i \leftarrow i + 1$
12      move points from $S$ to $B_i$
```

subset of P associated with ℓ. It is easy to check that, in doing so, we have chosen ℓ among the leaf nodes with probability $\text{cost}(P_\ell, q_\ell)/\sum_{j=1}^{i}\text{cost}(P_j, q_j)$.

In the second step, we choose the new sample point q_{i+1} from P_ℓ at random according to D^2, that is, each $p \in P_\ell$ is chosen with probability $D^2(p, q_\ell)/\text{cost}(P_\ell, q_\ell)$. In doing so, each point in P is sampled with a probability that is proportional to its squared distance to its center in the clustering induced by the partition of the leaf nodes (giving the clusters) and their sample points (being the centers). That is, we use the same distribution as the k-MEANS++ seeding does with the exception that the probability of choosing a point $p \in P_j$ is proportional to $D^2(p, q_j)$ rather than proportional to $D^2(p, \{q_1, \ldots, q_i\})$.

In the third step, we create two child nodes ℓ_1 and ℓ_2 of ℓ and compute the associated partition of P_ℓ as well as the corresponding attributes. We store at node ℓ_1 the point q_ℓ and at node ℓ_2 our new sample point q_{i+1}. Based on these two representative points, we partition P_ℓ into two subsets P_{ℓ_1} and P_{ℓ_2}. The set P_{ℓ_1} contains all the points from P_ℓ which are closer to q_ℓ than to q_{i+1}, that is,

$$P_{\ell_1} = \{p \in P_\ell \mid D(p, q_\ell) < D(p, q_{i+1})\}.$$

The set P_{ℓ_2} contains all the remaining points from P_ℓ, that is,

$$P_{\ell_2} = \{p \in P_\ell \mid D(p, q_{i+1}) \leq D(p, q_\ell)\}.$$

The node ℓ_1 is associated with the set P_{ℓ_1}, and ℓ_2 is associated with the set P_{ℓ_2}. We determine the weight attribute for the nodes ℓ_1 and ℓ_2 as described earlier. Recall here that the weight attribute of an inner node of T is defined as the sum of the

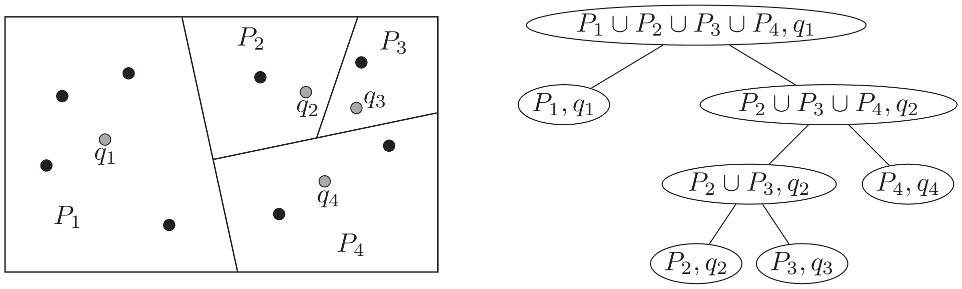

Fig. 2. Example illustrating a coreset tree for a point set in the plane. The right-hand side of the figure illustrates the coreset tree for a point set $P = P_1 \cup P_2 \cup P_3 \cup P_4$, where the sample points are chosen in the order q_1, q_2, q_4, q_3. The first piece of information contained in a node is the point set associated with the node. The second piece of information is the sample point associated with the node. The third attribute of a node is omitted. The left-hand side of the figure indicates how the coreset nodes are related to regions in the plane.

weights of its child nodes. Consequently, we update the weight of the parent node ℓ of ℓ_1 and ℓ_2 according to this. Afterward, this update is propagated upward, until we reach the root of the tree. Figure 2 shows a coreset tree for a small point set in the plain.

4.3. Extraction of the Coreset

As soon as the coreset tree T has m leaf nodes, we can construct our coreset. Let $q_1, q_2, \dots, q_m$ be the representative points stored at the leaf nodes of T. Furthermore, let Q_i denote the set of points from P that are closest to q_i (breaking ties arbitrarily). Then, we obtain the coreset $S = \{q_1, q_2, \dots, q_m\}$, where the weight of q_i is given by the number of points in Q_i.

5. THE STREAMING ALGORITHM

In this section, we describe our clustering algorithm for data streams. To this end, let m be a fixed-size parameter. First, we extract a small coreset of size m from the data stream by using the merge-and-reduce technique from Har-Peled and Mazumdar [2004], which is based on the theory of decomposable search problems of Bentley and Saxe [1980]. The key concept of this technique is as follows: The data is organized in a small number of samples, each representing $2^i m$ input points (for some integer i). Every time two samples representing the same number of input points exist, we take the union (merge) and create a new sample (reduce). A more detailed description of the merge-and-reduce technique is given in Section 5.1.

For the reduce step, we employ our new coreset construction, using the coreset trees, as explained in Section 4. A k-clustering can be obtained at any time by running any k-means algorithm on the coreset. Note that since the size of the coreset is much smaller than the size of the data stream, it is no longer inefficient to use algorithms that require random access on their input data. In our implementation, we run the k-MEANS++ algorithm from Arthur and Vassilvitskii [2007] on our coreset five times independently and choose the best clustering result obtained this way. We call the resulting algorithm STREAMKM++.

5.1. The Merge-and-Reduce Technique

In order to maintain a small coreset for all points in the data stream, we use the merge-and-reduce method [Bentley and Saxe 1980; Har-Peled and Mazumdar 2004]. For a data stream containing n points, the algorithm maintains $L = \lceil \log_2(n/m) + 2 \rceil$

buckets $B_0, B_1, \ldots, B_{L-1}$. Bucket B_0 can store any number between 0 and m points. In contrast, for $i \geq 1$, bucket B_i is either empty or contains exactly m points. The idea of this approach is that, at any time, if bucket B_i is full, it contains a coreset of size m representing $2^{i-1}m$ points from the data stream.

New points from the data stream are always inserted into the first bucket B_0. If bucket B_0 is full (i.e., contains m points), all points from B_0 need to be moved to bucket B_1. If bucket B_1 is empty, we are done. However, if bucket B_1 already contains m points, we compute a new coreset S of size m from the union of the $2m$ points stored in B_0 and B_1. Now, both buckets B_0 and B_1 are emptied, and the m points from coreset S are moved into bucket B_2 (unless, of course, bucket B_2 is also full in which case the process is repeated). Pseudocode for inserting a point from the data stream into the buckets is given in Algorithm 4.2.

At any time, it is possible to compute a coreset of size m for all the points in the data stream that we have seen so far. For this purpose, we compute a coreset from the union of the at most $m\lceil\log_2(n/m) + 2\rceil$ weighted coreset points stored in all the buckets $B_0, B_1, \ldots, B_{L-1}$.

The merge-and-reduce technique is based on the following two observations [Har-Peled and Mazumdar 2004].

—If S_1 and S_2 are (k, ε)-coresets for disjoint sets P_1 and P_2, respectively, then $S_1 \cup S_2$ is a (k, ε)-coreset for $P_1 \cup P_2$.
—If S_1 is a (k, ε)-coreset for S_2 and S_2 is a (k, ε')-coreset for S_3, then S_1 is a $(k, (1 + \varepsilon)(1 + \varepsilon') - 1)$-coreset for S_3.

Using these facts about unions of coresets and nested coreset constructions, one can show that if we compute a $(k, \varepsilon/(2\lceil\log_2(n/m) + 2\rceil))$-coreset of the two merged point sets in each reduce step, then the merge-and-reduce technique yields a (k, ε)-coreset of the input stream.

In order for us to be able to employ our coreset tree construction for the reduce step, it has to be generalized to input points with integer weights. This can be easily done as follows: Each time when we choose a new coreset point, we compute the probabilities of the points according to D^2, as described before, and then multiply each probability with the weight of the appropriate point. We also incorporate the point weights when we compute the weight attribute of a new leaf node. These two adaptations can be thought of as replacing each weighted point by multiple copies of the same point each having weight 1.

5.2. Complexity

Using our implementation, a single merge-and-reduce step is guaranteed to be executed in time $\mathcal{O}(dm^2)$ (or even in time $\Theta(dm \log m)$, if we assume the used coreset tree to be balanced). For a stream of n points, $\lceil n/m \rceil$ such steps are needed. Hence, the running time of all merge-and-reduce steps is $\mathcal{O}(dnm)$. The final merge-and-reduce step, to obtain a coreset of size m for the union of all buckets, can be done in time $\mathcal{O}(dm^2 \log(n/m))$. Finally, algorithm k-MEANS++ is executed five times on an input set of size m, requiring time $\Theta(dkm)$ per iteration. Summing up, the total running time of algorithm StreamKM++ is $\mathcal{O}(dnm)$, and the amortized processing time per data item is $\mathcal{O}(dm)$. Obviously, algorithm StreamKM++ needs at most $\Theta(dm \log(n/m))$ memory units. Hence, we obtain a low dependency on the dimension d, so our approach is suitable for high-dimensional data.

Of course, careful consideration has to be given to the choice of the coreset size parameter m. Our experiments show that a choice of $m = 200k$ is sufficient for a good clustering quality without sacrificing too much running time.

6. EMPIRICAL EVALUATION

We conducted several experiments on different datasets to evaluate the quality of algorithm StreamKM++.[1] A description of the datasets can be found in the Section 6.1. The computation on the biggest dataset, which is denoted by *BigCross*, was performed on a DELL Optiplex 620 machine with 3GHz Pentium D CPU and 2GB main memory, using Linux 2.6.9 kernel. For all remaining datasets, the computation was performed on a DELL Optiplex 620 machine with 3GHz Pentium D CPU and 4GB main memory, using Linux 2.6.18 kernel.

We compared algorithm StreamKM++ with two frequently used clustering algorithms for processing data streams, namely with algorithm BIRCH[2] [Zhang et al. 1996] and with algorithm StreamLS[3] [O'Callaghan et al. 2002; Guha et al. 2003]. On the smaller datasets, we also compared our algorithm with a vanilla implementation of Lloyd's algorithm [Lloyd 1982], using initial seeds either uniformly at random (algorithm k-means) or according to the nonuniform seeding from Arthur and Vassilvitskii [2007] (algorithm k-means++). All algorithms were compiled using g++ from the GNU Compiler Collection on optimization level 2. The quality measure for all experiments was the sum of squared distances, referred to as the cost of the clustering.

6.1. Datasets

Since synthetic datasets are typically easy to cluster, we focused our experiments on real-world datasets to obtain practically relevant results. Our main source was the UCI Machine Learning Repository [Asuncion and Newman 2007]. In the following, we give a brief description of all datasets used in our empirical evaluation.

Spambase[4] is a dataset that contains data about spam emails and nonspam emails including work and personal emails. Each data entry is a vector consisting of frequencies of certain words or characters occurring in the message and a class attribute that denotes whether the correspondig email was considered spam. After removing the classification attribute, 4,601 data points in 57 dimensions remained.

Intrusion[4,5] comprises data about TCP transmissions in a simulated environment. This simulation included different types of network attacks and intrusion attempts as well as normal network traffic. We used a 10% subset of the whole unlabeled dataset[6] and excluded all symbolic features. Eventually, 311,078 data points in 34 dimensions remained.

Covertype[4,7] contains cartographic data about some wilderness areas inside the Roosevelt National Forest of northern Colorado. The leading thought of analyzing this dataset is to be able to predict the forest cover type of specific regions from cartographic variables, which is a classification task. After removing the classification attribute, 581,012 data points in 54 dimensions remained.

The *Tower*[8] dataset consists of the RGB values of a 2,560 by 1,920 pixel image file. All 4,915,200 pixels are mapped into a 3-dimensional space of integer values between 0 and 255, representing the colors used in the image. Note that clustering techniques

[1]The source code, the documentation, and the datasets of our experiments can be found at `http://www.cs.upb.de/en/fachgebiete/ag-bloemer/research/clustering/streamkmpp/`.
[2]The source code can be found at `http://pages.cs.wisc.edu/~vganti/birchcode/`.
[3]The source code can be found at `http://infolab.stanford.edu/~loc/`.
[4]The dataset was contributed by the UCI Machine Learning Repository [Asuncion and Newman 2007].
[5]*Intrusion* dataset is part of the kddcup99 dataset.
[6]Available for free download at `http://kdd.ics.uci.edu/databases/kddcup99/kddcup.newtestdata_10_percent_unlabeled.gz`.
[7]Copyright by Jock A. Blackard, Colorado State University.
[8]*Tower* dataset was contributed by Gereon Frahling and is available for free download at `http://homepages.uni-paderborn.de/frahling/coremeans.html`.

Table I. Overview of the Datasets

	data points	dimension	type
Spambase	4,601	57	float
Intrusion	311,079	34	int, float
Covertype	581,012	54	int
Tower	4,915,200	3	int
Census 1990	2,458,285	68	int
BigCross	11,620,300	57	int
NormData	100,000	15	float

are frequently used for lossy image compression: Individual colors can be substituted with their corresponding cluster center.

The *Census 1990*[4] dataset consists of a 1% sample of the Public Use Microdata Samples (PUMS) person records, sampled from the full 1990 census set contributed by the U.S. Department of Commerce Census Bureau. Most of the data is citizen-related information such as personal income or age. The dataset has 2,458,285 data points in 68 dimensions. To our knowledge, it is one of the largest, free-accessible, naturally structured datasets available.

To run our algorithm on really huge datasets, we created the Cartesian product of the *Tower* and *Covertype* dataset. In this way, we got a naturally structured dataset that is large enough to test the ability of our algorithm to handle huge amounts of data. We used a 1.5GB-sized subset of the Cartesian product consisting of 11,620,300 data points with 57 attributes, which we refer to as the *BigCross* dataset.

To evaluate the impact of the number of well-separated clusters of a dataset, we also considered a number of synthetic datasets to which we collectively refer as the *Normdata* datasets. To generate these datasets, we used essentially the same construction that has already been used by [Arthur and Vassilvitskii 2007] to evaluate the k-MEANS++ algorithm. More precisely, for different values of k, we chose k "true" centers uniformly at random from a 15-dimensional hypercube of side length 100. We then randomly chose points from a uniform mixture of 15-dimensional normal distributions of variance 1 around these center points. In this way, we obtained k well-separated clusters. Each *Normdata* dataset consist of 100,000 points.

The size and the dimensionality of the datasets are summarized in Table I.

6.2. Parameters of the Algorithms

For algorithm BIRCH, we set all parameters of the experimental environment as recommended by the authors of BIRCH except for the memory settings. Like Guha et al. [2003], we observed that the CF-Tree had less leaves than it was allowed to use. The CF-Tree is the data structure used to compute the preclustering into the so-called clustering features (see also Section 2). The more leaves it has, the finer is the preclustering. Therefore, from time to time, BIRCH did not produce the correct number of centers, especially when the number of clusters k was high. For this reason, the memory settings had to be manually adjusted for each individual dataset. The complete list of parameters is given in Tables IV and V.

For algorithm STREAMKM++, we determined experimentally an appropriate coreset size m as a function of k. For obvious reasons, we need to choose $m \geq k$. To estimate an m that is sufficient to obtain good approximation results, we ran several experiments for different values of k and m on the datasets *Covertype* and *Tower*. Due to the randomized[9] nature of STREAMKM++, we conducted 10 runs for each fixed k and for each fixed m. Figure 3 shows the average running times and cost of the clusterings. Concerning the

[9]We used the Mersenne Twister PRNG [Matsumoto and Nishimura 1998].

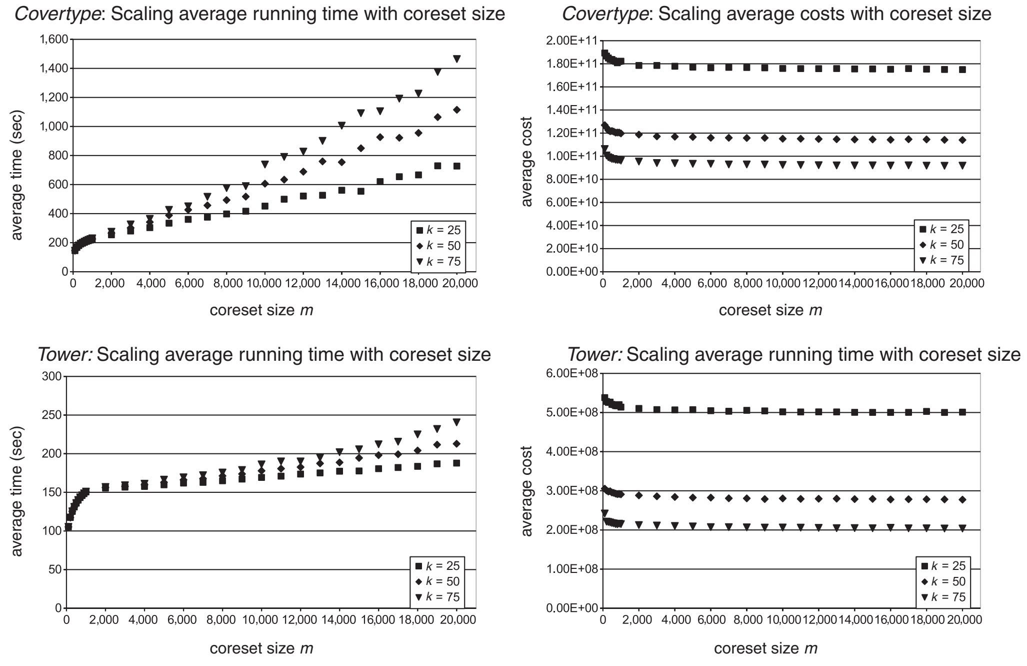

Fig. 3. Experimental results for different coreset sizes.

cost, we observed that, for coreset sizes that are only marginally larger than k, the quality of a clustering can be considerably improved by increasing the coreset size. In contrast, for coreset sizes of, say, $m = 100k$ or more, the quality improves only slightly with increasing coreset size. For instance, the cost of a 50-clustering of either dataset computed on 20,000 coreset points is only marginally smaller than the cost of a clustering computed on 10,000 coreset points. However, with respect to the running time, we observed that the growth of the running time depends roughly linear on the coreset size. Overall, we conclude the following. On the one hand, m should be chosen not too small (e.g., a very small multiple of k) because, for these values of m, the quality of a clustering can be easily improved, without sacrificing too much running time. On the other hand, m should not be chosen too large (e.g., a large multiple of k) because the increase in quality is only very small compared to clusterings for smaller coresets, but the running time is significantly higher. Therefore, we assume that our choice of $m = 200k$ provides a good trade-off for arbitrary datasets. However, smaller sizes such as $m = 20k$ or $m = 50k$ might still be sufficient to obtain very good clustering results on datasets with k well-separated clusters.

For algorithm StreamLS, the size of the data chunks is set equal to the coreset size m of algorithm StreamKM++. This is done to enable a fair comparison of both algorithms by allowing the same memory usage. We have to point out that, due to its nature, algorithm StreamLS does not always compute the prespecified number of cluster centers. In such a case, the difference varies from dataset to dataset and, usually, lies within a 20% margin from the specified number.

6.3. Comparison with BIRCH and StreamLS

To compare StreamKM++ with BIRCH and StreamLS, we conducted several experiments for different values of k on the four larger, real-world datasets, that is,

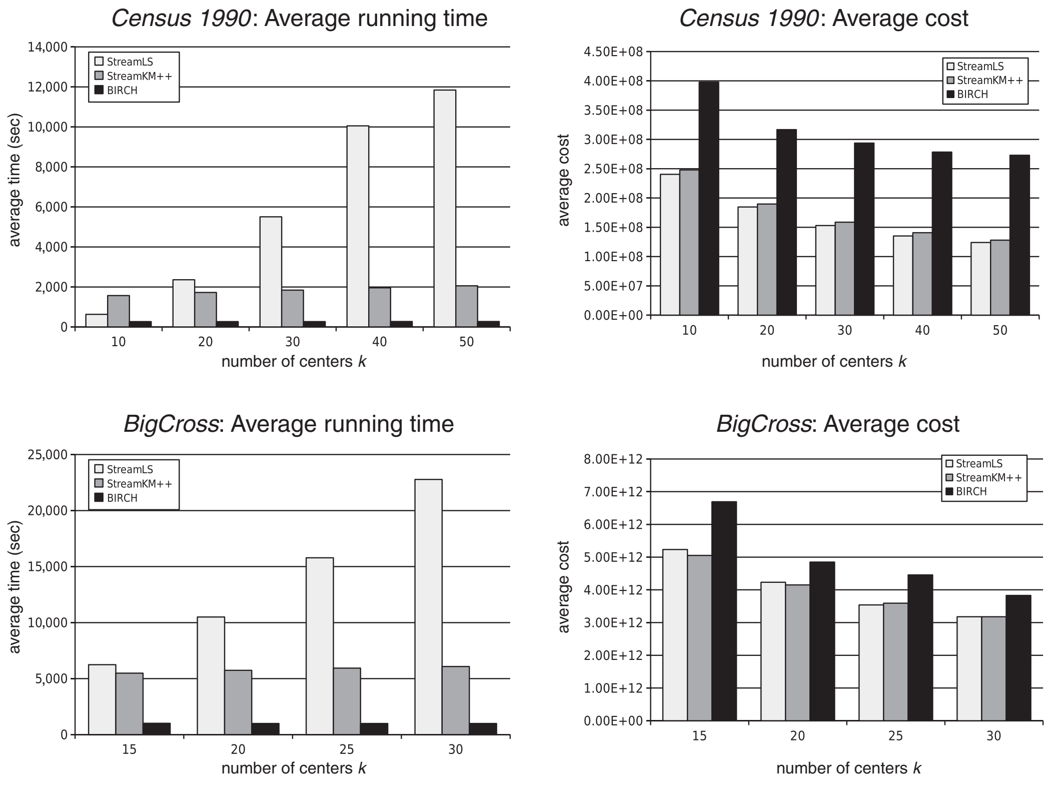

Fig. 4. Experimental results for *Census 1990* and *BigCross* datasets.

the datasets *Covertype*, *Tower*, *Census 1990*, and *BigCross*. In each of these experiments, we set $m = 200k$. For the randomized algorithms StreamKM++ and StreamLS, 10 experiments were conducted for each fixed k. For BIRCH, a single run was used, since it is a deterministic algorithm. The average running times and cost of the clusterings are summarized in Figures 4 and 5. The interested reader can find the concrete values of all experiments in Appendixes A.2 and A.3.

In our experiments, algorithm BIRCH had the best running time of all algorithms. However, this comes at the expense of a high k-means clustering cost. In terms of the sum of squared distances, algorithms StreamKM++ and StreamLS outperform BIRCH by up to a factor of 2. Furthermore, as already mentioned, one drawback of algorithm BIRCH is the need to manually adjust parameters to obtain a clustering with the desired number of centers.

By comparing StreamKM++ and StreamLS, we observed that the quality of the clusterings were on a par. More precisely, the absolute value of the cost of both algorithms lies within a $\pm 5\%$ margin from each other. In contrast to algorithm StreamLS, the number of centers computed by our algorithm always equals its prespecified value. Hence, the cost of clusterings computed by algorithm StreamKM++ tends to be more stable than the costs computed by algorithm StreamLS. The standard deviations of the experiments for $k = 20$ are given in Tables II and III. Appendix A.4 provides a complete overview of all the experiments.

In terms of the running time, it turns out that our algorithm scales much better with increasing number of centers than algorithm StreamLS does. While for about $k \leq 10$

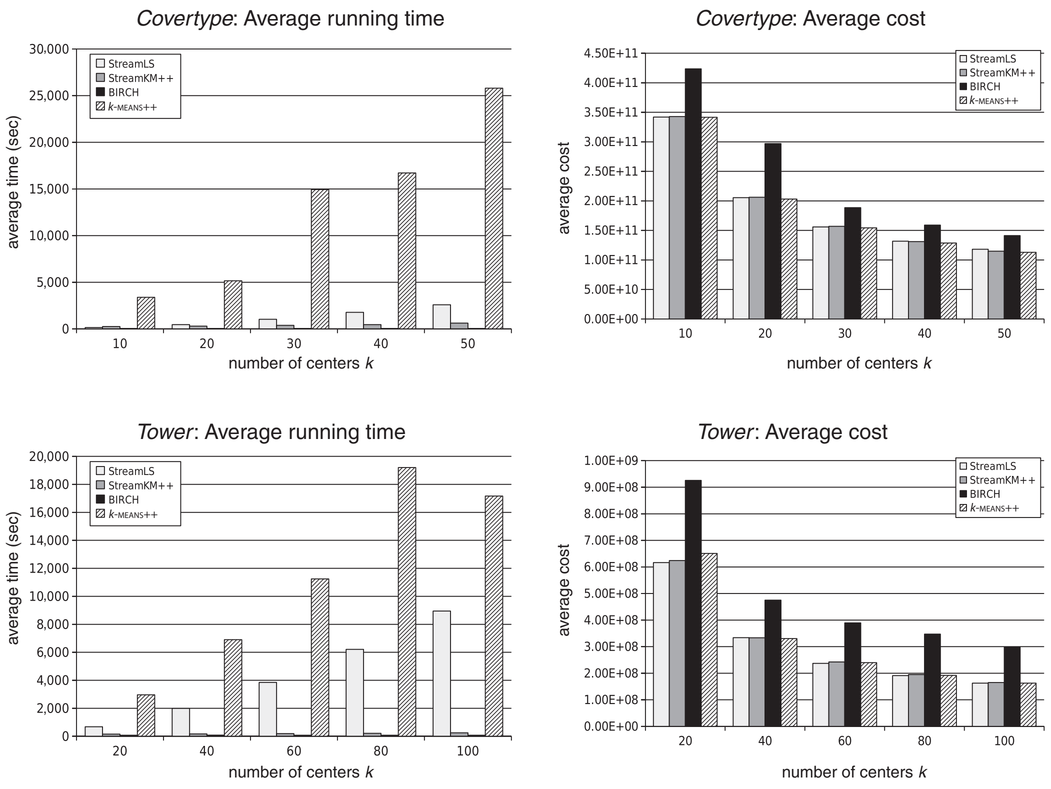

Fig. 5. Experimental results for *Covertype* and *Tower* datasets.

Table II. Standard Deviation of the Running Time for $k = 20$

$k = 20$	running time (in sec)		
	StreamKM++	StreamLS	*k*-means++
Spambase	1.09	—	3.88
Intrusion	3.22	—	98.11
Covertype	6.93	18.18	1,249.18
Tower	0.58	14.11	1,594.76
Census 1990	5.16	54.30	—
BigCross	11.49	162.44	—

Table III. Standard Deviation of the Cost for $k = 20$

$k = 20$	cost		
	StreamKM++	StreamLS	*k*-means++
Spambase	$6.49 \cdot 10^5$	—	$1.73 \cdot 10^6$
Intrusion	$8.54 \cdot 10^{10}$	—	$3.70 \cdot 10^{11}$
Covertype	$1.08 \cdot 10^9$	$1.03 \cdot 10^{10}$	$9.17 \cdot 10^8$
Tower	$7.31 \cdot 10^6$	$2.71 \cdot 10^7$	$4.39 \cdot 10^7$
Census 1990	$3.66 \cdot 10^6$	$3.14 \cdot 10^6$	—
BigCross	$2.46 \cdot 10^{10}$	$3.36 \cdot 10^{11}$	—

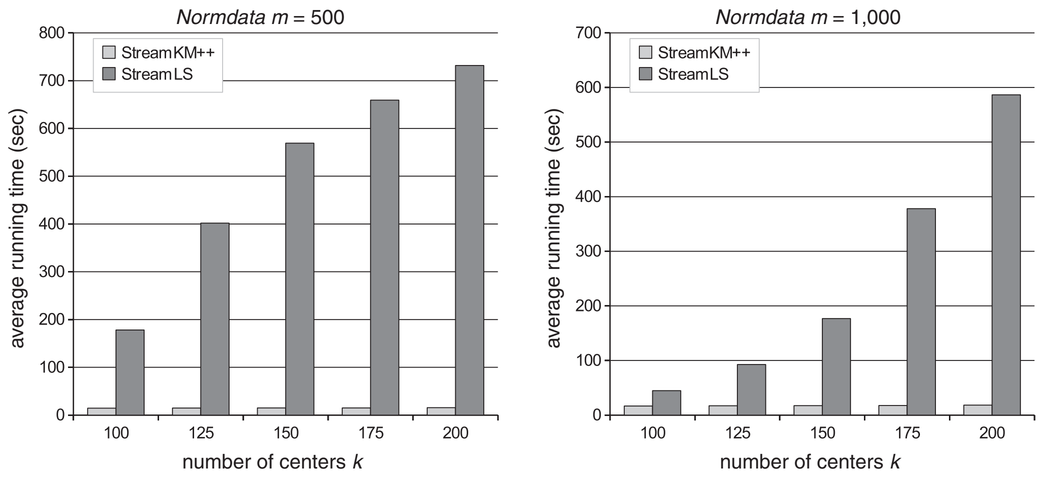

Fig. 6. Experimental results for *Normdata* datasets.

centers, StreamLS is sometimes faster than our algorithm, and for a larger number of centers, our algorithm easily outperforms StreamLS. For instance, on the dataset *Tower*, StreamKM++ computes a clustering with $k = 100$ centers in about 3% of the running time of StreamLS.

To investigate the impact of the number of clusters on the running time further, we conducted experiments on the synthetic datasets *Normdata* for different fixed values of k and m. As described earlier, for both StreamKM++ and StreamLS, 10 experiments were conducted for each fixed k and for each fixed m. The average running times of the clusterings are shown in Figure 6. Note that we omitted a figure presenting the average cost of the clusterings, because both StreamKM++ and StreamLS always found an optimal or near-optimal clustering. The interested reader can find the average values as well as the standard deviations for both running times and cost of the clusterings in the appendix. Figure 6 reveals the difference between the running times of StreamKM++ and StreamLS. The ratio between the running time needed by StreamKM++ and the running time needed by StreamLS is decreasing with increasing number of clusters. For $m = 500$, StreamKM++ computed the clusterings for $k = 100$ in about 9% of the running time of StreamLS and for $k = 200$ in about 2% of the running time of StreamLS. For $m = 1{,}000$, StreamKM++ computed the clusterings for $k = 100$ in about 38% of the running time of StreamLS, whereas for $k = 200$, it needed about 3% of the running time of StreamLS.

Overall, we conclude that if the first priority is the quality of the clustering, then our algorithm provides a good alternative to BIRCH and StreamLS, particularly if the number of cluster centers is large.

6.4. Comparison with *k*-means and *k*-means++

We also compared the quality of StreamKM++ with classical nonstreaming k-means algorithms. Because of their popularity, we have chosen k-means and the recent k-means++ as competitors. These algorithms are designed to work in a classical nonstreaming setting and, due to their need for random access on the data, are not suited for larger datasets. For this reason, we have run k-means only on the two smallest datasets, *Spambase* and *Intrusion*, while k-means++ has been evaluated only on the four smaller datasets, *Covertype*, *Tower*, *Spambase*, and *Intrusion*. For each fixed k, we conducted

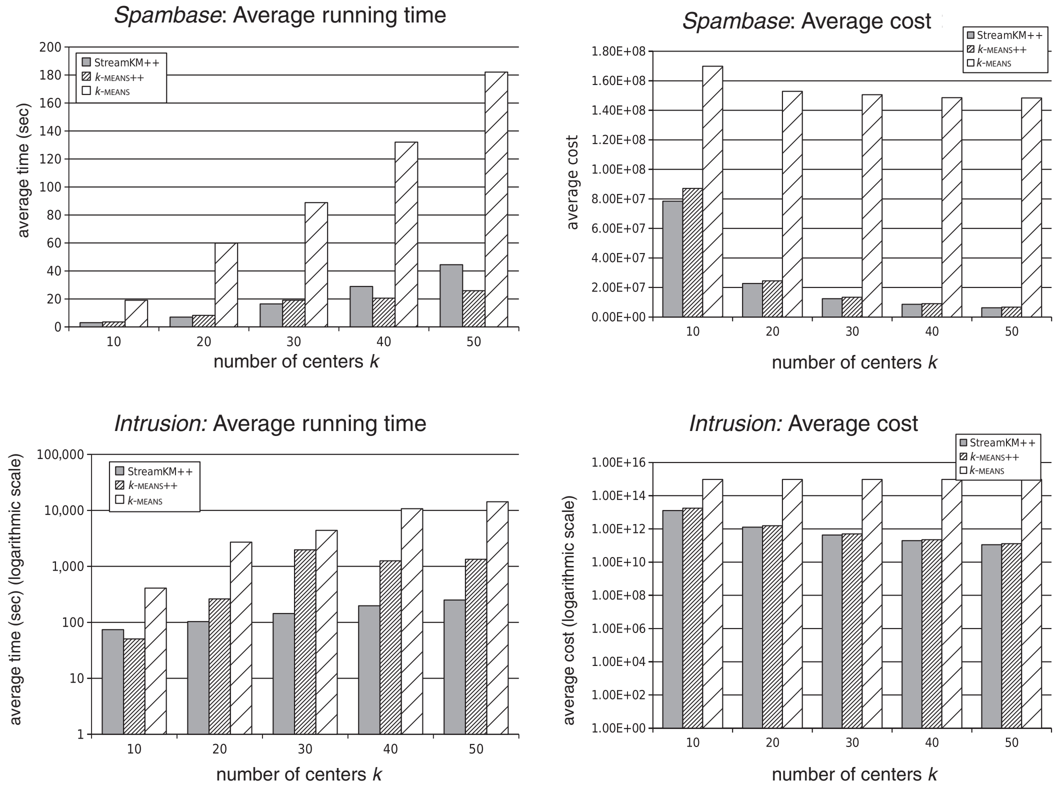

Fig. 7. Experimental results for *Spambase* and *Intrusion* datasets.

10 experiments. The results of these experiments are summarized in Figures 5 and 7. Please note that the results for the dataset *Intrusion* are on a logarithmic scale. The concrete values of all experiments can be found in Appendixes A.2 and A.3. The standard deviations of the experiments are given in Appendix A.4.

As expected, *k*-MEANS++ is clearly superior to the classical algorithm *k*-MEANS, both in terms of quality and running time. Comparing *k*-MEANS++ with our streaming algorithm, we find that on all datasets the quality of the clusterings computed by algorithm STREAMKM++ is on a par with or even better than the clusterings obtained by algorithm *k*-MEANS++. We conjecture that this is due to the fact that, in the last step of our algorithm, we run the *k*-MEANS++ algorithm five times on the coreset and choose the best clustering result obtained this way. On the other hand, for the experiments with the *k*-MEANS++ algorithm, we run the *k*-MEANS++ algorithm only once in each repetition of the experiment. However, the running time of *k*-MEANS++ is only comparable with algorithm STREAMKM++ for the smallest dataset, *Spambase*. Even for moderately large datasets, like dataset *Covertype*, we obtain that algorithm STREAMKM++ is orders of magnitude faster than *k*-MEANS++. We conclude that algorithm *k*-MEANS++ should only be used if the size of the dataset is not too large. For larger datasets, algorithm STREAMKM++ computes comparable clusterings in a significantly improved running time.

APPENDIX

A.1. Parameters of Algorithm BIRCH

Table IV. Manually Adjusted TotalMemSize as Percentage of the Dataset Size for Algorithm BIRCH

	Covertype	*Tower*	*Census 1990*	*BigCross*
$p =$	10	5	5	25

Table V. List of Parameters for Algorithm BIRCH

Parameter	Value
CorD	0
TotalMemSize (in bytes)	p% of dataset size
TotalBufferSize (in bytes)	5% of TotalMemSize
TotalQueueSize (in bytes)	5% of TotalMemSize
TotalOutlierTreeSize (in bytes)	5% of TotalMemSize
WMflag	0
W vector	(1,1,. . . ,1)
M vector	(0,0,. . . ,0)
PageSize (in bytes)	1,024
BDtype	4
Ftype	0
Phase1Scheme	0
RebuiltAlg	0
StatTimes	3
NoiseRate	0.25
Range	2,000
CFDistr	0
H	0
Bars vector	(100,100,. . . ,100)
K	number of clusters k
InitFt	0
Ft	0
Gtype	1
GDtype	2
Qtype	0
RefineAlg	1
NoiseFlag	0
MaxRPass	1

A.2. Running Time of Our Experiments

Table VI. Average Running Time of Our Experiments

		running time (in sec)				
dataset	k	StreamKM++	StreamLS	BIRCH	k-means++	k-means
Spambase	10	3.06	—	—	3.57	19.02
	20	7.04	—	—	8.22	59.85
	30	16.45	—	—	19.05	88.8
	40	28.93	—	—	20.54	132.03
	50	44.48	—	—	25.9	182.08
Intrusion	10	74.1	—	—	50.6	408.8
	20	103.1	—	—	262.4	2,711.3
	30	143.8	—	—	1,973.3	4,389.1
	40	197.6	—	—	1,257.0	10,733.7
	50	250.5	—	—	1,339.5	14,282.0
Covertype	10	245	147	44	3,389	—
	20	297	460	44	5,160	—
	30	378	1,027	44	14,933	—
	40	454	1,773	44	16,713	—
	50	617	2,588	44	25,803	—
Tower	20	157	679	77	2,960	—
	40	168	1,989	78	6,902	—
	60	187	3,849	77	11,247	—
	80	211	6,212	77	19,206	—
	100	248	8,946	77	17,161	—
Census 1990	10	1,571	631	271	—	—
	20	1,724	2,362	271	—	—
	30	1,839	5,504	271	—	—
	40	1,956	10,054	272	—	—
	50	2,057	11,842	272	—	—
BigCross	15	5,486	6,239	1,006	—	—
	20	5,738	10,502	998	—	—
	25	5,933	15,780	996	—	—
	30	6,076	22,779	996	—	—
Normdata	100	14.5	178.2	—	—	—
($m = 500$)	125	14.9	401.8	—	—	—
	150	15.1	569.3	—	—	—
	175	15.1	659.3	—	—	—
	200	15.6	731.8	—	—	—
Normdata	100	16.7	44.8	—	—	—
($m = 1,000$)	125	17.1	92.6	—	—	—
	150	17.5	176.9	—	—	—
	175	17.6	378.1	—	—	—
	200	18.3	586.7	—	—	—

A.3. Cost of Our Experiments

Table VII. Average Cost of Our Experiments

		cost				
dataset	k	StreamKM++	StreamLS	BIRCH	k-means++	k-means
Spambase	10	$7.85 \cdot 10^{7}$	—	—	$8.71 \cdot 10^{7}$	$1.70 \cdot 10^{8}$
	20	$2.27 \cdot 10^{7}$	—	—	$2.45 \cdot 10^{7}$	$1.53 \cdot 10^{8}$
	30	$1.24 \cdot 10^{7}$	—	—	$1.34 \cdot 10^{7}$	$1.51 \cdot 10^{8}$
	40	$8.64 \cdot 10^{6}$	—	—	$9.01 \cdot 10^{6}$	$1.49 \cdot 10^{8}$
	50	$6.29 \cdot 10^{6}$	—	—	$6.68 \cdot 10^{6}$	$1.48 \cdot 10^{8}$
Intrusion	10	$1.27 \cdot 10^{13}$	—	—	$1.75 \cdot 10^{13}$	$9.52 \cdot 10^{14}$
	20	$1.26 \cdot 10^{12}$	—	—	$1.55 \cdot 10^{12}$	$9.51 \cdot 10^{14}$
	30	$4.29 \cdot 10^{11}$	—	—	$4.96 \cdot 10^{11}$	$9.51 \cdot 10^{14}$
	40	$1.95 \cdot 10^{11}$	—	—	$2.25 \cdot 10^{11}$	$9.50 \cdot 10^{14}$
	50	$1.11 \cdot 10^{11}$	—	—	$1.29 \cdot 10^{11}$	$9.50 \cdot 10^{14}$
Covertype	10	$3.43 \cdot 10^{11}$	$3.42 \cdot 10^{11}$	$4.24 \cdot 10^{11}$	$3.42 \cdot 10^{11}$	—
	20	$2.06 \cdot 10^{11}$	$2.05 \cdot 10^{11}$	$2.97 \cdot 10^{11}$	$2.03 \cdot 10^{11}$	—
	30	$1.57 \cdot 10^{11}$	$1.56 \cdot 10^{11}$	$1.89 \cdot 10^{11}$	$1.54 \cdot 10^{11}$	—
	40	$1.31 \cdot 10^{11}$	$1.32 \cdot 10^{11}$	$1.59 \cdot 10^{11}$	$1.29 \cdot 10^{11}$	—
	50	$1.15 \cdot 10^{11}$	$1.18 \cdot 10^{11}$	$1.41 \cdot 10^{11}$	$1.13 \cdot 10^{11}$	—
Tower	20	$6.24 \cdot 10^{8}$	$6.16 \cdot 10^{8}$	$9.26 \cdot 10^{8}$	$6.51 \cdot 10^{8}$	—
	40	$3.34 \cdot 10^{8}$	$3.34 \cdot 10^{8}$	$4.75 \cdot 10^{8}$	$3.30 \cdot 10^{8}$	—
	60	$2.43 \cdot 10^{8}$	$2.37 \cdot 10^{8}$	$3.89 \cdot 10^{8}$	$2.40 \cdot 10^{8}$	—
	80	$1.95 \cdot 10^{8}$	$1.91 \cdot 10^{8}$	$3.47 \cdot 10^{8}$	$1.92 \cdot 10^{8}$	—
	100	$1.65 \cdot 10^{8}$	$1.63 \cdot 10^{8}$	$2.98 \cdot 10^{8}$	$1.63 \cdot 10^{8}$	—
Census 1990	10	$2.48 \cdot 10^{8}$	$2.40 \cdot 10^{8}$	$3.98 \cdot 10^{8}$	—	—
	20	$1.90 \cdot 10^{8}$	$1.85 \cdot 10^{8}$	$3.17 \cdot 10^{8}$	—	—
	30	$1.59 \cdot 10^{8}$	$1.53 \cdot 10^{8}$	$2.94 \cdot 10^{8}$	—	—
	40	$1.41 \cdot 10^{8}$	$1.35 \cdot 10^{8}$	$2.78 \cdot 10^{8}$	—	—
	50	$1.28 \cdot 10^{8}$	$1.24 \cdot 10^{8}$	$2.73 \cdot 10^{8}$	—	—
BigCross	15	$5.05 \cdot 10^{12}$	$5.23 \cdot 10^{12}$	$6.69 \cdot 10^{12}$	—	—
	20	$4.15 \cdot 10^{12}$	$4.23 \cdot 10^{12}$	$4.85 \cdot 10^{12}$	—	—
	25	$3.59 \cdot 10^{12}$	$3.54 \cdot 10^{12}$	$4.45 \cdot 10^{12}$	—	—
	30	$3.18 \cdot 10^{12}$	$3.18 \cdot 10^{12}$	$3.83 \cdot 10^{12}$	—	—
Normdata	100	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
($m = 500$)	125	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
	150	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
	175	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
	200	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
Normdata	100	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
($m = 1{,}000$)	125	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
	150	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
	175	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—
	200	$1.50 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	—	—	—

A.4. Standard Deviation of our Experiments

Table VIII. Standard Deviation of the Running Time of Our Experiments

		running time (in sec)			
dataset	k	StreamKM++	StreamLS	k-means++	k-means
Spambase	10	0.29	—	1.5	3.33
	20	1.09	—	3.88	6.36
	30	1.52	—	11.27	17.61
	40	6.56	—	6.97	26.95
	50	6.59	—	12.83	68.1
Intrusion	10	0.68	—	40.81	58.84
	20	3.22	—	98.11	499.7
	30	6.07	—	1, 263.44	345.6
	40	24.91	—	563.20	1, 306.2
	50	31.58	—	706.00	1, 190.78
Covertype	10	0.88	2.43	2, 295.85	—
	20	6.93	18.18	1, 249.18	—
	30	14.15	52.14	9, 653.06	—
	40	14.02	97.64	6, 838.93	—
	50	39.28	123.28	12, 231.98	—
Tower	20	0.58	14.11	1, 594.76	—
	40	1.79	50.83	2, 085.12	—
	60	3.96	58.27	3, 656.87	—
	80	7.95	122.65	5, 162.60	—
	100	11.34	315.31	1, 795.07	—
Census 1990	10	2.04	9.08	—	—
	20	5.16	54.3	—	—
	30	5.38	98.03	—	—
	40	23.31	193.00	—	—
	50	17.43	533.39	—	—
BigCross	15	10.49	93.6	—	—
	20	11.49	162.44	—	—
	25	15.69	226.38	—	—
	30	16.66	200.68	—	—
Normdata	100	0.07	1.22	—	—
$(m = 500)$	125	0.05	1.14	—	—
	150	0.05	2.19	—	—
	175	0.03	2.89	—	—
	200	0.03	4.05	—	—
Normdata	100	0.06	0.6	—	—
$(m = 1{,}000)$	125	0.06	1.32	—	—
	150	0.04	2.56	—	—
	175	0.08	3.96	—	—
	200	0.2	2.41	—	—

Table IX. Standard Deviation of the Cost of Our Experiments

		cost			
dataset	k	StreamKM++	StreamLS	k-means++	k-means
Spambase	10	$2.05 \cdot 10^6$	—	$9.57 \cdot 10^6$	$1.06 \cdot 10^6$
	20	$6.49 \cdot 10^5$	—	$1.73 \cdot 10^6$	$8.78 \cdot 10^4$
	30	$3.14 \cdot 10^5$	—	$9.51 \cdot 10^5$	$8.81 \cdot 10^4$
	40	$1.93 \cdot 10^5$	—	$5.31 \cdot 10^5$	$3.42 \cdot 10^6$
	50	$1.49 \cdot 10^5$	—	$2.47 \cdot 10^5$	$2.91 \cdot 10^6$
Intrusion	10	$1.39 \cdot 10^{12}$	—	$6.61 \cdot 10^{12}$	$3.09 \cdot 10^{11}$
	20	$8.54 \cdot 10^{10}$	—	$3.70 \cdot 10^{11}$	$8.20 \cdot 10^9$
	30	$3.13 \cdot 10^{10}$	—	$46.85 \cdot 10^{10}$	$2.54 \cdot 10^{10}$
	40	$7.03 \cdot 10^9$	—	$3.25 \cdot 10^{10}$	$1.53 \cdot 10^8$
	50	$6.01 \cdot 10^9$	—	$1.61 \cdot 10^{10}$	$6.82 \cdot 10^8$
Covertype	10	$2.47 \cdot 10^9$	$2.70 \cdot 10^{10}$	$3.63 \cdot 10^9$	—
	20	$1.08 \cdot 10^9$	$1.03 \cdot 10^{10}$	$9.17 \cdot 10^8$	—
	30	$1.49 \cdot 10^9$	$6.61 \cdot 10^9$	$6.12 \cdot 10^8$	—
	40	$8.38 \cdot 10^8$	$5.63 \cdot 10^9$	$6.64 \cdot 10^8$	—
	50	$5.68 \cdot 10^8$	$3.90 \cdot 10^9$	$2.92 \cdot 10^8$	—
Tower	20	$7.31 \cdot 10^6$	$2.71 \cdot 10^7$	$4.39 \cdot 10^7$	—
	40	$1.85 \cdot 10^6$	$1.65 \cdot 10^7$	$4.37 \cdot 10^6$	—
	60	$1.52 \cdot 10^6$	$1.55 \cdot 10^7$	$1.61 \cdot 10^6$	—
	80	$1.03 \cdot 10^6$	$9.63 \cdot 10^6$	$1.54 \cdot 10^6$	—
	100	$7.73 \cdot 10^5$	$1.03 \cdot 10^7$	$1.17 \cdot 10^6$	—
Census 1990	10	$5.02 \cdot 10^6$	$1.45 \cdot 10^5$	—	—
	20	$3.66 \cdot 10^6$	$3.14 \cdot 10^6$	—	—
	30	$1.61 \cdot 10^6$	$9.34 \cdot 10^5$	—	—
	40	$1.21 \cdot 10^6$	$8.13 \cdot 10^5$	—	—
	50	$1.01 \cdot 10^6$	$6.80 \cdot 10^5$	—	—
BigCross	15	$3.22 \cdot 10^{10}$	$1.75 \cdot 10^{11}$	—	—
	20	$2.46 \cdot 10^{10}$	$3.36 \cdot 10^{11}$	—	—
	25	$1.86 \cdot 10^{10}$	$1.76 \cdot 10^{11}$	—	—
	30	$1.94 \cdot 10^{10}$	$1.29 \cdot 10^{11}$	—	—
Normdata	100	0	0	—	—
($m = 500$)	125	0	0	—	—
	150	0	0	—	—
	175	0	0	—	—
	200	0	0	—	—
Normdata	100	0	0	—	—
($m = 1{,}000$)	125	0	0	—	—
	150	0	0	—	—
	175	0	0	—	—
	200	0	0	—	—

REFERENCES

Agarwal, P. K., Har-Peled, S., and Varadarajan, K. R. 2004. Approximating extent measures of points. *J. ACM 51,* 4, 606–635.

Aggarwal, A., Deshpande, A., and Kannan, R. 2009. Adaptive sampling for k-means clustering. In *Proceedings of the 12th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX '10)*. Springer, 15–28.

Ailon, N., Jaiswal, R., and Monteleoni, C. 2009. Streaming k-means approximation. In *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, Eds., 10–18.

Aloise, D., Deshpande, A., Hansen, P., and Popat, P. 2009. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learn. 75,* 2, 245–248.

ARTHUR, D., MANTHEY, B., AND RÖGLIN, H. 2009. *k*-means has polynomial smoothed complexity. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS '09)*. IEEE Computer Society, 405–414.

ARTHUR, D. AND VASSILVITSKII, S. 2007. `k-means++`: the advantages of careful seeding. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*. SIAM, 1027–1035.

ASUNCION, A. AND NEWMAN, D. J. 2007. UCI machine learning repository. University of California, Irvine, School of Information and Computer Sciences. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

BENTLEY, J. L. AND SAXE, J. B. 1980. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algor. 1,* 4, 301–358.

CHEN, K. 2009. On coresets for *k*-median and *k*-means clustering in metric and Euclidean spaces and their applications. *SIAM J. Comput. 39,* 3, 923–947.

DASGUPTA, S. 2008. The hardness of *k*-means clustering. Tech. rep. CS2008-0916, University of California.

FELDMAN, D., MONEMIZADEH, M., AND SOHLER, C. 2007. A PTAS for *k*-means clustering based on weak coresets. In *Proceedings of the 23rd ACM Symposium on Computational Geometry (SCG '07)*. ACM, 11–18.

FORGY, E. W. 1965. Cluster analysis of multivariate data: Efficiency versus interpretability of classifications. *Biometrics 21*, 768–780.

FRAHLING, G. AND SOHLER, C. 2005. Coresets in dynamic geometric data streams. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC '05)*. ACM, 209–217.

GUHA, S., MEYERSON, A., MISHRA, N., MOTWANI, R., AND O'CALLAGHAN, L. 2003. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Engin. 15,* 3, 515–528.

GUHA, S., MISHRA, N., MOTWANI, R., AND O'CALLAGHAN, L. 2000. Clustering data streams. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS '00)*. IEEE Computer Society, 359–366.

HAR-PELED, S. AND MAZUMDAR, S. 2004. On coresets for *k*-means and *k*-median clustering. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC '04)*. ACM, 291–300.

KANUNGO, T., MOUNT, D. M., NETANYAHU, N. S., PIATKO, C. D., SILVERMAN, R., AND WU, A. Y. 2004. A local search approximation algorithm for *k*-means clustering. *Computat. Geometry 28,* 2-3, 89–112.

KUMAR, A. AND KANNAN, R. 2010. Clustering with spectral norm and the *k*-means algorithm. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science (FOCS '10)*. IEEE Computer Society, 299–308.

LLOYD, S. P. 1982. Least squares quantization in PCM. *IEEE Trans. Infor. Theory 28,* 2, 129–137.

MACQUEEN, J. B. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. Vol. 1, University of California Press, 281–297.

MATOUSEK, J. 2000. On approximate geometric *k*-clustering. *Discrete Computat. Geometry 24,* 1, 61–84.

MATSUMOTO, M. AND NISHIMURA, T. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul. 8,* 1, 3–30.

O'CALLAGHAN, L., MEYERSON, A., MOTWANI, R., MISHRA, N., AND GUHA, S. 2002. Streaming-data algorithms for high-quality clustering. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*. IEEE Computer Society, 685–696.

SELIM, S. Z. AND ISMAIL, M. A. 1984. *k*-means-type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Trans. Pattern Anal. Mach. Intell. 6,* 1, 81–87.

VATTANI, A. 2009. *k*-means requires exponentially many iterations even in the plane. In *Proceedings of the 25th ACM Symposium on Computational Geometry (SoCG '09)*. ACM, 324–332.

ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. 1996. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*. ACM, 103–114.

Received December 2010; revised September 2011; accepted February 2012