StreamKM++: A Clustering Algorithms for Data Streams.

Conference Paper *in* Journal of Experimental Algorithmics · January 2010

DOI: 10.1145/2133803.2184450 · Source: DBLP

6 authors, including:

Christiane Lammersen
University of Bonn
11 PUBLICATIONS **158** CITATIONS

SEE PROFILE

Marcus Märtens
European Space Agency
25 PUBLICATIONS **238** CITATIONS

SEE PROFILE

StreamKM++: A Clustering Algorithm for Data Streams*

Marcel R. Ackermann† Christiane Lammersen‡ Marcus Märtens†
Christoph Raupach† Christian Sohler‡ Kamil Swierkot†

Abstract

We develop a new k-means clustering algorithm for data streams, which we call StreamKM++. Our algorithm computes a small weighted sample of the data stream and solves the problem on the sample using the k-means++ algorithm [1]. To compute the small sample, we propose two new techniques. First, we use a non-uniform 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 low dependency on the dimensionality of the data. Second, we propose a new data structure which we call a coreset tree. The use of these coreset trees significantly speeds up the time necessary for the non-uniform sampling during our coreset construction.

We compare our algorithm experimentally with two well-known streaming implementations (BIRCH [16] and StreamLS [4, 9]). 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). 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.

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, which are to be clustered, are very large and 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 harddisks, where a streaming access is orders of magnitude faster than random access.

One of the most widely used clustering algorithms is Lloyd's algorithm (sometimes also called *the k-means algorithm*) [6, 12, 13]. This 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 and (2) given a cluster, the best center of the cluster is the center of gravity (mean) of its points. Lloyd's algorithm applies these two local optimization steps repeatedly to the current solution, until no more improvement is possible. It is known that the algorithm converges to a local optimum [15] and no approximation guarantee can be given. Recently, Arthur and Vassilvitskii developed the k-means++ algorithm [1], which is a seeding procedure for Lloyd's k-means algorithm that guarantees a solution with certain quality and gives good practical results. However, the k-means++ algorithm (as well as Lloyd's algorithm) needs random access on the input data and is not suited for data streams.

In this paper, 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 [16]. BIRCH is a heuristic that computes a pre-clustering of the data into so-called clustering features and then clusters this pre-clustering using an agglomerative (bottom-up) clustering algorithm. Another well-known algorithm is StreamLS [4, 9], which partitions the input stream into chunks and computes for each chunk a clustering using a local search algorithm from [10]. 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 [5, 7, 8, 10, 11]. Many of these algorithms are based on applying the merge-and-reduce pro-

*Partially supported by Deutsche Forschungsgemeinschaft (DFG), grants Bl 314/6-1 and So 514/1-2.

†University of Paderborn, Department of Computer Science, 33095 Paderborn, Germany

‡TU Dortmund, Department of Computer Science, 44221 Dortmund, Germany

cedure from [11] to obtain a small coreset [3] of the data stream, i.e., a small weighted point set that approximates the points from the data stream with respect to the k-means clustering problem.

1.2 Our Contribution. We develop a new algorithm for k-means clustering in the data streaming model, which we call StreamKM++. Our streaming algorithm maintains a small sketch of the input using the merge-and-reduce technique [11], i.e., the data is organized in a small number of samples, each representing $2^i m$ input points (for some integer i and a fixed value m). Everytime when two samples representing the same number of input points exist we take the union (merge) and create a new sample (reduce).

For the reduce step, we propose a new coreset construction. Here, we focus on giving a construction that is suitable for high-dimensional data. Existing coreset constructions based on grid-computations [11, 8] 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 approach efficiently, we develop a new data structure, which we call *the coreset tree*.

We compare our algorithm experimentally with BIRCH and StreamLS, which are both frequently used to cluster data streams, as well as with the non-streaming 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++ does, but is a factor of 40 faster than algorithm 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.

2 Preliminaries

Let $\|\cdot\|$ denote the ℓ_2-norm on $\mathbb{R}^d$. By $\mathrm{d}(x,y) = \|x-y\|$ we denote the Euclidean distance and by $\mathrm{d}^2(x,y) = \|x-y\|^2$ the squared Euclidean distance of $x, y \in \mathbb{R}^d$. We use $\mathrm{d}(x,C) = \min_{c\in C} \mathrm{d}(x,c)$, $\mathrm{d}^2(x,C) = \min_{c\in C} \mathrm{d}^2(x,c)$, and $\mathrm{cost}(P,C) = \sum_{x\in P} \mathrm{d}^2(x,C)$ for $C, P \subset \mathbb{R}^d$. Analogously, for a weighted subset $S \subset \mathbb{R}^d$ with weight function $w : S \to \mathbb{R}_{\geq 0}$, we use $\mathrm{cost}_w(S,C) = \sum_{y\in S} w(y)\,\mathrm{d}^2(y,C)$. The *Euclidean k-means problem* is defined as follows.

Problem 1. *Given an input set $P \subset \mathbb{R}^d$ with $|P| = n$ and $k \in \mathbb{N}$, find a set $C \subset \mathbb{R}^d$ with $|C| = k$ that minimizes* $\mathrm{cost}(P,C)$.

Furthermore, by

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

we denote the cost of an optimal Euclidean k-means clustering of P.

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 2.1. *Let $k \in \mathbb{N}$ and $\varepsilon \leq 1$. A weighted multiset $S \subset \mathbb{R}^d$ with positive weight function $w : s \to \mathbb{R}_{\geq 0}$ and $\sum_{y\in S} w(y) = |P|$ is called (k,ε)-coreset of P iff for each $C \subset \mathbb{R}^d$ of size $|C| = k$ we have*

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

Our clustering algorithm maintains a small coreset in the data streaming model. In this model, the input is a sequence of points. Due to the long length of the sequence, algorithms are only allowed to perform one sequential scan over the data and to use local memory that is merely polylogarithmic in the size of the input stream.

3 Coreset Construction

Our coreset construction is based on the idea of the k-means++ seeding procedure from [1]. 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, like the grid-based methods from [11, 8], for instance.

In the following, let $P \subset \mathbb{R}^d$ with $|P| = n$. The k-MEANS++ seeding from [1] is an iterative process as follows:

1. Choose an initial point $q_1 \in P$ uniformly at random.
2. Let S be a set of points already chosen from P. Then, each element $p \in P$ is chosen with probability $\frac{\mathrm{d}^2(p,S)}{\mathrm{cost}(P,S)}$ as next element of S.
3. Repeat step 2 until S contains the desired number of points.

We say S is chosen *at random according to* d^2.

For an arbitrary fixed integer m our coreset construction is as follows. First, we chose 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 which are closest to q_i (breaking ties arbitrarily). Using weight function $w : S \to \mathbb{R}_{\geq 0}$ with $w(q_i) = |Q_i|$, we obtain the weighted set S as our coreset. Note that this construction is rather easy to implement and its running time has a merely linear dependency on the dimension d.

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 giving a rigorous proof to the fact that at least in low dimensional spaces, our construction indeed leads to small (k, ε)-coresets. Please note that there is no reason to assume that the size bound from Theorem 3.1 is tight.

THEOREM 3.1. *If* $m = \Theta\Big(\frac{k \log n}{\delta^{d/2} \varepsilon^d} \log^{d/2}\big(\frac{k \log n}{\delta^{d/2} \varepsilon^d}\big)\Big)$, *then with probability at least* $1 - \delta$ *the weighted multiset* S *is a* $(k, 6\varepsilon)$*-coreset of* P.

Proof. First, we need the following two lemmas. The first lemma is due to [1]. The proof of the second lemma can be found in Appendix A.

LEMMA 3.1. *Let* $S \subseteq P$ *be a set of* m *points chosen at random according to* d^2. *Then we have* $\mathrm{E}[\mathrm{cost}(P, S)] \leq 8(2 + \ln m)\mathrm{opt}_m(P)$.

LEMMA 3.2. *Let* $\gamma > 0$. *If* $m \geq \big(\frac{9d}{\gamma}\big)^{\frac{d}{2}} k \lceil \log(n) + 2 \rceil$, *then* $\mathrm{opt}_m(P) \leq \gamma\, \mathrm{opt}_k(P)$.

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. By the triangle inequality, we have $|\mathrm{cost}(P, C) - \mathrm{cost}_w(S, C)| \leq \sum_{p \in P} \big|\mathrm{d}^2(p, C) - \mathrm{d}^2(q_p, C)\big|$. By $P' = \big\{p \in P \,\big|\, \mathrm{d}(p, q_p) \leq \varepsilon \mathrm{d}(p, C)\big\}$ and $P'' = P \setminus P'$ we define a partition of P. Using the triangle inequality of the Euclidean distance, we obtain Proposition 3.1 and Proposition 3.2 below. The proofs of these propositions can be found in Appendix B and C.

PROPOSITION 3.1. *If* $p \in P'$, *then*

$$\big|\mathrm{d}^2(p, C) - \mathrm{d}^2(q_p, C)\big| \leq 3\varepsilon \mathrm{d}^2(p, C) \ .$$

PROPOSITION 3.2. *If* $p \in P''$, *then*

$$\big|\mathrm{d}^2(p, C) - \mathrm{d}^2(q_p, C)\big| \leq \frac{3}{\varepsilon} \mathrm{d}^2(p, q_p) \ .$$

Using Proposition 3.1 and 3.2, we find

$$\begin{aligned}
& \big|\mathrm{cost}(P, C) - \mathrm{cost}_w(S, C)\big| \\
\leq\ & \sum_{p \in P'} \big|\mathrm{d}^2(p, C) - \mathrm{d}^2(q_p, C)\big| \\
& + \sum_{p \in P''} \big|\mathrm{d}^2(p, C) - \mathrm{d}^2(q_p, C)\big| \\
\leq\ & 3\varepsilon \sum_{p \in P'} \mathrm{d}^2(p, C) + \frac{3}{\varepsilon} \sum_{p \in P''} \mathrm{d}^2(p, q_p) \\
\leq\ & 3\varepsilon \mathrm{cost}(P, C) + \frac{3}{\varepsilon} \mathrm{cost}(P, S) \, .
\end{aligned}$$

Using Lemma 3.1 and Markov's inequality, we obtain $\mathrm{cost}(P, S) \leq \frac{8}{\delta}(2 + \ln m)\mathrm{opt}_m(P)$ with probability at least $1 - \delta$. Hence, using Lemma 3.2 with $\gamma = \frac{\varepsilon^2 \delta}{8(2 + \ln m)}$, we have with high probability

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

for $m = \Theta\Big(\frac{k \log n}{\delta^{d/2} \varepsilon^d} \log^{d/2}\big(\frac{k \log n}{\delta^{d/2} \varepsilon^d}\big)\Big)$. Therefore, the theorem follows. □

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 data structure called *coreset*

tree which speeds up this computation. The advantage of the coreset tree is that it enables 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 $\Theta(\log k)$), we merely need time $\Theta(dn \log m)$ to compute all m coreset points. This intuition is supported by our empirical evaluation, 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++ seeding.

In the following, we explain the construction of the coreset tree in more detail. A description in pseudocode is given by Figure 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, an integer size(v), and a value cost(v). Here, point set P_v is the cluster associated with node v. Note that the set P_v only has 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. The representative q_v of a node v is obtained by sampling according to d^2 from P_v. 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, the attribute size(v) of a node v denotes the number of points in set P_v. For leaf nodes, the attribute cost(v) equals cost(P_v, q_v), which is the sum of squared distances over all points in P_v to q_v. The value cost(v) of an inner node v is defined as the sum of the cost of its children.

4.2 Construction of the Coreset Tree. To simplify descriptions, 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 choosing uniformly at random one point from P. Now, let us assume that our current tree has i leaf nodes $1, 2, \ldots, i$ and the corresponding sample points are $q_1, q_2, \ldots, q_i$. We obtain the next sample point q_{i+1}, a new cluster 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.
2. Choose a new sample point denoted by q_{i+1} from the subset P_ℓ at random.
3. Based on q_ℓ and q_{i+1}, split P_ℓ into two subclusters and create two child nodes of ℓ in T.

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 costs. More precisely, each child node v of the current node u is chosen with probability $\frac{\mathrm{cost}(v)}{\mathrm{cost}(u)}$. 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 contained in ℓ, and let P_ℓ be the subset of P corresponding to leaf ℓ.

In the second step, we choose a new sample point from P_ℓ at random according to d^2, i.e., each $p \in P_\ell$ is chosen with probability $\frac{\mathrm{d}^2(p,q_\ell)}{\mathrm{cost}(P_\ell,q_\ell)}$. In doing so, we sample each point from P with probability proportional to its distance to the sample points of the clustering induced by the partition of the leaf nodes and their sample points. That is, we are using the same distribution as the k-MEANS++ algorithm does with the exception that the partition is determined by the coreset tree rather than by assigning each point to the nearest sample point.

In the third step, we create two new leaf nodes ℓ_1 and ℓ_2 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 we store our new sample point q_{i+1}. We partition P_ℓ into two subsets $P_{\ell_1} = \{p \in P_\ell \mid d(p, q_\ell) < d(p, q_{i+1})\}$ and $P_{\ell_2} = P_\ell \setminus P_{\ell_1}$ and associate them with the corresponding nodes. Node ℓ becomes the parent node of the two new leaf nodes ℓ_1 and ℓ_2. We determine size and cost attributes for the nodes ℓ_1 and ℓ_2 as described above and update the cost of ℓ according to this. This update is propagated upwards, until we reach the root of the tree.

TreeCoreset(P, m):

1 choose q_1 uniformly at random from P
2 $root \leftarrow$ node with $q_{root} = q_1$, size$(root) = |P|$, cost$(root) =$ cost(P, q_1)
3 $S \leftarrow \{q_1\}$
4 **for** $i \leftarrow 2$ **to** m **do**
5 start at $root$, iteratively select random child node until a leaf ℓ is chosen
6 choose q_i according to d^2 from P_ℓ
7 $S \leftarrow S \cup \{q_i\}$
8 create two child nodes ℓ_1, ℓ_2 of ℓ and update size(ℓ) and cost(ℓ)
9 propagate update upwards to node $root$

Figure 1: Algorithm TreeCoreset

InsertPoint(p):

1 put p into B_0
2 **if** B_0 is full **then**
3 create empty bucket Q
4 move points from B_0 to Q
5 empty B_0
6 $i \leftarrow 1$
7 **while** B_i is not empty **do**
8 merge points from B_i and Q, store merged points in Q
9 empty B_i
10 $i \leftarrow i + 1$
11 move points from Q to B_i

Figure 2: Algorithm InsertPoint

4.3 The Coreset. Once we have constructed a coreset tree with m leaf nodes, let $q_1, q_2, \ldots, q_m$ denote the points associated with the leaf nodes. We obtain coreset $S = \{q_1, q_2, \ldots, q_m\}$ where the weight of q_i is given by the number of points that are associated with the leaf node of q_i.

5 The Algorithm

Now, we are able to 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 [11]. This streaming method is described in detail in the subsection below. For the reduce step, we employ our new coreset construction, using the coreset trees as given in Section 4. After that, a k-clustering can be obtained at any point of time by running any k-means algorithm on the coreset of size m. Note that since the size of the coreset is much smaller than (or even independent of) the size of the data stream, we are no longer prohibited from algorithms that require random access on their input data. In our implementation, we run the k-means++ algorithm from [1] on our coreset five times independently and choose the best clustering result obtained this way. We call the resulting algorithm StreamKM++.

5.1 The Streaming Method. In order to maintain a small coreset for all points in the data stream, we use the merge-and-reduce method from [11]. For a data stream containing n points, the algorithm maintains $L = \lceil \log_2(\frac{n}{m}) + 2 \rceil$ 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 point of 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 finished. However, if bucket B_1 already contains m points, we compute a new coreset Q of size m from the union of the $2m$ points stored in B_0 and B_1 by using the coreset construction described above. Now, both buckets B_0 and B_1 are emptied and the m points from coreset Q are moved into bucket B_2 (unless, of course, bucket B_2 is also full in which case the process is repeated). Algorithm InsertPoint for inserting a point from the data stream into the buckets is given in Figure 2.

At any point of 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 $m\lceil \log_2(\frac{n}{m}) + 2 \rceil$ points stored in all the buckets $B_0, B_1, \ldots, B_{L-1}$ by using the coreset tree construction and obtain the desired coreset of size m.

5.2 Running Time and Memory Usage. 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, $\lceil \frac{n}{m} \rceil$ such steps are needed. The amortized running time of all merge-and-reduce steps is at most $\mathcal{O}(dnm)$. The final merge of all buckets to obtain a coreset of size m can be done in time $\mathcal{O}(dm^2 \log \frac{n}{m})$. Finally, algorithm k-means++ is executed five times on an input set of size m, using time $\Theta(dkm)$ per iteration. Obviously, algorithm StreamKM++ uses at most $\Theta(dm \log \frac{n}{m})$

memory units. Hence, we obtain a low dependency on the dimension d and 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 next subsection. The computation on the biggest dataset, which is denoted by *BigCross*, was performed on a DELL Optiplex 620 machine with 3 GHz Pentium D CPU and 2 GB main memory, using Linux 2.6.9 kernel. For all remaining datasets, the computation was performed on a DELL Optiplex 620 machine with 3 GHz Pentium D CPU and 4 GB 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 [16] and with a streaming variant of the local search algorithm given in [4, 9] which we call StreamLS. On the smaller datasets, we also compared our algorithm with a classical implementation of Lloyd's k-means algorithm [12], using initial seeds either uniformly at random (algorithm k-means) or according to the non-uniform seeding from [1] (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, to be referred as costs of the clustering.

6.1 Datasets. Since synthetical datasets (like Gaussian distributed points near some uniformly distributed centers in $\mathbb{R}^d$) are typically easy to cluster, we use real-world datasets to obtain practically relevant results. Our main source for data was the UCI Machine Learning Repository [2] (datasets *Covertype*[2], *Census 1990*, *Intrusion*[3], and *Spambase*) as well as dataset *Tower*[4] from [8]. To test our algorithm on really huge datasets, we created the cartesian product of the *Tower* and *Covertype* dataset. We used a 1.5 GB sized subset of the cartesian product with 11 620 300 data points at 57 attributes, referred as the *BigCross* dataset in this paper. The size and dimensionality of the datasets is summarized in Table 1.

	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

Table 1: Overview of the datasets

6.2 Parameters of the Algorithms. In the following, we describe the experimental environment for the two streaming algorithms BIRCH and StreamLS. For algorithm BIRCH we set all parameters as recommended by the authors of BIRCH except for the memory settings. Like the authors in [9], we observed that the CF-Tree had less leaves than it was allowed to use. 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 Appendix H. Second, for algorithm StreamLS the size of the data chunks used by the streaming method from [4] is set equal to the coreset size $m = 200k$ of algorithm StreamKM++. 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. Due to the randomized[5] nature of the algorithms StreamKM++ and StreamLS, ten experiments were conducted for both algorithms and for each fixed k. For BIRCH, a single run was used, since it is a deterministic algorithm. We conducted the experiments on the four larger datasets, i.e., the datasets *Covertype*, *Tower*, *Census 1990*, and *BigCross*. The average running times and cost of the clusterings are summarized in Figure 3. The interested reader can find the concrete values of all experiments in the appendix.

In our experiments, algorithm BIRCH had the best running time of all algorithms. However, this

[1] The sourcecode, 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] Copyright by Jock A. Blackard, Colorado State University

[3] *Intrusion* dataset is part of the kddcup99 dataset.

[4] *Tower* dataset was contributed by Gereon Frahling and is available for free download at: `http://homepages.uni-paderborn.de/frahling/coremeans.html`

[5] We used the Mersenne Twister PRNG [14].

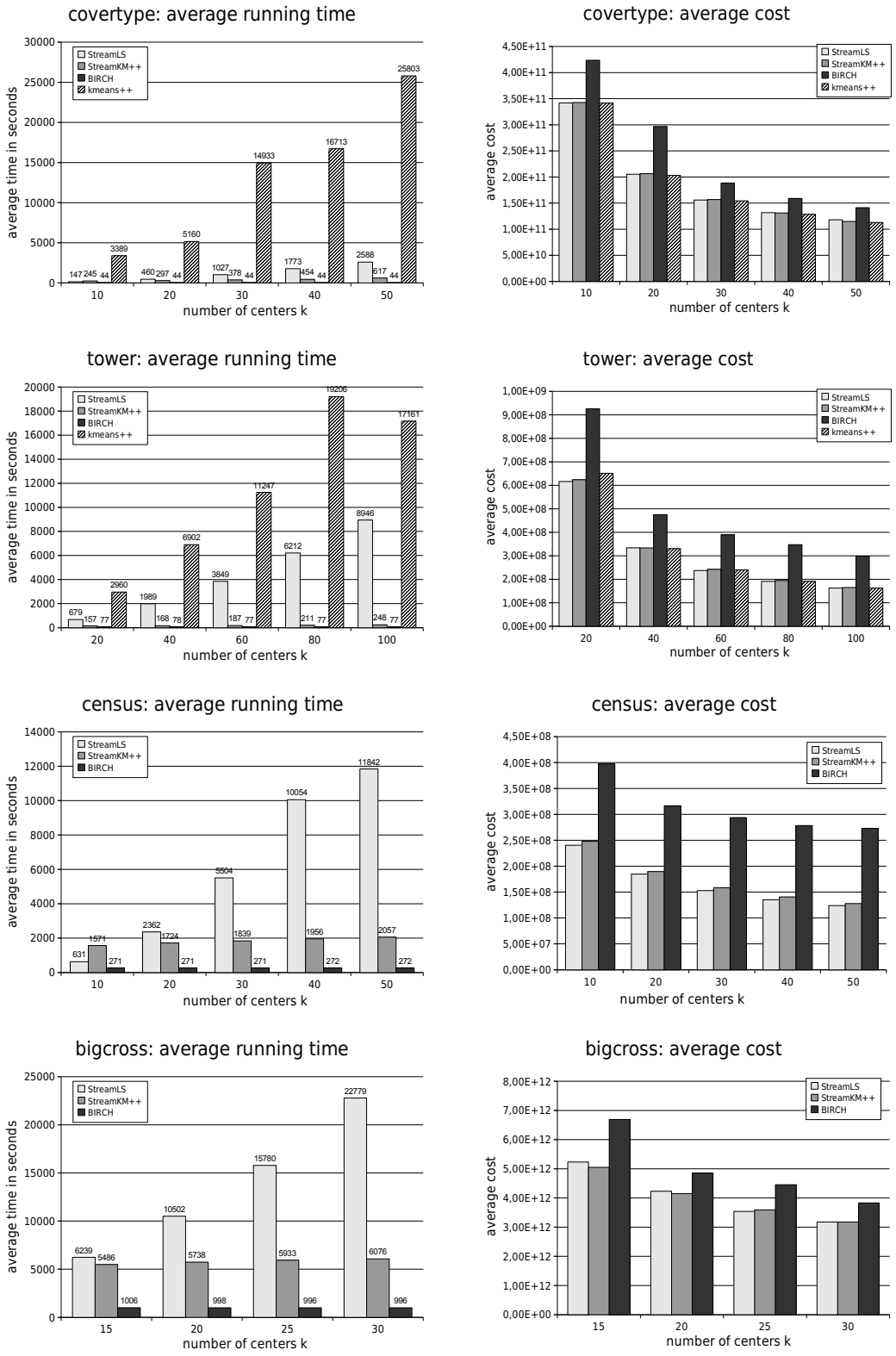

Figure 3: Experimental results for *Covertype*, *Tower*, *Census 1990*, and *BigCross* datasets

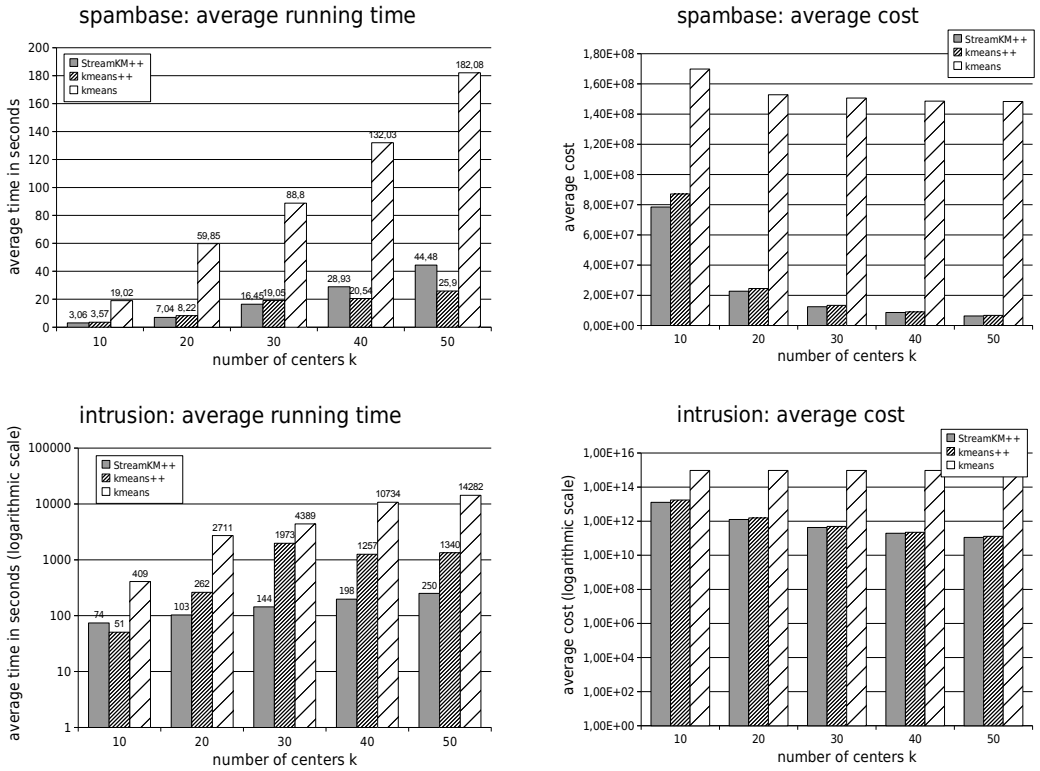

Figure 4: Experimental results for *Spambase* and *Intrusion* datasets

comes at the cost 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 of adjusting parameters manually 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 (see Table 2, for a complete overview of the standard deviations of our experiments see Appendix G). 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 $k \leq 10$ centers StreamLS is sometimes faster than our algorithm, for a larger number of centers our algorithm easily outperforms StreamLS. For $k = 100$ centers on the dataset *Tower*, the running times of both algorithms differed by a factor of about 30.

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, in particular, 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 non-streaming k-means algorithms. Because of their popularity, we have chosen the k-means algorithm and the recent k-means++ as competitor. These algorithms are designed to work in a classical non-streaming 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 ten experiments. The results of these experiments are summarized in Figure 4 (and, in part, in Figure 3). Please note that the results for dataset *Intrusion* are on a logarithmic scale. The concrete values of all experiments can be found in the appendix.

As expected, k-means++ is clearly superior to the

$k = 20$	running time			cost		
	StreamKM++	StreamLS	k-means++	StreamKM++	StreamLS	k-means++
Spambase	1.09	-	3.88	$6.49 \cdot 10^5$	-	$1.73 \cdot 10^6$
Intrusion	3.22	-	98.11	$8.54 \cdot 10^{10}$	-	$3.70 \cdot 10^{11}$
Covertype	6.93	18.18	1249.18	$1.08 \cdot 10^9$	$1.03 \cdot 10^{10}$	$9.17 \cdot 10^8$
Tower	0.58	14.11	1594.76	$7.31 \cdot 10^6$	$2.71 \cdot 10^7$	$4.39 \cdot 10^7$
Census 1990	5.16	54.30	-	$3.66 \cdot 10^6$	$3.14 \cdot 10^6$	-
BigCross	11.49	162.44	-	$2.46 \cdot 10^{10}$	$3.36 \cdot 10^{11}$	-

Table 2: Standard deviation for $k = 20$

classical k-means algorithm 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.

References

[1] D. Arthur and S. Vassilvitskii. `k-means++`: the advantages of careful seeding. *Proc. 18th ACM-SIAM Sympos. Discrete Algorithms*, pp. 1027–1035, 2007.

[2] A. Asuncion and D. J. Newman. UCI machine learning repository, 2007. University of California, Irvine, School of Information and Computer Sciences, `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[3] M. Badoiu, S. Har-Peled, and P. Indyk. Approximate Clustering via Coresets. *Proc. 34th ACM Sympos. Theory Comput.*, pp. 250–257, 2002.

[4] L. O'Callaghan, A. Meyerson, R. Motwani, N. Mishra, S. Guha. Streaming-Data Algorithms for High-Quality Clustering. ICDE 2002.

[5] K.Chen. On k-Median Clustering in High Dimensions. *Proc. 17th ACM-SIAM Sympos. Discrete Algorithms*, pp. 1177–1185, 2006.

[6] E. Forgey. Cluster Analysis of Multivariate Data: Efficiency vs. Interpretability of Classification. *Biometrics*, 21:768, 1965.

[7] D. Feldman, M. Monemizadeh, and C. Sohler. A PTAS for k-means clustering based on weak coresets. *Proc. ACM Sympos. Comput. Geom.*, pp. 11–18, 2007.

[8] G. Frahling and C. Sohler. Coresets in Dynamic Geometric Data Streams. *Proc. 37th ACM Sympos. Theory Comput.*, pp. 209–217, 2005.

[9] S. Guha, A. Meyerson, N. Mishra, R. Motwani, L. O'Callaghan. Clustering Data Streams: Theory and Practice. *IEEE Trans. Knowl. Data Eng.*, 15(3): 515–528, 2003.

[10] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. *Proc. IEEE Sympos. Found. Comput. Sci.*, pp. 359–366, 2000.

[11] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. *Proc. 36th ACM Sympos. Theory Comput.*, pp. 291–300, 2004.

[12] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28: 129–137, 1982.

[13] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. *Proc. 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1: 281–296, 1967.

[14] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. on Modeling and Computer Simulations*, 1998.

[15] S. Selim and M. Ismail. k-Means-Type Algorithms: A Generalized Convergence Theorem and Characterizations of Local Optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6: 81–87, 1984.

[16] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A new data clustering algorithm and its applications. *Journal of Data Mining and Knowledge Discovery*, 1(2): 141-182, 1997.

A Proof of Lemma 3.2

Let $C = \{c_1, \ldots, c_k\}$ be an optimal solution to the Euclidean k-means problem for P with $|P| = n$, i.e., $\mathrm{cost}(P, C) = opt_k(P)$. We consider an exponential grid around each c_i. The construction of this grid follows the one from [11].

Let $R = \frac{1}{n} opt_k(P)$ and, for $j = 1, 2, \ldots, \lceil \log(n)+2 \rceil$ and for each c_i, let Q_{ij} denote the axis-parallel square centered at c_i with side length $\sqrt{2^j R}$. We define recursively $U_{i0} = Q_{i0}$ and $U_{ij} = Q_{ij} \setminus Q_{i,j-1}$ for $j \geq 1$. Obviously, each $p \in P$ is contained within a U_{ij}, since otherwise we would have

$$\begin{aligned} \mathrm{d}^2(p, C) &> \frac{1}{4} 2^{\lceil \log(n)+2 \rceil} R \\ &\geq opt_k(P) \ , \end{aligned}$$

which is a contradiction.

For each i, j individually, we partition U_{ij} into small grid cells with side length $\sqrt{\frac{\gamma}{9d} 2^j R}$. For each grid cell which 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 $(\frac{9d}{\gamma})^{\frac{d}{2}} k \lceil \log(n)+2 \rceil$ grid cells and, hence, $|G| \leq m$.

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

$$\begin{aligned} opt_m(P) &\leq \mathrm{cost}(P, G) \\ &\leq \sum_{p \in P} \mathrm{d}^2\big(p, g_p\big) \ . \end{aligned}$$

Observe that, for $p \in U_{i0}$, we have $\mathrm{d}^2\big(p, g_p\big) \leq \frac{\gamma}{9} R$. On the other hand, for $p \in U_{ij}$ with $j \geq 1$, we find $\mathrm{d}^2(p, C) \geq 2^{j-3} R$. Therefore, in this case, we have

$$\mathrm{d}^2\big(p, g_p\big) \leq \frac{\gamma}{9} 2^j R \leq \frac{8\gamma}{9} \mathrm{d}^2(p, C) \ .$$

We obtain

$$\begin{aligned} opt_m(P) &\leq n\frac{\gamma}{9} R + \frac{8\gamma}{9} \sum_{p \in P} \mathrm{d}^2(p, C) \\ &= \frac{\gamma}{9} opt_k(P) + \frac{8\gamma}{9} opt_k(P) = \gamma opt_k(P). \end{aligned}$$

□

B Proof of Proposition 3.1

Assume $\mathrm{d}(p, C) \leq \mathrm{d}(q_p, C)$. Let c_p denote the element from C closest to p. By triangle inequality, we have

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

Hence, for the squared distances, we obtain

$$\mathrm{d}^2(q_p, C) \leq (1+\varepsilon)^2 \mathrm{d}^2(p, C) \leq (1+3\varepsilon)\mathrm{d}^2(p, C) \ .$$

and we have $\mathrm{d}^2(q_p, C) - \mathrm{d}^2(p, C) \leq 3\varepsilon \mathrm{d}^2(p, C)$.

Now assume $\mathrm{d}(q_p, C) < \mathrm{d}(p, C)$. Let c_s denote the element from C closest to q_p. Again, by triangle inequality, we have

$$\begin{aligned} \mathrm{d}(p, C) &\leq \mathrm{d}(p, c_s) \\ &\leq \mathrm{d}(q_p, c_s) + \mathrm{d}(p, q_p) \\ &\leq \mathrm{d}(q_p, C) + \varepsilon \mathrm{d}(p, C) \ , \end{aligned}$$

since $p \in P'$. Therefore, $(1-\varepsilon)\mathrm{d}(p, C) \leq \mathrm{d}(q_p, C)$. For the squared distances, we obtain

$$\begin{aligned} \mathrm{d}^2(q_p, C) &\geq (1 - 2\varepsilon + \varepsilon^2)\mathrm{d}^2(p, C) \\ &> (1-2\varepsilon)\mathrm{d}^2(p, C) \ . \end{aligned}$$

Hence, we get

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

□

C Proof of Proposition 3.2

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

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

□

D Numerical Values for Spambase and Intrusion

		running time (in sec)			cost		
dataset	k	StreamKM++	k-means++	k-means	StreamKM++	k-means++	k-means
Spambase	10	3.06	3.57	19.02	$7.85 \cdot 10^7$	$8.71 \cdot 10^7$	$1.70 \cdot 10^8$
	20	7.04	8.22	59.85	$2.27 \cdot 10^7$	$2.45 \cdot 10^7$	$1.53 \cdot 10^8$
	30	16.45	19.05	88.8	$1.24 \cdot 10^7$	$1.34 \cdot 10^7$	$1.51 \cdot 10^8$
	40	28.93	20.54	132.03	$8.64 \cdot 10^6$	$9.01 \cdot 10^6$	$1.49 \cdot 10^8$
	50	44.48	25.9	182.08	$6.29 \cdot 10^6$	$6.68 \cdot 10^6$	$1.48 \cdot 10^8$
Intrusion	10	74.1	50.6	408.8	$1.27 \cdot 10^{13}$	$1.75 \cdot 10^{13}$	$9.52 \cdot 10^{14}$
	20	103.1	262.4	2711.3	$1.26 \cdot 10^{12}$	$1.55 \cdot 10^{12}$	$9.51 \cdot 10^{14}$
	30	143.8	1973.3	4389.1	$4.29 \cdot 10^{11}$	$4.96 \cdot 10^{11}$	$9.51 \cdot 10^{14}$
	40	197.6	1257.0	10733.7	$1.95 \cdot 10^{11}$	$2.25 \cdot 10^{11}$	$9.50 \cdot 10^{14}$
	50	250.5	1339.5	14282.0	$1.11 \cdot 10^{11}$	$1.29 \cdot 10^{11}$	$9.50 \cdot 10^{14}$

Table 3: Average running time and average cost for the experiments on *Spambase* and *Intrusion*

E Numerical Values for Covertype and Tower

		running time (in sec)			
dataset	k	StreamKM++	StreamLS	BIRCH	k-means++
Covertype	10	245	147	44	3389
	20	297	460	44	5160
	30	378	1027	44	14933
	40	454	1773	44	16713
	50	617	2588	44	25803
Tower	20	157	679	77	2960
	40	168	1989	78	6902
	60	187	3849	77	11247
	80	211	6212	77	19206
	100	248	8946	77	17161

Table 4: Average running time for the experiments on *Covertype* and *Tower*

		cost			
dataset	k	StreamKM++	StreamLS	BIRCH	k-means++
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	$1.65 \cdot 10^{8}$	$1.63 \cdot 10^{8}$	$2.98 \cdot 10^{8}$	$1.63 \cdot 10^{8}$

Table 5: Average cost for the experiments on *Covertype* and *Tower*

F Numerical Values for BigCross and Census 1990

		running time (in sec)			cost		
dataset	k	StreamKM++	StreamLS	BIRCH	StreamKM++	StreamLS	BIRCH
BigCross	15	5486	6239	1006	$5.05 \cdot 10^{12}$	$5.23 \cdot 10^{12}$	$6.69 \cdot 10^{12}$
	20	5738	10502	998	$4.15 \cdot 10^{12}$	$4.23 \cdot 10^{12}$	$4.85 \cdot 10^{12}$
	25	5933	15780	996	$3.59 \cdot 10^{12}$	$3.54 \cdot 10^{12}$	$4.45 \cdot 10^{12}$
	30	6076	22779	996	$3.18 \cdot 10^{12}$	$3.18 \cdot 10^{12}$	$3.83 \cdot 10^{12}$
Census 1990	10	1571	631	271	$2.48 \cdot 10^{8}$	$2.40 \cdot 10^{8}$	$3.98 \cdot 10^{8}$
	20	1724	2362	271	$1.90 \cdot 10^{8}$	$1.85 \cdot 10^{8}$	$3.17 \cdot 10^{8}$
	30	1839	5504	271	$1.59 \cdot 10^{8}$	$1.53 \cdot 10^{8}$	$2.94 \cdot 10^{8}$
	40	1956	10054	272	$1.41 \cdot 10^{8}$	$1.35 \cdot 10^{8}$	$2.78 \cdot 10^{8}$
	50	2057	11842	272	$1.28 \cdot 10^{8}$	$1.24 \cdot 10^{8}$	$2.73 \cdot 10^{8}$

Table 6: Average running time and average cost for the experiments on *BigCross* and *Census 1990*

G Standard Deviation 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	-	1263.44	345.6
	40	24.91	-	563.20	1306.2
	50	31.58	-	706.00	1190.78
Covertype	10	0.88	2.43	2295.85	-
	20	6.93	18.18	1249.18	-
	30	14.15	52.14	9653.06	-
	40	14.02	97.64	6838.93	-
	50	39.28	123.28	12231.98	-
Tower	20	0.58	14.11	1594.76	-
	40	1.79	50.83	2085.12	-
	60	3.96	58.27	3656.87	-
	80	7.95	122.65	5162.60	-
	100	11.34	315.31	1795.07	-
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	-	-

Table 7: Standard deviation of the running time 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	$3.13 \cdot 10^{10}$	-	$6.85 \cdot 10^{10}$	$2.54 \cdot 10^{10}$
	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	$1.94 \cdot 10^{10}$	$1.29 \cdot 10^{11}$	-	-

Table 8: Standard deviation of the cost of our experiments

H Parameters of Algorithm BIRCH

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

Table 9: Manually adjusted TotalMemSize percentage 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)	1024
BDtype	4
Ftype	0
Phase1Scheme	0
RebuiltAlg	0
StatTimes	3
NoiseRate	0.25
Range	2000
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

Table 10: List of parameters for algorithm BIRCH