BICO: BIRCH meets Coresets for k-means clustering[*]

Hendrik Fichtenberger, Marc Gillé, Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler

TU Dortmund, Germany

We design a data stream algorithm for the k-means problem, called BICO, that combines the data structure of the SIGMOD Test of Time award winning algorithm BIRCH [27] with the theoretical concept of coresets for clustering problems. The k-means problem asks for a set C of k centers minimizing the sum of the squared distances from every point in a set P to its nearest center in C. In a data stream, the points arrive one by one in arbitrary order and there is limited storage space.

BICO computes high quality solutions in a time short in practice. First, BICO computes a summary S of the data with a provable quality guarantee: For every center set C, S has the same cost as P up to a $(1+\varepsilon)$-factor, i. e., S is a *coreset*. Then, it runs k-means++ [5] on S.

We compare BICO experimentally with popular and very fast heuristics (BIRCH, MacQueen [24]) and with approximation algorithms (StreamKM++ [2], StreamLS [18, 26]) with the best known quality guarantees. We achieve the same quality as the approximation algorithms mentioned with a much shorter running time, and we get much better solutions than the heuristics at the cost of only a moderate increase in running time.

1 Introduction

Clustering is the task to partition a set of objects into groups such that objects in the same group are similar and objects in different groups are dissimilar. There is a huge amount of work on clustering both in practice and in theory. Typically, theoretic work

[*] This research was partly supported by DFG grants BO 2755/1-1 and SO 514/4-3 and within the Collaborative Research Center SFB876, project A2. The final authenticated version is available online at https://doi.org/10.1007/978-3-642-40450-4_41.

focuses on exact solutions or approximations with guaranteed approximation factors, while practical algorithms focus on speed and results that are reasonably good on the particular data at hand.

We study the k-means problem, which given a set of points P from $\mathbb{R}^d$ asks for a set of k centers such that the cost defined as the sum of the squared distances of all points in P to their closest center is minimized. The centers induce a clustering defined by assigning every point to its closest center.

For this problem, the algorithm most used in practice is Lloyd's algorithm, an iterative procedure that converges to a local optimum after a possibly exponential number of iterations. An improved algorithm known as k-means++ by Arthur and Vassilvitskii [6] has a $\mathcal{O}(\log k)$ approximation guarantee.

Big Data is an emerging area of computer science. Nowadays, data sets arising from large scale physical experiments or social networks analytics are far too large to fit in main memory. Some data, e. g., produced by sensors, additionally arrives one by one, and it is desirable to filter it at arrival without intermediately storing large amounts of data. Considering k-means in a data stream setting, we assume that points arrive in arbitrary order and that there is limited storage capacity. Neither Lloyd's algorithm nor k-means++ work in this setting, and both would be too slow even if the data already was on a hard drive [2].

A vast amount of approximation algorithms were developed for the k-means problem in the streaming setting. They usually use the concept of *coresets*. A coreset is a small weighted set of points S, that ensures that if we compute the weighted clustering cost of S for any given set of centers C, then the result will be a $(1 + \varepsilon)$-approximation of the cost of the original input.

Approximation algorithms are rather slow in practice. Algorithms fast in practice are usually heuristics and known to compute bad solutions on occasions. The best known one is BIRCH [27]. It also computes a summary of the data, but without theoretical quality guarantee.

This paper contributes to the field of interlacing theoretical and practical work to develop an algorithm good in theory and practice. An early work in this direction is StreamLS [18, 26] which applies a local search approach to chunks of data. StreamLS is significantly outperformed by Stream-KM++[2] which computes a coreset and then solves the k-means problem on the coreset by applying k-means++. Stream-KM++ computes very good solutions and is reasonably fast for small k. However, especially for large k, its running time is still far too large for big data sets.

We develop BICO, a data stream algorithm for the k-means problem which also computes a coreset and achieves very good quality in practice, but is significantly faster than Stream-KM++.

Related Work. To solve k–means in a stream, there are three basic approaches. First, one can apply an online gradient descent such as MacQueen's algorithm [24]. Such an algorithm is usually the fastest available, yet the computed solution has poor performance in theory and (usually) also in practice.

The other two approaches compute summaries of the data for further processing. Summaries should be small to speed up the optimization phase, they should have a good quality, and their computation should be fast. Summary computing algorithms either update the summary one point at a time or read a batch of points and process them together.

Theoretical analysis has mostly focused on the latter scenario and then relies on the *merge & reduce* framework originally by Bentley and Saxe [7] and first applied to clustering in [4]. Informally speaking, a coreset construction can be defined in a non-streaming manner and then be embedded into the merge & reduce framework. The computed coreset is by a factor of $\log^{t+1} n$ larger than the non-streaming version where t is the exponent of ε^{-1} in the coreset size. The running time is increased to $2C(m) \cdot n/m$ where m is the batch size and $C(m)$ is the time of the non-streaming version of the coreset construction. Thus, the asymptotic running time is not increased (for at least linear $C(m)$), but from a practical point of view this overhead is not desirable. Another drawback is that the size of the computed coresets usually highly depends on $\log n$. Streaming algorithms using summaries include [8, 10, 11, 12, 20, 21, 22]. In particular, StreamLS uses a batch approach, and Stream-KM++ uses merge & reduce. There is usually a high dependency on $\log n$ in merge & reduce constructions. If d is not a constant, the smallest dependency on the dimension is achieved by [12] and this coreset has a size of $\mathcal{O}(k^2 \cdot \varepsilon^{-4} \log^5 n)$ (which is independent of d).

For pointwise updates, in particular notice the construction in [15] computing a streaming coreset of size $\mathcal{O}(k \cdot \log n \cdot \varepsilon^{-(d+2)})$. For low dimensions, i. e., if d is a constant, this is the lowest dependency on k and $\log n$ of any coreset construction. This is due to the fact that the coreset is maintained without merge & reduce. The time to compute the streaming coreset is $\tilde{\mathcal{O}}(n \cdot \rho + k \cdot \log n \cdot \rho \cdot \epsilon^{-(d+2)})$ with $\rho = \log(n\Delta/\epsilon)$ where Δ is the spread of the points, i. e., the maximal distance divided by the smallest distance of two distinct points.

Pointwise updates are usually preferable for practical purposes. Probably the best known practical algorithm is BIRCH [27]. It reads the input only once and computes a summary by pointwise updates. Then, it solves the k-means problem on the summary using agglomerative clustering.

The summary that BIRCH computes consists of a tree of so-called Clustering Features. A Clustering Feature (CF) summarizes a set of points by the sum of the points, the number of points and the sum of the squared lengths of all points. BIRCH has no theoretical quality guarantees and does indeed sometimes perform badly in practice [17, 19].

We are not aware of other very popular data stream algorithms for the k-means problem. There is a lot of work on related problems, for example CURE [17] which requires more than one pass over the data, DBSCAN [9] which is not center based, CLARANS [25] which is typically used when centers have to be chosen from the dataset and is not particularly optimized for points in Euclidean space and ROCK [16] and COBWEB [14] which are designed for categorical attributes.

Our Contribution. We develop BICO, a data stream algorithm based on the data structure of BIRCH. Both algorithms compute a summary of the data, but while the summary computed by BIRCH can be arbitrarily bad as we show at the start of Section 3, we show that BICO computes a coreset S, so for every set of centers C, the cost of the input point set P can be approximated by computing the cost of S. For constant dimension d, we bound the size m of our coreset by $\mathcal{O}(k \cdot \log n \cdot \varepsilon^{-(d+2)})$ and show that BICO needs $\mathcal{O}\left(N(m) \cdot (n + m \log n\Delta)\right)$ time to compute it where $N(m)$ is the time needed to solve a nearest neighbor problem within m points. Trivially, $N(m) = \mathcal{O}(m)$. By using range query data structures, $N(m) = \mathcal{O}(\log^{d-1} m)$ can be achieved at the cost of $\mathcal{O}(m \log^{d-1} m)$ additional space [3]. Notice that the size of the coreset is asymptotically equal to [15].

We implement BICO and show how to realize the algorithm in practice by introducing heuristic enhancements. Then we compare BICO experimentally to two heuristics, BIRCH and MacQueen's k-means algorithm, and to two algorithms designed for high quality solutions, Stream-KM++ and StreamLS. BICO computes solutions of the same high quality as Stream-KM++ and StreamLS which we believe to be near optimal. For small k, BICOs running time is only beaten by MacQueen's and in particular, BICO is 5-10 times faster than Stream-KM++ (and more for StreamLS). For larger k, BICO needs to maintain a larger coreset to keep the quality up. However, BICO can trade quality for speed. We do additional testruns showing that with different parameters, BICO still beats the cost of MacQueen and BIRCH in similar running time. We believe that BICO provides the best quality-speed trade-off in practice.

2 Preliminaries

The k-means problem. Let $P \subseteq \mathbb{R}^d$ be a set of points in d-dimensional Euclidean space with $|P| = n$. For two points $p, q \in P$, we denote their Euclidean distance by $||p - q|| := \sqrt{\sum_{i=1}^{d}(p_i - q_i)^2}$. The k-means problem asks for a set C of k points in $\mathbb{R}^d$ (called *centers*) that minimizes the sum of the squared distances of all points in P to their closest point in C, i. e., the objective is $\min_{C \subset \mathbb{R}^d, |C|=k} \sum_{p \in P} \min_{c \in C} ||p - c||^2 =: \min_{C \subset \mathbb{R}^d, |C|=k} \operatorname{cost}(P, C)$.

The weighted k-means cost is defined by $\operatorname{cost}_w(P, C) := \sum_{p \in P} w(p) \min_{c \in C} ||p - c||^2$ for any weight function $w : P \to \mathbb{R}^+$, and the weighted k-means objective then is $\min_{C \subset \mathbb{R}^d, |C|=k} \operatorname{cost}_w(P, C)$. For a point set P, we denote the centroid of P as $\mu(P) := \frac{1}{|P|} \sum_{p \in P} p$. The k-means objective function satisfies the following well-known equation that allows to compute $\operatorname{cost}(P, \{c\})$ via $\mu(P)$.

Fact 1. *Let $P \subset R^d$ be a finite point set. Then the following equation holds: $\sum_{p \in P} ||p - c||^2 = \sum_{p \in P} ||p - \mu(P)||^2 + |P| ||\mu(P) - c||^2$.*

BIRCH. We only describe the main features of BIRCH's preclustering phase. The algorithm processes given points on the fly, storing them in a so called *CF Tree* where CF is the abbreviation of *Clustering Feature.*

Definition 2 (Clustering Feature). *Let $P := \{p_1, \ldots, p_n\} \subset \mathbb{R}^d$ be a set of n d-dimensional points. The* Clustering Feature *CF_P of P is a 3-tuple (n, s_1, S_2) where n is number of points, $s_1 = \sum_{i=1}^{n} p_i$ is the linear sum of the points, and $S_2 = \sum_{i=1}^{n} ||p_i||^2$ is the sum of the squared lengths of the points.*

The usage of Clustering Features are the main space reduction concept of BIRCH. Notice that given a Clustering Feature $CF_P = (n, s_1, S_2)$, the squared distances of all points in a point set P to *one* given center c can be calculated *exactly* by $\mathrm{cost}\,(P, \{c\}) = \sum_{i=1}^{n} ||p_i||^2 - n \cdot ||\mu(P) - 0||^2 + n \cdot ||c - \mu_i||^2 = S_2 - \frac{1}{n}||s_1||^2 + n||c - s_1/n||^2$.

When using Clustering Features to store a small summary of points, the quality of the summary decreases when storing points together in one CF that should be assigned to different centers later on. If we summarize points in a CF and later on get centers where all these points are closest to the same center, then their clustering cost can be computed with the CF without any error. Thus, the idea of BIRCH is to heuristically identify points that are likely to be clustered together. For this purpose, they use the following insertion process.

The first point in the input opens the first CF, i. e., a CF only containing the first point is created. Then, iteratively, the next points are added. For a new point p, BIRCH first looks for the CF which is 'closest' to p. Let CF_S be an arbitrary existing CF in the CF tree of BIRCH and recall that CF_S represents the set of points S. The distance between p and CF_S is defined as

$$\sum_{q \in S \cup \{p\}} (q - (\sum_{q' \in (S \cup \{p\})} q')/(|S|+1))^2 - \sum_{q \in S} (q - (\sum_{q' \in S} q')/|S|)^2. \qquad (1)$$

Let CF_{S^*} be the CF closest to p. Then, p is added to CF_{S^*} if the radius

$$\sqrt{\frac{\sum_{p \in (S^* \cup \{p\})} (q - \mu_{S^*})^2}{|S| + 1}}$$

is smaller than a given threshold t. If the radius exceeds the threshold, then p opens a new CF.

BIRCH works with increasing thresholds when processing the input data. It starts with threshold $t = 0$ and then increases t whenever the number of CFs exceeds a given space bound, calling a rebuilding algorithm to shrink the tree. This algorithm ensures that the number of CFs is decreased sufficiently. Notice that CFs cannot be split again, so the rebuilding might return a different tree than the one computed directly with the new threshold.

Coresets. BIRCH decides heuristically how to group the points into subclusters. We aim at refining this process such that the reduced set is not only small but is also guaranteed to approximate the original point set. More precisely, we aim at constructing a coreset:

Definition 3 ([21])**.** *A* (k, ε)-coreset *is a subset* $S \subset \mathbb{R}^d$ *weighted with a weight function* $w' : S \to \mathbb{R}^d$ *such that for all* $C \subset \mathbb{R}^d$, $|C| = k$, *it holds that* $|\mathrm{cost}_{w'}(S, C) - \mathrm{cost}(P, C)| \leq \varepsilon \cdot \mathrm{cost}(P, C)$.

3 BICO: Combining BIRCH and Coresets

The main problem with the insertion procedure of BIRCH is that the decision whether points are added to a CF or not is based on the increase of the radius of the candidate

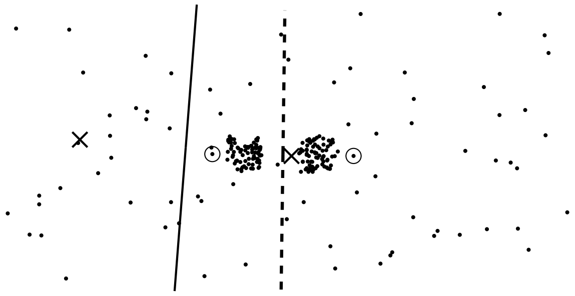

Figure 1: An example created by drawing 150 points uniformly at random from the areas around $(-0.5, 0)$ and $(0, 0.5)$ and 75 points from $[-4, -2] \times [4, 2]$. BIRCH computed the centers marked by x leading to the partitioning by the solid line. BICO computed the same centers in 10 independent runs, marked by circles and the dashed line partitioning.

CF. Figure 1 shows a point set that was generated with two rather close but clearly distinguishable clusters plus randomly added points serving as noise. The problem for BIRCH is that the distance between the two clusters is not much larger than the average distance between the points in the noise. We see that BIRCH merges the clusters together and thus later computes only one center for them while the second center is placed inside the noise.

Our lower bound example for the quality guarantee of BIRCH follows this intuition. It looks similar to Figure 1, but is multi-dimensional and places the points deterministically in a structured way useful for the theoretical analysis.

Let $d > 1$ and define the point sets $P_1 := \{(7, 3i_2, \ldots, 3i_d) \mid i_2, \ldots, i_d \in \{-2, -1, 0, 1, 2\}\}$ and $P_2 := \{(-7, 3i_2, \ldots, 3i_d) \mid i_2, \ldots, i_d \in \{-2, -1, 0, 1, 2\}\}$. Let R be the set of $(n - |P_1| - |P_2|)/2$ points at position $(1, 0, \ldots, 0)$ and equally many points at position $(-1, 0, \ldots, 0)$ and set $\tilde{P} = P_1 \cup P_2 \cup R$.

Theorem 4. *For $c > 0$, input $\tilde{P}$ and threshold $T > 0$, BIRCH either has $\Omega(n^{1/c})$ CFs or the computed solution has cost of at least $\Omega\left((n^{1-\frac{1}{c}}/\log n) \cdot OPT\right)$.*

The basic algorithm. Like BIRCH, BICO uses a tree whose nodes correspond to CFs. The tree has no distinguished root, but starts with possibly many CFs in level 1 (which can be seen as children of an imaginary root node). When we open a CF, we keep the first point in the CF as its *reference point*. The first point in the stream just opens a CF in level 1. Now, for each new point, we first try to add the point into an existing CF. We start on the first level $i = 1$. We try to insert the current point to the nearest CF in level i. The insertion fails if all CFs are far away, i.e. the distance between the current point and all CF reference points is larger than the *radius* R_i of CFs on level i, or the nearest CF is full, i.e. the cost of the point set represented by the CF is greater or equal than a threshold parameter T. In the first case we open a new CF with the current point as reference point in level i and in the second case we recurse and try to insert the point into the children of the nearest CF which are on level $i+1$.

The algorithm is given in Algorithm 1. We assume that $n_{\max}$ is large enough to ensure that line 13 is never executed. We denote the CF that a point r is reference point of as $CF(r)$. By nearest(p, S) we refer to the reference point closest to p of all CFs in S. By children$(CF(r))$ we denote the set of CFs that are children of $CF(r)$ in the tree. For sake of shorter notation, we also use a virtual point ρ with virtual CF $CF(\rho)$ for the root node of the tree.

Theorem 5. *Let $\varepsilon > 0$, $f(\varepsilon) = (2 \cdot (\log n) \cdot 4^d \cdot \sqrt{40}^{d+2})/(\varepsilon^{d+2})$, $OPT/(k \cdot f(\varepsilon)) \leq T \leq 2 \cdot OPT/(k \cdot f(\varepsilon))$ and $R_i = \sqrt{T/(8 \cdot 2^i)}$. The set of centroids s_1/n', where (n', s_1, S_2) is a CF resulting from Algorithms 1, weighted with n' is a (k, ε)-coreset of size $O(k \cdot \log n \cdot \varepsilon^{-(d+2)})$ if the dimension d is a constant.*

Algorithm 1: Update mechanism where T and R_i are fixed parameters

```
   input: $p \in \mathbb{R}^d$
 1 Set $f = \rho$, $S = \text{children}\,(CF(\rho))$ and $i = 1$;
 2 if $S = \emptyset$ or $||p - \text{nearest}(p, S)|| > R_i$ then
 3     Open new CF with reference point $p$ in level $i$ as child of $CF(f)$;
 4 else
 5     Set $r := \text{nearest}(p, S)$;
 6     if $\text{cost}_{CF}(CF(r) \cup \{p\}) \leq T$ then
 7         Insert $p$ in $CF(r)$;
 8     else
 9         Set $S := \text{children}\,(CF(r))$;
10         Set $f := r$ and $i := i + 1$;
11         Goto line 2;

12 if number of current CFs $> n_{\max}$ then
13     Start rebuilding algorithm;
```

The Rebuilding Algorithm. Above, we assumed that we know the cost of an optimal solution beforehand. To get rid of this assumption, we start with a small threshold, increase it if necessary and use the rebuilding algorithm given in Algorithm 2 to adjust the tree to the new threshold. If we start with a T smaller than $OPT/(k \cdot f(\varepsilon))$ and keep doubling it, then at some point T will be in $[OPT/(k \cdot f(\varepsilon)), 2 \cdot OPT/(k \cdot f(\varepsilon))]$, and at this point in time our coreset size of $O(k \cdot \log n \cdot \varepsilon^{-(d+2)})$ is sufficient to store a (k, ϵ)-coreset. Until this point, we will need rebuilding steps, but we will not lose quality.

The aim of the rebuilding algorithm is to create a tree which is similar to the tree which would have resulted from using the new threshold in the first place. Let R'_i be the radii before and let R_i be the radii after one iteration of the rebuilding algorithm. Notice that $R_i = \sqrt{2 \cdot T'/(8 \cdot 2^i)} = \sqrt{T'/(8 \cdot 2^{i-1})} = R'_{i-1}$. Thus, if a CF is not moved up and thus its level is increased by one, the radius will remain the same. This is nice as thus the CF in the same level automatically satisfy that the reference points are not within the radii of their neighbors. However, other properties of the tree do no longer hold in the original way: (1) If a CF $CF(r)$ on level i is inserted to a CF $CF(r')$ on level $i-1$, i. e., $CF(r)$ becomes a new child of $CF(r')$ or they are merged, it is possible that some points which are represented by $CF(r)$ are not within the radius R_{i-1} of r'. But the distance can be bounded by $R_{i-1} + R_i$ such that the CFs on each level do not overlap too much. (2) The rebuilding algorithm can not split CFs. Thus, the set of CFs is different compared to a run where the new threshold was used in the first place. In total, these changes slightly increase the coreset size compared to Theorem 5, but it can be still bounded by $O(k \cdot \log n \cdot \varepsilon^{-(d+2)})$, and the weighted set of centroids also remains a (k, ε)-coreset.

Algorithm 2: Rebuilding algorithm when number of CFs gets too large

```
Set $T := 2 \cdot T$;
Create a new empty level 1 (which implicitly increases the number of all existing levels);
Let $S_1$ be the empty set of Clustering Features in level 1;
Let $S_2$ be the set of all Clustering Features in level 2;
for all Clustering Features $X \in S_2$ with reference point $p$ do
    if $S_1 = \emptyset$ or $||p - nearest(p, S_1)||^2 > R_1$ then
        Move $X$ from $S_2$ to $S_1$;
        Notice that this implicitly moves all children of $X$ one level up;
    else
        if $cost\,(X \cup CF(nearest(p, S_1))) \leq T$ then
            Insert $X$ into $CF(nearest(p, S_1))$;
        else
            Make $X$ a child of $CF(nearest(p, S_1))$;
Traverse through the CF tree and, if possible, merge CFs into parent CFs
```

Running Time. When trying to insert a point on level i, we need to decide whether the point is within distance R_i of its nearest neighbor, and if so, we need to locate the nearest neighbor. Let $N(m)$ denote the time needed for this, depending on the coreset size $m \in \mathcal{O}(k \cdot \log n \cdot \varepsilon^{-(d+2)})$. The running time of BICO is $\mathcal{O}(N(m) \cdot n)$ plus the time needed for the rebuilding steps.

A rebuilding step needs to go through all m elements of the coreset and to insert them into the new-build tree. This takes $\mathcal{O}(m \cdot N(m))$ time. The number of rebuilding steps depends on how we choose the start value for T. We proved that BICO computes a (k, ε)-coreset for large enough m. In particular, this means that for any $m + 1$ points, BICO contracts at least two of them during the process. We use this observation by scanning through the first $m + 1$ points and calculating the minimal distance d_0 between two points (here, just ignore multiple points at the same position). Notice that if $T < d_0^2$, we are not able to merge any two points into one Clustering Feature. We set $T = d_0^2$. Then, T cannot be too small, because otherwise we cannot contract the m points.

The cost of any clustering is bounded from above by $n \cdot \Delta_{\max}$ where $\Delta_{\max}$ is the maximal distance between any two points. Our start value for T is bounded from below by the smallest distance $\Delta_{\min}$ between any two points. Thus, the factor between the start and end value of T is bounded by $n \cdot \frac{\Delta_{\max}}{\Delta_{\min}}$. The fraction $\Delta := \frac{\Delta_{\max}}{\Delta_{\min}}$ is called the *spread* of the points. With each rebuilding step we double T, and thus the number of rebuilding steps is bounded by $\log(n \cdot \Delta)$.

Corollary 6. *BICO computes a coreset for a set of n points in $\mathbb{R}^d$ given as an input only data stream in time bounded by $\mathcal{O}(N(m)(n + \log(n\Delta)m))$ using $\mathcal{O}(m)$ space where $m \in \mathcal{O}(k \cdot \log n \cdot \varepsilon^{-(d+2)})$ is the coreset size and d is constant.*

4 Experiments

Algorithms. We compare BICO with Stream-KM++, StreamLS, BIRCH and MacQueen's k-means algorithm. Stream-KM++ also aims at a trade-off between quality and speed which makes it most relevant for our work. BIRCH is the most relevant practical algorithm. We include MacQueen because it performed very well on one data set and is very fast. We use the author's implementations for Stream-KM++ [2], BIRCH [27] and StreamLS [18, 26] and an open source implementation of MacQueen's k-means [13]. We use the same parameters for BIRCH as in [2] except that we increase the memory to 26% on BigCross and 8% on Census in order to enable BIRCH to compute solutions for our larger k.

Setting. All computations were performed on seven identical machines with the same hardware configuration (2.8 Ghz Intel E7400 with 3 MB L2 Cache and 8 GB main memory). BICO and k-means++ are implemented in C++ and compiled with gcc 4.5.2. The source code for the algorithms, the testing environment and links to the other algorithms' source codes will appear at our website[1].

After computing the coreset, we determine the final solution via five weighted k-means++ runs (until convergence) and chose the solution with best cost on the coreset. The implementation of BICO differs from Section 3 in two points.

Coreset Size. Our theoretic analysis gives a worst case bound on the space needed by BICO even for adversarial inputs. On average instances, we expect that $\mathcal{O}(k)$ Clustering Features suffice to get a very good solution. The authors in [2] used the size $200k$ for Stream-KM++ which we also opted to use for both Stream-KM++ and BICO in our line of experiments. This leads to a asymptotic running time of $\mathcal{O}\left(k \cdot (n + k \log n\Delta)\right)$ for BICO.

Filtering. A large part of the running time of BICO is spent for nearest neighbor queries on the first level of the tree. We speed it up by an easy heuristic: All CF reference points are projected to d one-dimensional subspaces chosen uniformly at random at the start of BICO. Let p be a new point. Since only reference points that are close to p in each subspace are candidates for the cluster feature we are searching for, we take the subspace where the number of points within distance R_1 of p is smallest and only iterate through these to find the nearest neighbor.

Datasets. We used the four largest data sets evaluated in [2], *Tower*, *Covertype* and *Census* from the UCI Machine Learning Repository [1] and *BigCross*, which is a subset

[1] http://ls2-www.cs.uni-dortmund.de/bico

of the Cartesian product of *Tower* and *Covertype*, created by the authors of [2] to have a very large data set. Additionally, we use a data set we call CalTech128 which is also large and has higher dimension. It consists of 128 SIFT descriptors [23] computed on the Caltech101 object database.

	BigCross	CalTech128	Census	CoverType	Tower
Number of Points (n)	11620300	3168383	2458285	581012	4915200
Dimension (d)	57	128	68	55	3
Total size ($n \cdot d$)	662357100	405553024	167163380	31955660	14745600

Experiments. On Census, Tower and BigCross, we ran tests with all values for k from [2], and $k = 250$ and $k = 1000$ in addition. On CalTech128, we tested $k = 50, 100, 250$ and $k = 1000$. We repeated all randomized algorithms 100 times and the diagrams show the mean values. In all diagrams, the bar of BICO is composed of two bars on top of each other corresponding to the core BICO part and the k-means++ part of BICO. We did not find parameters that enabled BIRCH to compute centers on CalTech128. Due to tests that we did with modified versions of CalTech128 we believe that the implementation is not able to handle data with a dimension like CalTech128.

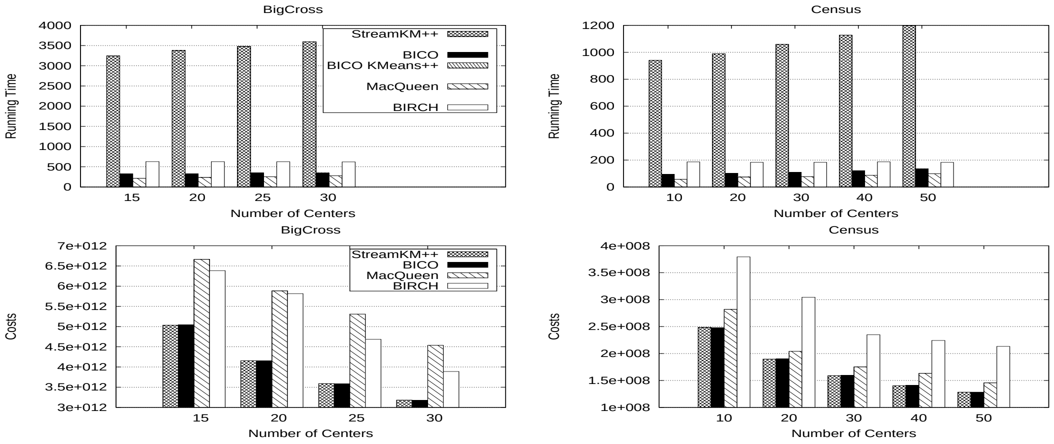

Figure 2: Running times (in seconds) and costs for datasets BigCross and Census

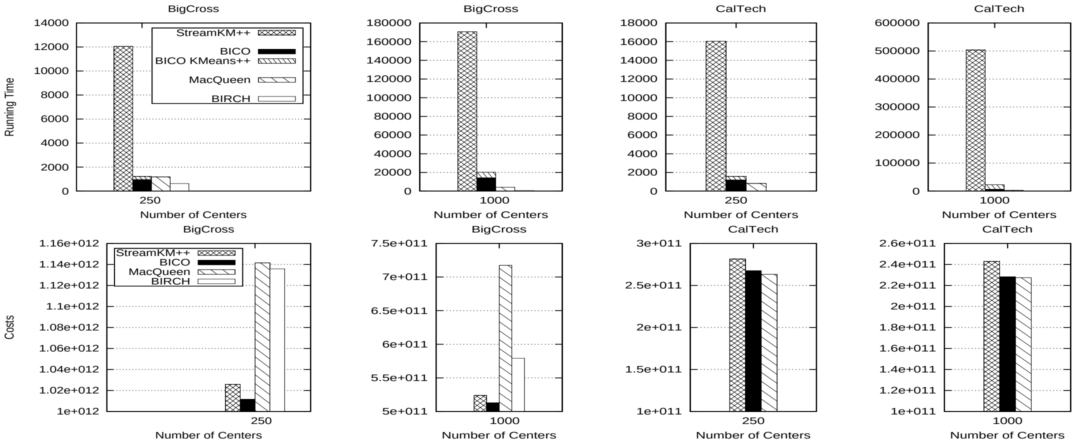

Figure 3: Running times and costs for datasets BigCross and CalTech for large k

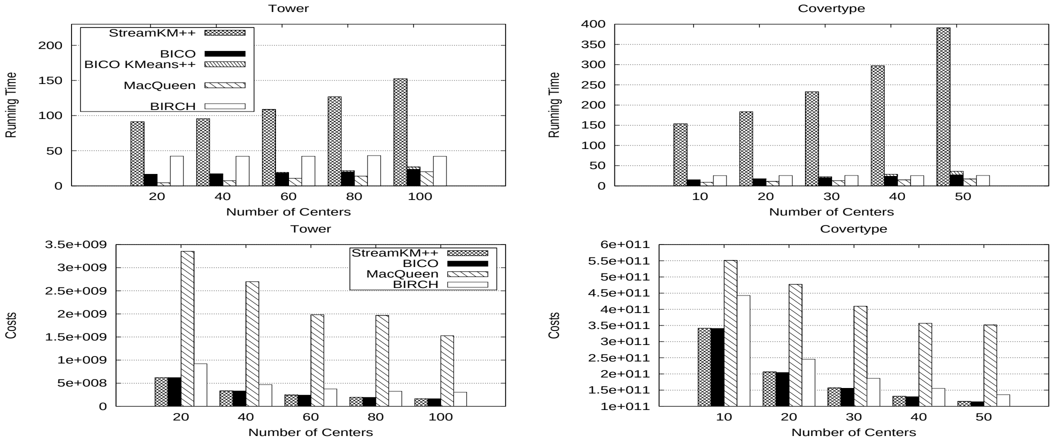

Figure 4: Running times and costs for datasets Tower and Covertype

Comparison with Stream-KM++ and StreamLS. StreamLS, Stream-KM++ and BICO all have comparable solution quality (see Figures 2, 4, 3). StreamLS, however, is rather slow such that we did not include it in the diagrams and did not include StreamLS in the tests for larger k. The running times of Stream-KM++ and BICO both depend on the number of centers which is reasonable because more centers require a larger coreset size which induces more effort to keep the coreset up to date. However, BICO is 5-10 times faster and applicable to much higher values of k (see Figure 2, 3, 4).

Comparison with BIRCH and MacQueen. BIRCH and MacQueen both tend to compute much worse solutions. MacQueen performs okay on Census, really well on CalTech128, but badly on BigCross and worse on Tower and Covertype (see Figures 2, 3, 4). MacQueen starts being faster than BIRCH, but is slower for larger k (compare BigCross in Figure 2 and 3) because BIRCH does not adjust for larger k. The running time of BICO is nearly always lower than that of BIRCH for small k (see Figures 2, 4) and within two times the running time of MacQueen in most experiments. For large k, BICO has significantly larger running time. If the dataset is high-dimensional like CalTech128, this is mainly due to k-means++, while on data sets with a lot of points of lower dimension like BigCross, the core part of BICO is dominating.

BICO for large k. We point out that BICO is still practical for large k despite the large running times when adjusted. We chose BigCross because a running time of 4.6 hours is unfavourable and because here, the running time is due to core BICO and cannot be tackled by improving the k-means++ implementation (which is implemented without any speed-ups). By reducing the coreset size, the running time of BICO decreases. We lower it until BICO runs in 619 seconds compared to a running time of 616 seconds by BIRCH and 4241 seconds by MacQueen. The solution computed by BICO is still significantly better than the solutions by MacQueen and BIRCH.

Remark. Notice that many proof and experiment details are omitted due to space restrictions and will appear in a long version of the paper.

Acknowledgements. We thank René Grzeszick for providing the data set CalTech128 and for pointing out the implementation of MacQueen's k-means algorithm. We also thank Frank Hellweg, Daniel R. Schmidt and Lukas Pradel for their help with the implementation and the example in Figure 1, and anonymous referees for their valuable comments concerning the presentation of the paper.

References

[1] D. J. Newman A. Asuncion. UCI machine learning repository, 2007. URL `http://www.ics.uci.edu/$\sim$mlearn/{MLR}epository.html`.

[2] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler. Streamkm++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[3] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56, 1999.

[4] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51(4):606–635, 2004.

[5] D. Arthur and S. Vassilvitskii. How slow is the k-means method? In *Proc. of the 22nd SoCG*, pages 144–153, 2006.

[6] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proc. of the 18th SODA*, pages 1027–1035, 2007.

[7] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[8] K. Chen. On coresets for k-median and k-means clustering in metric and euclidean spaces and their applications. *SIAM Journal on Computing*, 39(3):923–947, 2009.

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

[10] D. Feldman and M. Langberg. A unified framework for approximating and clustering data. In *Proc. of the 43th STOC*, pages 569–578, 2011.

[11] D. Feldman, M. Monemizadeh, and C. Sohler. A PTAS for k-means clustering based on weak coresets. In *Proc. of the 23rd SoCG*, pages 11–18, 2007.

[12] D. Feldman, M. Schmidt, and C. Sohler. Constant-size coresets for k-means, pca and projective clustering. In *Proc. of the 24th SODA*, pages 1434 –1453, 2012.

[13] G. A. Fink and T. Plötz. Open source project ESMERALDA. URL `sourceforge.net/projects/esmeralda/files/esmeralda/open%201.20/`.

[14] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139–172, 1987.

[15] G. Frahling and C. Sohler. Coresets in dynamic geometric data streams. In *Proc. of the 37th STOC*, pages 209–217, 2005.

[16] S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. *Inform. Systems*, 25(5):345–366, 2000.

[17] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Inform. Systems*, 26(1):35–58, 2001.

[18] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE TKDE*, 15(3):515–528, 2003.

[19] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Inform. Systems*, 17(2-3):107–145, 2001. ISSN 0925-9902.

[20] S. Har-Peled and A. Kushal. Smaller coresets for k-median and k-means clustering. *Discrete & Computational Geometry*, 37(1):3–19, 2007.

[21] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *Proc. of the 36th STOC*, pages 291–300, 2004.

[22] M. Langberg and L. J. Schulman. Universal epsilon-approximators for integrals. In *Proc. of the 21th SODA*, pages 598–607, 2010.

[23] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[24] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Berkeley Symp. on Math., Stat., and Prob.*, pages 281–297, 1967.

[25] R. T. Ng and J. Han. Clarans: A method for clustering objects for spatial data mining. *IEEE TKDE*, 14(5):1003–1016, 2002.

[26] L. O'Callaghan, A. Meyerson, R. Motwani, N. Mishra, and S. Guha. Streaming-data algorithms for high-quality clustering. In *Proc. of the 18th ICDE*, pages 685–694, 2002.

[27] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.