Streaming k-Means Clustering with Fast Queries

Yu Zhang[1]
Dept. of Elec. and Computer Eng.
Iowa State U., USA
yuz1988@iastate.edu

Kanat Tangwongsan[2]
Computer Science Program
Mahidol U. International College, Thailand
kanat.tan@mahidol.edu

Srikanta Tirthapura[1]
Dept. of Elec. and Computer Eng.
Iowa State U., USA
snt@iastate.edu

***Abstract*—We present methods for k-means clustering on a stream with a focus on providing fast responses to clustering queries. Compared to the current state-of-the-art, our methods provide substantial improvement in the query time for cluster centers while retaining the desirable properties of provably small approximation error and low space usage. Our algorithms rely on a novel idea of "coreset caching" that systematically reuses coresets (summaries of data) computed for recent queries in answering the current clustering query. We present both theoretical analysis and detailed experiments demonstrating their correctness and efficiency.**

I. Introduction

Clustering is a fundamental method for understanding and interpreting data that seeks to partition input objects into groups, known as *clusters*, such that objects within a cluster are similar to each other, and objects in different clusters are not. A clustering formulation called k-means is simple, intuitive, and widely used in practice. Given a set of points S in a Euclidean space and a parameter k, the objective of k-means is to partition S into k clusters in a way that minimizes the sum of the squared distance from each point to its cluster center.

In many cases, the input data is not available all at once but arrives as a continuous, possibly unending, sequence. This variant, known as *streaming k-means clustering*, requires an algorithm to maintain enough state to be able to incrementally update the clusters as more data arrive. Furthermore, when a query is posed, the algorithm is required to return k cluster centers, one for each cluster, for the data observed so far.

Prior work on streaming k-means (e.g. [1, 2, 3, 4]) has mainly focused on optimizing the memory requirements, leading to algorithms with provable approximation quality that only use space polylogarithmic in the input stream's size [1, 2]. However, these algorithms require an expensive computation to answer a query for cluster centers. This can be a serious problem for applications that need answers in (near) real-time, such as in network monitoring and sensor data analysis. Our work aims at improving the clustering query-time while keeping other desirable properties of current algorithms.

To understand why current solutions to streaming k-means clustering have a high query runtime, let us review the framework they use. At a high level, an incoming data stream $\mathcal{S}$ is divided into smaller "chunks" $\mathcal{S}_1, \mathcal{S}_2, \ldots,$. Each chunk is summarized into a compact representation, known as a "coreset" (for example, see [5]). The resulting coresets may still not all fit into the memory of the processor. Hence, multiple coresets are further merged recursively into higher-level coresets, forming a hierarchy of coresets, or a "coreset tree". When a query arrives, all active coresets in the coreset tree are merged together, and a clustering algorithm such as k-`means++` [6] is applied on the result, outputting k cluster centers. The query time is consequently proportional to the number of coresets that need to be merged together. In prior algorithms, the total size of all these coresets could be as large as the whole memory itself, which causes the query time to often be prohibitively high.

A. Our Contributions

We present three algorithms (`CC`, `RCC`, and `OnlineCC`) for streaming k-means clustering that asymptotically and practically improve upon prior algorithms' response time of a query while retaining guarantees on memory efficiency and solution quality of the current state-of-the-art. We provide theoretical analysis, as well as extensive empirical evaluation, of the proposed algorithms.

At the heart of our algorithms is the idea of "coreset caching" that to our knowledge, has not been used before in streaming clustering. It works by reusing coresets that have been computed during previous (recent) queries to speedup the computation of a coreset for the current query. In this way, when a query arrives, the algorithm has no need to combine all coresets currently in memory; it only needs to merge a coreset from a recent query (stored in the coreset cache) along with coresets of points that arrived after this query.

Our theoretical results are summarized in Table I. Throughout, let n denote the number of points observed in the stream so far. We measure an algorithm's performance in terms of both running time (further separated into query and update) and memory consumption. The query cost, stated in terms of q, represents the expected amortized cost per input point assuming that the total number of queries does not exceed n/q—or that the average interval between queries is $\Omega(q)$. The update cost is the average (i.e., amortized) per-point processing cost, taken over the entire stream. The memory cost is expressed in terms of words assuming that each point is d-dimensional and can be stored in $O(d)$ words. Furthermore, let m denote a user-defined parameter that determines the coreset size (m is set independent of n and is often $O(k)$ in practice); r denote a user-defined parameter that sets the merge degree of the coreset tree; and $N = n/m$ be the number of "base buckets."

In terms of accuracy, each of our algorithms provides a provable $O(\log k)$-approximation to the optimal k-means solution—that is, the quality is comparable, up to constants, to what we would obtain if we simply stored all the points

[1]Supported in part by the US National Science Foundation through awards 1527541 and 1632116

[2]Corresponding Author

Name	Query Cost (per point)	Update Cost (per point)	Memory Used	Coreset level returned at query after N batches
Coreset Tree (CT)	$O\left(\frac{kdm}{q} \cdot \frac{r\log N}{\log r}\right)$	$O(kd)$	$O\left(md\frac{r\log N}{\log r}\right)$	$\log_r N$
Cached Coreset Tree (CC)	$O\left(\frac{kdm}{q} \cdot r\right)$	$O(kd)$	$O\left(md\frac{r\log N}{\log r}\right)$	$2\log_r N$
Recursive Cached Coreset Tree (RCC)	$O\left(\frac{kdm}{q} \log\log N\right)$	$O(kd\log\log N)$	$O(mdN^{1/8})$	$O(1)$
Online Coreset Cache (OnlineCC)	usually $O(1)$ worst case $O\left(\frac{kdm}{q} \cdot r\right)$	$O(kd)$	$O\left(md\frac{r\log N}{\log r}\right)$	$2\log_r N$

TABLE I. The accuracy and query cost of different clustering methods.

so far in memory and ran an (expensive) batch k-means++ algorithm at query time. Further scrutiny reveals that for the same target accuracy (holding constants in the big-O fixed), our simplest algorithm "Cached Coreset Tree" (CC) improves the query runtime of a current state-of-the-art, CT[1], by a factor of $O(\log N)$—at the expense of a (small) constant factor increase in memory usage. If more flexibility in tradeoffs among the parameters is desired, coreset caching can be applied recursively. Using this scheme, the "Recursive Cached Coreset Tree" (RCC) algorithm can be configured so that it has a query runtime that is a factor of $O(\log N/\log\log N)$ faster than CT, and yields better solution quality than CT, at the cost of a small polynomial factor increase in memory.

In practice, a simple sequential streaming clustering algorithm, due to MacQueen [7], is known to be fast but lacks good theoretical properties. We derive an algorithm, called OnlineCC, that combines ideas from CC and sequential streaming to further enhance clustering query runtime while keeping the provable clustering quality of RCC and CC.

We also present an extensive empirical study of the proposed algorithms, in comparison to the state-of-the-art k-means algorithms, both batch and streaming. The results show that our algorithms yield substantial speedups (5–100x) in query runtime and total time, and match the accuracy of streamkm++ for a broad range of datasets and query arrival frequencies.

B. Related Work

When all input is available at the start of computation (batch setting), Lloyd's algorithm [8], also known as the k-means algorithm, is a simple, widely-used algorithm. Although it has no quality guarantees, heuristics such as k-means++ [6] can generate a starting configuration such that Llyod's algorithm will produce provably-good clusters.

In the streaming setting, Sequential k-means [7] is the earliest streaming k-means method, which maintains the current cluster centers and applies one iteration of Lloyd's algorithm for every new point received. Because it is fast and simple, Sequential k-means is commonly used in practice (e.g., Apache Spark mllib [9]). However, it cannot provide any guarantees on the quality [10]. BIRCH [11] is a streaming clustering method based on a data structure called the CF Tree; it produces cluster centers through agglomerative hierarchical clustering on the leaf nodes of the tree. CluStream[12] constructs "microclusters" that summarize subsets of the stream, and further applies a weighted k-means algorithm on the microclusters. STREAMLS [3] is a divide-and-conquer method based on repeated application of a bicriteria approximation algorithm for clustering. A similar divide-and-conquer algorithm based on k-means++ is presented in [2]. Invariably, these methods have high query-processing cost and are not suitable for applications that require fast query response. In particular, at the time of query, they require merging multiple data structures, followed by an extraction of cluster centers, which is expensive.

Har-Peled and Mazumdar [5] present coresets of size $O(\varepsilon^{-d}k\log n)$ for summarizing n points k-means, and also show how to use the merge-and-reduce technique based on the Bentley-Saxe decomposition [13] to derive a small-space streaming algorithm using coresets. Further work [14, 15] has reduced the size of a k-means coreset to $O(kd/\varepsilon^6)$. A close cousin to ours, streamkm++ [1] (essentially the CT scheme) is a streaming k-means clustering algorithm that uses the merge-and-reduce technique along with k-means++ to generate a coreset. Our work improves on streamkm++ w.r.t. query runtime.

Roadmap: We present preliminaries in Section II, background for streaming clustering in Section III and then the algorithms CC, RCC, and OnlineCC in Section IV, along with their proofs of correctness and quality guarantees. We then present experimental results in Section V.

II. Preliminaries and Notation

We work with points from the d-dimensional Euclidean space $\mathbb{R}^d$ for integer $d > 0$. Each point is associated with a positive integral weight (1 if unspecified). For points $x, y \in \mathbb{R}^d$, let $D(x, y) = \|x - y\|$ denote the Euclidean distance between x and y. Extending this notation, the distance from a point $x \in \mathbb{R}^d$ to a point set $\Psi \subset \mathbb{R}^d$ is $D(x, \Psi) = \min_{\psi\in\Psi} \|x - \psi\|$. In this notation, the *k-means clustering problem* is as follows:

Problem 1 (k-means Clustering) *Given a set $P \subseteq \mathbb{R}^d$ with n points and a weight function $w: P \to \mathbb{Z}^+$, find a point set $\Psi \subseteq \mathbb{R}^d$, $|\Psi| = k$, that minimizes the objective function*

$$\phi_\Psi(P) = \sum_{x\in P} w(x) \cdot D^2(x, \Psi) = \sum_{x\in P} \min_{\psi\in\Psi} \left(w(x) \cdot \|x - \psi\|^2\right).$$

Streams: A stream $\mathcal{S} = e_1, e_2, \ldots$ is an ordered sequence of points, where e_i is the i-th point observed by the algorithm. For $t > 0$, let $\mathcal{S}(t)$ denote the first t entries of the stream: $e_1, e_2, \ldots, e_t$. For $0 < i \leq j$, let $\mathcal{S}(i, j)$ denote the substream $e_i, e_{i+1}, \ldots, e_j$. Define $S = \mathcal{S}(1, n)$ be the whole stream observed until e_n, where n is, as before, the total number of points observed so far.

k-means++ Algorithm: Our algorithms rely on a batch k-means algorithm as a subroutine. We use the k-means++ algorithm [6], which has the following properties:

Theorem 1 (Theorem 3.1 in [6]) *On an input set of n points $P \subseteq \mathbb{R}^d$, the k-means++ algorithm returns a set Ψ of k centers such that $\mathbf{E}[\phi_\Psi(P)] \leq 8(\ln k + 2)\phi_{OPT}(P)$ where $\phi_{OPT}(P)$ is the*

[1] CT is essentially the streamkm++ algorithm [1] and [2] except it has a more flexible rule for merging coresets.

Algorithm 1: Stream Clustering Driver

```
1  def StreamCluster-Update(H, p)
       ▷ Insert points into D in batches of size m
2      H.n ← H.n + 1
3      Add p to H.C
4      if (|H.C| = m) then
5          H.D.Update(H.C, H.n/m)
6          H.C ← ∅
7  def StreamCluster-Query()
8      C_1 ← H.D.Coreset()
9      return k-means++(k, C_1 ∪ H.C)
```

optimal k-means clustering cost for P. The time complexity of the algorithm is $O(nkd)$.

Coresets and Their Properties: Our clustering method builds on the concept of a *coreset*, a small-space representation of a weighted point set that (approximately) preserves desirable properties of the original point set. A variant suitable for k-means clustering is as follows:

Definition 1 (k-means Coreset) *For a weighted point set $P \subseteq \mathbb{R}^d$, integer $k > 0$, and parameter $0 < \varepsilon < 1$, a weighted set $C \subseteq \mathbb{R}^d$ is said to be a (k, ε)-coreset of P for the k-means metric, if for any set Ψ of k points in $\mathbb{R}^d$, we have*

$$(1 - \varepsilon)\phi_\Psi(P) \leq \phi_\Psi(C) \leq (1 + \varepsilon)\phi_\Psi(P)$$

Throughout, we use the term "coreset" to always refer to a k-means coreset. When k is clear from the context, we simply say an ε-coreset. For integer $k > 0$, parameter $0 < \varepsilon < 1$, and weighted point set $P \subseteq \mathbb{R}^d$, we use the notation $\texttt{coreset}(k, \varepsilon, P)$ to mean a (k, ε)-coreset of P. We use the following observations from [5].

Observation 1 ([5]) *If C_1 and C_2 are each (k, ε)-coresets for disjoint multi-sets P_1 and P_2 respectively, then $C_1 \cup C_2$ is a (k, ε)-coreset for $P_1 \cup P_2$.*

Observation 2 ([5]) *Let k be fixed. If C_1 is ε_1-coreset for C_2, and C_2 is a ε_2-coreset for P, then C_1 is a $((1 + \varepsilon_1)(1 + \varepsilon_2) - 1)$-coreset for P.*

While our algorithms can work with any method for constructing coresets, an algorithm due to Feldman and Langberg [15] provides the following guarantees:

Theorem 2 (Theorem 15.7 in [15]) *Let $0 < \delta < \frac{1}{2}$ and let n denote the size of point set P. There exists an algorithm to compute $\texttt{coreset}(k, \varepsilon, P)$ with probability at least $1 - \delta$. The size of the coreset is $O\left(\frac{kd + \log(1/\delta)}{\varepsilon^6}\right)$, and the construction time is $O\left(ndk + \log^2\left(\frac{1}{\delta}\right)\log^2 n + |\texttt{coreset}(k, \varepsilon, P)|\right)$.*

III. Streaming Clustering and Coreset Trees

To provide context for how algorithms in this paper will be used, we describe a generic "driver" algorithm for streaming clustering. We also discuss the coreset tree (CT) algorithm. This is both an example of how the driver works with a specific implementation and a quick review of an algorithm from prior work that our algorithms build upon.

Algorithm 2: Coreset Tree Update

```
   ▷ Input: bucket b
1  def CT-Update(b)
2      Append b to Q_0
3      j ← 0
4      while |Q_j| ≥ r do
5          U ← coreset(k, ε, ∪_{B∈Q_j} B)
6          Append U to Q_{j+1}
7          Q_j ← ∅
8          j ← j + 1
9  def CT-Coreset ()
10     return ⋃_j {⋃_{B∈Q_j} B}
```

A. Driver Algorithm

The "driver" algorithm (presented in Algorithm 1) is initialized with a specific implementation of a clustering data structure $\mathcal{D}$ and a batch size m. It internally keeps state inside an object $\mathcal{H}$. It groups arriving points into batches of size m and inserts into the clustering data structure at the granularity of a batch. $\mathcal{H}$ stores additional state, the number of points received so far in the stream $\mathcal{H}.n$, and the current batch of points $\mathcal{H}.C$. Subsequent algorithms in this paper, including CT, are implementations for the clustering data structure $\mathcal{D}$.

B. CT: r-way Merging Coreset Tree

The *r-way coreset tree* (CT) turns a traditional batch algorithm for coreset construction into a streaming algorithm that works in limited space. Although the basic ideas are the same, our description of CT generalizes the coreset tree of Ackermann et al. [1], which is the special case when $r = 2$.

The Coreset Tree: A coreset tree Q maintains *buckets* at multiple levels. The buckets at level 0 are called *base buckets*, which contain the original input points. The size of each base bucket is specified by a parameter m. Each bucket above that is a coreset summarizing a segment of the stream observed so far. In an r-way CT, level ℓ has between 0 and $r - 1$ (inclusive) buckets, each a summary of r^ℓ base buckets.

Initially, the coreset tree is empty. After observing n points in the stream, there will be $N = \lfloor n/m \rfloor$ base buckets (level 0). Some of these base buckets may have been merged into higher-level buckets. The distribution of buckets across levels obeys the following invariant:

> If N is written in base r as $N = (s_q, s_{q-1}, \ldots, s_1, s_0)_r$, with s_q being the most significant digit (i.e., $N = \sum_{i=0}^{q} s_i r^i$), then *there are exactly s_i buckets in level i.*

How is a base bucket added? The process to add a base bucket is reminiscent of incrementing a base-r counter by one, where merging is the equivalent of transferring the carry from one column to the next. More specifically, CT maintains a sequence of sequences $\{Q_j\}$, where Q_j is the buckets at level j. To incorporate a new bucket into the coreset tree, CT-Update, presented in Algorithm 2, first adds it at level 0. When the number of buckets at any level i of the tree reaches r, these buckets are merged, using the coreset algorithm, to form a single bucket at level $(i + 1)$, and the process is repeated until there are fewer than r buckets at all levels of the tree. An

example of how the coreset tree evolves after the addition of base buckets is shown in Figure 1.

How to answer a query? The algorithm simply unions all the (active) buckets together, specifically $\bigcup_j\{\bigcup_{B\in Q_j} B\}$. Notice that the driver will combine this with a partial base bucket before deriving the k-means centers.

We present lemmas stating the properties of the CT algorithm. Due to space constraints, some of the proofs are omitted, and can be found in the long version of the paper [16]. We use the following definition in proving clustering guarantees.

Definition 2 (Level-ℓ Coreset) *For $\ell \in \mathbb{Z}_{\geq 0}$, a (k, ε, ℓ)-coreset of a point set $P \subseteq \mathbb{R}^d$, denoted by $\texttt{coreset}(k, \varepsilon, \ell, P)$, is as follows: The level-0 coreset of P is P. For $\ell > 0$, a level-ℓ coreset of P is a coreset of the union of C_i's (i.e., $\texttt{coreset}(k, \varepsilon, \cup_{i=1}^t C_i)$), where each C_i is a level-ℓ_i coreset, $\ell_i < \ell$, of P_i such that $\{P_j\}_{j=1}^t$ forms a partition of P.*

Lemma 1 *For a point set P, parameter $\varepsilon > 0$, and integer $\ell \geq 0$, if $C = \texttt{coreset}(k, \varepsilon, \ell, P)$ is a level ℓ-coreset of P, then $C = \texttt{coreset}(k, \varepsilon', P)$ where $\varepsilon' = (1+\varepsilon)^\ell - 1$.*

Fact 1 *After observing N base buckets, the number of levels in the coreset tree CT satisfies $\ell = \max\{j | Q_j \neq \emptyset\}$, is $\ell \leq \frac{\log N}{\log r}$.*

The accuracy of a coreset is given by the following lemma, since it is clear that a level-ℓ bucket is a level-ℓ coreset of its responsible range of base buckets.

Lemma 2 *Let $\varepsilon = (c \log r)/\log N$ where c is a small enough constant. After observing stream $S = \mathcal{S}(1, n)$, a clustering query StreamCluster-Query returns a set of k centers Ψ of S whose clustering cost is a $O(\log k)$-approximation to the optimal clustering for S.*

The memory and time cost of CT is as follows:

Lemma 3 *Let N be the number of buckets observed so far. Algorithm CT, including the driver, takes amortized $O(kd)$ time per point, using $O\left(\frac{mdr \log N}{\log r}\right)$ memory. The amortized cost of answering a query is $O\left(\frac{kdm}{q} \cdot \frac{r \log N}{\log r}\right)$ per point.*

As evident from the above lemma, answering a query using CT is expensive compared to the cost of adding a point. More precisely, when queries are made rather frequently—every q points, $q < O(rm \cdot \log_r N) = \widetilde{O}(rkd \cdot \log_r N)$—the cost of query processing is asymptotically greater than the cost of handling point arrivals. We address this issue in the next section.

IV. Clustering Algorithms with Fast Queries

This section describes algorithms for streaming clustering with an emphasis on query time.

A. Algorithm CC: Coreset Tree with Caching

The CC algorithm uses the idea of "coreset caching" to speed up query processing by reusing coresets that were constructed during prior queries. In this way, it can avoid merging a large number of coresets at query time. When compared with CT, the CC algorithm can answer queries faster, while maintaining nearly the same processing time per element.

In addition to the coreset tree CT, the CC algorithm also has an additional *coreset cache*, cache, that stores a subset of coresets that were previously computed. When a new query has to be answered, CC avoids the cost of merging coresets from multiple levels in the coreset tree. Instead, it reuses previously cached coresets and retrieves a small number of additional coresets from the coreset tree, thus leading to less computation at query time.

However, the level of the resulting coreset increases linearly with the number of merges a coreset is involved in. For instance, suppose we recursively merged the current coreset with the next arriving batch to get a new coreset, and so on, for N batches. The resulting coreset will have a level of $\Theta(N)$, which can lead to a very poor clustering accuracy. Additional care is needed to ensure that the level of a coreset is controlled while caching is used.

Details: Each cached coreset is a summary of base buckets 1 through some number u. We call this number u as the *right endpoint* of the coreset and use it as the key/index into the cache. We call the interval $[1, u]$ as the "span" of the bucket. To explain which coresets are cached by the algorithm, we introduce the following definitions.

For integers $n > 0$ and $r > 0$, consider the unique decomposition of n according to powers of r as $n = \sum_{i=0}^{j} \beta_i r^{\alpha_i}$, where $0 \leq \alpha_0 < \alpha_1 \ldots < \alpha_j$ and $0 < \beta_i < r$ for each i. The β_is can be viewed as the non-zero digits in the representation of n as a number in base r. Let $\texttt{minor}(n, r) = \beta_0 r^{\alpha_0}$, the smallest term in the decomposition, and $\texttt{major}(n, r) = n - \texttt{minor}(n, r)$. Note that when n is of the form βr^α where $0 < \beta < r$ and $\alpha \geq 0$, $\texttt{major}(n) = 0$.

For $\kappa = 1 \ldots j$, let $n_\kappa = \sum_{i=\kappa}^{j} \beta_i r^{\alpha_i}$. n_κ can be viewed as the number obtained by dropping the κ smallest non-zero digits in the representation of n as a number in base r. The set $\texttt{prefixsum}(n, r)$ is defined as $\{n_\kappa | \kappa = 1 \ldots j\}$. When n is of the form βr^α where $0 < \beta < r$, $\texttt{prefixsum}(n, r) = \emptyset$.

For instance, suppose $n = 47$ and $r = 3$. Since $47 = 1 \cdot 3^3 + 2 \cdot 3^2 + 2 \cdot 3^0$, we have $\texttt{minor}(47, 3) = 2, \texttt{major}(47, 3) = 45$, and $\texttt{prefixsum}(47, 3) = \{27, 45\}$.

CC caches every coreset whose right endpoint is in $\texttt{prefixsum}(N, r)$. When a query arrives after N batches, the task is to compute a coreset whose span is $[1, N]$. CC partitions $[1, N]$ as $[1, N_1] \cup [N_1 + 1, N]$ where $N_1 = \texttt{major}(N, r)$. Out of these two intervals, $[1, N_1]$ is already available in the cache, and $[N_1 + 1, N]$ is retrieved from the coreset tree, through the union of no more than $(r-1)$ coresets. This needs a merge of no more than r coresets. This is in contrast with CT, which may need to merge as many as $(r-1)$ coresets at each level of the tree, resulting in a merge of up to $(r-1)\log N$ coresets at query time. The algorithm for maintaining the cache and answering clustering queries is shown in Algorithm 3. See Figure 1 for an example of how the CC algorithms updates the cache and answers queries using cached coresets.

Note that to keep the size of the cache small, as new base buckets arrive, CC-Update will ensure that "stale" or unnecessary coresets are removed.

The following lemma relates what the cache stores with the number of base buckets observed so far, guaranteeing that Algorithm 3 can find the required coreset.

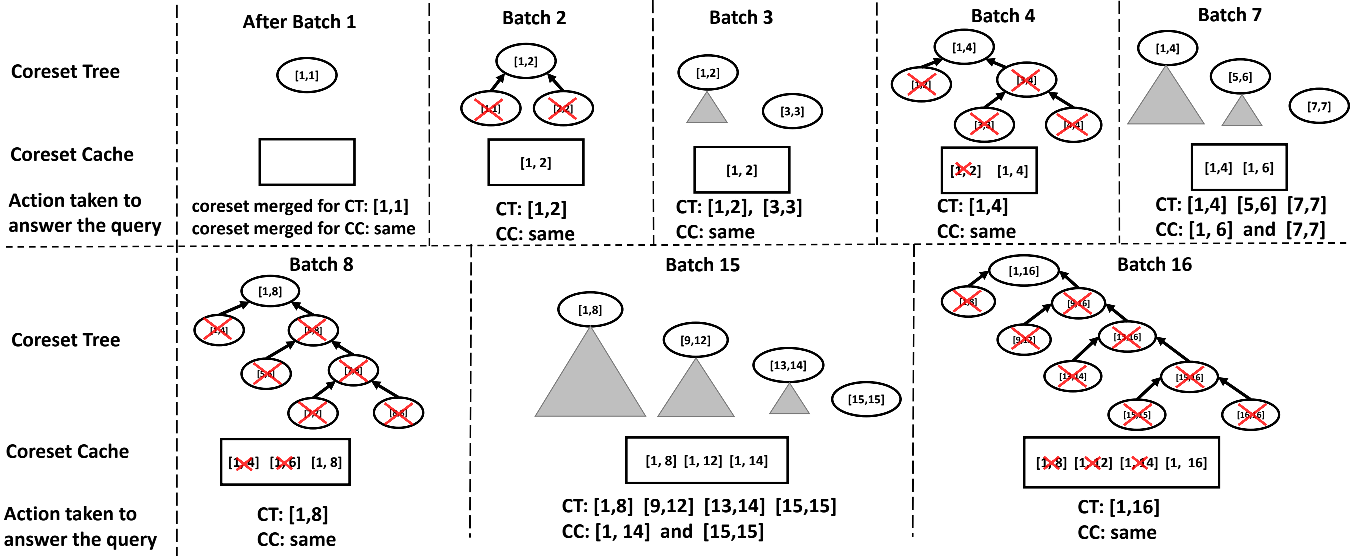


Fig. 1. Illustration of Algorithm CC, showing the states of coreset tree and cache after batch 1, 2, 3, 4, 7, 8, 15 and 16. The notation $[l, r]$ denotes a coreset of all points in buckets l to r, both endpoints inclusive. The coreset tree consists of a set of coresets, each of which is a base bucket or has been formed by merging multiple coresets. Whenever a coreset is merged into a another coreset (in the tree) or discarded (in the cache), the coreset is marked with an "X". We suppose that a clustering query arrives after seeing each batch, and describe the actions taken to answer this query (1) if only CT was used, or (2) if CC was used along with CT.

Lemma 4 *Immediately before base bucket N arrives, each $y \in \texttt{prefixsum}(N, r)$ appears in the key set of* `cache`.

Proof: Proof is by induction on N. The base case $N = 1$ is trivially true, since $\texttt{prefixsum}(1, r)$ is empty. For the inductive step, assume that at the beginning of batch N, each $y \in \texttt{prefixsum}(N, r)$ appears in `cache`. By Fact 2, we know that $\texttt{prefixsum}(N+1, r) \subseteq \texttt{prefixsum}(N, r) \cup \{N\}$. Using this, every bucket with a right endpoint in $\texttt{prefixsum}(N+1, r)$ is present in `cache` at the beginning of batch $(N+1)$, except for the coreset with right endpoint N. But the algorithm adds the coreset for this bucket to the cache, if r divides N. Hence, the inductive step is proved. ■

Fact 2 *Let $r \geq 2$. For each $N \in \mathbb{Z}_+$, $\texttt{prefixsum}(N+1, r) \subseteq \texttt{prefixsum}(N, r) \cup \{N\}$.*

Since $\texttt{major}(N, r) \in \texttt{prefixsum}(N, r)$ for each N, we can always retrieve the bucket with span $[1, \texttt{major}(N, r)]$ from `cache`.

Lemma 5 *When queried after inserting batch N, Algorithm* `CC-Coreset` *returns a coreset whose level in no more than $\left\lceil \frac{2 \log N}{\log r} \right\rceil - 1$.*

Proof: Let $\chi(N)$ denote the number of non-zero digits in the representation of N as a number in base r. We show that the level of the coreset returned by Algorithm `CC-Coreset` is no more than $\left\lceil \frac{\log N}{\log r} \right\rceil + \chi(N) - 1$. Since $\chi(N) \leq \left\lceil \frac{\log N}{\log r} \right\rceil$, the lemma follows.

The proof is by induction on $\chi(N)$. If $\chi(N) = 1$, then $\texttt{major}(N, r) = 0$, and the coreset is retrieved directly from the coreset tree Q. By Fact 1, each coreset in Q is at a level no more than $\lceil \frac{\log N}{\log r} \rceil$, and the base case follows. Suppose the claim was true for all N such that $\chi(N) = t$. Consider N such that $\chi(N) = (t+1)$. The algorithm computes $N_1 = \texttt{major}(N, r)$, and retrieves the coreset with span $[1, N_1]$ from the cache. Note that $\chi(N_1) = t$. By the inductive hypothesis, b, the coreset for span $[1, N_1]$ is at a level $\left\lceil \frac{\log N}{\log r} \right\rceil + t - 1$. The coresets for span $[N_1 + 1, N]$ are retrieved from the coreset tree; note there are multiple such coresets, but each of them is at a level no more than $\left\lceil \frac{\log N}{\log r} \right\rceil$, using Fact 1. Their union is denoted by a. The level of the final coreset for span $[1, N]$ is no more than $\left\lceil \frac{\log N}{\log r} \right\rceil + t$, proving the inductive case. ■

Let the accuracy parameter $\varepsilon = \frac{c \log r}{2 \log N}$, where $c < \ln 1.1$.

Lemma 6 *After observing N batches, Algorithm* `StreamCluster-Query` *when using clustering data structure* CC*, returns a set of k points whose clustering cost is within a factor of $O(\log k)$ of the optimal k-means clustering cost.*

Proof: We provide a sketch of the proof here. From Lemma 5, we know that the level of a coreset returned is no more than $\left\lceil \frac{2 \log N}{\log r} \right\rceil - 1$. Using Lemma 1, the returned coreset, say C, is an ε'-coreset where $\varepsilon' = \left[\left(1 + \frac{c \log r}{2 \log N}\right)^{\frac{2 \log N}{\log r}} - 1 \right] \leq \left[e^{\left(\frac{c \log r}{2 \log N}\right) \cdot \frac{2 \log N}{\log r}} - 1 \right] < 0.1$.

Let P denote the point set consisting all buckets from 1 till N. Algorithm `StreamCluster-Query` returns clusters Ψ by running k-`means++` on a coreset $C(P)$ with an approximation factor no more than 0.1. Suppose Ψ_1 was the optimal cluster centers for P and Ψ_2 was the optimal cluster centers for $C(P)$. From Theorem 1, we know that Ψ is an $O(\log k)$ approximation to the optimal on $C(P)$, i.e. Ψ_2. By the property of k-means coresets (Definition 1), we know that the cost of Ψ_1 on $C(P)$ is close to the cost of Ψ_1 on P, and the cost of Ψ on $C(P)$ is close to the cost of Ψ on P. Since the cost of Ψ_1 on $C(P)$ is at most the cost of Ψ_2 on $C(P)$, we conclude that the cost of Ψ on P is within $O(\log k)$ of the optimal clustering of P. ■

Lemma 7 *Algorithm 3 processes a stream of points using*

Algorithm 3: Coreset Tree with Caching: Algorithm Description

```
1  def CC-Init(r, k, ε)
2      Remember the parameters r, k, and ε.
3      Q ← CT-Init(r, k, ε) ▷ The coreset tree
4      cache ← ∅
5  def CC-Update(b, N)
       ▷ b is a batch and N is the number of batches
         received from stream so far.
6      Remember N
7      Q.CT-Update(b, N)
       ▷ May need to insert a coreset into cache
8      if r divides N then
9          c ← CC-Coreset()
10         Add coreset c to cache using key N
11         Remove from cache each bucket whose key
           does not appear in prefixsum(N + 1)
12 def CC-Coreset()
       ▷ Return a coreset of points in buckets 1 till N
13     N1 ← major(N, r) and N2 ← minor(N, r)
14     Let N2 = βr^α where α and β < r are positive integers
       ▷ a is the coreset for buckets
         N1 + 1, N1 + 2, ..., (N1 + N2) = N and is retrieved
         from the coreset tree
15     a ← ∪_{B∈Q_α} B
       ▷ b is the coreset spanning [1, N1], retrieved from
         the cache
16     if N1 is 0 then
17         b ← ∅
18     else
19         b ← cache.lookup(N1)
20     C ← coreset(k, ε, a ∪ b)
21     return C
```

amortized time $O(kd)$ per point, using memory of $O\left(\frac{mdr \log N}{\log r}\right)$. The amortized cost of answering a query is $O\left(\frac{kdm}{q} \cdot r\right)$.

Proof: The runtime for Algorithm CC-Update is the sum of the times to update the coreset tree Q and to update cache. We know from Lemma 3 that the time to update the coreset is $O(kd)$ per point. To update the cache, CC-Update inserts a new coreset into the cache every r batches. The cost of computing this coreset is $O(kmdr)$. Averaging over the mr points in r batches, the cost of maintaining cache is $O(kd)$ per point. The overall update time for CC-Update is $O(kd)$ per point.

The coreset tree Q uses space $O\left(\frac{mdr \log N}{\log r}\right)$. After processing batch N, cache only stores those buckets b corresponding to prefixsum$(N + 1, r)$. The number of such buckets possible is $O(\log N/\log r)$, so that the space cost of cache is $O(md \log N/\log r)$. The space complexity follows.

At query time, Algorithm CC-Coreset combines no more than r buckets, out of which there is no more than one bucket from the cache, and no more than $(r - 1)$ from the coreset tree. It is necessary to run k-means++ on $O(mr)$ points using time $O(kdmr)$. Since there is a query every q points, the amortized query time per point is $O\left(\frac{kdmr}{q}\right)$. ∎

B. Algorithm RCC: Recursive Coreset Cache

There are still a few issues with the CC data structure. First, the level of the coreset finally generated is $O(\log_r N)$. Since theoretical guarantees on the approximation quality of clustering worsen with the number of levels of the coreset, it is natural to ask if the level can be further reduced to $O(1)$. Moreover, the time taken to process a query is linearly proportional to r; we wish to reduce the query time even more. While it is natural to aim to simultaneously reduce the level of the coreset as well as the query time, at first glance, these two goals seem to be inversely related. It seems that if we decreased the level of a coreset (better accuracy), then we will have to increase the merge degree, which would in turn increase the query time. For example, if we set $r = \sqrt{N}$, then the level of the resulting coreset is $O(1)$, but the query time will be $O(\sqrt{N})$.

In the following, we present a solution RCC that uses the idea of coreset caching in a recursive manner to achieve both a low level of the coreset, as well as a small query time. In our approach, we keep the merge degree of nodes relatively high, thus keeping the levels of coresets low. At the same time, we use coreset caching even within a single level of a coreset tree, so that there is no need to merge r coresets at query time. Special care is required for coreset caching in this case, so that the level of the coreset does not increase significantly.

For instance, suppose we built another coreset tree with merge degree 2 for the $O(r)$ coresets within a single level of the current coreset tree, this would lead to a level of $\log r$. At query time, we will need to aggregate $O(\log r)$ coresets from this tree, in addition to a coreset from the coreset cache. So, this will lead to a level of $O\left(\max\left\{\frac{\log N}{\log r}, \log r\right\}\right)$, and a query time proportional to $O(\log r)$. This is an improvement from the coreset cache, which has a query time proportional to r and a level of $O\left(\frac{\log N}{\log r}\right)$.

We can take this idea further by recursively applying the same idea to the $O(r)$ buckets within a single level of the coreset tree. Instead of having a coreset tree with merge degree 2, we use a tree with a higher merge degree, and then have a coreset cache for this tree to reduce the query time, and apply this recursively within each tree. This way we can approach the ideal of a small level and a small query time. We are able to achieve interesting tradeoffs, as shown in Table II. To keep the level of the resulting coreset low, along with the coreset cache for each level, we also maintain a list of coresets at each level, like in the CT algorithm. To merge coresets to a higher level, we use the list, rather than the recursive coreset cache.

More specifically, the RCC data structure is defined inductively as follows. For integer $i \geq 0$, the RCC data structure of order i is denoted by RCC(i). RCC(0) is a CC data structure with a merge degree of $r_0 = 2$. For $i > 0$, RCC(i) consists of:

- cache(i), a coreset cache storing previous coresets.
- For each level $\ell = 0, 1, 2, \ldots$, there are two structures. One is a list of buckets L_ℓ, similar to the structure Q_ℓ in a coreset tree. The maximum length of a list is $r_i = 2^{2^i}$. Another is an RCC$_\ell$ structure which is a RCC structure of a lower order $(i - 1)$, which stores the same information as L_ℓ, except, in a way that can be quickly retrieved during a query.

The main data structure $\mathcal{R}$ is initialized as $\mathcal{R} =$ RCC-Init(ι), for a parameter ι, to be chosen. Note that ι is the highest order

Algorithm 4: RCC-Init(i)

1 $\mathcal{R}$.order $\leftarrow i$, $\mathcal{R}$.cache $\leftarrow \emptyset$, $\mathcal{R}.r \leftarrow 2^{2^i}$
  ▷ *N is the number of batches so far*
2 $\mathcal{R}.N \leftarrow 0$
3 **foreach** $\ell = 0, 1, 2, \ldots$ **do**
4   $\mathcal{R}.L_\ell \leftarrow \emptyset$
5   **if** $\mathcal{R}$.*order* > 0 **then** $\mathcal{R}.R_\ell \leftarrow$ RCC-Init(order $- 1$)
6
7 **return** R

Algorithm 5: $\mathcal{R}$.RCC-Update(b)

  ▷ *b is a batch of points*
1 $\mathcal{R}.N \leftarrow \mathcal{R}.N + 1$
  ▷ *Insert b into $\mathcal{R}.L_0$ and merge if needed*
2 Append b to $\mathcal{R}.L_0$.
3 **if** $\mathcal{R}$.*order* > 0 **then**
4   recursively update $\mathcal{R}.R_0$ by $\mathcal{R}.R_0$.RCC-Update(b)
5
6 $\ell \leftarrow 0$
7 **while** ($|\mathcal{R}.L_\ell| = \mathcal{R}.r$) **do**
8   $b' \leftarrow$ BucketMerge($\mathcal{R}.L_\ell$)
9   Append b' to $\mathcal{R}.L_{\ell+1}$
10   **if** $\mathcal{R}$.*order* > 0 **then**
11   recursively update $\mathcal{R}.R_{\ell+1}$ by $\mathcal{R}.R_{\ell+1}$.RCC-Update(b)
12
    ▷ *Empty the list of coresets*
13
14   $\mathcal{R}.L_\ell \leftarrow \emptyset$; ▷ *Empty the cache*
15
16   **if** $\mathcal{R}$.*order* > 0 **then**
17     $\mathcal{R}.R_\ell \leftarrow$ RCC-Init($\mathcal{R}$.order $- 1$)
18
19 **if** $\mathcal{R}.r$ *divides* $\mathcal{R}.N$ **then**
20   Bucket $b' \leftarrow \mathcal{R}$.RCC-Coreset()
21   Add b' to $\mathcal{R}$.cache with right endpoint $\mathcal{R}.N$
22   From $\mathcal{R}$.cache, remove all buckets b'' such that right(b'') $\notin$ prefixsum($\mathcal{R}.N + 1$)

of the recursive structure. This is also called the "nesting depth" of the structure.

Lemma 8 *When queried after inserting N batches, Algorithm 6 using RCC(ι) returns a coreset whose level is $O\left(\frac{\log N}{2^\iota}\right)$. The amortized time cost of answering a clustering query is $O\left(\frac{kdm}{q}\iota\right)$ per point.*

Proof: Algorithm 6 retrieves a few coresets from RCC of different orders. From the outermost structure $R_\iota = \text{RCC}(\iota)$, it retrieves one coreset c from cache(ι). Using an analysis similar to Lemma 5, the level of b_ι is no more than $2\frac{\log N}{\log r_\iota}$.

Note that for $i < \iota$, the maximum number of coresets that will be inserted into RCC(i) is $r_{i+1} = r_i^2$. The reason is that inserting r_{i+1} buckets into RCC(i) will lead to the corresponding list structure for RCC(i) to become full. At this point, the list and the RCC(i) structure will be emptied out in Algorithm 5. From each recursive call to RCC(i), it can be similarly seen

Algorithm 6: $\mathcal{R}$.RCC-Coreset()

1 $B \leftarrow \mathcal{R}$.RCC-Getbuckets
2 $C \leftarrow$ coreset(k, ϵ, B)
3 **return** bucket ($C, 1, \mathcal{R}.N, 1 + \max_{b \in B}$ level(b))

Algorithm 7: $\mathcal{R}$.RCC-Getbuckets()

1 $N_1 \leftarrow$ major($\mathcal{R}.N, \mathcal{R}.r$)
2 $b_1 \leftarrow$ retrieve bucket with right endpoint N_1 from $\mathcal{R}$.cache
3 Let ℓ^* be the lowest numbered non-empty level among $\mathcal{R}.L_i, i \geq 0$.
4 **if** $\mathcal{R}$.*order* > 0 **then**
5   $B_2 \leftarrow \mathcal{R}.R_{\ell^*}$.RCC-Getbuckets()
6 **else**
7   $B_2 \leftarrow \mathcal{R}.L_{\ell^*}$
8 **return** $\{b_1\} \cup B_2$

that the level of a coreset retrieved from the cache is at level $2 \cdot \frac{\log r_i}{\log r_{i-1}}$, which is $O(1)$. The algorithm returns a coreset formed by the union of all the coresets, followed by a further merge step. Thus, the coreset level is one more than the maximum of the levels of all the coresets returned, which is $O(\frac{\log N}{\log r_\iota})$.

For the query cost, note that the number of coresets merged at query time is equal to the nesting depth of the structure ι. The query time equals the cost of running k-means++ on the union of all these coresets, for a total time of $O(kdm\iota)$. The amortized per-point cost of a query follows. ■

Lemma 9 *The memory consumed by RCC(ι) is $O(mdr_\iota)$. The amortized processing time is $O(kd\iota)$ per point.*

Proof: First, we note in RCC(i) for $i < \iota$, there are $O(1)$ lists L_ℓ. The reason is as follows. It can be seen that in order to get a single bucket in list L_2 within RCC(i), it is necessary to insert $r_i^2 = r_{i+1}$ buckets into RCC(i). Since this is the maximum number of buckets that will be inserted into RCC(i), there are no more than three levels of lists within each RCC(i) for $i < \iota$.

We prove by induction on i that RCC(i) has no more than $6r_i$ buckets. For the base case, $i = 0$, and we have $r_0 = 2$. In this case, RCC(0) has three levels, each with no more than 2 buckets. The number of buckets in the cache is also a constant for r_0, so that the total memory is no more than 6 buckets, due to the lists in different levels, and no more than 2 buckets in the cache, for a total of $8 = 4r_0$ buckets. For the inductive case, consider that RCC(i) has no more than three levels. The list at each level has no more than r_i buckets. The recursive structures R_ℓ within RCC(i) themselves have no more than $6r_{i-1}$ buckets. Adding the constant number of buckets within the cache, we get the total number of buckets within RCC(i) to be $3r_i + 4r_{i-1} + 2 = 3r_i + 6\sqrt{r_i} + 2 \leq 6r_i$, for $r_i \geq 16$, i.e. $i \geq 2$. Thus if ι is the nesting depth of the structure, the total memory consumed is $O(mdr_\iota)$, since each bucket requires $O(md)$ space.

For the processing cost, when a bucket is inserted into $\mathcal{R} = \text{RCC}(\iota)$, it is added to list L_0 within $\mathcal{R}$. The cost of maintaining these lists in $\mathcal{R}$ and $\mathcal{R}$.cache, including merging into higher level lists, is amortized $O(kd)$ per point, similar to the analysis in Lemma 7. The bucket is also recursively inserted into a RCC($\iota - 1$) structure, and a further structure within, and the

ι	coreset level at query	Query cost (per point)	update cost per point	Memory
$\log\log N - 3$	$O(1)$	$O(\frac{kdm}{q}\log\log N)$	$O(kd\log\log N)$	$O(mdN^{1/8})$
$\log\log N/2$	$O(\sqrt{\log N})$	$O(\frac{kdm}{q}\log\log N)$	$O(kd\log\log N)$	$O(md2^{\sqrt{\log N}})$

TABLE II. Possible tradeoffs for the RCC(ι) algorithm, based on the parameter ι, the nesting depth of the structure.

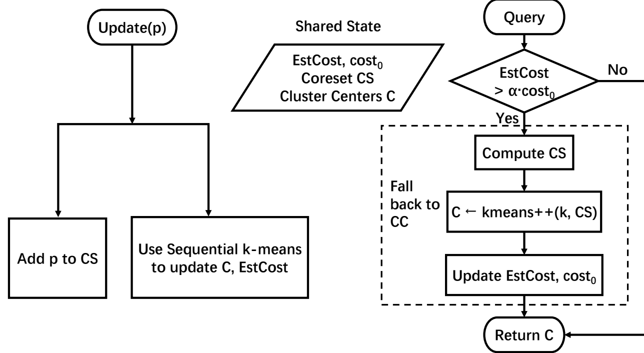


Fig. 2. Illustration of Algorithm OnlineCC.

amortized time for each such structure is $O(kd)$ per point. The total time cost is $O(kd\iota)$ per point. ■

Different tradeoffs are possible by setting ι to specific values. Some examples are shown in the Table II.

C. *Online Coreset Cache: a Hybrid of* CC *and* Sequential *k*-means

If we break down the query runtime of the algorithms considered so far, we observe two major components: (1) the construction of the coreset of all points seen so far, through merging stored coresets; and (2) the k-means++ algorithm applied on the resulting coreset. The focus of the algorithms discussed so far (CC and RCC) is on decreasing the runtime of the first component, coreset construction, by reducing the number of coresets to be merged at query time. But they still have to pay the cost of the second component k-means++, which is substantial in itself, since the runtime of k-means++ is $O(kdm)$, where m is the size of the coreset. To make further progress, we have to reduce this component. However, the difficulty in eliminating k-means++ at query time is that without an approximation algorithm such as k-means++, we have no way to guarantee that the returned clustering is an approximation to the optimal.

This section presents an algorithm, OnlineCC, which only occasionally runs k-means++ at query time, and most of the time, uses a much cheaper method that costs $O(1)$ to compute the clustering centers. OnlineCC uses a combination of CC and the Sequential k-means algorithm [7] (aka. Online Lloyd's algorithm) to maintain the cluster centers quickly while also providing a guarantee on the quality of clustering. Like Sequential k-means, it incrementall updates the current set of cluster centers for each arriving point. However, while Sequential k-means can process incoming points (and answer queries) extremely quickly, it cannot provide any guarantees on the quality of answers, and in some cases, the clustering quality can be very poor when compared with say, k-means++. To prevent such deterioration in clustering quality, our algorithm (1) occasionally falls back to CC, which is provably accurate, and (2) runs Sequential k-means so long as the clustering cost does not get much larger than the previous time CC was used. This ensures that our clusters always have a provable quality with respect to the optimal.

Algorithm 8: The Online Coreset Cache: A hybrid of CC and Sequential k-means algorithms

1 **def** *OnlineCC-Init*(k, ε, α)
2   Remember coreset approximation factor ε, merge-degree r, and parameter $\alpha > 1$ the threshold to switch the query processing to CC
   ▷ *C is the current set of cluster centers*
3   Initialize C by running k-means++ on set $\mathcal{S}_0$ consisting of the first $O(k)$ points of the stream
   ▷ *$cost_0$ is the clustering cost during the previous "fallback" to CC; EstCost is an estimate of the clustering cost of C on the stream so far*
4    $cost_0 \leftarrow$ clustering cost of C on $\mathcal{S}_0$
5   EstCost $\leftarrow cost_0$
6    $Q \leftarrow$ CC-Init(r, k, ε)

▷ *On receiving a new point p from the stream*
7 **def** *OnlineCC-Update*(p)
8   Assign p to the nearest center c_p in C
9   EstCost $\leftarrow$ EstCost $+ \|p - c_p\|^2$
   ▷ *centroid of c_p and p where w is the weight of c_p*
10    $c'_p \leftarrow (w \cdot c_p + p)/(w + 1)$
11   Update center c_p in C to c'_p
12   Add p to the current batch b. If $|b| = m$, then Q.CC-Update(b)

13 **def** *OnlineCC-Query*()
14   **if** *EstCost* $> \alpha \cdot cost_0$ **then**
15      $CS \leftarrow Q$.CC-Coreset$() \cup b$, where b is the current batch that has not been inserted into Q
16      $C \leftarrow k$-means++(k, CS)
17      $cost_0 \leftarrow \phi_C(CS)$, the k-means cost of C on CS
18   EstCost $\leftarrow cost_0/(1 - \varepsilon)$
19   **return** C

To accomplish this, OnlineCC also processes incoming points using CC, thus maintaining coresets of substreams of data seen so far. When a query arrives, it typically answers them in $O(1)$ time using the centers maintained using Sequential k-means. If, however, the clustering cost is significantly higher (by more than a factor of α for a parameter $\alpha > 1$) than the previous time that the algorithm fell back to CC, then the query processing again returns to CC to regenerate a coreset. One difficulty in implementing this idea is that (efficiently) maintaining an estimate of the current clustering cost is not easy, since each change in cluster centers can affect the contribution of a number of points to the clustering cost. To reduce the cost of maintenance, our algorithm keeps an upper bound on the clustering cost; as we show further, this is sufficient to give a provable guarantee on the quality of clustering. Further details on how the upper bound on the clustering cost is maintained, and how Sequential k-means and CC interact are shown in Algorithm 8, with a schematic illustration in Figure 2.

We state the properties of Algorithm OnlineCC. Proofs that are omitted can be found in [16].

Lemma 10 *In Algorithm 8, after observing point set P, if C is the current set of cluster centers, then* `EstCost` *is an upper bound on* $\phi_C(P)$.

Lemma 11 *When queried after observing point set P, the* `OnlineCC` *algorithm (Algorithm 8) returns a set of k points C whose clustering cost is within* $O(\log k)$ *of the optimal k-means clustering cost of P, in expectation.*

Proof: Let $\phi^*(P)$ denote the optimal k-`means` cost for P. We will show that $\phi_C(P) = O(\log k) \cdot \phi^*(P)$. There are two cases:

Case I: When C is directly retrieved from `CC`, Lemma 6 implies that $\mathbf{E}[\phi_C(P)] \leq O(\log k) \cdot \phi^*(P)$. This case is handled through the correctness of `CC`.

Case II: The query algorithm does not fall back to `CC`. We first note from Lemma 10 that $\phi_C(P) \leq$ `EstCost`. Since the algorithm did not fall back to `CC`, we have `EstCost` $\leq \alpha \cdot cost_0$. Since $cost_0$ was the result of applying `CC` to P_0, we have from Lemma 6 that $cost_0 \leq O(\log k) \cdot \phi^*(P_0)$. Since $P_0 \subseteq P$, we know that $\phi^*(P_0) \leq \phi^*(P)$. Putting together the above four inequalities, we have $\phi_C(P) = O(\log k) \cdot \phi^*(P)$. ■

V. Experimental Evaluation

This section describes an empirical study of the proposed algorithms, in comparison to the state-of-the-art clustering algorithms. Our goals are twofold: to understand the relative clustering accuracy and running time of different algorithms in the context of continuous queries, and to investigate how they behave under different settings of parameters.

A. Datasets

Our experiments use a number of real-world or semi-synthetic datasets, based on data from the UCI Machine Learning Repositories [17]. These are commonly used datasets in benchmarking clustering algorithms. Table III provides an overview of the datasets used.

Dataset	Number of Points	Dimension	Description
`Covtype`	581,012	54	forest cover type
`Power`	2,049,280	7	household power consumption
`Intrusion`	494,021	34	KDD Cup 1999
`Drift`	200,000	68	derived from US Census 1990

TABLE III. An overview of the datasets used in the experiments.

The *`Covtype`* dataset models the forest cover type prediction problem from cartographic variables. The dataset contains 581,012 instances and 54 integer attributes. The *`Power`* dataset measures electric power consumption in one household with a one-minute sampling rate over a period of almost four years. We remove entries with missing values, resulting in a dataset with 2,049,280 instances and 7 floating-point attributes. The *`Intrusion`* dataset is a 10%-subset of the *KDD Cup 1999* data. The task was to build a predictive model capable of distinguishing between normal network connections and intrusions. After ignoring symbolic attributes, we have a dataset with 494,021 instances and 34 floating-point attributes. To erase any potential special ordering within data, we randomly shuffle each dataset before using it as a data stream.

However, each of the above datasets, as well as most datasets used in previous works on streaming clustering, is not originally a streaming dataset; the entries are only read in some order and consumed as a stream. To better model the evolving nature of data streams and drifts center locations, we generate a semi-synthetic dataset, called `Drift`, which we derive from the `USCensus1990` dataset [17] as follows: The method is inspired by Barddal [18]. The first step is to cluster the `USCensus1990` dataset to compute 20 cluster centers and for each cluster, the standard deviation of the distances to the cluster center. Following that, the synthetic dataset is generated using the Radial Basis Function (RBF) data generator from the MOA stream mining framework [19]. The RBF generator moves the drifting centers with a user-given direction and speed. For each time step, the RBF generator creates 100 random points around each center using a Gaussian distribution with the cluster standard deviation. In total, the synthetic dataset contains 200,000 and 68 floating-point attributes.

B. Experimental Setup and Implementation Details

We implemented all the clustering algorithms in Java, and ran experiments on a desktop with Intel Core i5-4460 3.2GHz processor and 8GB main memory.

Algorithms: Our baseline algorithms are two prominent streaming clustering algorithms. (1) the `Sequential k-means` algorithm due to MacQueen [7], which is frequently implemented in clustering packages today. For `Sequential k-means` clustering, we use the implementation in Apache Spark MLLib [9], though ours is modified to run sequentially. Furthermore, the initial centers are set by the first k points in the stream instead of setting by random Gaussians, to ensure no clusters are empty. (2) We also implemented `streamkm++` [1], a current state-of-the-art algorithm that has good practical performance. `streamkm++` can be viewed as a special case of `CT` where the merge degree r is 2. The bucket size is $10k$, where k is the number of centers. [2]

For `CC`, we set the merge degree to 2, in line with `streamkm++`. For `RCC`, we use a maximum nesting depth of 3, so the merge degrees for different structures are $N^{\frac{1}{2}}, N^{\frac{1}{4}}$ and $N^{\frac{1}{8}}$, respectively. For `OnlineCC`, the threshold α is set to 1.2. To compute the `EstCost` after each fallback to `CC`, we need to know the value of the clusters' standard deviation D. This value is estimated using the coreset which is representative of the whole data received.

We use the batch k-`means++` algorithm as the baseline for clustering accuracy. This is expected to outperform every streaming algorithm. The k-`means++` algorithm, similar to [1, 2], is used to derive coresets. We also use k-`means++` as the final step to construct k centers of from the coreset, and take the best clustering out of five independent runs of k-`means++`; each instance of k-`means++` is followed by up to 20 iterations of Lloyd's algorithm to further improve clustering quality. The number of clusters k tested are 10, 15, 20, 25, 30. Finally, for each statistic, we report the median value from five independent runs of each algorithm to improve robustness.

We divide the input stream into batches of q points each, and present a query for cluster centers at the end of each batch.

[2] A larger bucket size such as $200k$ can yield slightly better clustering quality. But this led to a high runtime for `streamkm++`, especially when queries are frequent, so we use a smaller bucket size.

Hence, there is one query per batch of input points.

Metrics: We use three metrics: clustering accuracy, runtime and memory cost. The clustering accuracy is measured using the k-means cost, also known as the within cluster sum of squares (SSQ). We measure the average runtime of the algorithm per batch, as well as the total runtime over all batches. The runtime is split into two parts, (1) update time, the time required to update internal data structures upon receiving new data, and (2) query time, the time required to answer clustering queries. Finally, we consider the memory consumption through measuring the number of points stored in the coreset tree and the coreset cache. From the number of points, we estimate the number of bytes used, assuming that each dimension of a data point consumes 8 bytes (the size of a double-precision floating-point number).

C. Discussion of Experimental Results

Accuracy (k-means cost): Consider Figures 3 and 4. Figure 4 shows the k-means cost versus k when the query interval is 100 points. For the Intrusion data, the result of Sequential k-means is not shown since its cost much larger (by a factor of about 10^5) than the other methods. Not surprisingly, for all algorithms studied, the clustering cost decreases with k. For all the datasets, Sequential k-means always achieves the highest k-means cost, in some cases (such as Intrusion), much higher than other methods. This shows that Sequential k-means is consistently worse than the other methods, when it comes to clustering accuracy—this is as expected, since unlike the other methods, Sequential k-means does not have a theoretical guarantee on clustering quality. A similar trend is also observed on the plot with the k-means cost versus the number of points received, Figure 3.

The other algorithms, streamkm++, CC, RCC, and OnlineCC all achieve very similar clustering cost, on all datasets. In Figure 4, we also show the cost of running a batch algorithm k-means++ (followed by iterations of Lloyd's algorithm). We found that the clustering costs of the streaming algorithms are nearly the same as that of running the batch algorithm, which can see the input all at once! Indeed, we cannot expect the streaming clustering algorithms to perform any better than this.

Theoretically, it was shown that clustering accuracy improves with the merge degree r. Hence, RCC should have the best clustering accuracy (lowest clustering cost). But we do not observe such behaviors experimentally; RCC and streamkm++ show similar accuracy. However, their accuracy matches that of batch k-means++. A possible reason for this may be that our theoretical analyses of streaming clustering methods is too conservative, and/or there is some structure within real data that we can better exploit to predict clustering accuracy.

Update Time: Figure 6 shows the average update time per batch, versus the number of clusters, when the query interval $q = 100$. The update time of streamkm++, CC and OnlineCC all increase linearly with the number of centers, as the amortized update time is proportional to k. Note that CC and OnlineCC have slightly higher update time than streamkm++, since they have to update the coreset cache, unlike streamkm++. This shows that the overhead of maintaining the cache is small. The update time of OnlineCC is nearly the same as CC. Among the four algorithms, RCC has the largest update time, as it updates multiple levels of caches.

Query Time: Figure 7 shows the time taken for answering a single query versus the number of clusters, when $q = 100$. We see that OnlineCC has the fastest query time, followed by RCC, and CC, and finally by streamkm++. Note that the y-axis in Figure 7 is in log scale. We note that OnlineCC is significantly faster than the rest of the algorithms. For instance, it is about two orders of magnitude faster than streamkm++ for $q = 100$. This shows that the algorithm succeeds in achieving significantly faster queries than streamkm++, while maintaining the same clustering accuracy.

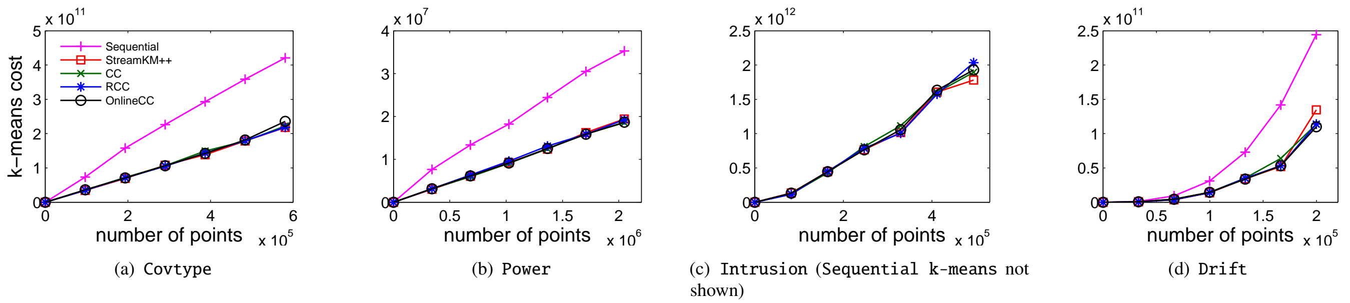
Total Time: Figure 8 shows the average runtime per batch (the sum of the update time and query times), as a function of k, for $q = 100$. For streamkm++, the query time dominates the update time, hence the total time is close to query time. For OnlineCC, however, the update time is greater than the query time, hence the total time is substantially larger than its query time. Overall, the total time of OnlineCC is still nearly 5–10x faster than streamkm++. Figure 5 shows the average runtime per batch versus the number of batches. We observe that for streamkm++, the average runtime per batch increases with the number of batches. This is because the query cost of streamkm++ increases with the number of batches, since the depth of the coreset tree increases as more points are inserted. In contrast, CC always answers a query through merging a constant number of coresets, and its query cost does not increase with the number of batches.

We next consider how the runtime varies with q, the query interval. Figure 9 shows the algorithm total run time as a function of the query interval q. Note that the update time does not change with q, and is not shown here. The trend for query time is similar to that shown for total time, except that the differences are more pronounced. We note that the total time for OnlineCC is consistently the smallest, and does not change with an increase in q. This is because OnlineCC essentially maintains the cluster centers on a continuous basis, while occasionally falling back to CC to recompute coresets, to improve its accuracy. For the other algorithms, including CC, RCC, and streamkm++, the query time and the total time decrease as q increases (and queries become less frequent).

Memory Usage: Finally, we report the memory cost in Table IV using $k = 30$; the trends are identical for other values of k. Evidently, streamkm++ uses the least memory since it only maintains the coreset tree. Because it also maintains a coreset cache, CC requires additional memory. Even then, CC's memory cost is less than 2x that of streamkm++. The memory cost of OnlineCC is similar to CC while RCC has the highest memory cost. This shows that the marked improvements in speed requires only a modest increase in the memory requirement, making the proposed algorithms practical and appealing.

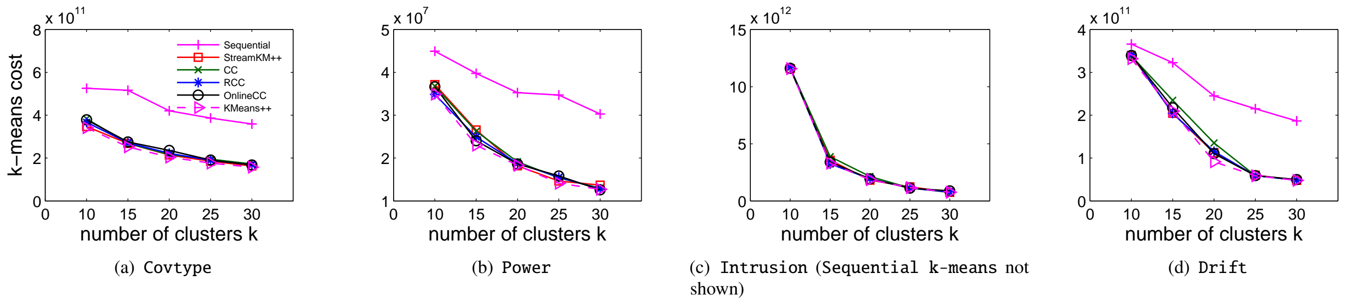
VI. Conclusion

We presented new streaming algorithms for k-means clustering. Compared to prior methods, our algorithms significantly speedup query processing, while offering provable guarantees on accuracy and memory cost. The general framework of "coreset caching" may be applicable to other streaming algorithms built around the Bentley-Saxe decomposition. For instance, applying it to streaming k-median seems natural. Many other open questions remain, including (1) improved handling of



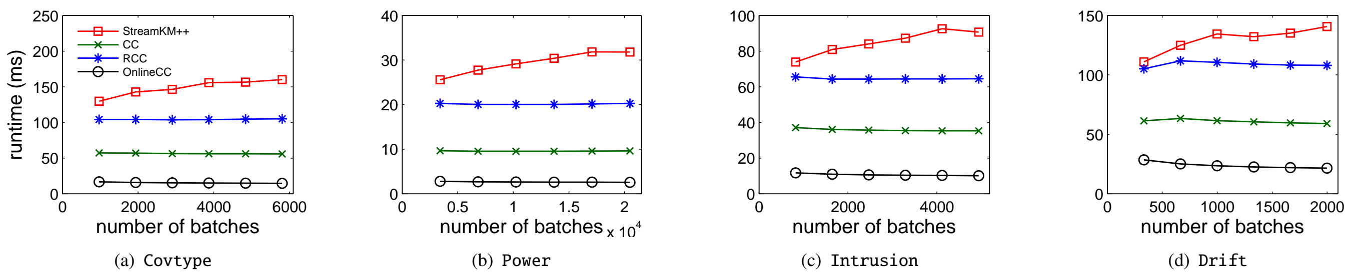
(a) Covtype (b) Power (c) Intrusion (Sequential k-means not shown) (d) Drift

Fig. 3. k-means cost vs. number of points. The number of clusters k is 20.



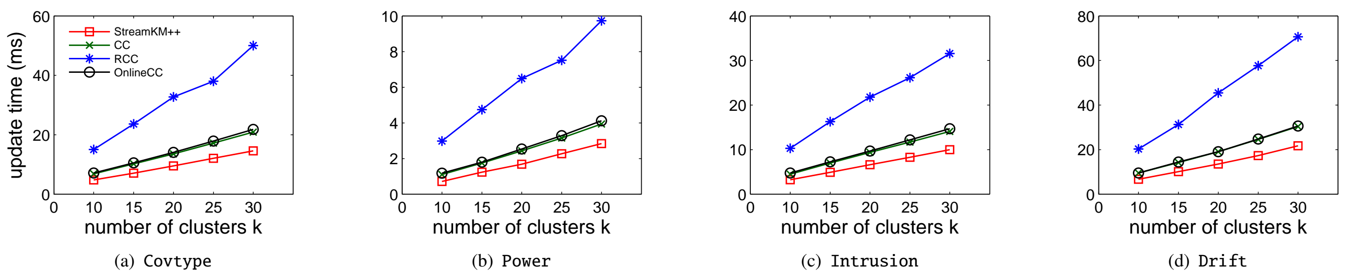
(a) Covtype (b) Power (c) Intrusion (Sequential k-means not shown) (d) Drift

Fig. 4. k-means cost vs. number of clusters k. The cost is computed at the end of observing all the points. k-means cost of Sequential k-means on Intrusion dataset is not shown in Figure (c), since it was orders of magnitude larger than the other algorithms.



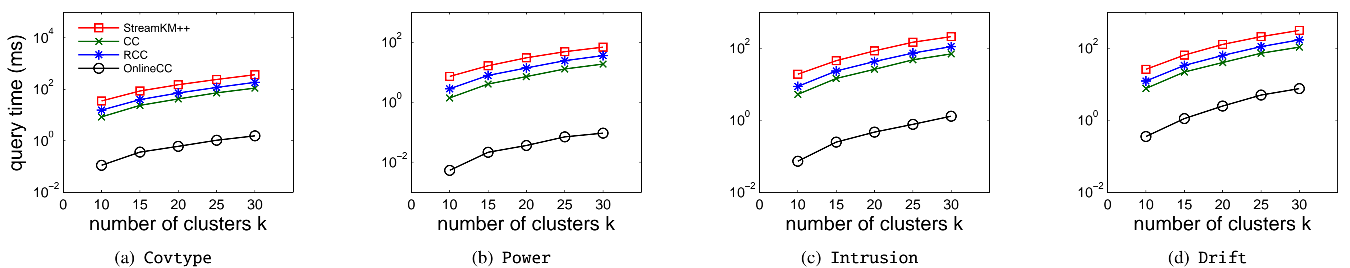
(a) Covtype (b) Power (c) Intrusion (d) Drift

Fig. 5. Average runtime per batch (milliseconds) vs. number of batches. The number of clusters k is 20.



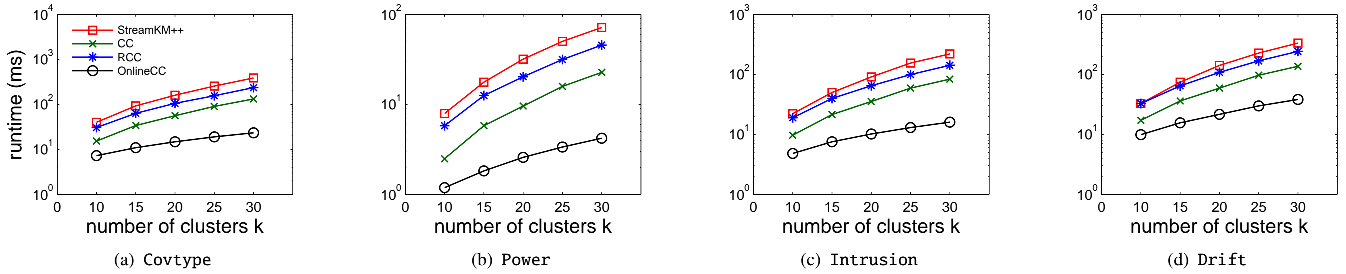
(a) Covtype (b) Power (c) Intrusion (d) Drift

Fig. 6. Average update time per batch (milliseconds) vs. number of clusters k. The query interval q is 100 points.



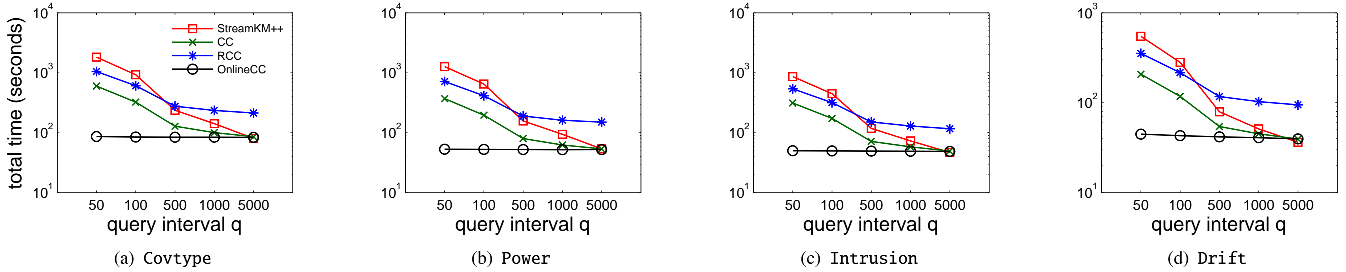
(a) Covtype (b) Power (c) Intrusion (d) Drift

Fig. 7. Average query time per batch (milliseconds) vs. number of clusters k. The query interval q is 100 points.



(a) Covtype (b) Power (c) Intrusion (d) Drift

Fig. 8. Average runtime per batch (milliseconds) vs. number of clusters k. The runtime is the sum of update time (per batch) and the query time (per batch). The query interval q is 100 points.



(a) Covtype (b) Power (c) Intrusion (d) Drift

Fig. 9. Total time (seconds) vs. query interval q. The total time is for the entire dataset overall batches. For every q points, there is a query for the cluster centers. The number of centers k is 20.

Dataset	Memory cost in points				Memory cost in Megabytes (MB)			
	streamkm++	CC	RCC	OnlineCC	streamkm++	CC	RCC	OnlineCC
Covtype	3300	6000	31800	6030	1.43	2.59	13.74	2.60
Power	3900	7200	64800	7230	0.22	0.40	3.63	0.4
Intrusion	3300	6000	31800	6030	0.90	1.63	8.65	1.64
Drift	3000	5400	22800	5430	1.63	2.94	12.40	2.95

TABLE IV. Memory cost of algorithms, number of clusters k is set to 30 .

concept drift, through the use of time-decaying weights, and (2) clustering on distributed and parallel streams.

References

[1] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "Streamkm++: A clustering algorithm for data streams," *J. Exp. Algorithmics*, vol. 17, no. 1, pp. 2.4:2.1–2.4:2.30, 2012.

[2] N. Ailon, R. Jaiswal, and C. Monteleoni, "Streaming k-means approximation," in *NIPS*, 2009, pp. 10–18.

[3] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams: Theory and practice," *IEEE TKDE*, vol. 15, no. 3, pp. 515–528, 2003.

[4] M. Shindler, A. Wong, and A. Meyerson, "Fast and accurate k-means for large datasets," in *NIPS*, 2011, pp. 2375–2383.

[5] S. Har-Peled and S. Mazumdar, "On coresets for k-means and k-median clustering," in *STOC*, 2004, pp. 291–300.

[6] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *SODA*, 2007, pp. 1027–1035.

[7] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[8] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Information Theory*, vol. 28, no. 2, pp. 129–136, 1982.

[9] X. Meng, J. K. Bradley, B. Yavuz, and et al., "MLlib: Machine Learning in Apache Spark," *J. Machine Learning Research*, vol. 17, pp. 1235–1241, 2016.

[10] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "A local search approximation algorithm for k-means clustering," *Computational Geometry*, vol. 28, no. 2 - 3, pp. 89 – 112, 2004.

[11] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," in *SIGMOD*, 1996, pp. 103–114.

[12] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *PVLDB*, 2003, pp. 81–92.

[13] J. L. Bentley and J. B. Saxe, "Decomposable searching problems i. static-to-dynamic transformation," *Journal of Algorithms*, vol. 1, pp. 301 – 358, 1980.

[14] S. Har-Peled and A. Kushal, "Smaller coresets for k-median and k-means clustering," *Discrete Computational Geometry*, vol. 37, no. 1, pp. 3–19, 2007.

[15] D. Feldman and M. Langberg, "A unified framework for approximating and clustering data," in *STOC*, 2011, pp. 569–578.

[16] Y. Zhang, K. Tangwongsan, and S. Tirthapura, "Streaming algorithms for k-means clustering with fast queries," https://arxiv.org/abs/1701.03826.

[17] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[18] J. P. Barddal, H. M. Gomes, F. Enembreck, and J. P. Barthes, "SNCStream+: Extending a high quality true anytime data stream clustering algorithm," *Information Systems*, vol. 62, pp. 60 – 73, 2016.

[19] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive Online Analysis," *J. Machine Learning Research*, vol. 11, pp. 1601–1604, 2010.