The Computer Journal Advance Access published April 9, 2015

doi:10.1093/comjnl/bxv023

Aggregate Profile Clustering for Streaming Analytics

Mehmet Ali Abbasoğlu, Buğra Gedik* and Hakan Ferhatosmanoğlu

Department of Computer Engineering, Bilkent University, Çankaya, Ankara 06800, Turkey
**Corresponding author: bgedik@cs.bilkent.edu.tr*

Many analytic applications require analyzing user interaction data. In particular, such data can be *aggregated* over a window to build user activity *profiles*. Clustering such *aggregate profiles* is useful for grouping together users with similar behaviors, so that common models could be built for them. In this paper, we present an approach for clustering profiles that are incrementally maintained over a stream of updates. Owing to the potentially large number of users and high rate of interactions, maintaining profile clusters can have high processing and memory resource requirements. To tackle this problem, we apply distributed stream processing. However, in the presence of distributed state, it is a major challenge to partition the profiles over nodes such that memory and computation balance is maintained, while keeping the clustering accuracy high. Furthermore, in order to adapt to potentially changing user interaction patterns, the partitioning of profiles to nodes should be continuously revised, yet one should minimize the migration of profiles so as not to disturb the online processing of updates. We develop a re-partitioning technique that achieves all these goals. To achieve this, we keep micro-cluster summaries at each node and periodically collect these summaries at a central node to perform re-partitioning. We use a greedy algorithm with novel affinity heuristics to revise the partitioning and update the routing tables without introducing a lengthy pause. We showcase the effectiveness of our approach using an application that clusters customers of a telecommunications company based on their aggregate calling profiles.

Keywords: aggregate profile clustering; data streaming; distributed clustering

Received 10 July 2014; revised 22 March 2015
Handling editor: Munevver Kokuer

1. INTRODUCTION

In data-intensive online services, besides the relatively static user personal information, there is an important source of dynamic user data in the form of user interaction streams. For instance, in telecommunications services, call detail records (CDRs) contain information to generate calling profiles of users (weekend caller, international caller, etc.), whereas in social media services, micro-blog posts contain information to create rich user profiles. In general, whenever a user of the service performs an online interaction over the provided service, a log entry that contains the details of the action streams into the data centers of the service provider. This stream of data can be fed to analytics used to improve operations, such as forecasting for resource provisioning; marketing, such as user segmentation for campaign management and sales, such as regression for churn prediction.

Data streams containing user interaction data can be aggregated, often over a recent window, to build robust user activity profiles. We call these *aggregate profiles*. Such profiles can provide insights about how users benefit from the service, and can be used to model the behavior of users. Clustering aggregate profiles is essential for many analytic applications. For example, user segmentation by clustering is a fundamental operation for churn analysis [1] and user equity management [2]. Also, modeling and forecasting the user behavior patterns is more effective when applied on users with similar behavior rather than on individual users [3]. As these examples motivate, many analytics operate on clusters of aggregate user profiles. We name these *clustered analytics*. Because of the continuous and live nature of these analytics and the potentially dynamic behavior of users, there is a need to maintain the aggregate user profiles in a clustered manner.

However, performing analytics on a large set of user profiles requires high processing resources. Similarly, keeping a recent history of interactions (such as a sliding window) for maintaining aggregate profiles requires high memory resources. Given the potentially large number of profiles, maintaining these clusters on a single machine may not be feasible, especially when the cost to process each profile update is high (e.g. updating forecasting models), the rate of incoming updates is high or the profiles are large in terms of size. Furthermore, in most real-world scenarios the updates (user interactions) are not only used for the sole purpose of cluster maintenance and clustered analytics, but for miscellaneous processing, such as enrichment, model scoring, visualization, etc. Thus, the need for parallel and distributed processing is paramount.

In this paper, we present our solution for scalable aggregate profile clustering in a streaming setting. We employ *partitioned stateful parallelism* to achieve scale. In particular, we partition the incoming stream over a set of processing nodes based on a profile id attribute and have each node process its portion of the substream, keeping a subset of the clusters and the associated state needed to maintain the aggregate profiles. Importantly, we want to make sure that each node gets assigned similar amount of processing load, since the slowest node will form a bottleneck for the system. Similarly, and for memory constrained scenarios, each node needs to store similar amounts of state for maintaining the profiles.

There are a number of challenges in achieving this. First, in order to distribute the incoming updates over the set of processing nodes, we need a way of *partitioning* them such that each update is routed to the node that contains profiles similar to its own. It is important to note that the similarity here applies to the aggregate profiles, and *not* to the updates. Initially, there is no information on the profile clusters, and as a result the partitioning could be hash based. Thus, after some time all nodes will form similar clusters. This is undesirable, since similar profiles are not co-located on the same machine and as the number of nodes increases, the fidelity of the clusters will decrease. Most importantly, this also means that performing clustered analytics in a distributed manner will result in poor accuracy. Fortunately, as we learn more about the nature of the profiles and frequencies of the partitioning attribute values, we can incrementally update our partitioning scheme and migrate profiles as needed, in order to increase the clustering quality.

Second, the re-partitioning has to make sure that each node gets a similar-sized flow of updates (good *processing balance*). Similarly, and for memory-constrained scenarios, each node should get around the same amount of state used to compute the aggregate profiles (good *memory balance*). Furthermore, the changes in the partitioning function should be incremental, so that the migration of profiles does not cause a long pause in processing (low *migration overhead*).

Third, the timing of the re-partitioning needs to be arranged carefully. Unnecessary updates to the partitioning would result in pauses in the processing and slow down the operation of system. Therefore, the system needs to monitor the clustering qualities and decide accordingly whether to update the partitioning scheme, or not.

Our solution relies on a smart re-partitioning technique to solve these problems. Concretely, it adaptively adjusts the assignment of profiles to nodes such that the memory and processing balance is improved, while still maintaining a high clustering accuracy and keeping the migration cost low. The system monitors the processing nodes' clustering quality and when it drops under a predefined threshold, it starts the re-partitioning operation. During the re-partitioning operation, each node creates micro-clusters and computes vectors that summarize the memory and processing requirements of the profiles stored in the micro-clusters, as well as the centroid and radius information for the micro-clusters. These summaries take a small amount of space and are collected at a central node. This node uses the micro-cluster summaries to come up with a new partitioning. To create the new partitioning, we use a greedy algorithm that iterates over the micro-clusters and computes the *affinity* of each micro-cluster to each node. The best assignment that maximizes the affinity is taken. We use a novel definition of affinity, which considers clustering quality, balance and the migration cost.

We developed an application [4] for telco customer segmentation using CDR data, based on the streaming aggregate profile clustering solution we propose in this paper. The solution relies on distributed stream processing middleware Storm [5] for processing updates and maintaining the aggregate profile clusters in memory, and HBase [6] for partial fault tolerance and for facilitating the migration. We use this system to study the impact of various workload, system and algorithmic parameters on the performance and accuracy of our re-partitioning algorithm.

The rest of this paper is organized as follows. Section 2 overviews relevant work from the literature. Section 3 formalizes our problem. Section 4 details our technical solution to the re-partitioning problem and Section 5 describes the system architecture used to implement a scalable distributed aggregate profile clustering system. Section 6 presents our evaluation and Section 7 concludes the paper.

2. RELATED WORK

Clustering can be defined as the task of grouping a set of objects in such a way that objects in the same group are more similar in some sense to each other than to those in other groups. Clustering is a fundamental task in exploratory data mining and is commonly used in statistical data analysis. To create a good classification of existing work on clustering and better underline the uniqueness of our work, we classify the approaches in the literature using the following dimensions:

($D1$) *Nature of the processing*:

(i) *Centralized*: There is a single processing node that performs the clustering.
(ii) *Distributed*: Multiple nodes are used to perform the clustering. In general, one or more of these nodes are specialized for cross-node arbitration.
(iii) *Decentralized*: All nodes perform the same task to construct the clustering.

($D2$) *Dynamicity of the dataset*:

(i) *Static*: Clustered objects do not change, or change infrequently. As a result, the clustering is offline.
(ii) *Dynamic*: Clustered objects are updated frequently. As a result, the clusters are maintained in an online manner, via incremental schemes.

($D3$) *Home location of the clustered objects*:

(i) *Centralized*: Clustered objects are stored at a single node.
(ii) *Distributed*: Clustered objects are partitioned over nodes.

($D4$) *Mobility of clustered objects*:

(i) *Fixed*: Clustered objects do not change their home locations (nodes), i.e. they do not migrate.
(ii) *Flexible*: Clustered objects can migrate across nodes according to their similarity to other objects.

($D5$) *Source of updates to the clustered objects*:

(i) *Local*: The updates are generated at the home location of the clustered objects.
(ii) *Remote*: The updates are routed to the home locations of the clustered objects.

($D6$) *Nature of the updates*:

(i) *Complete*: Each update replaces the clustered object completely. Typically, clustered objects have the same format as the updates.
(ii) *Partial*: Each update contributes to the value of the clustered object. Typically, clustered objects are formed by aggregating the updates.

We classify the existing clustering methodologies into five main categories and characterize them according to the aforementioned dimensions. A summary of this analysis is given in Table 1.

2.1. Traditional clustering

In traditional clustering, a single node is responsible for performing the clustering. Objects to be clustered are static and reside on the same node that is responsible for performing the clustering. Examples include k-means [7] (centroid-based), DBSCAN [8] (density-based), EM [9] (probability distribution based), as well as various other methods, such as link-based clustering used to create hierarchical clusters [10]. Our focus in this paper is the distribution and partitioning strategies used for distributed aggregate profile clustering in a streaming setting. We use some of the traditional algorithms mentioned here, such as k-means and EM, as part of our solution.

2.2. Distributed and parallel implementations of traditional clustering

Distributed and parallel versions of the traditional clustering algorithms have been developed to provide speed-up and scale-up. In these implementations, multiple nodes and/or CPUs are used to perform the clustering. There is often a master node, which is responsible for establishing cross-node arbitration and data partitioning. The data can be distributed or centralized, but the processing is always distributed.

As an example, PDBSCAN [11] is a parallel clustering algorithm based on DBSCAN, designed for knowledge discovery in large datasets. PDBSCAN uses a shared-nothing architecture, therefore it can be scaled up to hundreds of nodes. In a similar fashion, parallel k-means [12] offers a parallel version of the k-means algorithm on shared-nothing architectures. Projects like Mahout [13] and Oryx [14] provide

TABLE 1. Characterization of different clustering methodologies.

Clustering methodology	Nature of processing	Dataset dynamicity	Home location	Data mobility	Update source	Nature of updates
This work	Distributed	Dynamic	Distributed	**Flexible**	Remote	**Partial**
Traditional clustering	Centralized	Static	Centralized	Fixed	—	—
Distributed/parallel traditional clustering	Distributed	Static	Centralized distributed	Fixed	—	—
Distributed clustering for remote monitoring	Distributed	Dynamic	Distributed	Fixed	Local	Complete
Data clustering in sensor/P2P networks	Decentralized	Dynamic	Distributed	Fixed	Local	Complete
Incremental data stream clustering	Centralized	Dynamic	Centralized	Fixed	Remote local	Complete

Partial updates and flexible data mobility differentiate our problem setup from others.

distributed implementations of several traditional clustering algorithms based on the Map/Reduce [15] framework. Mahout [13] also offers a streaming k-means algorithm [16], which makes one pass through the static data and produces cluster centroids. These centroids are later used by a second phase to reduce the number of clusters to k. The term 'streaming' applies to the single scan nature of the algorithm's core. It is still a batch algorithm that does not deal with updates.

Distributed and parallel approaches to clustering utilize multiple cpus/machines for clustering data, yet they target static and immobile datasets, and thus are not applicable in our scenario.

2.3. Distributed clustering for remote monitoring

In distributed clustering for remote monitoring, the clustering task as well as the clustered objects are distributed over multiple nodes. Furthermore, the clustered objects are updated frequently. Each object has a fixed home location, and the updates originate from these home locations.

For remote monitoring, Januzaj *et al.* [17] propose clustering data locally at each node and extract suitable representatives out of these clusters. The representatives are then sent to a global master node, where a complete clustering based on local representatives is performed. As another example, Cormode *et al.* [18] also propose to perform clustering in-place at the home locations of the clustered objects, send the resulting summary information to a central location and form high-accuracy clusters there, while minimizing the communication and computational cost.

These approaches are similar to ours in one aspect: local clusterings are combined to form a global clustering. However, unlike our setup, update routing and migration concerns do not exist in these systems as the data mobility is not present and the updates are always local.

2.4. Data clustering in sensor/peer-to-peer networks

Clustering in sensor networks and peer-to-peer (P2P) environments is done without the use of a central node. In such systems, each node holds some data value, e.g. a local sensor reading or a series of them, and participates in a distributed clustering algorithm. We classify these clustering approaches as having decentralized nature of processing. An overview of distributed clustering with a particular focus on sensor networks can be found in [19].

Eyal *et al.* [20] propose a solution where numerous interconnected sensor nodes partition their data into multiple clusters, and describe each cluster concisely. They observe that direct distance computation is not sufficient to provide good clustering results, and for this reason they develop a generic algorithm that models the values as a Gaussian mixture model. Gedik *et al.* [21] propose ASAP, an adaptive-sampling approach to data collection in sensor networks. ASAP uses sensing-driven clustering to group nodes into clusters. This clustering technique tries to form clusters that contain sensor nodes that are not only spatially close, but also their sensor readings are close.

Clustering approaches in sensor and P2P networks offer decentralized processing of distributed and dynamic datasets. Unlike these works, our problem setup involves a tightly coupled cluster environment. Furthermore, in our setup, the clustered profiles can change their home locations, resulting in data mobility.

2.5. Incremental data stream clustering

Incremental data stream clustering is used in cases where the data are evolving dynamically. A single node is responsible for maintaining clusters in an online manner, using incremental schemes. The data are scanned only once.

Balanced iterative reducing and clustering using hierarchies [22] is an unsupervised data mining algorithm used to perform hierarchical clustering over incrementally and dynamically incoming data. Guha *et al.* [23] also study k-Median problem in the streaming context and provides a new stream clustering algorithm that is based on a facility location algorithm.

Discussion

Our problem has two unique aspects that sets it apart from these earlier approaches. The first is the partial updates. Each streaming update modifies an aggregate profile we keep. Profiles are the clustered objects. The update does not completely replace the profile, but instead modifies its value. This has an implication in routing: it is the similarity of the profiles, not updates, that decides how incoming updates are to be routed. Second, in our setting, the home locations of the profiles can change. The only requirement is that, similar profiles (the ones in the same cluster) stay in the same machine as much as possible. Otherwise, the fidelity of the clusters decreases, lowering the accuracy of clustered analytics performed in a distributed manner. Recall that, for performing clustered analytics, one needs to build models on a per-cluster basis.

3. PROBLEM DEFINITION

In this section, we define clustering quality, balance quality and migration quality as three key metrics and formalize our aggregate profile clustering problem as an optimization problem built on these metrics.

Using clustering quality, we measure how successful our clustering is compared with a centralized clustering approach. Using balance quality, we measure how balanced the load on the nodes of the system are. Migration quality captures the cost of the migration in terms of the amount of data moved with respect to the total state size. Finally, we present an overall

quality formulation which is a combined measure that relies on the clustering, balance and migration qualities.

3.1. Notation

Let S denote a stream of updates, where $u \in S$ denotes an update. We use $\iota(u) \in D$ to denote the value of the profile id for the update, where D is the domain of the profile ids. Let $P(d)$ denote the aggregate profile for profile id $d \in D$. We assume that $P(d)$ is a multi-dimensional vector. We use $f(d)$ to denote the frequency of updates with profile id $d \in D$.

We define a partitioning function $p : D \rightarrow [0 \ldots N)$ that maps each profile id to a node, where we have N nodes. When an update u is received, it is forwarded to the node at index $p(d)$, where $d = \iota(u)$. There, it contributes to the aggregate profile information $P(d)$, via the transformation

$$\langle P(d), S(d) \rangle \leftarrow \gamma(P(d), S(d), u).$$

Here, γ is an aggregation function and $S(d)$ is the state maintained to compute it continuously for profile id d. For instance, $S(d)$ could be a sliding window of updates maintained for computing holistic aggregates [24] or a vector of moments for maintaining simple aggregates. As such, for some applications this state could be constant size (e.g. for simple aggregates), whereas in others it could be linear in the size of the window kept (e.g. for holistic aggregates).

At time step s, the partitioning function will be updated from p_{s-1} to p_s, with the goal of keeping the clustering quality high, the processing and/or memory loads balanced and the migration cost low. We now define each of these metrics.

3.2. Clustering quality

Let $\mathcal{C}_i$ be the set of clusters on node i after applying a local clustering algorithm $\mathcal{A}$, that is, $\mathcal{C}_i = \mathcal{A}(\{P(d) : p(d) = i \wedge d \in \mathcal{D}\})$. Let $\mathcal{C}$ be the set of all clusters from all nodes, that is, $\mathcal{C} = \bigcup_{i \in [0 \ldots N)} \mathcal{C}_i$. Further, assume that $\mathcal{C}^*$ denotes the clustering that would be formed if the same clustering algorithm is applied on all profiles, that is, $\mathcal{C}^* = \mathcal{A}(\{P(d) : d \in \mathcal{D}\})$.

We define the *clustering quality* as the *normalized mutual information* (NMI) [25] between the ideal clustering denoted by $\mathcal{C}^*$ and the distributed clustering we computed denoted by $\mathcal{C}$:

$$Q^c = NMI(\mathcal{C}^*, \mathcal{C}) \tag{1}$$

NMI is defined as

$$NMI(X, Y) = \frac{H(X) + H(Y) - H(X, Y)}{(H(X) + H(Y))/2}, \tag{2}$$

where $H(X)$ is the entropy of the clustering X and $H(X, Y)$ is the joint entropy of X and Y.

Note that our goal is not to measure the base clustering quality, which could be done via traditional measures, such as V-Measure [26]. Instead, we aim to compare our distributed clustering results with the clustering that is formed when all profiles are collected at a single node (using the same clustering method). If our distributed clustering results are exactly same as the single node ones, the NMI value, thus the clustering quality, will become 1. As the two start to diverge, the NMI values become smaller (with a lower bound of 0).

Here, it is important to use a clustering algorithm, $\mathcal{A}$, whose parameter settings are not impacted by the number of nodes, N. For instance, for small number of dimensions a density-based clustering algorithm (such as DB-scan [8]) will be effective. For k-means-based algorithms, the distribution of k over the N nodes will be a problem. To alleviate this, k-means algorithms that use automatic determination of the k value can be used, such as those that rely on the Bayesian Information Criterion metric to determine k [27].

For this work, we have used EM clustering [9], a distribution-based clustering algorithm. The WEKA [28] implementation of the EM clustering can determine the number of clusters automatically, which makes it easy to use in our setup, as it avoids having to adjust the number of clusters based on the node count.

3.3. Balance quality

Let $R_i = \sum_{p(d)=i} f(d) \cdot \beta(|S(d)|)$ denote the processing cost required to handle the updates for the profiles assigned to the ith node. Here, β is a function that defines the relationship between the amount of state maintained and the required processing to update the aggregate profile. For instance, a simple incremental aggregation, such as average, would have $\beta(x) \propto 1$ (independent of state size). In contrast, a holistic aggregate, such as computing quantiles, could require $\beta(x) \propto x$.

We define the *processing balance quality* as $Q^{pb} = 1 - \text{CoV}(\{R_i\})$, where CoV is the coefficient of variation (ratio of standard deviation to mean). When the standard deviation in the balance is 0, then the balance quality is 1. When the deviation reaches a single node's share of the load (i.e. the mean), then the quality reaches 0. Let $S_i = \sum_{p(d)=i} |S(d)|$ denote the size of state stored on the ith note to maintain the profiles assigned to it. We define the *memory balance quality* as $Q^{mb} = 1 - \text{CoV}(\{S_i\})$.

Depending on the nature of the state maintained ($S(d)$ for profile d), the memory may or may not be a concern. For instance, if the state is constant size and small, then it may fit on a single machine. In this case, we can take the *balance quality* as $Q^b = Q^{pb}$, ignoring the memory balance. On the other hand, when the state is linear in the frequency ($|S(d)| \propto f(d)$), such as for a non-invertible aggregation γ defined over time-based sliding window, then the memory balance may factor into the balance quality and thus we take $Q^b = (Q^{pb} + Q^{mb})/2$. Other combinations are possible.

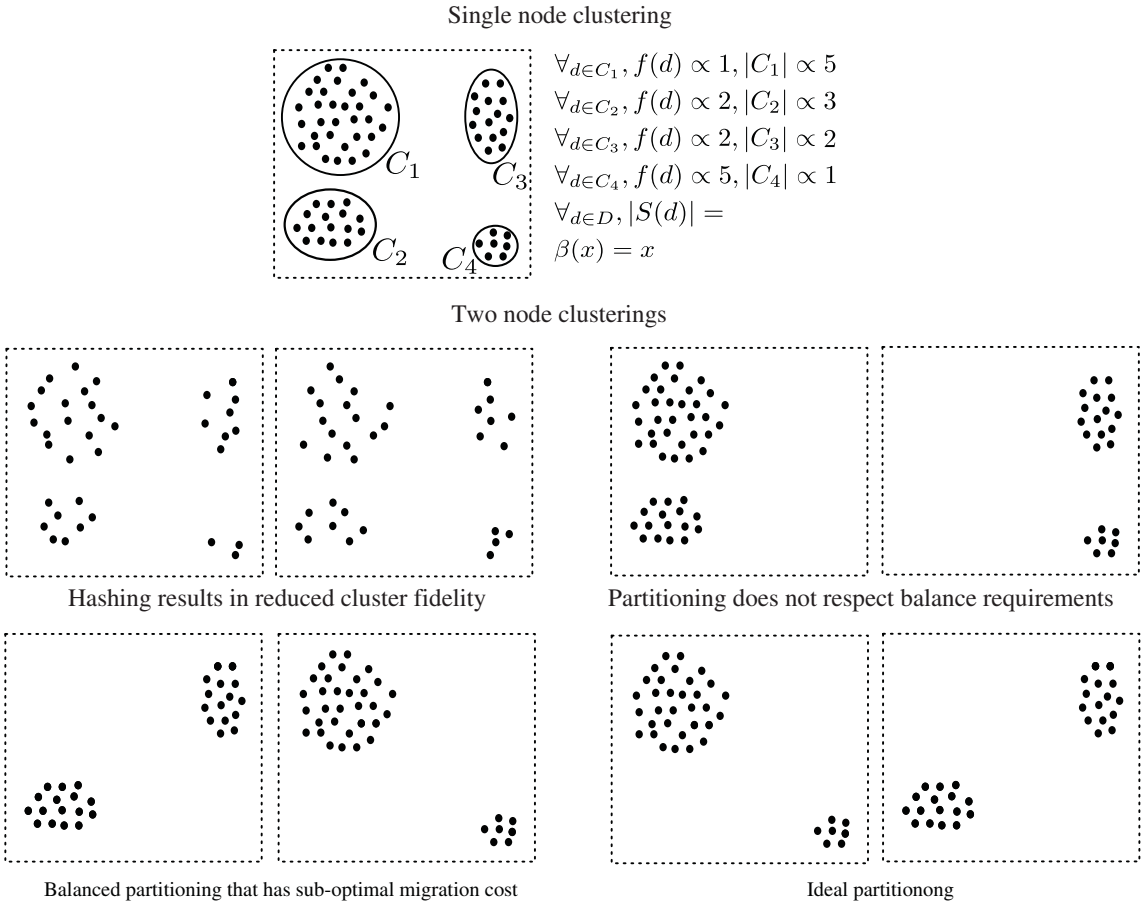

FIGURE 1. An illustration of alternative partitionings.

3.4. Migration quality

As the system learns more about the frequencies of the partitioning attribute values and the nature of the profile clusters, the partitioning scheme needs to be incrementally updated and profiles should be migrated as needed. During this operation, the migration overhead should be low to avoid lengthy pauses. Therefore, we define the migration quality to quantify the amount of data movement during the migration. In particular, we divide the amount of migrated state to the total state size in order to derive a normalized metric.

We formalize the migration quality as follows:

$$Q^m = 1 - \frac{\sum_{d\in D} |S(d)| \cdot \mathbf{1}(p'(d) \neq p(d))}{\sum_{d\in D} |S(d)|}. \quad (3)$$

Here, p' is the previous partitioning function and $\mathbf{1}$ denotes the indicator function.[1] When there is no migration, the migration quality is 1. When the entire state needs to move, then the migration quality is 0.

[1] Produces 1 if the Boolean condition holds, 0 otherwise.

3.5. Overall quality

We now define the overall quality based on the previously defined metrics: clustering quality, balance quality and migration quality. There is a clear trade-off between clustering quality and balance quality. When the importance of balance quality is set high, the clustering quality may suffer, as keeping the balance quality high may necessitate splitting clusters into several subclusters that are placed on different nodes. Therefore, there is a need to strike a good balance between clustering quality and balance quality.

We denote the overall quality as Q, and define it as

$$Q = (\sigma \cdot Q^c + (1 - \sigma) \cdot Q^b) \cdot Q^m. \quad (4)$$

Here, $\sigma \in [0, 1]$ adjusts the relative importance of clustering quality versus load balance, which is set to 0.5 by default.

3.6. Sample illustration

Figure 1 illustrates a toy scenario with four clusters of profiles, namely C_1, C_2, C_3 and C_4. Among these, C_1 is the largest in terms of the number of profiles ($|C_1| \propto 5$), but the frequency

of updates for profiles in this cluster is the lowest ($f(d) \propto 1, \forall_{d \in C_1}$[2]). On the other hand, C_4 has the lowest number of profiles ($|C_4| \propto 1$), but the highest frequency of updates ($f(d) \propto 5, \forall_{d \in C_4}$). We see that C_2 and C_3 have values in between the two extremes ($|C_2| \propto 3$, $|C_3| \propto 2$ and $f(d) \propto 2, \forall_{d \in C_2 \cup C_3}$). In this example, we assume that the state kept for aggregation is constant and equal to a window size w ($|S(d)| = w, \forall d \in D$) and the processing time for updates is linear in the state size ($\beta(x) = x$). Given that the processing load for a cluster is $p(C_i) = \sum_{d \in C_i} f(d) \cdot \beta(|S(d)|)$ and memory size is $m(C_i) = \sum_{d \in C_i} |S(d)|$, we have the following characteristics for the clusters: $\langle p(C_1), p(C_2), p(C_3), p(C_4) \rangle = \langle 5, 6, 4, 5 \rangle$ and $\langle m(C_1), m(C_2), m(C_3), m(C_4) \rangle = \langle 5, 3, 2, 1 \rangle$.

Figure 1 shows four different partitionings for $N = 2$. The first alternative represents hashing. The problem with this alternative is that the fidelity of the clusters are reduced. As N increases, this alternative will further degrade with respect to the cluster quality. The second alternative puts together clusters that are similar on the same node (akin to clustering the clusters). However, this is not a good partitioning, as it does not balance the load. In this alternative, the processing load for the first node is ≈11, whereas that of the second node is ≈9. The third alternative balances the load perfectly (10 and 10), has the same clustering result as the single node case, but compared with the last alternative, it has higher migration cost (6 versus 5). As a result, the best alternative is the rightmost one.

In the example scenario, the best partitioning keeps the individual clusters unaltered. However, it is important to note that it may not be possible to keep the clusters intact in all cases, especially when the number of nodes is high and there are clusters that are large in size. This is because the load balance requirement may necessitate dividing a cluster across nodes.

4. SOLUTION

Our goal is to cluster a large number of profiles, which are formed by aggregating streaming updates. Maintaining these profile clusters on a single machine may not feasible, especially when the rate of the updates is high (resulting in high processing cost) or the amount of state kept is high (resulting in high memory cost). Thus, we propose a distributed streaming approach to cluster the aggregate profiles. Our approach forms and maintains the clusters on a set of hosts, so that updates on the profiles and clustered analytics defined on them can be executed in parallel.

4.1. Overview

In our solution, profiles are distributed to nodes according to their similarities to each other, while at the same time considering the balance across the nodes with respect to processing and memory costs.

Initially, the incoming stream is partitioned over processing nodes by a simple hash function using a key attribute from the update (such as the caller id in a CDR) and each node processes its portion of the substream. Aggregate profiles are incrementally built on the processing nodes. Periodically and/or adaptively, a re-partitioning procedure is performed to improve the partitioning.

During the re-partitioning, each node applies k-means clustering with a relatively high k parameter to create *micro-clusters*. Micro-clusters are then sent to the master node. After the master node gets all micro-clusters from processing nodes, a new partition mapping is created and the system pauses shortly until it is put into active use.

For each micro-cluster, the master node ranks the processing nodes and assigns the micro-cluster to the most appropriate node. Ranking is performed using three considerations:

(1) keep the clustering quality high, by placing micro-clusters that are close to each other on the same nodes;
(2) keep the total processing and/or memory load balanced;
(3) minimize the amount of state migration resulting from updating the partitioning function.

After the master node creates a new partition mapping (which maps profile ids to processing nodes), the splitter node is informed about the new partition mapping and the processing nodes are informed about the changes in the set of profiles they are maintaining. Then the state migrations are performed for the profiles whose node mappings have changed. After the migration operation is complete, the system resumes processing again. Keeping the amount of state migrated small is important for minimizing the disturbance to the processing flow during this adaptation.

4.2. Updating the partitioning function

We update the partitioning function periodically by collecting summary information at a master node. The update is performed such that the load balance and the clustering quality are kept high, while incurring low migration cost.

At step $s = 0$, we set the partitioning function to the consistent hash function $\mathcal{H}_N$, that is, $p_0(d) = \mathcal{H}_N(d)$. For the purpose of updating the partitioning function, each node creates micro-clusters [29] over the profiles they maintain. The summaries of these micro-clusters are then sent to a master node, which computes a new partitioning function p_s.

A micro-cluster, denoted as $M \subset D$, keeps a set of profile ids. It is summarized as a 5-tuple: $\hat{M} = \langle o, r, p, m, l \rangle$. Here, o denotes the centroid of the micro-cluster, that is, $\hat{M} \cdot o = \sum_{d \in M} P(d)/|M|$. The radius of the micro-cluster is denoted by r. We have $\hat{M} \cdot r = \max_{d \in M} \|P(d) - \hat{M} \cdot o\|$. The total processing cost for the profiles in the micro-cluster is denoted as p. We have $\hat{M} \cdot p = \sum_{d \in M} f(d) \cdot \beta(|S(d)|)$. The total

[2] In this example, the frequency is taken as equal for all updates in the same cluster.

memory cost for the state associated with the profiles of the micro-cluster is denoted by m. We have $\hat{M} \cdot m = \sum_{d \in D} |S(d)|$. Finally, l denotes the current location (node maintaining it) of the micro-cluster. We have $\hat{M} \cdot l = p_{s-1}(d), d \in M$.

The master node, upon receiving all the micro-cluster summaries, creates a new partitioning function. For this purpose, we use the greedy procedure described in Algorithm 1. The algorithm iterates over the micro-clusters and for each micro-cluster it makes a node assignment. We consider micro-clusters in decreasing order of their state sizes during the assignment.

While assigning micro-clusters to nodes, the algorithm makes use of a heuristic metric and picks the assignment that maximizes this metric. Let $\mathcal{M} = \{M_i\}$ be the list of all micro-clusters, and assume that $i-1$ assignments are made and we are to make an assignment for the ith micro-cluster, M_i. In order to do this, we first compute the *affinity* of this micro-cluster to each node. Let $A(M_i, j)$ denote the affinity of M_i to the jth node. We set $\forall d \in M_i$, $p_s(d) = \text{argmax}_{j \in [0...N)} A(M_i, j)$. That is, the node for which the micro-cluster has the highest affinity becomes the new mapping for all the profiles within the micro-cluster.

Affinity has three aspects to it: the *clustering disaffinity* denoted by A^c, the *balance disaffinity* denoted by A^b and the *migration affinity* denoted by A^m. Let $\mathcal{M}_l$ denote the set of micro-clusters assigned to the lth node so far, i.e. $\mathcal{M}_l = \{M \mid p_s(d) = l \wedge d \in M \in \mathcal{M}\}$.

4.2.1. *Clustering disaffinity*

Clustering disaffinity computes how far a micro-cluster is to a processing node in the multi-dimensional space. k micro-cluster members of the processing node that are closest to the micro-cluster at hand are found, and the sum of their distances is calculated. We normalize this value with the sum of all clustering disaffinities toward all nodes.

The clustering disaffinity is formalized as follows:

$$A^c(M_i, j) = \frac{\sum_{x \in \text{min-}k(L_{i,j})} x}{\sum_{l \in [0...N)} \sum_{x \in \text{min-}k(L_{i,l})} x}, \tag{5}$$

where $L_{i,j} = \{\|\hat{M}_i \cdot o - \hat{M} \cdot o\| \mid M \in \mathcal{M}_j\}$ and min-k is a function that takes the smallest k elements from a list. By $L_{i,j}$, we represent the list of distances from the micro-cluster centroid $\hat{M}_i \cdot o$ to the centroids of micro-clusters that are assigned to the jth node so far; $k \geq 1$ is a parameter of our algorithm. The clustering disaffinities sum up to 1, that is, $\sum_{j \in [0...N)} A^c(M_i, j) = 1$.

4.2.2. *Balance disaffinity*

One of our aims is to assign a similar-sized flow of updates to each processing node. Therefore, we use balance disaffinity to compute how much processing capacity is used in a processing node after the micro-cluster at hand is assigned to it, relative to the total used capacity. If this number is high compared with other possible node assignments, then it is not a good idea to assign the micro-cluster to the node under consideration, as it will hurt the balance.

The balance disaffinity, for processing, is formalized as

$$A^b(M_i, j) = \frac{C_{i,j}}{\sum_{l \in [0...N)} C_{i,l}}, \tag{6}$$

where $C_{i,j} = \hat{M}_i \cdot p + \sum_{M \in \mathcal{M}_j} \hat{M} \cdot p$. Here, $C_{i,j}$ represents the amount of processing capacity used up on the jth node once the micro-cluster M_i is placed on it, also considering all the previous micro-clusters that were placed on that node. The balance disaffinities sum up to 1, that is, $\sum_{j \in [0...N)} A^b(M_i, j) = 1$. The balance disaffinity for memory is defined similarly, by replacing p with m.

4.2.3. *Migration affinity*

To calculate the migration affinity, we compute how much migration is performed so far and compare it to the maximum migration cost possible. As expected, migration affinity will decrease as the total amount of migration so far increases. However, there is a slight problem. Recall that micro-clusters are considered in the order of their state sizes. Thus, the initial ones considered would have large state sizes. Furthermore,

Algorithm 1: UPDATEPARTITIONING($\mathcal{M}$, N, O)

Param : $\mathcal{M}$, micro-clusters
Param : N, number of nodes
Param : O, ordering policy
$p \leftarrow \{\}$; ▷ The partitioning function to be constructed
$\mathcal{M}' \leftarrow$ SORT($\mathcal{M}$, O); ▷ Order micro-clusters
for $M \in \mathcal{M}'$ **do** ▷ For each micro-cluster, in order
  $i \leftarrow -1$; $a \leftarrow 0$; ▷ Best assignment and affinity
  for $j \in [0..N)$ **do** ▷ For each node
    ▷ Compute the affinity of M to node j
    $v \leftarrow A^m(M, j) \cdot (1 - ((1-\alpha) \cdot A^c(M, j) + \alpha \cdot A^b(M, j)))$
    if $v > a$ **then** $\langle i, a \rangle \leftarrow \langle j, v \rangle$; ▷ Update best
  $p[d] = i, \forall d \in M$; ▷ Create profile mappings
return p; ▷ Return the fully constructed mapping

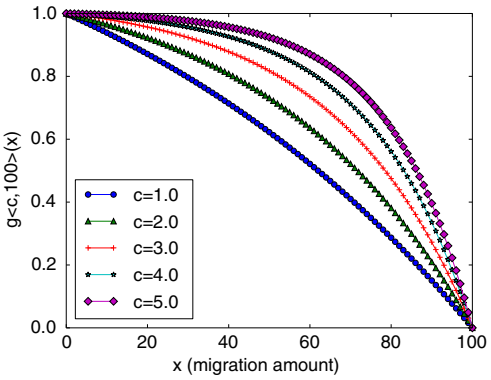

FIGURE 2. g functions for different c values.

since migration cost depends on the binary variable of whether the micro-cluster was on the same machine before or not, the migration affinity will dominate our decisions for the initial set of assignments. This is not desirable, as there will be plenty of opportunities to improve the migration cost. To alleviate this problem, we want the rate at which migration affinity decreases to increase with increasing migration cost. This means that migration affinity has lower impact initially, when the total migration cost is still small compared with the maximum allowed. This allows more flexibility for our algorithm to make micro-cluster assignments initially.

The migration affinity is defined as follows:

$$A^m(M_i, j) = g\langle c, G\rangle(R + \hat{M}_i \cdot m \cdot \mathbf{1}(\hat{M}_i \cdot l \neq j)), \quad (7)$$

where $R = \sum_{i\in[0\ldots N)} \sum_{M\in\mathcal{M}_i} (\hat{M} \cdot m \cdot \mathbf{1}(\hat{M} \cdot l \neq i))$ represents the total migration cost so far. We see that $G = (2/N) \cdot \sum_{M\in\mathcal{M}} \hat{M} \cdot m$ represents the maximum allowable migration cost (taken as the twice the amount of state per node[3]); $g\langle c, G\rangle(x)$ is a function of the form $y = a - b \cdot e^{c\cdot x/G}$ that satisfies $g\langle c, G\rangle(0) = 1$ and $g\langle c, G\rangle(G) = 0$; c is a parameter that adjusts the skew of the function. For instance, for $G = 100$, we get the graph shown in Fig. 2. The motivation for having the g function is to penalize migrations less when initially there is a high budget for migration. We use a c value of 2 in our experiments.

4.2.4. *Overall affinity*

Given these definitions, we define the overall affinity as follows:

$$A(M_i, j) = A^m(M_i, j) \cdot (1 - ((1-\alpha) \cdot A^c(M_i, j) + \alpha \cdot A^b(M_i, j))). \quad (8)$$

Here, $((1-\alpha) \cdot A^c(M_i, j) + \alpha \cdot A^b(M_i, j))$ defines the combined clustering and balance disaffinity, where $\alpha \in [0\ldots 1]$ adjusts the importance of one compared with the other. As the value of α increases, the clustering becomes less decisive compared with the balance. Subtracting the combined clustering and balance disaffinity from 1 gives the combined affinity, which we multiply with the migration affinity to get the overall affinity value.

[3]Which is the amount of migration resulting from two nodes completely exchanging their state.

4.3. Handling edge cases

There are a few edge cases to handle with the partitioning algorithm we have described so far. The first one is about the proper computation of the clustering disaffinity when a node has no micro-clusters assigned so far. The second is about the over-sensitivity of the balance disaffinity when there are only a few assignments made so far. We now describe how these issues are resolved in our system.

4.3.1. *Initial condition handling for clustering disaffinity*

At the beginning of the re-partitioning procedure, the processing nodes are empty (nothing is assigned to them yet). For the first assignments, it is impossible to calculate the clustering dissaffinty, as the formula relies on the existing assignments to compute a distance. To be able to compute a clustering disaffinity for a node that has no assignments so far, we come up with initial cluster centers for each node. In particular, we take all micro-clusters and use k-means clustering to create N clusters out of them. We take the centroids of the resulting clusters and assign each one to one of the nodes as that node's initial cluster center. When the clustering disaffinity is to be computed for a micro-cluster that has no assignments, this initial cluster center is used to compute the distance.

4.3.2. *Start-up phase handling for balance disaffinity*

During the start-up phase of the algorithm, the balance disaffinity may prevent micro-clusters that are close to each other to be assigned to the same node, as that might hurt load balance. However, initial imbalances are not that important, as there would be plenty of opportunities for correcting them later in the assignment process. To capture this, we scale the importance of the load disaffinity (originally α) by a *scaler function*. We denote the scaler function as l and define it as follows:

$$l\langle d, L\rangle(x) = \begin{cases} a' - b' \cdot e^{d\cdot x/L}, & x < L/10, \\ 1, & \text{otherwise.} \end{cases} \quad (9)$$

The function takes as a parameter the total amount of load assigned to the nodes so far; L is the maximum amount of load to be assigned and d is a parameter that adjusts the skew of the function. After 10% of the load is assigned, the scaler function defaults to 1. Furthermore, the scaler function l satisfies $l\langle d, L\rangle(0) = 0$ and $l\langle d, L\rangle(L/10) = 1$. For instance, for $L = 100$, we get the graph shown in Fig. 3. The motivation for having the l function is to gradually increase the penalty due to the load imbalance. We use a d value of 2 in our experiments.

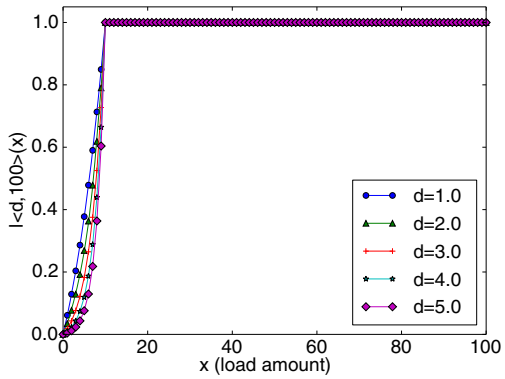

FIGURE 3. l functions for different d values.

4.4. Adaptive re-partitioning

As we explained earlier, there is a small pause in the processing when the system updates the partitioning. If there are many unnecessary re-partitionings, the system would slow down. Therefore, the timing of the re-partitioning must be arranged carefully. At a given moment in the processing, the system needs to detect if re-partitioning is required or not. To handle this, we store the micro-clustering results of the previous re-partitioning process, and compare it with the current micro-clustering results. Comparison is done using the NMI metric (see Section 2). Each processing node in the system does its own comparison, and if the comparison result is lower than a predefined threshold (in range [0, 1]), then the processing node sends a *re-partition* signal to the master node, which starts the re-partitioning.

4.5. Parameter discussion

There are three parameters that are important for the performance of the system. The first one is the number of closest neighbors we use when computing the clustering disaffinity (k). Using too few neighbors reduces the robustness of the disaffinity computation, as a single outlier may pull a micro-cluster to a non-optimal partition. Too many neighbors will result in high disaffinity values for all nodes. We experiment with the value of k in Section 6.

The second parameter that is important is α, which is used to adjust the relative impacts of balance disaffinity and cluster disaffinity. We set this value to 0.5 by default, giving them equal importance. However, depending on the application needs, this parameter can be adjusted to trade off throughput for better clustering accuracy. We study this effect in Section 6.

Third, there is the threshold used for adaptive re-partitioning. Setting this threshold too low may cause frequent re-partitionings, whereas high values may result in reduced clustering quality due to drifts in the profile values. We experimentally evaluate this in Section 6 as well.

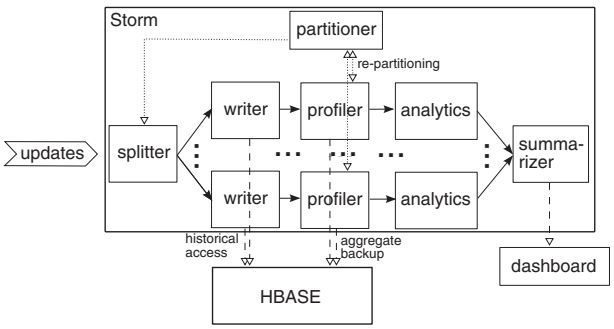

FIGURE 4. The architecture of the aggregate profile clustering system.

5. IMPLEMENTATION DETAILS

We implemented a profile clustering system and an analytic application on top of it. The application is about customer segmentation over CDR data for tariff optimization.

5.1. The profile clustering system

We implemented our profile clustering system with the help of a Storm distributed stream processing platform and the HBase key-value store. Figure 4 depicts the system architecture.

The profile updates stream into the system and are processed by a topology[4] that runs on the Storm distributed stream processing system. The updates are tuplized and partitioned using the Splitter operator. The splitted flows first go through the Writer operator, which persists the updates to the HBase distributed key-value store for historical access. This parallel write feature is not strictly needed for our aggregate profiling technique, but is part of our analytics platform.

The updates are then sent to the Profiler operators, which are responsible for updating the in-memory profiles and performing clustering. The profiler interacts with the Partitioner operator, which in turn interacts with the Splitter, for implementing the re-partitioning. In particular, when re-partitioning is initiated, the Partitioner asks the Splitter to pause the flow. After all in-flight tuples are processed, the micro-clusters are shipped from the Profilers to the Partitioner. The Partitioner executes the re-partitioning algorithm and computes the new partitioning. Using this partitioning, it computes migration schedules and sends these to the Profilers. To minimize the coupling between Profilers, the actual migration of state is performed through HBase. Each Profiler writes to HBase the state that it no longer has to keep. After a synchronization step, it also borrows the state that it needs to maintain from now on. Once the state migration is completed, the Partitioner sends the new partitioning to the Splitter operator, which installs it and resumes the flow.

[4] A flow graph of operators.

The Profilers also use the HBase store to back up their state periodically, to support fault tolerance. While the profile maintenance is not sensitive to short-term tuple loss, this backup is needed to avoid losing long-term aggregations that are computed over large time scales.

Also note that the Partitioner is not keeping state across re-partitionings. If it fails between re-partitionings, it can simply be restarted. It will construct its state from the information it gets from the Profilers, during the next re-partitioning. If it fails during a re-partitioning, the adaptation step would be skipped for the current period.

5.2. The customer segmentation application

We built an application that uses the aggregate profiling system outlined so far to perform customer segmentation for tariff optimization [4]. The system uses CDRs as profile updates and builds aggregate customer calling profiles.

Telco companies provide their customers with tariffs that regulate base fees and call charges according to call types. Correctly defining tariffs not only benefits the customers by lowering their bills, but also it benefits the telco companies, as they can analyze customer orientation better and develop the necessary infrastructure and better optimize resources.

To define well-targeted tariffs, telco companies need to understand call patterns of their customer base. Whenever a customer makes a call, a CDR is sent to the data center of the telco company. The CDR has a caller associated with it and contains information about the call, such as call target, call time, call duration, etc. When CDRs of a customer are aggregated, customer call patterns can be understood.

The CDRs are processed to compute customer calling profiles. We define a number of features, based on the kind of the destination number of the call (local, trunk, GSM, international, PRS), based on the time of the call (night time, daytime, weekday, weekend), as well as the length of the call (short, long). For each call category, we maintain separate aggregates of the percentage of calls falling into that category. These form a call profile vector, which is updated each time a new call is received. The clusters are maintained over these aggregate call profiles.

Each profile has tariff information associated with it and thus the resulting clusters from our distributed aggregate profile clustering solution have labeled points with tariff information. Our goal is to perform tariff optimization by detecting *poorly defined tariffs* and *potential new tariffs*.

The main idea is that customers who have similar call patterns should have the same tariff, assuming that the tariffs are well defined. The system analyzes the clustering results and detects if a tariff is scattered over many clusters where it is a minority, or concentrated on a few clusters where it is well represented. In the former case, we conclude that the tariff is not reaching its target audience, therefore it is a poorly defined tariff.

Using a similar line of thought, the majority of customers in a cluster should have the same tariff if there is a tariff that meets the expectations of the clustered customer group. Therefore, clusters with high entropy are identified as candidates for creating *new tariffs*.

Figure 5 shows the demo dashboard of the Customer Segmentation Application, where different tariffs are shown with different colors and different clusters are shown with different shapes. Various visual aids are used to show how much a tariff is scattered over clusters as well as how much a cluster is scattered over tariffs.

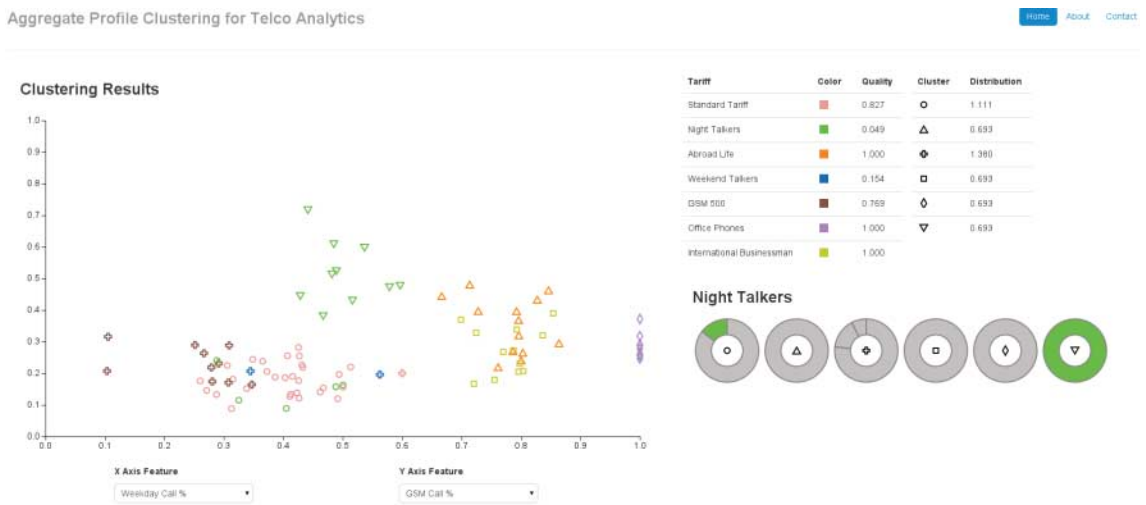

FIGURE 5. A sample screenshot from the demo dashboard of the Customer Segmentation Application.

6. EXPERIMENTAL EVALUATION

We have performed an experimental study with the aim of evaluating the efficacy of the proposed solution. In this section, we first describe the experimental setup and then discuss our experimental results in detail.

6.1. Experimental setup

We describe the specifications of the machines and system software used in our experimentation, the properties of the CDR dataset used as our workload and the default configurations used for the system parameters.

6.1.1. *Machines*

Evaluating the scalability of our solution requires the use of multiple machines. We use kernel-based virtual machines on 2 IBM System x3650 M4 servers to create a multi-machine environment. Each host machine has two 8-core Intel Xeon 2.00 GHz processors, 96 GB RAM and 5TB 10K rpm hard disks. In total, the host machines have 32 cores. They are connected to each other via a 10Gbit Ethernet switch. Our guest machines have a single 2.00 GHz virtual core, 5 GB RAM and 100 GB hard disk. The guest machines run a CentOS 6.4 operating system, have version 1.7.0 of Java, 1.0.3 version of Hadoop [30], 0.94.9 version of HBase [6] and 0.8.2 version of Storm [5] installed.

6.1.2. *CDR dataset*

To evaluate the system with varying workload characteristics, there is a need for a data provider. We built a synthetic CDR generator for this purpose. CDRs are generated according to predefined customer profiles. Predefined customer profiles have target, time and duration features, and each of those features have different values with their associated probabilities of appearance. Predefined profiles also have shift parameters to model the changes in call behaviors of customers. The shift parameters consist of a *direction* that defines the target toward which the profiles will move, and a *probability* value that indicates the probability of that shift happening at each time step.

Before the CDR generator starts, the system builds its customer base by selecting one of the predefined profiles for each customer using a Zipf distribution [31]. When CDRs are being generated, target, time and duration values are determined by using probabilities that are defined in the customer profiles. The generator periodically checks for profile shifts based on the customers' shift probabilities. If a shift happens, the customer is shifted toward the direction that is contained within its predefined profile.

6.1.3. *Experimental parameters*

Given that our focus is on scalability, *Number of Nodes* is one of the main parameters we study.

There are also parameters of our solution as well (see Section 4.5). In particular, there is the k value representing the number of nearest neighbors used in defining the clustering disaffinity as part of our heuristic algorithm. There is the α value used in Equation (8) to adjust the relative importance of balance disaffinity and cluster disaffinity, while computing the overall quality. Finally, there is the adaptive re-partitioning threshold.

There are also workload parameters. To experiment and evaluate the proposed solution, we feed the system with different types of customer bases. *Number of Profile Types* defines the number of predefined profiles used by the CDR generator, which can be considered as the number of ideal clusters in the dataset. As we mentioned earlier, the customer base is built by assigning each customer to one of the predefined profiles. Profile selection is performed using a Zipf distribution, and the Z parameter is used to adjust the skew of the distribution. In other words, it is the skew in the sizes of the ideal clusters. To generate profiles, we use a Gaussian distribution, where the standard deviation used for the profile attributes is taken as 0.1 times the mean. In effect, our workload generation model can be seen as a Gaussian mixture.

The aforementioned parameters will be analyzed separately, assuming other variables are assigned to their default values in the process. Default values are given in Table 2.

Let us denote the *Number of Nodes* as N. The system is fed with $10^5 \cdot N$ CDRs from $1000 \cdot N$ customers. In the k-means clustering part of the algorithm, where we compute micro-clusters, k is taken as $\lceil 1000/30 \rceil$. In other words, each micro-cluster contains on average, 30 profile summaries.

6.1.4. *Evaluation metrics*

We evaluate the proposed solution with two fundamental metrics: *quality* and *execution time*. There are four quality metrics: *Cluster Quality*, *Balance Quality*, *Migration Quality* and *Overall Quality*. Quality definitions and formulations are given in Section 3. There are four execution time metrics to analyze: *Micro-cluster Time*, *Partition Time*, *Migration Time* and *Total Time*. Micro-cluster Time is the average time it takes for the system to perform the micro-clustering, which is performed in parallel by the nodes. Partition Time is the average time the system spends on building a new

TABLE 2. Default values of the experimental variables.

Parameter	Default value
Number of nodes	16
Number of nearest neighbors	10
Alpha	0.5
Number of profile types	6
Standard deviation of profiles	0.1 × mean
Zipf Z	1

partitioning from the micro-cluster summaries, which is done at a centralized node. *Migration Time* is the average pause time while the system migrates micro-clusters to their new locations. *Total Time* is the average time it takes the system to complete the entire process. It is important to note that micro-clustering time can be excluded from the total time if micro-clusters are maintained incrementally [29], rather than being computed in batch before the partitioning.

6.2. Experiment results

We now present our experimental results, studying the impact of experimental parameters on the evaluation metrics.

6.2.1. *Scalability*

To test scalability, we investigate the change in the quality metrics by running the proposed solution with varying number of nodes.

Figure 6 plots different quality measures as a function of the number of nodes used. We observe from the figure that as the number of nodes increases, cluster and balance qualities decrease, whereas the migration quality initially decreases but then switches to an increasing trend. Cluster quality decreases because the system is forced to split some of the clusters across multiple nodes, but in order to prevent a drastic decrease in cluster quality, it also sacrifices balance quality. As a result of the decrease in both cluster quality and balance quality, overall quality also decreases but this is tolerable, since the system offers scalability when clustering large number of profiles. The behavior of the migration quality is worthy of further elaboration. When the number of nodes is small, migration quality is sacrificed to keep together the clusters as much as possible and achieve high cluster quality. However, as the number of nodes become large, it becomes impossible not to split clusters, and thus the amount of migrations required decreases. The switching point at which the migration cost starts to increase has a relation with the number of true clusters in the data, which we investigate next.

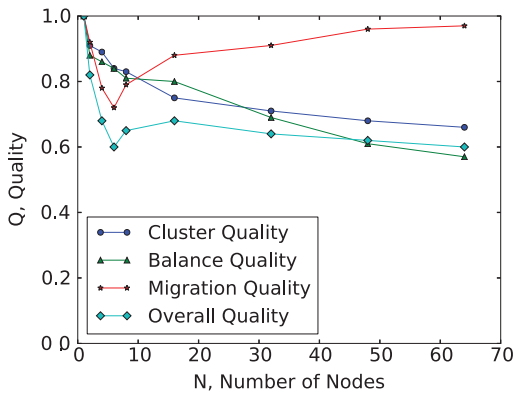

FIGURE 6. Impact of the number of nodes on the quality metrics.

As we observe from Fig. 6, up to $N = 6$, the migration quality drastically decreases. Recall from the Table 2 that the number of predefined profiles is 6, and there is skew in the sizes of the number of profiles belonging to a certain type, because of the Zipf distribution used. Our solution tries to fit every predefined profile into one processing node, but some of the profiles have to be split and migrated to other nodes due to the skew in sizes. As a result, the migration quality drops drastically. We experiment and validate our reasoning about this quality drop in Fig. 7.

Figure 7 plots the migration quality as a function of the number of nodes, for different number of profile types. When the number of profile types is small, we see a purely increasing trend in the migration quality. When the number of profile types is large, we see a purely decreasing trend in the migration. Between these two extremes, we see an elbow in the migration quality. The higher the number of profile types, the further toward the right (higher number of nodes) is the elbow's location. This behavior further supports our earlier claim that the migration quality switches back to an increasing trend after we are forced to split the profile types across nodes.

6.2.2. *Clustering versus balance*

As mentioned earlier, when we define the overall quality in Equation (8), α is used for adjusting the relative importance of balance disaffinity and cluster disaffinity. For lower values of α, cluster affinity has more importance than balance affinity in our heuristic algorithm. Conversely, for higher values of α, balance affinity has more importance.

Figure 8 plots the quality measures as a function of α. It shows that for lower values of α, the proposed solution cannot achieve balanced distribution and tries to collect all similar clusters to one node, and cluster quality becomes high, but balance quality becomes too low. Conversely, for higher values of α, the proposed solution just tries to achieve good balance, resulting in a decrease in the cluster quality. When we analyze

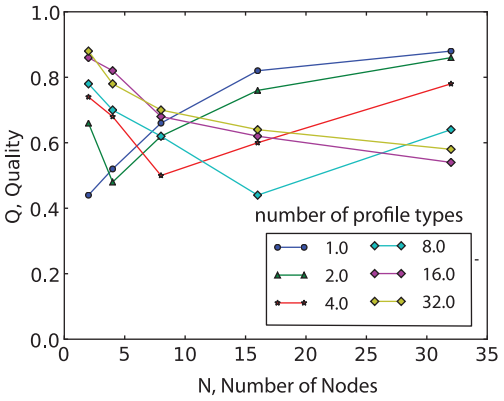

FIGURE 7. Impact of the number of profile types on migration quality.

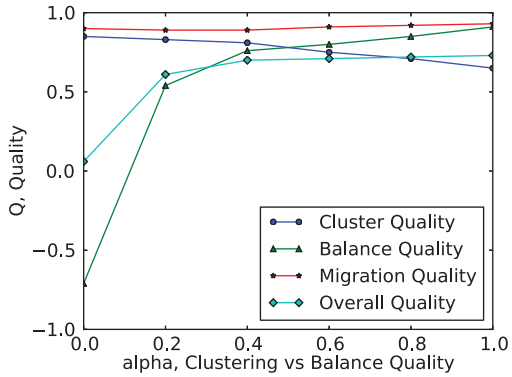

FIGURE 8. Impact of α on the quality measures.

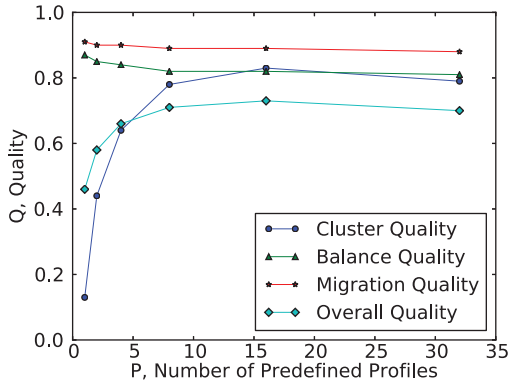

FIGURE 9. Impact of number of profile types on the quality metrics.

overall quality, it reaches its maximum point around $\alpha = 0.4$ and stays stable.

The figure also shows that even for $\alpha = 1$, the balance quality is not 1, as migration quality is still a factor in the overall partitioning process.

6.2.3. *Predefined clusters experiment*

To analyze the behavior of the proposed solution with different types of customer bases, we run experiments on different datasets, where the *Number of Profiles* is altered to experiment with different number of clusters that exist in the customer base.

Figure 9 plots the quality metrics as a function of the number of profile types. It shows that overall quality increases as the number of profile types increases, but up to some point. After reaching optimal quality, it stays relatively stable.

When there are more nodes than customer profile types, cluster quality decreases. The reason behind this is that, again, one profile type is forced to be split into multiple nodes in order to preserve balance quality. On the other hand, having more profile types than nodes does not harm the quality.

6.2.4. *Cluster size*

We experiment our proposed solution with varying sizes of profile clusters. As mentioned before, customer distribution to

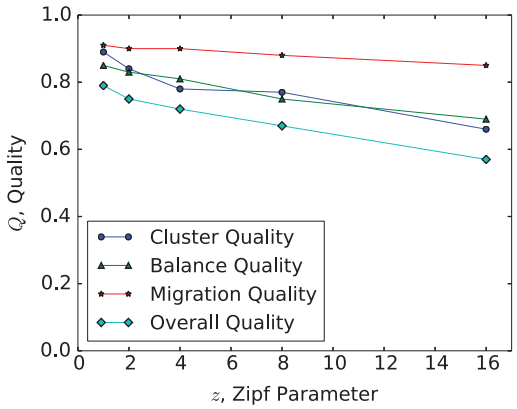

FIGURE 10. Impact of varying values of Zipf Z on the quality metrics.

profile types is done using a Zipf distribution. For low values of Z, customers have a more balanced distribution over the clusters. Increasing values of Z creates skew.

Figure 10 plots the quality measures as a function of the Zipf skew parameter Z. We observe that the balance and cluster qualities decrease as the skew is increased. The clustering quality is affected the most. As the skew increases, we get a few large clusters. Such clusters will not fit into a single node, and as a result they must be split. This results in decreased clustering quality. While the situation is hopeless for very large clusters, the clustering concern tries to improve the situation as much as possible for the other clusters that can still be located on the same node. However, this happens at the cost of reduced load balance, and thus the balance suffers as well, as the skew increases.

6.2.5. *Number of nearest neighbors*

Recall that the cluster disaffinity is calculated using the average of distances to the k nearest micro-clusters in the target node. To investigate the effect of the number of nearest neighbors on the quality measures, we experiment with varying k values.

Figure 11 plots the quality measures as a function of the number of neighbors (k) used for computing the clustering disaffinity. As we can see from the figure, increasing the number of nearest neighbors results in decreasing the cluster quality, but balance and migration qualities increase slightly.

For high values of k, the average distance of micro-clusters to nodes becomes very similar to each other and similar clusters are formed in all nodes, albeit with decreased fidelity. This provides additional flexibility for migration and balance, as clustering is often a conflicting goal.

6.2.6. *Variance in profile attributes*

Figure 12 plots the quality measures as a function of the standard deviation in the profile attributes. As we can see from the figure, with low variance in the profiles (deviation $\leq 0.1\times$ mean), the quality is high (0.75 to $\approx$1). As the variance

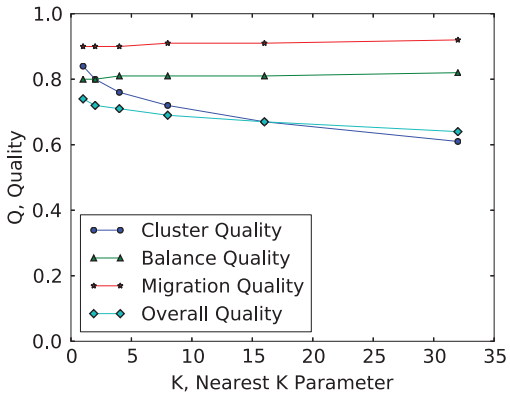

FIGURE 11. Impact of the number of neighbors on the quality metrics.

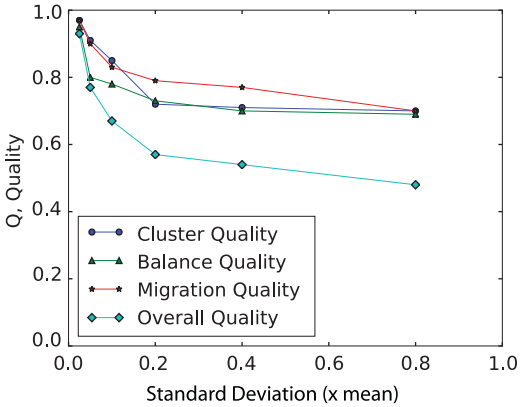

FIGURE 12. Impact of the standard deviation in profile attributes on the quality metrics.

increases, the quality suffers. This is because as the clusters become larger, the distributed approach suffers against the centralized one, as it must divide some of the clusters to maintain balance properties. Note that with further increase in variance, the decrease in clustering quality stops, as the clusters become indistinguishable from each other. However, the drop in migration quality continues, since high variance in profile updates result in profiles shifting locations, causing migration. The trend in the migration quality also impacts the overall quality. Since for high variance in profile attribute values we do not have meaningful clusters, the increased reduction in quality due to decreasing migration quality is not a concern for realistic scenarios.

6.2.7. *Execution time*

To analyze the execution time of our distributed clustering algorithm, we run our system with varying number of nodes.

Figure 13 plots the execution time of the distributed clustering steps (in seconds) as a function of the number of nodes. We observe that as the number of nodes increases, the total time and the micro-cluster time decrease (less profiles per

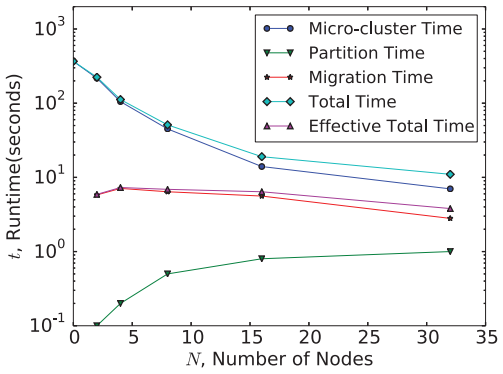

FIGURE 13. Impact of the number of nodes on the clustering time.

node), but the partition time (the centralized part) increases. Since the processing load for micro-clustering is shared across the nodes in the system, it is expected that the micro-clustering time decreases with increasing number of nodes. On the other hand, the increase in the potential target nodes for each micro-cluster considered processed by the centralized partitioning algorithm results in increased execution time as N increases.

We note that the micro-clustering time and thus the total time is quite high in terms of their absolute values in seconds. However, it is important to note that in practice the micro-clustering can be done incrementally [29] and thus the effective total time for re-partitioning can be taken as the sum of the partition time and migration time, which is under half a minute.

6.2.8. *Adaptive re-clustering experiment*

To experiment with our adaptive approach to re-clustering, we built two different dynamic update schedules. As we explained in Section 6.1.2, the CDR generator periodically checks for possible profile shifts. For this experiment, we use two different setups for the profile shifts. The first one, which we call the *fast* setup, has a shift probability that is twice of the second one, which we call the *slow* setup.

Figure 14 plots the behavior of adaptive re-partitioning for fast changing profiles, and similarly Fig. 15 plots the behavior for the slow changing profiles. *Adaptive Re-clustering(n)* shows the local clustering quality results for the nth node. In Fig. 14, cluster quality decreases rapidly, and both adaptive re-clustering and periodic re-clustering require the same number of re-clusterings. In contrast, in Fig. 15, cluster quality decreases slowly. In this setup, periodic re-clustering performs six re-clustering steps, whereas adaptive re-clustering performs only two re-clustering steps within the same time frame, yet still keeps the cluster quality above 0.9 (same threshold used in Fig. 14). As the results show, adaptive re-clustering prevents unnecessary re-clustering steps, avoiding costly pauses. The figures also show that not performing any re-clustering results in the cluster quality to drop significantly as time progresses.

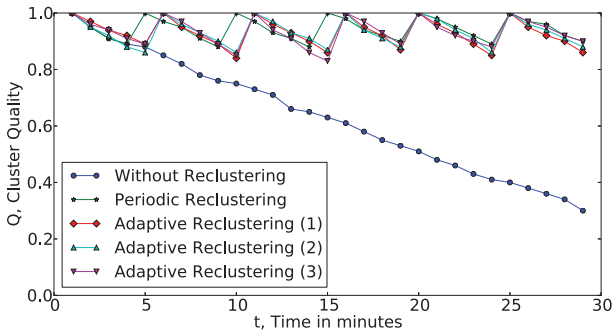

FIGURE 14. Adaptive re-clustering behavior for fast changing profiles.

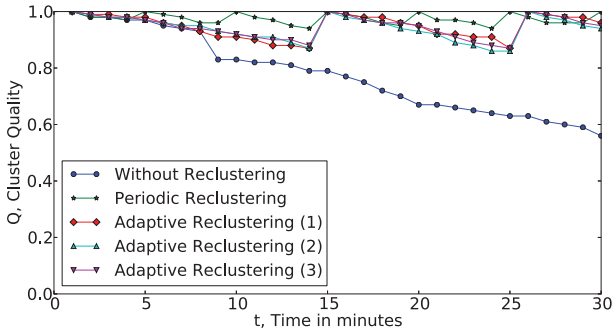

FIGURE 15. Adaptive re-clustering behavior for slow changing profiles.

7. CONCLUSION

In this paper, we introduced the problem of distributed aggregate profile clustering in a streaming setup, with the aim of enabling scalable clustered analytics on frequently updated user profile information in data-intensive services. We proposed a solution that employs partitioned stateful parallelism and heuristic re-partitioning techniques. It makes scalable clustered analytics possible by partitioning the profiles over a set of nodes, where the partitioning maintains both good load balance and good clustering accuracy (close to the case of a centralized clustering). Furthermore, our solution maintains such a partitioning when there are changes in the profiles, via lightweight adaptive re-partitioning that minimizes the migration cost. We have evaluated the performance of our proposed solution on a customer segmentation application using a CDR generator, and studied the impact of various system parameters on the performance metrics. Our evaluation shows that our solution can scale as the number of nodes increases, can provide both good clustering quality (keeps individual clusters on a single node as much as possible) and good balance (places similar amount of CPU/memory load on each node), and does this while incurring low migration cost. Furthermore, our adaptive re-clustering technique is more effective compared with periodic re-clustering as it can adapt to the rate of change in the profiles to minimize unnecessary re-clusterings.

FUNDING

This study was funded in part by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under grants EEEAG #111E217 and #112E271.

REFERENCES

[1] Hung, S.-Y., Yen, D.C. and Wang, H.-Y. (2006) Applying data mining to telecom churn management. *Expert Syst. Appl.*, **31**, 515–524.

[2] Brusco, M.J., Cradit, J.D. and Tashchian, A. (2003) Multicriterion clusterwise regression for joint segmentation settings: an application to customer value. *J. Market. Res.*, **40**, 225–234.

[3] Gür, I. and Güvercin, M. (2014) Scaling forecasting algorithms using clustered modeling. *Very Large Databases J.*, doi:10.1007/s00778-014-0363-0.

[4] Abbasoğlu, M.A., Gedik, B. and Ferhatosmanoğlu, H. (2013) Aggregate profile clustering for telco analytics. *Proc. VLDB Endow.*, **6**, 1234–1237 [demo paper].

[5] Storm. http://storm-project.net/ (accessed March 2013).

[6] HBase. http://hbase.apache.org/ (accessed March 2013).

[7] MacQueen, J.B. (1967) Some methods for classification and analysis of multivariate observations. *Berkeley Symp. Math. Stat. Probab.*, **1**, 281–297.

[8] Ester, M., Kriegel, H.-P., Sander, J. and Xu, X. (1996) A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *ACM Int. Conf. Knowledge Discovery and Data Mining (SIGKDD)*, August 2–4, 1996, Portland, Oregon, USA, pp. 226–231.

[9] McLachlan, G.J. and Krishnan, T. (2008) *The EM Algorithm and Extensions* (2nd ed.). Wiley, Hoboken, NJ, USA.

[10] Han, J. and Kamber, M. (2005) *Data Mining: Concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[11] Xu, X. and Kriegel, H.-P. (1999) A fast parallel clustering algorithm for large spatial databases. *Data Min. Knowl. Discov.*, **3**, 263–290.

[12] Dhillon, I.S. and Modha, D.S. (2000) A Data-Clustering Algorithm on Distributed Memory Multiprocessors. *Workshop on Large-Scale Parallel KDD Systems (as part of ACM SIGKDD)*, August 15, 1999, San Diego, CA, USA, pp. 245–260.

[13] Mahout. http://mahout.apache.org/ (accessed August 2013).

[14] Oryx. https://github.com/cloudera/oryx (accessed March 2014).

[15] Dean, J. and Ghemawat, S. (2010) MapReduce: a flexible data processing tool. *Commun. ACM*, **53**, 72–77.

[16] Shindler, M., Wong, A. and Meyerson, A.W. (2011) Fast and Accurate k-means for Large Datasets. *Annual Conf. Neural Information Processing Systems (NIPS)*, December 12–14, 2011, Granada, Spain, pp. 2375–2383.

[17] Januzaj, E., Kriegel, H.-P. and Pfeifl, M. (2013) Towards Effective and Efficient Distributed Clustering. *IEEE Int. Conf.*

Data Mining (as part of IEEE ICDM), November 19–22, 2003, Melbourne, Florida, USA, pp. 49–58.

[18] Cormode, G., Muthukrishnan, S. and Zhuang, W. (2007) Conquering the Divide: Continuous Clustering of Distributed Data Streams. *IEEE Int. Conf. Data Engineering (ICDE)*, April 15–20, 2007, The Marmara Hotel, Istanbul, Turkey, pp. 1036–1045.

[19] C. da Silva, J., Giannella, C., Bhargava, R., Kargupta, H. and Klusch, M. (2005) Distributed data mining and agents. *Eng. Appl. Artif. Intell.*, **18**, 791–807.

[20] Eyal, I., Keidar, I. and Rom, R. (2011) Distributed data clustering in sensor networks. *Distrib. Comput.*, **24**, 207–222.

[21] Gedik, B., Liu, L. and Yu, P.S. (2007) ASAP: an adaptive sampling approach to data collection in sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, **18**, 1766–1783.

[22] Zhang, T., Ramakrishnan, R. and Livny, M. (1996) BIRCH: An Efficient Data Clustering Method for Very Large Databases. *ACM Int. Conf. Management of Data (SIGMOD)*, June 4–6, 1996, Montreal, Quebec, Canada, pp. 103–114.

[23] Guha, S., Meyerson, A., Mishra, N., Motwani, R. and O'Callaghan, L. (2003) Clustering data streams: theory and practice. *IEEE Trans. Knowl. Data Eng.*, **15**, 515–528.

[24] Cormode, G., Garofalakis, M., Muthukrishnan, S. and Rastogi, R. (2005) Holistic Aggregates in a Networked World: Distributed Tracking of Approximate Quantiles. *ACM Int. Conf. Management of Data (SIGMOD),* June 14–16, 2005, Baltimore, Maryland, USA.

[25] Danon, L., Diaz-Guilera, A. and Duch, J. (2005) Comparing community structure identification. *J. Statist. Mech. Theory Exp.*, **2005**, P09008.

[26] Rosenberg, A. and Hirschberg, J. (2007) V-Measure: A Conditional Entropy-Based External Cluster Evaluation. *Joint Conf. Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, June 28–30, 2007, Prague, Czech Republic, pp. 410–420.

[27] Pelleg, D. and Moore, A.W. (2000) X-Means: Extending k-Means with Efficient Estimation of the Number of Clusters. *Int. Conf. Machine Learning (ICML)*, June 29 – 2, 2000, Stanford University, Stanford, CA, USA, pp. 727–734.

[28] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, H.I. (2009) The weka data mining software. *ACM SIGKDD Explor. Newslett.*, **11**, 225–234.

[29] Aggarwal, C.C., Han, J., Wang, J. and Yu, P.S. (2003) A Framework for Clustering Evolving Data Streams. *Int. Conf. on Very Large Databases Conf. (VLDB)*, September 9–12, 2003, Berlin, Germany, pp. 81–92.

[30] Hadoop. http://hadoop.apache.org/ (retrieved August 2013).

[31] Manning, C.D. and Schütze, H. (1999) *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.