C-Cube: Elastic Continuous Clustering in the Cloud

Zhenjie Zhang [1], Hu Shu [2], Zhihong Chong [2], Hua Lu [3], Yin Yang [1]

[1]Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore
{zhenjie,yin.yang}@adsc.com.sg

[2]*School of Computer Science and Engineering, Southeast University, China*
{hshu.seu, chong.seu}@gmail.com

[3]*Department of Computer Science, Aalborg University, Denmark*
luhua@cs.aau.dk

***Abstract*— Continuous clustering analysis over a data stream reports clustering results incrementally as updates arrive. Such analysis has a wide spectrum of applications, including traffic monitoring and topic discovery on microblogs. A common characteristic of streaming applications is that the amount of workload fluctuates, often in an unpredictable manner. On the other hand, most existing solutions for continuous clustering assume either a central server, or a distributed setting with a fixed number of dedicated servers. In other words, they are not *elastic*, meaning that they cannot dynamically adapt to the amount of computational resources to the fluctuating workload. Consequently, they incur considerable waste of resources, as the servers are under-utilized when the amount of workload is low.**

This paper proposes C-Cube, the first elastic approach to continuous streaming clustering. Similar to popular cloud-based paradigms such as MapReduce, C-Cube routes each new record to a processing unit, e.g., a virtual machine, based on its hash value. Each processing unit performs the required computations, and sends its results to a lightweight aggregator. This design enables dynamic adding/removing processing units, as well as replacing faulty ones and re-running their tasks. In addition to elasticity, C-Cube is also effective (in that it provides quality guarantees on the clustering results), efficient (it minimizes the computational workload at all times), and generally applicable to a large class of clustering criteria. We implemented C-Cube in a real system based on Twitter Storm, and evaluated it using real and synthetic datasets. Extensive experimental results confirm our performance claims.

I. Introduction

Clustering is a fundamental problem in data management, with numerous applications, e.g., in information retrieval [30], network design [9], multimedia analysis [21], etc. In a nutshell, clustering divides a set of unlabeled objects into groups that are not pre-defined, such that objects in the same group are similar to each other, and objects in different groups are dissimilar. In many practical applications involving streaming data, e.g., traffic jam monitoring and topic discovery on microblogs, it is desirable to run clustering analysis continuously, and report results in real time. Meanwhile, as is common in streaming applications, the amount of workload may fluctuate in an unpredictable manner. For instance, there can be a sudden surge of microblog posts when a major event occurs, leading to significantly higher computational costs for clustering the posts. Reporting results in real-time at such peak workload may require a large amount of computational resources, whereas resource demand at non-peak periods is often significantly lower. These characteristics motivate an *elastic* solution, which dynamically adjusts the amount of computational resources based on the current workload. The cloud platform fits these requirements well, as it provides high efficiency, reliablity and elasticity. The goal of this work, thus, is to design and implement an elastic algorithm for real-time, continuous clustering analysis on top of the cloud platform.

Although both streaming clustering and elastic query processing are well-studied in the literature, their combination, i.e., elastic continuous clustering, remains a challenging problem, and we are not aware of any existing solution for this purpose. As reviewed in Section 2, the majority of previous streaming clustering methods are restricted to a centralized server. Further, existing parallelized streaming clustering algorithms are inherently inelastic, because they assume a fixed number of computation nodes with complex internal states. Consequently, changing the number of nodes incurs high migration overhead for rebuilding these states. Finally, previous cloud-based clustering algorithms are limited to static datasets, which are not suitable for streaming data. Rerunning such a method at every timestamp is inefficient, since it typically imposes a high initialization cost.

In this paper, we present *C-Cube*, a general and elastic streaming framework to support a variety of clustering algorithms. C-Cube follows a simple and yet powerful design called *elastic operator*. Figure 1 shows an example of an elastic operator, which performs the task of one logical operator in the execution plan of a query. Internally, an elastic operator contains a *dispatcher*, a set of *processing units*, and an *aggregator*. All processing units share the same functionality, i.e., the main function of the entire elastic operator. When a new tuple arrives, the dispatcher routes it to one of the processing units based on its hash value. The corresponding processing unit then processes the tuple, and sends the results to the aggregator. Finally, the aggregator produces the output of the operator based on the results received from the processing units. As we show in the paper, this design achieves elasticity by dynamically adjusting the number of processing units. Besides elasticity, this design also enables fault tolerance

(by replacing faulty processing units) and load balancing (by modifying the routing strategy at the dispatcher). Further, since an elastic operator is encapsulated as one logical operator, it can be readily used in current data stream management systems by replacing existing operator implementations.

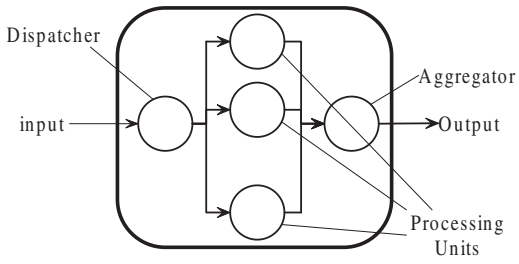


Fig. 1. Example elastic operator

Applying the elastic operator design to streaming clustering, however, is difficult. The main challenge lies in that the clustering results depend on all attributes of all objects, rather than a subset of them. Meanwhile, since clustering is an NP-hard problem, practical solutions often return approximate answers. As we review in Section 2, methods with good result quality are often difficult to parallelize. C-Cube tackles these problems using a verification-reclustering scheme: it verifies the clustering results computed at a previous timestamp, and only re-runs the clustering algorithm when the verifier module determines that the previous results no longer fit the current data distribution. This architecture shifts a considerable portion of the workload to the verification module, which is performed by an elastic operator. Further, for a wide class of distance-based clustering criteria, C-Cube guarantees that the quality of outputs is within a user-specified constant factor to the optimal clustering results. Finally, C-Cube handles efficiently and effectively object insertions and deletions, as well as joins and leaves of computational nodes. We have implemented C-Cube on top of a real streaming cloud platform *Twitter Storm*, and conducted extensive experiments with real data. The results confirm that C-Cube works very well on large scale datasets with a small amount of cloud computation resources. To the best of our knowledge, this is the first implementation of general clustering algorithm fully compatible with mainstream cloud systems.

Our main contributions are summarized as follows.

1) We propose a general framework C-Cube for elastic execution of continuous clustering on the cloud platform, for a large class of distance-based clustering criteria.
2) We devise a verification-reclustering architecture in C-Cube that only re-executes the clustering module when the previous results no longer satisfies the user-specified quality guarantee.
3) We derive the theory for analyzing the quality of past clustering results on current data, which generally supports the insert/remove operators on both objects and computing nodes.
4) We implement C-Cube on a real-time cloud-based analytic platform, *Twitter Storm*, and conduct extensive performance studies using real Twitter dataset. The results demonstrate that C-Cube can handle large-scale continuous clustering efficiently, with high result quality.

The remainder of the paper is organized as follows. Section II reviews the existing studies on clustering algorithms, especially on constant approximation algorithms, streaming and distributed clustering techniques. Section III provides the preliminaries on distance-based clustering algorithms. Section IV analyzes the optimality of continuous clustering and elaborates on the verification theory behind our C-Cube framework. Section V presents the details of C-Cube framework, introducing the algorithms for all basic operations used in the framework. Section VI covers the implementation details with Twitter Storm and discusses the experimental results on both synthetic and real datasets. Section VII concludes this paper and addresses some future research directions.

II. Related Work

A. Clustering Algorithms

Clustering is a well studied topic in computer science. While hundreds of different clustering algorithms were proposed in the literature, we mainly focus on the most common class, which consists of *distance-based* clustering methods.

In distance-based clustering, a distance function is employed to measure the dissimilarity between any two objects in the specific domain, e.g., Euclidean space. The goal of clustering is to partition the objects into clusters, to minimize the sum of distances from all objects in a cluster to the cluster center, or (equivalently) the sum of pairwise distances between the objects in every individual cluster. Methods in this category often differ on the distance functions adopted. K-means clustering [25], for example, uses Euclidean distance as its distance function, and aims to minimize the aggregate distance between each object and the geometric center of the corresponding cluster. Due to the use of centroids as cluster centers, k-means clustering is limited to datasets that lie in an Euclidean space with finite dimensionality. In contrast, k-median clustering eliminates this restriction by using data objects as cluster center rather that the centroids; thus, k-median clustering applies to datasets in any metric space. Both k-means and k-median are known to be hard problems, such that it takes exponential time in terms of the number of objects to find the global optimal clustering.

Due to the hardness, computer scientists resorted to approximate schemes to distance-based clustering problems, looking for algorithms always outputting clustering results with constant approximation guarantee. Arora [4] designed the first approximation algorithm to k-median problem in Euclidean space. After that, a series of improvements were made to improve the approximation bound, e.g., [6], [13], [12], [22]. Arya et al. [6] proposed a $(5 + 2/k)$-approximate k-median clustering in any metric space, using simple local search heuristic. Kanungo et al. [22] applied the similar strategy on k-means, and obtained similar constant approximation. Arthor and Vassilvitskii [5] designed a randomized algorithm

to achieve $O(\ln k)$-approximation in expectation. Note that C-Cube framework does not rely on the underlying clustering algorithm, which is therefore applicable with any of the algorithms mentioned above.

B. Distributed Clustering

Continuous clustering over streaming data has also received extensive research effort in the last decade. Guha et al. [19] presented the first clustering algorithm on data streams. The approach relies on the strategy of combining local clustering results to ensemble the global clustering results. It is extensible to a distributed environment where each node is responsible for clustering partial data in the stream. However, this approach requires that there are a fixed number of computational nodes. In particular, since the local clustering cannot be further partitioned, this approach does not support arbitrary adjustments on computational resource on cloud platform. A similar strategy was proposed by Zhang et al. [31] which summarizes local samples using the concept of coreset. Their technique provides an effective scheme on in-network sample generation and aggregation, which is exploited to reduce the communication cost within the network. Our problem formulation, however, focuses on application with updates from external source of cloud system, with goal of minimization on computation cost and response time on continuous clustering. Zhang et al. [32] presented a centralized solution for continuous clustering on moving objects, following the framework of verification-and-reclustering. This solution is difficult to extend to distributed environment, because the complicated aggregation function needs inputs from all updates. Cormode et al. [17] designed a distributed clustering algorithm for k-center clustering, which aims to minimize the maximal distance from the objects to the representative centers. Our distance-based clustering uses summation in the distance aggregation, which forbids the direct employment of their scheme. Moreover, it is important to emphasize that all of the algorithms above do not support *elasticity*, i.e. re-scaling adaptive to the update workload.

C. Batch and Stream Processing in the Cloud

Since the original proposal of MapReduce [18] only supports batch-based data processing, some systems are built to overcome this limitation. Haloop [11], for example, supports loops of operations, meaning that the output of a previous map-reduce task is fed into a subsequent map-reduce task as input. Therefore, Haloop naturally supports batch-based iterative clustering algorithms, such as the standard k-means algorithm. Mahout [1] is a mature open-source machine learning library adopting MapReduce as its underlying infrastructure. In the machine learning community, Chu et al. [16] presented systematic studies on the possibility of implementing machine learning methods on MapReduce, which include clustering algorithms.

Traditional data stream management systems (DSMSs), e.g., Aurora [3], STREAM [7], NiagaraCQ [14], etc., generally do not consider the unique properties of the cloud system, such as elasticity. Similarly, methods proposed for traditional DSMSs, e.g., adaptive query processing [8], just-in-time scheduling [28], dynamic plan migration [33], [29], etc., cannot be applied directly for cloud-based systems. DEDUCE [24] is an extension of IBM's System S DSMS that allows the user to write streaming processing jobs in MapReduce style. It automatically translates the MapReduce program into streaming processing language and deploys the computation procedures on the distributed platform. However, users cannot implement more complicated stream processing algorithm using DEDUCE, if the algorithm is not fully consistent with the MapReduce paradigm.

III. Preliminaries

This section introduces the problem definition of continuous clustering and covers the preliminaries for the clustering verification theory that is to be presented in Section IV. Assume that a dynamic object set D contains n objects, i.e., $D = \{o_1, o_2, \ldots, o_n\}$. For each object o_i, we use $o_i^{(t)}$ to denote the location of o_i at time t. $D^{(t)}$ consists of the locations of all objects at time t. A center set C consists of k centers, i.e., $C = \{c_1, c_2, \ldots, c_k\}$, each of which is in the same domain of the objects. The goal of *distance-based clustering* is to identify a cluster center set to minimize the aggregation on distances from the centers and the objects. Specifically, a distance function maps a pair of locations to a non-negative real number, i.e. $\mathsf{dist}(\cdot,\cdot)$, to measure the dissimilarity between these two locations (usually those of a center and an object). The cost of the clustering center set C with respect to the objects at time t is thus $\mathsf{cost}(D^{(t)}, C)$, i.e.,

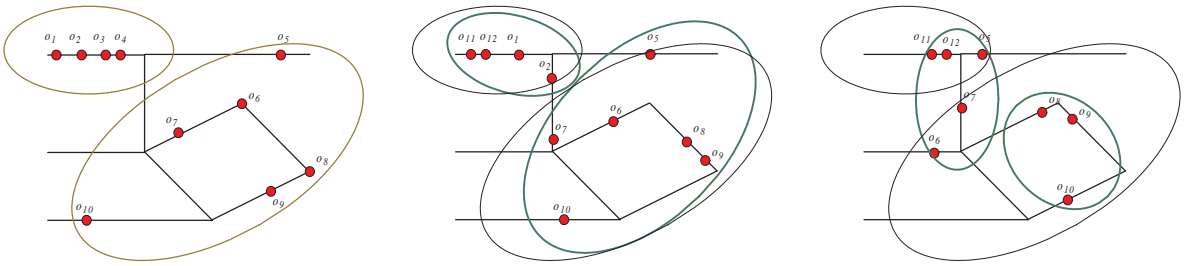
$$\mathsf{cost}\left(D^{(t)}, C\right) = \sum_{o_i \in D} \min_{c_j \in C} \mathsf{dist}\left(o_i^{(t)}, c_j\right) \qquad (1)$$

Here, $\sum$ is an aggregation operator which sums up the minimum distances from $o_i^{(t)}$ to the centers in C. Equation (1) is a general model for a variety of distance-based clustering problems when employing different distance functions as the underlying measure criteria. In this paper, we investigate the possibility of designing general streaming processing platform to support *any* distance-based clustering algorithm, when the distance function satisfies certain general property. In particular, we consider two common distance functions below, namely squared Euclidean distance and general metric distance.

Example 1: k-Means Clustering
Assume that every $o_i^{(t)}$ lies in a d-dimensional Euclidean space. Let $\mathsf{dist}(\cdot,\cdot)$ be squared Euclidean distance. The function in Equation (1) becomes the cost function for k-means clustering.

Example 2: k-Median Clustering
Let $\mathsf{dist}(\cdot,\cdot)$ be a metric distance defined for every $o_i^{(t)}$ in certain metric space. The function in Equation (1) is then the cost function for k-median clustering.

Given the object locations in $D^{(t)}$ at time t and a positive integer k, a clustering center set $C_*^{(t)}$ is optimal in terms of $D^{(t)}$, if it is the minimizer in Equation (1). While the problem



(a) Moving vehicles at time t (b) Moving vehicles at time $t+1$ (c) Moving vehicles at time $l > t+1$

Fig. 2. An example of traffic monitoring on consecutive timestamps

of calculating the optimal clustering center set $C_*^{(t)}$ is known to be NP-hard in most cases, there exist several effective approximate schemes to find sub-optimal solutions with certain relaxed optimality guarantee. Given an approximation factor $\alpha > 1$, a clustering center set $C^{(t)}$ is α-approximate to the optimal center set $C_*^{(t)}$, if

$$\frac{\mathsf{cost}\left(D^{(t)}, C^{(t)}\right)}{\mathsf{cost}\left(D^{(t)}, C_*^{(t)}\right)} \leq \alpha \tag{2}$$

Given a static dataset D and a cluster size parameter k, there have been constant approximation algorithms for distance-based clustering problems, e.g., $O(\log k)$-approximate k-means [5], $(5 + 2/k)$-approximate k-median [6]. However, the problem of continuous clustering on the objects in D is much more difficult. In this paper, we design a framework to fully utilize existing constant approximate clustering algorithms. Given an α-approximate clustering algorithm, we target at continuously maintaining β-approximate clustering center sets for every timestamp t, with new approximation factor $\beta > \alpha$. A naive solution to the problem is retrieving new location of every object o_i on every timestamp and re-running the α-approximate clustering algorithm on each $D^{(t)}$ independently. This method is unpractical on cloud platform, due to the high overhead on the repeated clustering procedures on each timestamp. Instead, our framework keeps collecting the updates from the objects and invokes the re-clustering only when the previous clustering result is no longer a β-approximate optimal clustering to the current timestamp.

We use an example of traffic monitoring to explain the basic principles behind our framework. In Figure 2, vehicles (shown as red dots) are moving on a road network (black lines). Given the 2-clustering result on time t, it is unnecessary to recompute the clustering at $t+1$, because the data distribution remains similar. At time l ($l > t+1$), the optimal structure of 2-clustering changes significantly. To keep an update-to-date clustering, the system must calculate a new 2-clustering result based on the current locations of the vehicles.

Although the re-clustering could reuse the existing standard clustering algorithm, it is more efficient to measure the quality of the previous clustering result on the current data and update the result incrementally only if it is necessary. In this paper, we focus on designing an elastic cloud based architecture that effectively estimates optimality of clusterings in real time. Mathematically, given the α-approximate center set $C^{(t)}$ calculated at time t, and the current updates $D^{(l)}$ from the objects, the problem of *clustering verification* is how to accurately evaluate the validity of the following inequality.

$$\frac{\mathsf{cost}\left(D^{(l)}, C^{(t)}\right)}{\mathsf{cost}\left(D^{(l)}, C_*^{(l)}\right)} \leq \beta \tag{3}$$

To guarantee that our framework always maintains β-approximate clustering, the evaluation on Inequality (3) should only allow false positive but no false negative. In other words, we are only allowed to overestimate $\frac{\mathsf{cost}\left(D^{(l)}, C^{(t)}\right)}{\mathsf{cost}\left(D^{(l)}, C_*^{(l)}\right)}$ so that the system never misses clustering updates when necessary.

In the next section, we propose a theory for verifying clustering results that are obtained from distance-based clustering techniques. The theory is to be applied to our C-Cube framework design in Section V.

IV. Clustering Verification Theory

A. Centralized Verification

To evaluate Inequality (3), the key is to estimate $\mathsf{cost}(D^{(l)}, C^{(t)})$ and $\mathsf{cost}(D^{(l)}, C_*^{(l)})$ separately and accurately. When the system receives all updates from the objects at every timestamp, it is pretty easy to exactly calculate the cost $\mathsf{cost}(D^{(l)}, C^{(t)})$ by aggregating the distances. The denominator $\mathsf{cost}(D^{(l)}, C_*^{(l)})$ is much more difficult for evaluation, mainly because it is impossible or too expensive to compute the optimal clustering at each timestamp l. To avoid any false negative in the inequality evaluation, an alternative and more practical solution is deriving a lower bound on $\mathsf{cost}(D^{(l)}, C_*^{(t)})$, rendering an overestimation on $\frac{\mathsf{cost}(D^{(l)}, C^{(t)})}{\mathsf{cost}(D^{(l)}, C_*^{(l)})}$. To derive an inexpensive lower bound calculation scheme, we discuss certain general properties of the distance function. First, the distance function $\mathsf{dist}(x, y)$ must satisfy the basic conditions of identifiability, positivity, and symmetry, i.e., 1) $\mathsf{dist}(x, y) = 0$ if and only if $x = y$; 2) $\forall x, y: \ \mathsf{dist}(x, y) \geq 0$; and 3) $\forall x, y: \ \mathsf{dist}(x, y) = \mathsf{dist}(y, x)$. Second, the distance function must follow the generalized triangle inequality below.

Definition 1: **Generalized Triangle Inequality** Given a positive constant $\tau \leq 1$, a distance function

$\mathsf{dist}(\cdot,\cdot)$ satisfies τ-relaxed triangle inequality, if $\mathsf{dist}(x,y) + \mathsf{dist}(y,z) \geq \tau \cdot \mathsf{dist}(x,z)$.

It is straightforward that any metric distance naturally follows the relaxed triangle inequality with $\tau = 1$. Squared Euclidean distance, on the other hand, is not a metric distance function, but remains consistent with the definition of the general triangle inequality with $\tau = 0.5$ [27].

For a positive integer n, a permutation σ is a one-to-one mapping from integers in $[1,\ldots,n]$ to $[1,\ldots,n]$, i.e., every integer i corresponds to another integer $\sigma(i)$. The following theorem lays the foundation for our general evaluation theory, using the concept of permutation.

Theorem 1: If $C^{(t)}$ is an α-approximate clustering upon $D^{(t)}$, with any permutation σ of size n, the optimal clustering cost on data $D^{(l)}$ is lower bounded by

$$\begin{aligned} & \mathsf{cost}\left(D^{(l)}, C_*^{(l)}\right) \\ \geq & \sum_{o_i}\left(\frac{\tau}{\alpha}\cdot \min_{c_j\in C^{(t)}} \mathsf{dist}\left(o_i^{(t)}, c_j\right) - \mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)\right) \end{aligned}$$

Proof: First, $C^{(t)}$ is an α-approximate clustering on $D^{(t)}$. Thus, we have

$$\frac{1}{\alpha}\mathsf{cost}\left(D^{(t)}, C^{(t)}\right) \leq \mathsf{cost}\left(D^{(t)}, C_*^{(t)}\right) \tag{4}$$

Then, we derive the connection between the optimal clusterings $C_*^{(t)}$ and $C_*^{(l)}$. We can show that

$$\begin{aligned} & \mathsf{cost}\left(D^{(t)}, C_*^{(t)}\right) \\ \leq & \mathsf{cost}\left(D^{(t)}, C_*^{(l)}\right) \\ = & \sum_{o_i} \min_{c_j^{(l)}\in C_*^{(l)}} \mathsf{dist}\left(o_i^{(t)}, c_j^{(l)}\right) \\ \leq & \frac{1}{\tau}\sum_{o_i}\left(\min_{c_j^{(l)}\in C_*^{(l)}} \mathsf{dist}\left(o_{\sigma(i)}^{(l)}, c_j^{(l)}\right) + \mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)\right) \\ = & \frac{1}{\tau}\left(\mathsf{cost}\left(D^{(l)}, C_*^{(l)}\right) + \sum_{o_i}\mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)\right) \end{aligned} \tag{5}$$

The first inequality is due to the optimality of $C_*^{(t)}$ on $D^{(t)}$. The second inequality exploits the property of generalized triangle inequality, i.e., $\tau \cdot \mathsf{dist}(o_i^{(t)}, c_j^{(l)}) \leq \mathsf{dist}(o_{\sigma(i)}^{(l)}, c_j^{(l)}) + \mathsf{dist}(o_i^{(t)}, o_{\sigma(i)}^{(l)})$. ■

Given Theorem 1, we can rewrite a relaxed evaluation condition for Inequality (3) as follows:

$$\frac{\sum_i \min_{c_j\in C^{(t)}} \mathsf{dist}\left(o_i^{(l)}, c_j\right)}{\sum_i\left(\frac{\tau}{\alpha}\cdot \min_{c_j\in C^{(t)}} \mathsf{dist}\left(o_i^{(t)}, c_j\right) - \mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)\right)} \leq \beta \tag{6}$$

To illustrate the meaning of Equation (6), we present an example on the left side of Figure 3, where 2-dimensional object locations are shown as white circles at time t and grey circles at time l. Given a permutation, as is indicated by

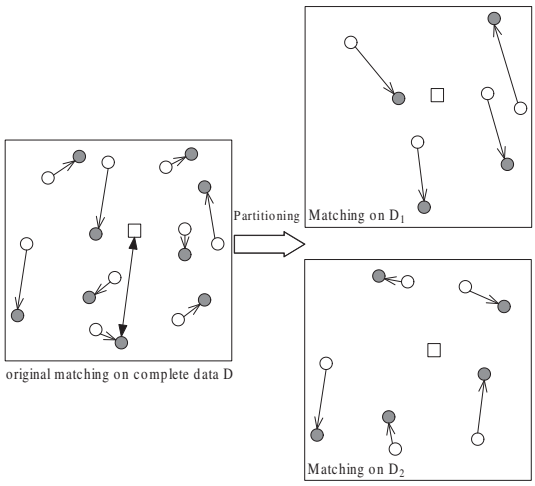


Fig. 3. An example of permutation and matching in 2-dimensional space

directed arrows, we model the change of the distribution by the pairwise distance between the matching object locations across timestamps. When the distances between matching locations are relatively small, i.e., it is significantly smaller than the distance to the cluster center (shallow square), the distribution is stable, implying the previous clustering remains accurate. In particular, when $\mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right) = 0$ for all i, the dataset does not change at all, even if the permutation does not pair the same object across the timestamps. This property is reflected in the new Inequality (6), in which small pairwise distance on the permutation leads to a smaller ratio.

B. Distributed Verification

While Inequality (6) is simple to evaluate, there remain certain computational difficulties to overcome. To maximize the denominator on the lefthand side, the system needs to find a permutation to minimize $\sum_{o_i}\mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)$. This is equivalent to the bipartite minimal matching problem, when all the updates are received and processed by a single server in the system. There are polynomial algorithms, e.g., the Hungarian algorithm [26], to solve the bipartite minimal matching problem, incurring cubic complexity with respect to the data size.

To extend the verification technique in a cloud environment with high extensibility and elasticity, we further consider how to distribute the evaluation on multiple computation nodes, when each node only receives the updates from a small fraction of objects. Assume the object set is arbitrarily partitioned into m subsets, i.e. $D = D_1 \cup D_2 \ldots \cup D_m$, such that $D_i \cap D_j = \emptyset$ for every i and j. Given such a partitioning, we define a special class of permutations, called *constrained permutation*.

Definition 2: A permutation σ is a valid constrained permutation with respect to the partitioning $(D_1,\ldots,D_m)$, if $o_{\sigma(i)} \in D_r$ for each $o_i \in D_r$.

Based on the concept of constrained permutation, instead of finding the global permutation that minimizes

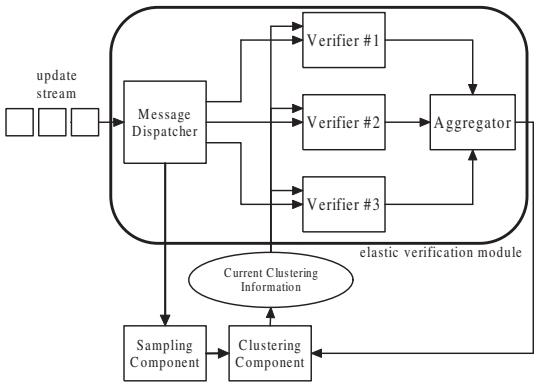


Fig. 4. Architecture Framework of C-Cube

$\sum_{o_i} \mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)$, we resort to the optimal constrained permutation with respect to the specific partitioning $(D_1, D_2, \ldots, D_m)$. In Figure 3, we partition the object set into D_1 and D_2. Generally speaking, the pairwise matching cost increases, because the best matching pairs on the complete data may not be assigned to the same partition. The right-top object in D_1, for examples, is matched to a grey point further than its partner in complete data D. However, the increase on the pairwise matching cost is limited. To validate this intuition, we provide a formal analysis on the convergence property on matching within constrained permutations.

Assume every object is identically and independently drawn from a distribution. Every object o_i is assigned to a data partition D_r uniformly, i.e. $\Pr(o_i \in D_r) = 1/m$ for every $1 \leq r \leq m$. In the following, we analyze the optimality of pairwise matching distance sum $\sum_{o_i} \mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)$, when every object satisfies the uniform assignment assumption.

Theorem 2: If σ^* is the optimal permutation from $D^{(t)}$ to $D^{(l)}$ and σ is the optimal constrained permutation from $D^{(t)}$ to $D^{(l)}$ with respect to partitioning $(D_1, D_2, \ldots, D_m)$ under uniform assignment assumption, the following convergence property holds:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{o_i \in D} \left(\mathsf{dist}\left(o_i^{(t)}, o_{\sigma^*(i)}^{(l)}\right) - \mathsf{dist}\left(o_i^{(t)}, o_{\sigma(i)}^{(l)}\right)\right) = 0$$

The theorem above implies that each partition D_r is a subsample set of the underlying data distributions at time t and time l. Therefore, each intra-partition pairwise matching partly reflects the difference between the distributions, and each such difference uniformly converges to the exact global value when the sampling rate is large enough. Based on this observation, we design the C-Cube framework in next section, which fully exploits the distributed clustering verification theory.

V. C-Cube Framework

A. Overview

The basic architecture of C-Cube is shown in Figure 4. Each large rectangle denotes a computation node (i.e., virtual machine), which is the minimum independent computation unit on cloud platforms. Specifically, *Message Dispatcher* works as a gateway of the system, responsible for handling the update messages from the objects and distributing the messages to other components in the system. The *verifier* and the *aggregator* together form the core component of C-Cube. They continuously verify the optimality of the current clustering upon receiving updates from the message dispatcher. Since most of the computation resource in C-Cube is spent on clustering verification, the parallelism of this component is the focus of our system design. Our verifier-aggregator scheme works as follows. The updates from the objects are hashed onto different verifiers in the system. Each verifier U_r maintains a permutation σ_r over the active objects attached with U_r. Statistics are collected by U_r and forwarded to the aggregator. Based on the statistics from the verifiers, the aggregator makes an overall evaluation on the quality of the clustering and decides if it is necessary to trigger the clustering algorithm in the clustering component.

Algorithm 1 Verifiers's Message Handler (new update $o_i^{(l)}$ from object o_i)

1: Update T_r with $o_i^{(l)}$ for object o_i
2: Run Hungarian algorithm to update the mapping σ_r
3: Evaluate S_r by Equation (7)
4: Evaluate B_r by Equation (8)
5: Send (S_r, B_r) to the aggregator

Algorithm 2 Aggregator's Message Handler ((S_r, B_r) from Verifier U_r)

1: Update S_r and B_r in its database
2: Calculate Ratio= $\frac{\sum_r S_r}{\sum_r B_r}$
3: **if** Ratio$> \beta$ **then**
4: Trigger the clustering component
5: **end if**

In addition, the *sampling* component records every update message from the objects and maintains samples of the object locations based on a specified sampling rate. When the *clustering* component receives the command to start a new clustering, it kicks off a clustering thread on the current samples available in the sampling component. Note that the underlying clustering algorithm does not necessarily run on a single virtual machine. Distributed clustering algorithm with certain number of independent virtual machines is naturally supported in the framework. After the clustering component finishes re-clustering on the current samples, it updates the *current clustering information* with the new clustering result. The new clustering information is thus automatically pushed to the verifiers to update the verification process.

B. Object Updates in C-Cube

There are three types of object updates in C-Cube, including *status update*, *object insertion* and *object deletion*. In the following, we describe the updating protocol with all these operations in detail.

To handle status update, each verifier U_r maintains four data structures, including 1) an active object list $D_r \subseteq D$; 2)

the location $D_r^{(t)}$ for objects in D_r updated at previous re-clustering time t; 3) latest update statuses T_r for objects in D_r; and 4) a local constrained permutation σ_r mapping from statuses in T_r to statuses in $D_r^{(t)}$. When object o_i updates its status by a message $\left(o_i, o_i^{(l)}\right)$ at time l, the verifier first refreshes the corresponding entry in T_r, i.e. writing $o_i^{(l)}$ into T_r. The verifier then runs the Hungarian algorithm with the current σ_r as the initial matching. Since there is one and only one update in T_r, the Hungarian algorithm terminates in $O(|D_r|^2)$ time instead of $O(|D_r|^3)$ time as a complete re-execution of Hungarian algorithm takes. The new assignment, i.e., a new permutation σ_r' is used to replace the previous permutation σ_r. Given the new permutation, the verifier is capable of calculating the following two statistical measures:

$$S_r = \sum_{o_i \in D_r} \min_{c_j \in C^t} \mathsf{dist}\left(o_i^{(l)}, c_j\right) \tag{7}$$

$$B_r = \sum_{o_i \in D_r} \left(\frac{\tau}{\alpha} \min_{c_j \in C^t} \mathsf{dist}\left(o_i^{(t)}, c_j\right) - \mathsf{dist}\left(o_i^{(t)}, o_{\sigma_r(i)}^{(l)}\right)\right) \tag{8}$$

The new S_r and B_r are sent to the aggregator for further processing. The complete pseudo-code of verifier is summarized in Algorithm 1. The total computation time for every individual update is $O(|D_r|^2)$, since the update of permutation takes $O(|D_r|^2)$ time and the calculation of S_r and B_r is finished in $O(|D_r|)$ time.

On the aggregator node, it maintains the latest (S_r, B_r) from every verifier U_r. After receiving a message from U_r with the new (S_r, B_r), the aggregator refreshes the record for U_r in its own database and evaluate Inequality (6) in an equivalent way as

$$\frac{\sum_r S_r}{\sum_r B_r} \leq \beta \tag{9}$$

If the inequality above is no longer satisfied, the aggregator sends a new message to the clustering component to trigger a new round of re-clustering computation. The pseudo-code of the aggregator is listed in Algorithm 2. If the statistical information from the verifiers is kept in main memory, the computation cost of the aggregator on each message is constant. Therefore, the total computation cost on an object update only depends on the number of objects maintained by the verifier.

So far, we assume that there is no insertion or deletion on the objects in D. In the following, we extend our discussion to discard this assumption. In particular, we present insertion and deletion mechanisms in C-Cube for volatile objects under surveillance. Note that Inequality (6) provides guarantee on clustering quality verification, when the number of objects maintained by the system is fixed. Therefore, instead of discussing insertion/deletion operations separately, we consider object set update on two cases, 1) when the object maintained by U_r at time l is more than the number of objects at a previous re-clustering time t, i.e., $|D_r^{(l)}| > |D_r^{(t)}|$; and 2) when the object maintained by U_r at time l is less, i.e., $|D_r^{(l)}| < |D_r^{(t)}|$.

Growing Case: When $\left|D_r^{(l)}\right| > \left|D_r^{(t)}\right|$, there does not exist a one-to-one mapping σ_r. Fortunately, the lower bound on the optimal clustering remains valid, even when some objects at time l do not have a matching counterpart at time t. This is because the optimal distance-based clustering on a subset of objects always renders a lower cost [19]. Based on this property, we slightly change Inequality (6) by calculating the lower bound only on objects with a good mapping. This leads us to the new computation formula B_r, such that

$$B_r' = \sum_{o_{\sigma_r(i)}^{(t)} \in D_r^{(t)}} \left(\frac{\tau}{\alpha} \min_{c_j \in C^t} \mathsf{dist}\left(o_i^{(t)}, c_j\right) - \mathsf{dist}\left(o_i^{(t)}, o_{\sigma_r(i)}^{(l)}\right)\right)$$

The rest of the algorithms for both the verifiers and the aggregator remain the same. We therefore skip the details of the algorithms.

Shrinking Case: When $\left|D_r^{(l)}\right| < \left|D_r^{(t)}\right|$, the problem of object insertion/deletion is more complicated, since the previous lower bound estimation is no longer valid. To resolve this issue in practice, we make a general assumption on the disappearance ratio of the objects and design a simple mechanism for the shrinking case.

Assumption 1: At any time l, the number of objects is no less than the number of objects at another time t by scaling factor γ, i.e., $|D^{(l)}| \geq \gamma |D^{(t)}|$ for a constant $0 < \gamma \leq 1$.

Based on the above assumption and the Chernoff bound [20], the number of objects maintained by any verifier U_r at time l is no less than $(\gamma - \sqrt{-2\gamma \ln \delta})|D_l^{(t)}|$ with probability at least $1-\delta$. It means that the number of objects at each verifier is lower bounded by a constant ratio with high probability. This observation helps us transform the problem of *shrinking* to the problem of *growing*, if the system maintains another clustering and sample set under sampling rate $\gamma - \sqrt{-2\gamma \ln \delta}$.

Specifically, the system keeps another sample set on the objects at time t, i.e., the clustering re-computation time. There are thus two approximate clustering results maintained by the system. Whenever an verifier finds the number of objects it maintains is shrinking, it automatically switches to the clustering on the smaller sample set. By sharing the sampling random seed with every verifier, each verifier is capable of identifying which location in $D_r^{(t)}$ is included in the sample set. This enables the verifier to use the right subset of $D_r^{(t)}$ for permutation computation, when a shrinking case happens.

The overhead of this scheme is small. Each verifier only needs to keep track of the objects in the sample set, with a simple binary label on each object in the local database. The clustering component calculates two clusterings at full size and at a lower sampling rate respectively. However, when γ is close to 1, i.e., the object number does not change dramatically, these two clusterings are similar to each other. In such cases, it is sufficient for the clustering component to calculate only one clustering center set for both of them.

The aforementioned technique alleviates the problem when the above assumption holds. We have a simple solution when this assumption does not hold. For each verifier, a counter is used to track the number of active objects. When the counter implies that there are less than $\gamma|D_r^{(t)}|$ objects alive, it immediately sends a message to the aggregator to request a re-clustering. This helps every verifier refresh its matching base from $D_r^{(t)}$ to $D_r^{(l)}$, since the new clustering is for the latest object updates. This only causes a very small number of re-clustering operations, unless the number of objects fluctuates rapidly, which is rare in practice.

C. Elasticity, Load Balancing and Fault Tolerance

C-Cube achieves elasticity, load balancing, and fault tolerance mainly through the elastic operator design. Elasticity is achieved by dynamically adjusting the number of processing units. Load balancing is reflected by the fact that the dispatcher routes the objects randomly to processing units, based on their hash values. Fault tolerance is performed by simply replacing a faulty processing unit, and re-execute it. Specifically, each processing unit regularly reports its performance statistics to a performance monitor of the elastic operator. When the monitor detects that the number of processing units is insufficient to handle the current workload, it dynamically allocates new computational nodes as additional processing units in the operator. Conversely, if the processing units are under-utilized, the operator removes a portion of processing units, so that the computation power of the operator matches the current workload.

When the processing unit changes, the dispatcher updates the routing protocol based on the current number of processing units. The dispatcher modifies the hash function for assigning objects to processing units. To migrate the local information for each object o_i, including status of o_i on previous re-clustering time t and the latest status update $o_i^{(l)}$ on time l, the dispatcher sends the new hash function to each processing unit. In the verification module, for instance, the verifier U_r sends local information with each $o_i \in D_r$ to the corresponding new verifier for o_i. To minimize the migration cost, we adopt consistent hashing [23] for this purpose, which minimizes the number of objects migrated from one verifier to another.

In C-Cube, the verification module is realized through the elastic operator design, but the clustering component still applies existing algorithms. To achieve elasticity, one approach is to dynamically adjust the the sampling rate in the sampling component, which changes the expected workload of the clustering component. Implementing the clustering component is hard, and remains an open problem. Nevertheless, as our experiments demonstrate, in C-Cube the majority of the workload is performed by the elastic verification module.

VI. Experiments

A. Experiment Setup

In the experiments, we test C-Cube with two different clustering criteria, namely *k-means clustering* and *k-median clustering*. For k-means clustering, we employ k-means++ [5] as the underlying clustering algorithm, which outputs $O(\ln k)$-approximate clustering results by expectation. For k-median clustering, we use the local search algorithm [6], which outputs $(5+2/k)$-approximate results. The continuous clustering algorithms are applied on two very different problem domains, i.e. k-means clustering on *Moving Object Analysis* and k-median clustering on *Microblog Analysis*. In the following, we give detailed descriptions of the datasets in these two domains.

Moving Object Analysis: Clustering on moving objects is an important topic for traffic analysis and location-based services. Given the continuous clusterings results reported by the system, the user can identify potential traffic jams in the road network and understand the dynamics of traffic jams by watching the trends of the clusterings in consecutive timestamps. In our experiments, we use the popular synthetic moving object trajectory generator developed by Brinkhoff [10]. The generated dataset contains trajectories in 100 timestamps. We initialize 100,000 moving objects under slow speed on the map of Oldenburg. At each timestamp, the simulator generates 500 new moving objects. When an object terminates its journey, its location is set to $(-1, -1)$, and the object is removed at the following timestamp. Every object reports its location to the system at every timestamp. These locations are fed into our cloud-based system for k-means clustering analysis.

Microblog Analysis: Microblog systems, e.g., Twitter, have recently attracted extensive attentions in both academia and industry, because they greatly accelerate the information flowing among people. Clustering techniques can be used to capture the fresh and popular topics upon the arrivals of the tweets. In our experiments, we test C-Cube using the real data previously used [15]. This dataset records 28,688,584 tweets from 2,168,939 twitter users, from October 2006 to November 2009. After removing stop words and numbers, we picked 1,000 most popular words as keywords and transfer every tweet to a vector of size 1,000. The vector records the number of keywords appearing in the corresponding tweets. After removing all tweets without any keyword, there remain 22,613,449 tweet vectors with 3.75 non-zero entries on average. To analyze the trend of the topics discussed on Twitter, we continuously calculate the k-median clustering with L_1 distance on the tweet vectors, over a sliding window of consecutive 100,000 tweets.

Implementation: We run our experiments on a cluster with 9 PCs running Ubuntu system of version 10.0.4. Each PC is equipped with up to 2GB main memory, and a 1.8 GHz Intel dual core CPU. All PCs are connected through a regular network router. We implemented C-Cube on top of Twitter Storm 0.6.2. One of the PCs acts as the Zookeeper server, which is responsible for clustering resource maintenance. Another two PCs run as the Nimbus nodes for the cluster and the Kestrel message queue server respectively. The rest of the PCs in the cluster are supervisor nodes controlled by the Nimbus server.

Some details of our implementation and configurations on the C-Cube framework are given as follows. Following the

Cluster ID	Important keywords				
Cluster 566	C1: kids, school	C2: program, gotta, affiliate, power	C3: money, save	C4: studio, posted, video	C5: live, music, party, hotel
	C6: sleep, night, hour	C7: facebook, excited	C8: long, weekend, ready	C9: webinar, global, food	C10: google, voice, search
	C11: read, dark, book	C12: nice, app, list	C13: spring	C14: snow, weather, rain	C15: idea, hard, crazy
	C16: ebay, check, store	C17: whats, area, favorite	C18: isnt, working, people		
Cluster 567	C1: view, paul, latest, click	C2: meet, service	C3: kids, song, training	C4: working, website, hard	C5: article, magazine, read, interesting
	C6: enjoying, friend, weekend	C7: spring, finished	C8: iphone	C9: radio, guest	C10: wedding, lives, post, people
	C11: interesting, read, article	C12: sharing, article, business	C13: fans, music, facebook	C14: worst, world, home	C15: health, care
	C16: morning, work	C17: snow, rain, weather	C18: gonna, gotta, youre	C19: forget, feel, youre	
Cluster 568	C1: weekly, updated	C2: talking, idea	C3: havent, heard, years	C4: tips, find, google	C5: phone, cell, call
	C6: read, sell, book	C7: beautiful, out!, website	C8: gave, night, start	C9: problems, twitter!, finding	C10: games, watch, youtube
	C11: luck, pretty, stuff	C12: spring, finished	C13: real, estate	C14: updated, work, email	C15: gotta, hear, watching

TABLE I

CONTINUOUS CLUSTERING RESULTS ON *Twitter* DATASET

architecture shown in Figure 4, the Kestrel message queue server works as the *message dispatcher*. In Twitter Storm, each continuous analytic job is called a *topology*, consisting of two types of nodes, i.e. *spout* nodes and *bolt* nodes. We map C-Cube to a topology, in which the message dispatcher and verifiers run as *spout* nodes, and the aggregator is a *bolt* node. Clustering component is run as an independent *bolt* node, which receives a copy of every update message and samples them at 1% sampling rate. The clustering component maintains a sample pool of fixed size, 1% of the number of estimated active objects. Twitter Storm allows the user to manually specify the maximal number of nodes running at the same time. Unless specified otherwise, we set the parallelism parameters as follows: maximum 2 message dispatchers, maximum 100 verifiers and 1 aggregator. More details on the meanings of these parameters are available in Twitter Storm's documentation [2].

One implementation difficulty with *Twitter Storm* is to add new virtual machines into the cloud system when other nodes are still working. To circumvent this problem, our implementation on Twitter Storm first starts a maximal number of virtual machines at the beginning of the analytic process. Nevertheless, C-Cube only uses a fraction of the virtual machines and keeps other virtual machines in idle. Based on the workload, particularly when the system throughput is below the update arrival rate, C-Cube automatically changes the hashing function, and includes more virtual machines into the computation. This scheme incurs little overhead, since the idle virtual machines spend negligible computation resources of the cloud platform.

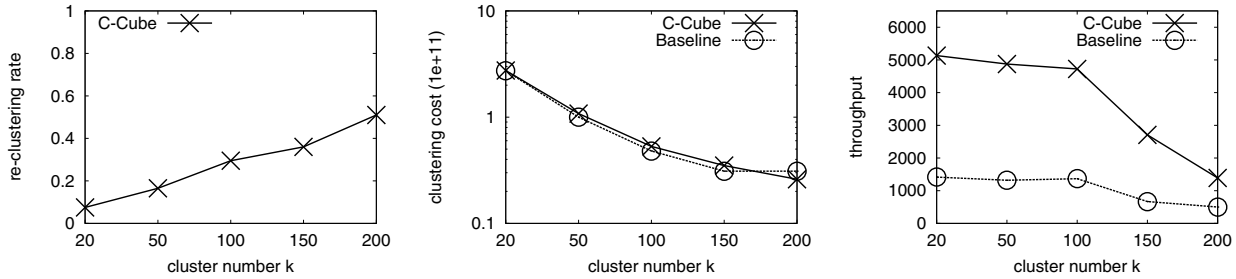
B. Clustering Results

In this section, we present continuous clustering results on *Twitter* dataset to illustrate the usefulness of C-Cube on topic analysis. While more complicated models have been developed for advanced information retrieval and natural language processing, we show that k-median clustering on popular words in tweets is sufficient to extract interesting patterns and trends of the topics. These clustering results are calculated by employing $k = 20$ on a sliding window of 100,000 tweets. In Table I, we list the clustering results continuously reported by the C-Cube framework. For each cluster, we identify the most frequent keywords from the tweets in the corresponding cluster. Some small clusters that contain no more than 20 tweets are excluded in the table, leading to the number of clusters less than 20.

From Table I, we observe a variety of popular topics lasting for a long period of time. In the following, we use Cx-y to denote the x'th cluster of clustering at timestamp #y. Topics related to mobile phones include C12-566 (on mobile phone applications), C8-567 (about the iPhone) and C6-568 (on phone calls). Google is another well covered topic, which is included in C10-566 (on Google Voice) and C4-568 (tips on using Google Search). Spring is another topic commonly discussed in the tweets, which appears in clusterings C13-566, C7-567 and C12-568. Weather is also a hot topic in the discussion as it appears in C14-566 and C17-567.

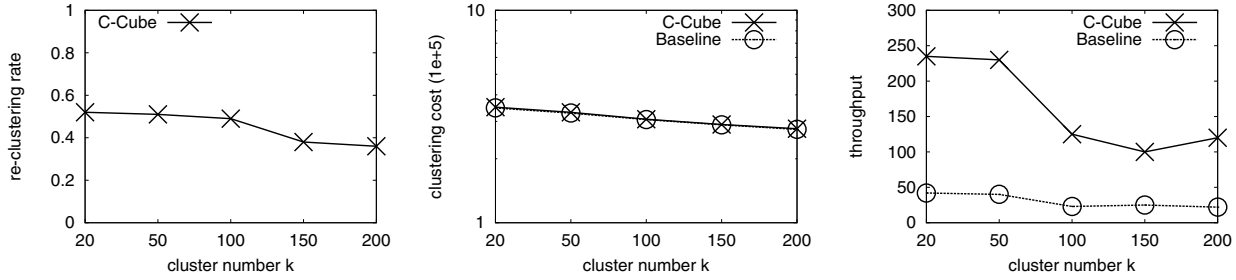
Some other topics last only for a short period of time. For example, C6-566 covers words "sleep" and "night" because these tweets are sent at late night. Words "morning" and "work" start to appear in C16-567 when the users go to work in the morning. Similar observations can be found on words such as "weekend".

Finally, there is an emerging topic found in C9-568, which shows that a large number of users are complaining about certain problems with Twitter. This implies that real-time clustering information can be helpful for system administrators to quickly pinpoint possible problems based on the users' tweets.



(a) Re-clustering rate (b) Clustering quality (c) Throughput

Fig. 5. Impact of number of clusters k on *Moving Object*



(a) Re-clustering rate (b) Clustering quality (c) Throughput

Fig. 6. Impact of number of clusters k on *Twitter*

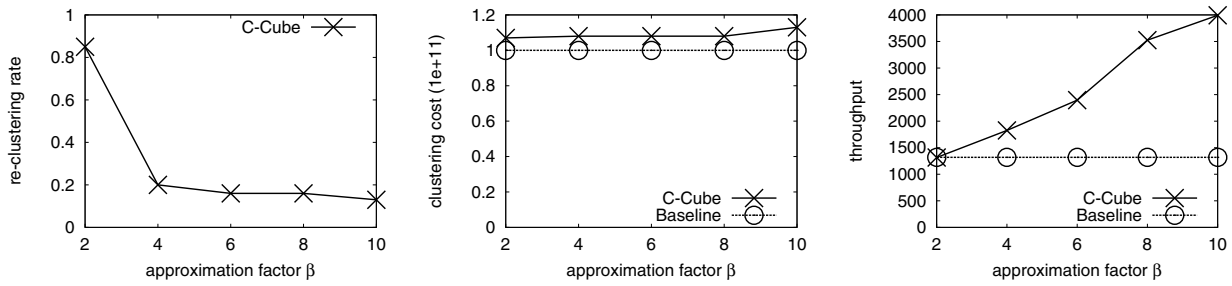
C. System Performance

In this part of the section, we focus on the empirical studies on the system performance of C-Cube. To the best of our knowledge, there is no existing system which is capable of running real-time continuous clustering algorithms on the cloud platform. To validate the usefulness of C-Cube, we included a baseline approach (referred to as *Baseline* in the rest of the section) in our experiments. Specifically, Baseline is also implemented on top of Twitter Storm, in which the system maintains a pool of bolts, and assigns an idle bolt at each timestamp to handle all object updates and perform *re-clustering*, i.e., re-computing the clustering results from scratch for each timestamp. We measure the performance of the systems in terms of *re-clustering rate*, *cluster cost*, *system throughput* and *maximal response time*. The re-clustering rate is the ratio of timestamps that the underlying clustering algorithm performs re-clustering. For instance, Baseline's re-clustering rate is always 1 as it does this at every timestamp. The cluster quality is the average cost according to the clustering criterion, e.g., the sum of object-center distances in k-means and k-median clustering. The system throughput is the average number of updates the system processes every second (wall clock time). In the rest of this section, we analyze the effects of (1) number of clusters k, (2) approximation factor β, (3) number of verifiers used in C-Cube, (4) workload change rate, and (5) number of machines in the cluster. The default values for these parameters are $k = 20$, $\beta = 8$ and 250 verifiers.

We first report the impact of the number of clusters k on *Moving Object* and *Twitter* in Figures 5 and 6, respectively. The experiments on both datasets show C-Cube achieves a dramatic reduction on the re-clustering rate. Referring to Figure 5 (a), C-Cube only recalculates the clusterings on less than 10% of the timestamps for $k = 20$ on *Moving Object*. The re-clustering rate on *Moving Object* grows with the number of clusters k, whereas the trend is reversed on *Twitter* as shown in Figure 6 (a). This is because of the unstable clusters when the system forcefully partitions them into more groups/clusters. *Twitter*, in contrast, consists of a large number of tweets that address a variety of different topics, leading to more stable clusterings and less re-clusterings with large k. Meanwhile, the difference in result quality between C-Cube and Baseline is negligible on both datasets, which is verified by the results shown in Figures 5(b) and 6(b).

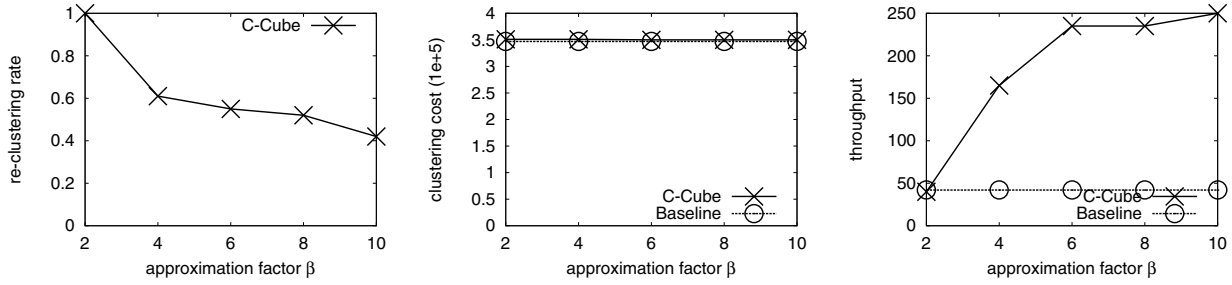
The average throughput of C-Cube is about 5 times higher than that of the Baseline on both datasets, according to the results shown in Figures 5(c) and 6(c). This shows the C-Cube is significantly better on handling large numbers of updates from the objects. When increasing the number of clusters, the throughput of both approaches declines. This is expected, since the cost of re-clustering increases with k. Nevertheless, we always observe considerable performance gap between C-Cube and Baseline, when k is larger than 2.

We report the results on varying the approximation parameter β in Figures 7 and 8. Note that in practice, β should be set with a higher value than the approximation factor of the underlying clustering algorithm. We use smaller values in our experiments anyway, in order to show how C-Cube performs even in such extreme cases. Referring to Figures 7(a) and 8(a), the re-clustering rates of C-Cube on *Moving Object* and *Twitter* are 0.8 and 1 respectively when $\beta = 2$, meaning that C-Cube needs to re-compute the clustering results at



(a) Re-clustering rate (b) Clustering quality (c) Throughput

Fig. 7. Impact of parameter β on *Moving Object*



(a) Re-clustering rate (b) Clustering quality (c) Throughput

Fig. 8. Impact of parameter β on *Twitter*

almost every timestamp. Consequently, the throughput of C-Cube degenerates to that of Baseline, referring to the results when $\beta = 2$ in Figures 7(c) and 8(c). On the other hand, as shown in Figures 7(b) and 8(b), such a small value of β does not improve the clustering quality significantly; the quality score is almost constant with all values of β, except when $\beta = 10$, where we observe a slight cost increase on the *Moving Object* data. This result suggests that a moderate β is sufficient in practice.

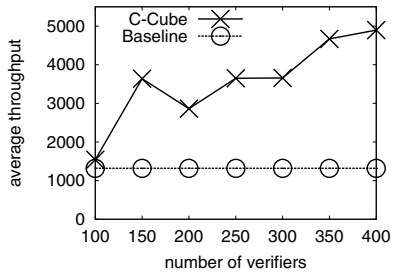


Fig. 9. Impact of the number of verifiers on *Moving Object*

The previous experiments focus on the parameters controlling the clustering procedure. Next, we test the impact of cloud system settings, particularly the number of verifiers (i.e., virtual machines) available to C-Cube. Note that Baseline uses exactly one virtual machine, i.e. *bolt* node, for every timestamp. In Figure 9, we present the results of average throughput of C-Cube, on *Moving Object*. It shows that adding more numbers of virtual machines, which is significantly larger than the actual cores in the physical cluster, improves the system throughput. This is due to the computational time saved on the bipartite matching procedure when each verifier is responsible for matching less objects. It implies that most of the CPU cycles in C-Cube are spent on the bipartite matching in the verification instead of the re-clustering computation.

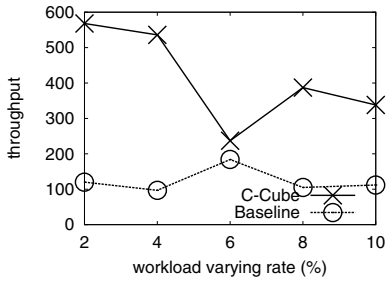
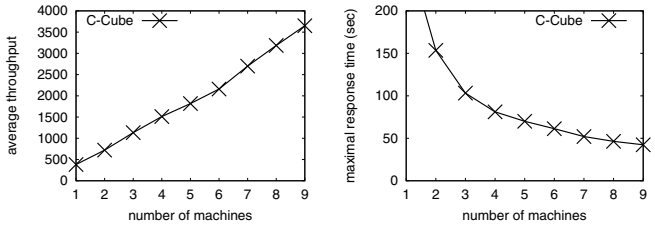


Fig. 10. Impact of workload change rate on *Twitter*

We also test the effect of workload change rate using the *Twitter* dataset. In particular, we control the number of active objects at every timestamp by adjusting the sliding window size on tweets. On the first timestamp, the clustering results are computed using the first 50,000 tweets; in each of the the following timestamps, we uniformly select the number of tweets within $1 \pm x\%$ of the previous timestamp, where x is the workload change rate under our investigation. We vary x from 2 to 10, while the other parameters are fixed at their defaults. Figure 10 presents the results on re-clustering rate and average system throughput. It is seen that C-Cube remains robust when the workload changes slowly. When x reaches 10%, the re-clustering rate of C-Cube grows to 0.6; nonetheless, C-Cube's throughput at $x = 10\%$ is still about 3 times as high as that of Baseline.

Finally, we evaluate C-Cube by varying the number of machines in the cluster, while the number of virtual machines



(a) System throughput (b) Maximal response time

Fig. 11. Impact of cluster size on *Moving Object*

is fixed to 250. Figure 11 presents the results on system average throughput and the maximal response time. Here, maximal response time is the longest time period between the timestamp when an object update is issued that triggers the re-clustering procedure and the time when the new clustering result is output by the system. The results in Figure 11 show that the average throughput of C-Cube scales linearly to the number of physical machines. Again, this confirms that most of the computation in C-Cube is on clustering verification, which can be effectively accelerated when more verifiers are available. When there are more than 5 machines used in the cluster, C-Cube is able to produce the response within 1 minute, which is often sufficient for daily traffic monitoring applications.

VII. Conclusion

In this paper, we present C-Cube, a general framework for real-time clustering on a cloud platform. C-Cube supports a variety of clustering criteria, e.g., k-means and k-median clustering. It provides robust guarantees on the quality of the clustering results, as well as elasticity in the presence of changing workload. C-Cube can be easily implemented on a general-purpose cloud platform, e.g., Twitter Storm. Empirical studies validate the superiority of C-Cube on both high effectiveness in identifying meaningful clusters and considerable performance gains compared to alternatives. It shows that C-Cube achieves generality, elasticity, quality and efficiency at the same time.

Acknowledgments

Zhang and Yang are supported by Human Sixth Sense Programme (HSSP) at ADSC from Singapore's A*STAR. Shu and Chong are supported by the National Natural Science Foundation of China under grant No. 60973023. Chong is also supported by the Key Lab of Computer Network & Information Integration, Ministry of Education, Southeast University. Chong is the corresponding author.

References

[1] http://mahout.apache.org/.
[2] https://github.com/nathanmarz/storm.
[3] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.
[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD conference*, pages 49–60, 1999.
[5] D. Arthur and S. Vassilvitskii. k-means++: The advantage of careful seeding. In *SODA*, 2007.
[6] V. Arya, N. Garg, R. Khandekar, K. Munagala, and V. Pandit. Local search heuristic for k-median and facility location problems. In *STOC*, pages 21–29, 2001.
[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.
[8] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, 2005.
[9] S. Bandyopadhyay and E. J. Coyle. An energy efficient hierarchical clustering algorithm for wireless sensor networks. In *INFOCOM*, 2003.
[10] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
[11] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.
[12] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *FOCS*, pages 378–388, 1999.
[13] M. Charikar, S. Guha, E. Tardos, and D. B. Shmoys. A constant-factor approximation algorithm for the k-median problem (extended abstract). In *STOC*, pages 1–10, 1999.
[14] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.
[15] M. D. Choudhury, Y.-R. Lin, H. Sundaram, K. S. Candan, L. Xie, and A. Kelliher. How does the data sampling strategy impact the discovery of information diffusion in social media? In *ICWSM*, 2010.
[16] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.
[17] G. Cormode, S. Muthukrishnan, and W. Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *ICDE*, pages 1036–1045, 2007.
[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[19] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng.*, 15(3):515–528, 2003.
[20] T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Inf. Process. Lett.*, 33(6), 1990.
[21] A. Hinneburg and D. A. Keim. An efficient approach to clustering in large multimedia databases with noise. In *SIGKDD*, pages 58–65, 1998.
[22] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: analysis and implementation, 2002.
[23] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.
[24] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. Deduce: at the intersection of mapreduce and stream processing. In *EDBT*, pages 657–662, 2010.
[25] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.
[26] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, pages 67–71. Dover Publications, 1998.
[27] C. G. Plaxton. Approximation algorithms for hierarchical location problems. In *STOC*, pages 40–49, 2003.
[28] Y. Yang and D. Papadias. Just-in-time processing of continuous queries. In *ICDE*, 2008.
[29] Y. Yang, D. Papadias, J. Krämer, and B. Seeger. Hybmig: a hybrid approach to dynamic plan migration for continuous queries. *IEEE TKDE*, 19(3):398–411, 2007.
[30] H.-J. Zeng, Q.-C. He, Z. Chen, W.-Y. Ma, and J. Ma. Learning to cluster web search results. In *SIGIR*, pages 210–217, 2004.
[31] Q. Zhang, J. Liu, and W. Wang. Approximate clustering on distributed data streams. In *ICDE*, pages 1131–1139, 2008.
[32] Z. Zhang, Y. Yang, A. K. H. Tung, and D. Papadias. Continuous k-means monitoring over moving objects. *IEEE Trans. Knowl. Data Eng.*, 20(9):1205–1216, 2008.
[33] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data sreams. In *SIGMOD*, 2004.