# SpeCH: A Scalable Framework for Data Placement of Data-intensive Services in Geo-distributed Clouds

Ankita Atrey[a,*], Gregory Van Seghbroeck[a], Higinio Mora[b], Filip De Turck[a], Bruno Volckaert[a]

[a] *Internet Technology and Data Science Lab (IDLAB), Ghent University, imec, Technologiepark-Zwijnaarde 126, B-9052, Ghent, Belgium*
[b] *Department of Computer Science Technology and Computation, University of Alicante, Alicante, Spain*

---

## Abstract

The advent of big data analytics and cloud computing technologies has resulted in wide-spread research on the data placement problem. Since data-intensive services require access to multiple datasets within each transaction, traditional schemes of uniformly partitioning the data into distributed nodes, as employed by many popular data stores like HDFS or Cassandra, may cause network congestion thereby affecting system throughput. In this article, we propose a scalable and unified framework for data-intensive service data placement into geographically distributed clouds. The proposed framework introduces a new paradigm for partitioning a set of data-items into geo-distributed clouds using *Spectral Clustering on Hypergraphs*, and is therefore called *SpeCH*.

Scaling spectral methods to large workloads is challenging, since computing the spectra of the hypergraph laplacian is a computationally intensive task. SpeCH provides two solutions to tackle this problem: (1) an algorithm, called *SpectralApprox*, that leverages randomized techniques for obtaining low-rank approximations of the hypergraph matrix with bounded guarantees, thereby significantly improving the efficiency of spectral clustering while also providing high quality solutions in practice; (2) an algorithm, called *SpectralDist*, that exploits the highly parallel nature of the spectral clustering algorithm and uses Apache Spark to speed-up the process while retaining the same quality guarantees as the exact algorithm. Additionally, being distributed in nature, SpectralDist enables SpeCH to perform data placement on workloads that require resources beyond the capacity of a single machine. Experiments on a real-world trace-based online social network dataset show that the SpeCH is effective, efficient, and scalable. Empirically, *SpectralApprox* is comparable in efficacy on the evaluated metrics, while being up to 10 times faster in execution time when compared to state-of-the-art techniques. On the other hand, though *SpectralApprox* is 7-8 times faster when compared to *SpectralDist*, in terms of efficacy on the evaluated metrics the latter is up to 50% better.

*Keywords:* Data placement, Geo-distributed clouds, Location-based services, Online social networks, Scalability, Spectral Clustering, Hypergraphs, Approximation, Distribution, Apache Spark

---

## 1. Introduction

With the emergence of *Cloud computing*, *Big Data*, and *Internet of Things (IoT)*, the rate at which data is being generated is increasing at an exponential rate (insideBIGDATA Editorial Team, 2017). For instance, the amount of data managed by Internet giants like Google and Facebook is of the order of *thousands of petabytes* (Schultz, 2017). Although the advancements in modern hardware, big data, and cloud computing technologies have enabled development of several distributed systems that have significantly enriched the field of scalable data management, effective strategies for data partitioning and placement are cardinal to the performance of such systems. Popular distributed data processing systems such as Hadoop (White, 2012) and, more recently, Apache Spark (Zaharia et al., 2016) distribute data uniformly across servers and perform parallel computations on the constructed small subsets of data on each server independently (Golab et al., 2014; Atrey et al., 2018). While uniform data partitioning schemes using hashing work well for MapReduce style workloads that can be easily parallelized, they are not suitable for *data-intensive* workloads that require access to multiple datasets within each transaction (Yu and Pan, 2015; Golab et al., 2014; Zhao et al., 2016b; Shabeera et al., 2017; Zhao et al., 2016a). In these scenarios uniform partitioning may result in a huge volume of data migrations thereby leading to network congestion and eventually reduced system throughput, especially in the case of geographically distributed data-centers where inter-datacenter communication latencies and costs are relatively high. Thus, there is a need for specialized data placement strategies for *data-intensive services*.

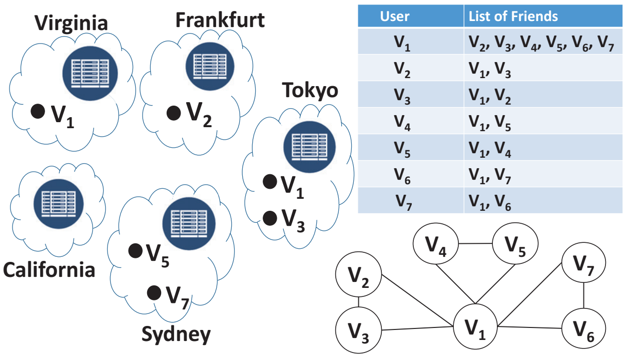
### 1.1. Use Case: Location Based OSN Service

Data-intensive services are becoming increasingly common in a plethora of real-world scenarios, namely – *online social networks (OSNs)*, *content distribution networks (CDNs)* etc. The use case under investigation is a location-based OSN service as portrayed in Fig 1. A sample Facebook social network

---

[*] Corresponding author.
*Email addresses:* `ankita.atrey@ugent.be` (Ankita Atrey), `gregory.vanseghbroeck@ugent.be` (Gregory Van Seghbroeck), `hmora@ua.es` (Higinio Mora), `filip.deturck@ugent.be` (Filip De Turck), `bruno.volckaert@ugent.be` (Bruno Volckaert)

*Preprint submitted to Journal of Network and Computer Applications* *February 6, 2019*



| User | List of Friends |
|---|---|
| $V_1$ | $V_2, V_3, V_4, V_5, V_6, V_7$ |
| $V_2$ | $V_1, V_3$ |
| $V_3$ | $V_1, V_2$ |
| $V_4$ | $V_1, V_5$ |
| $V_5$ | $V_1, V_4$ |
| $V_6$ | $V_1, V_7$ |
| $V_7$ | $V_1, V_6$ |

Figure 1: **Use Case: A location-based online social network service.**

is represented using a graph where each vertex corresponds to a user and undirected edges between two vertices represent friendship. In this network, users $\{v_1, v_5\}$ are friends of the user $v_4$. Similarly $\{v_6, v_1\}$ are friends of $v_7$. The list of all the friends of every user is also portrayed in a table in Fig. 1. There exists a notion of a data-item corresponding to each user of the social network, which represents the most recent snapshot (ex: profile picture, post etc.) of her profile. Additionally, each user can register a check-in, which is denoted by her user-id assigned to a data-center nearest (in geographical distance) to her check-in location. In Fig. 1, the user $v_1$ has registered two check-ins, at data-centers located in Virginia and Tokyo respectively, while user $v_2$ has checked-in at Frankfurt. Moreover, each user check-in requires retrieval of the data from their friends, constituting a data-request pattern triggered by this check-in. For example, while registering a check-in in Sydney the user $v_5$ may want to tag/mention some of her friends. This would require the data-items corresponding to her friends $\{v_1, v_4\}$ to be available at the Sydney data-center, thereby triggering a data request for transferring data-items corresponding to $\{v_1, v_4\}$ to Sydney.

Motivated by the use-case discussed above, the problem of *scalable* data placement for *data-intensive* services in data-centers that are *distributed geographically* across the world is the topic of research tackled in this article. This problem presents multiple challenges that are listed below:

1. Data-intensive geo-distributed cloud services operate at a massive scale. For instance, OSNs (ex: Facebook, Twitter) and CDNs (ex: YouTube, Netflix) contain billions of users and videos respectively. Thus, designing effective data placement strategies capable of scaling gracefully to large workloads remains an important challenge.

2. Additionally, the ubiquity of cloud computing and increased reliance of people across the globe on online services like OSNs and CDNs, requires data to be stored in data-centers that are geographically distributed. For instance, in the case of CDN services like *YouTube* the hosted content is stored in data-centers located around the world. It is highly likely that the set of content retrieved by a user query may be stored in different data-centers across the globe. Similar is the case for OSNs as well.

3. Since the users of OSN services like *Facebook* may register *check-ins* at various locations across the world, not only the data-items but also the source locations of data requests are geographically distributed.

While data placement has been studied extensively (Golab et al., 2014; Yuan et al., 2010; Ebrahimi et al., 2015; Jiao et al., 2014) (detailed literature review present in Sec. 2), literature on geo-distributed data-intensive services is relatively scarce (Yu and Pan, 2017). Any successful solution to this problem should provide two capabilities, namely – capturing and improving (1) data-item – data-item associations (i.e., the number of times two data-items were requested together); and (2) data-item – data-center associations (i.e., the number of times a data-item was requested at a given data-center). State-of-the-art methods proposed in (Nishtala et al., 2013) and (Agarwal et al., 2010) are capable of improving data-item – data-item, and data-item – data-center associations respectively, however, these methods cannot jointly handle both aspects. To jointly incorporate both aspects, (Yu and Pan, 2015, 2017) recently proposed a multi-objective data placement algorithm using *hypergraphs*. Hypergraphs offer a powerful representation by presenting a natural way of capturing multi-way relationships. Similar to graph partitioning however, hypergraph partitioning is NP-Hard. To this end, the authors use heuristic partitioning algorithms available in a publicly available tool – PaToH (Catalyurek, 2011), to efficiently partition large hypergraphs. Further, in our previous work (Atrey et al., 2018), we proposed a data placement strategy using spectral clustering on hypergraphs, which was made efficient by performing eigen decomposition using low rank approximations of the hypergraph matrix.

Despite several interesting proposals to address the data placement problem for data-intensive services, they cannot be considered *scalable* in true sense. Specifically, the state-of-the-art methods proposed by Yu et al. (Yu and Pan, 2017) and Atrey et al. (Atrey et al., 2018), employ heuristics and approximate methods respectively to tackle the scalability challenge while maintaining good empirical efficacy and efficiency. However, the use of *distributed algorithms* and recent big data frameworks has not been explored in this field. The importance of distribution stems from the fact that workloads of today seldom fit into the memory of a single machine. To bridge this gap, in this article, we propose a unified framework, *SpeCH*, to *scalably* solve the problem of data placement for data-intensive services in geo-distributed clouds. Under the SpeCH framework, we propose two algorithms, namely – *SpectralApprox* and *SpectralDist*. The former uses fast approximate eigen decomposition methods with bounded quality guarantees thereby being up to *10 times* more efficient when compared to the state-of-the-art, while the latter leverages in-memory distribution offered by Apache Spark to address the scalability challenge while portraying high empirical efficacy by being up to *50% better* on the evaluated metrics. In sum, SpeCH facilitates achieving a *better efficacy-efficiency trade-off*. Key contributions of this work are as follows:

- We study the data placement problem in a challenging and close to real-world setting of *data-intensive services in geo-distributed data-centers* (Sec. 3), where traditional methods of hash based partitioning that are prominent in systems like Hadoop and Spark do not perform well.

- We propose a novel framework, SpeCH, which offers two scalable algorithms (SpectralApprox and SpectralDist) to solve the data placement problem for geo-distributed data-intensive services through *Spectral Clustering on Hypergraphs* (Secs. 4 and 5). While SpectralApprox provides good quality approximations and superior efficiency running on a single machine, SpectralDist portrays superior quality and the capability to scale to very large workloads that cannot fit in a single machine.
- Through experiments on a real-world trace-based social network dataset (Secs. 6 and 7), we show that the proposed spectral clustering algorithms are scalable and effective.

## 2. Related Work

The data placement problem has been studied extensively in the literature spanning a wide-variety of research areas, both from the perspective of execution environments: ranging from distributed systems (Chervenak et al., 2007; Golab et al., 2014) to cloud computing environments (Li et al., 2018; Ferdaus et al., 2017; Yu et al., 2012; Guo and Wang, 2013); application areas: scientific workflows (Yuan et al., 2010; Ebrahimi et al., 2015; Liu and Datta, 2011), online social network services (Jiao et al., 2014; Han et al., 2017), location aware data placement for geo-distributed cloud services (Yu and Pan, 2017; Zhang et al., 2016; Yu and Pan, 2015, 2016; Agarwal et al., 2010), and many more. Here, we provide an overview of the existing research that overlaps with the work presented in this article.

Research on data placement in geo-distributed clouds has increasingly gained popularity over the years. Owing to multiple challenges as discussed in Sec. 1, specialized solutions have been devised to solve this problem. The biggest challenge for data placement algorithms in such scenarios is that data migrations from one location to another are significantly more expensive when compared to other real-world scenarios like grids, clusters, or private clouds, which are typically more geographically confined when compared to the spread of public cloud data-centers.

Agarwal et al. proposed a system Volley (Agarwal et al., 2010), to perform automatic data placement in geographically distributed data-centers. The proposed system possesses the capability to capture data-item – data-center associations, however, it lacked the capability for handling data-item – data-item associations. Rochman et al. (Rochman et al., 2013) design robust data placement algorithms to ensure that a large fraction of region specific requests are served at a lower cost, while managing the highly dynamic nature of user requests. In (Huguenin et al., 2012), a user generated content (UGC) dataset (with more than 650000 YouTube videos) is used to show the correlation between the content locality and geographic locality, thereby highlighting the importance of data-item – data-center associations. Zhang et al. (Zhang et al., 2016) propose an integer programming based data placement algorithm capable of minimizing the data communication cost while honoring the storage capacity of geo-distributed data-centers as well.

Researchers have also focused on different aspects of geo-distributed data placement, such as placement in multi-clouds and the design of specialized replication strategies. Jiao et al. (Jiao et al., 2014) formulate a multi-objective social network aware optimization problem that performs data placement by building a model framework, which takes multiple objectives like minimizing the carbon footprint, inter-cloud traffic etc. into consideration. Further, Han et al. (Han et al., 2017) introduce an adaptive data placement algorithm for social network services in a multicloud environment, which adapts to the changing data traffic for performing intelligent data migration decisions. Kayaaslan et al. proposed a document replication framework (Kayaaslan et al., 2013) to deal with the scalability issues, where, documents are replicated on data-centers based on region specific user interests. Shankarnarayanan et al. propose replication strategies (Shankaranarayanan et al., 2014) for a class of cloud storage systems denoted as quorum-based systems (viz. Cassandra, Dynamo) capable of solving the data placement problem cognizant of various location-aware metrics like location of geo-distributed data-centers, inter-datacenter communication costs etc.

Of late, literature has seen an increased use of (hyper)graph-based techniques for data placement. Saha et al. prove that the data placement problem can be reduced to the well known graph partitioning problem and propose an integer linear programming solution (Golab et al., 2014). The authors also propose two heuristics to reduce the data communication cost for data-intensive scientific workflows and join-intensive queries in distributed systems. In (Quamar et al., 2013), the authors study the problem of Online Transaction Processing (OLTP) workloads in cloud computing environments, and propose a scalable workload aware data partitioning and data placement approach called SWORD to reduce the partitioning overhead. SWORD utilizes a two phase approach: in the first phase a workload is modeled as a hypergraph which is further compressed by using hash partitioning. Later the compressed hypergraph is partitioned to retrieve the placement output. Hypergraph based partitioning solutions (Catalyurek et al., 2007) have also been used in grid and distributed computing environments previously.

Having said that, the current state-of-the-art for data placement in geo-distributed clouds is also comprised of (hyper)graph-based techniques. Yu et al. (Yu and Pan, 2015, 2016, 2017) propose data placement strategies using hypergraph modeling and publicly available partitioning heuristics (Catalyurek, 2011) for data-intensive services. While hypergraph-based modeling facilitates capturing of both data-item – data-item and data-item – data-center associations, the hypergraph partitioning heuristics available in (Catalyurek, 2011) facilitate these techniques to scale to large datasets. Recently, Atrey et al. (Atrey et al., 2018) presented a spectral clustering algorithm (which has been *comprehensively* extended as a unified framework *SpeCH* in this article) that employed the use of low-rank approximations of the hypergraph laplacian to obtain superior efficiency and scalability while retaining the

Table 1: **Summary of notations used.**

| Item | Definition |
|---|---|
| $V$ | The set of users in the social network $V = \{v_1, v_2, \ldots, v_n\}$. |
| $E$ | The set of edges in the social network $\forall e = (v_x, v_y) \in E$. |
| $\mathbf{Adj}(v)$ | The set of friends of the user $v \mid v \in V$. |
| $\mathcal{D}$ | The set of data-items $\mathcal{D} = \{d(v_1), d(v_2), \ldots, d(v_n)\}$. |
| $\mathcal{L}$ | The set of data-centers and their locations $\mathcal{L} = \{L_1, L_2, \ldots, L_l\}$. |
| $\mathcal{R}$ | The set of request patterns $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$. |
| $\mathcal{C}$ | The set of user check-ins $\mathcal{C} = \{C_k = (R_i, L_j) \mid \exists R_i \in \mathcal{R}, L_j \in \mathcal{L}\}$. |
| $\Pi$ | The hypergraph incidence matrix. |
| $\mathcal{W}_\Pi$ | The hyperedge weight matrix. |
| $\Phi$ | The desired data-center storage distribution. |
| $\mathcal{P}(\mathcal{D})$ | A partition on the set of data-items $\mathcal{D}$. |
| $\Gamma(L_j)$ | Cost (per unit) of outgoing traffic from data-center $L_j$. |
| $\kappa(L_j, L_{j'})$ | Inter data-center latency (directed) between $L_j$ and $L_{j'}$. |
| $\mathcal{S}(L_j)$ | Storage cost (per unit) of data-center $L_j$. |
| $\mathcal{N}(R_i)$ | Average number of data-centers accessed by request $R_i$. |

same efficacy as portrayed by (Yu and Pan, 2017).

Despite wide-spread research for data placement in geo-distributed clouds, to the best of our knowledge, none of the existing state-of-the-art methods are capable of gracefully scaling to large datasets while retaining high quality/efficacy. Specifically, as discussed in Sec. 1 the existing works either incorporate the use of heuristics (Yu and Pan, 2017) or approximations (Atrey et al., 2018) to achieve high efficiency and scalability while compromising on placement efficacy. To this end, the research presented in this article proposes a unified framework – *SpeCH*, capable of scalably performing data placement of data-intensive services into geographically distributed clouds, through a novel paradigm of data partitioning using Spectral Clustering on Hypergraphs. Specifically, SpeCH exposes two algorithms: SpectralApprox, which uses fast approximate eigen decomposition methods with bounded quality guarantees to achieve speed-up of up to 10 times over the state-of-the-art; and SpectralDist, which leverages in-memory distribution to address the scalability challenge while portraying high empirical efficacy by being up to 50% better on the evaluated metrics. In other words, SpeCH, along with its two algorithms SpectralApprox and SpectralDist, provides a holistic and unified solution to the data placement problem for data-intensive services.

## 3. Problem Statement

In this section, we first introduce some basic concepts of data placement, which is followed by a formal description of the data placement problem for data-intensive services in geo-distributed data-centers. Table 1 summarizes the notations used in the rest of the article.

**Definition 1** (Data-items ($\mathcal{D}$).)**.** *A data-item is defined as an atomic unit of data storage and transfer in the context of data placement. $\mathcal{D}$ denotes the set of data-items, where $|\mathcal{D}| = n$.*

**Definition 2** (Data-centers ($\mathcal{L}$).)**.** *A data-center constitutes a set of resources to store the data-items and perform different computational tasks on the stored data-items. Each data-center is hosted at a location $L_j$ and is denoted using the set $\mathcal{L}$, where $|\mathcal{L}| = l$.*

Given a scientific workflow management system, the workflow components may be considered as data-items while the computing and data-storage resources could be considered as data-centers. Similarly, for relational database management systems, database tables can be considered as data-items and the database server as a data-center.

Usually for large scale systems, the data-items are distributed across data-centers and might require migrations from one data-center to another for enabling various tasks to execute properly. More specifically, two database tables that are required to be joined for completing a specific task might be stored on two different data-centers, and thus, either table needs to be migrated to ensure proper task execution. The data-items that are potential candidates for migrations constitute a data-request, which is formally defined as follows.

**Definition 3** (Data-request Patterns ($\mathcal{R}$).)**.** *A data-request pattern $R \in \mathcal{R}$ is comprised of a set of data-items $D \subseteq \mathcal{D}$ that are required to be present together in a single data-center $L_j$ for a given task to be executed. The data-items ($d_i \in D$) that are not stored in $L_j$ are communicated from the data-centers in which they are stored to $L_j$. The set of data-request patterns denoted as $\mathcal{R}$ represent the system workload.*

Given a set of data-items $\mathcal{D}$, a set of data-centers $\mathcal{L}$, and a set of data-request patterns $\mathcal{R}$, the objective of the classical data placement problem is to intelligently place the data-items across data-centers so as to minimize the overall communication cost resulting from migration[1] of data-items corresponding to different data-requests.

Since our focus is on data placement of data-intensive services, which is being studied in this article in the context of location based OSN services, we next describe concepts specific to OSN services and their relation to the problem studied in this article.

A location based online social network (Fig. 1) possesses two aspects: (1) a social network connecting users with their friends, and (2) a capability for the users to register *check-ins* at potentially different locations across the globe.

**Definition 4** (Social Network. ($G(V,E)$))**.** *A social network with n individuals and m social ties can be denoted as a graph $G(V,E)$, where $V$ is the set of nodes representing the users of the social network, $|V| = n$, and $E$ is the set of edges (representing friend relationships) between any two nodes, $E \subseteq V \times V$, $|E| = m$.*

The set $\mathcal{D}$ contains $n$ data-items corresponding to each user $v \in V$ of the social network, where the data-item for a user $v$ is denoted as $d(v)$.

Moving ahead, a check-in depicts a social network user visiting any location in the world. Each user check-in is composed of two parts: (1) a data-request pattern, and (2) a location. As discussed in Sec. 1, the data-request corresponding to a

---

[1] Migration of data-items may involve additional overheads such as data-item retrieval delays, packet loss etc. For the sake of brevity, the focus of this article is on minimizing the communication cost alone, however, both the SpeCH framework and its associated data-placement algorithms are generic, and not restricted in their scope based on this assumption.

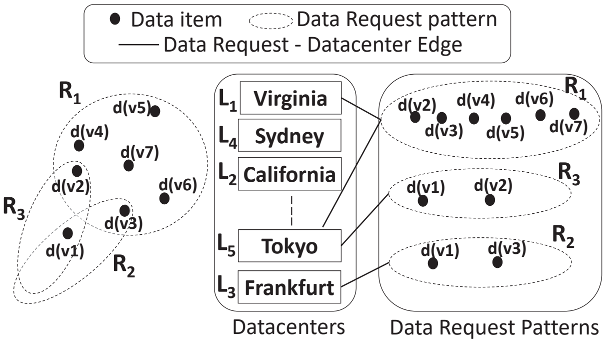


Figure 2: **Mapping of different requests to geo-distributed data-centers.**

user check-in requires retrieval of the data-items of her friends. Thus, for a check-in by a user $v$ the data-request pattern is represented as $R(v) = \{d(u) \mid u \in \mathbf{Adj}(v)\}$. Moreover, the location of a user check-in is decided as the data-center location closest (in distance) to the actual physical location of the user check-in. Given this information, a check-in is formally defined as follows:

**Definition 5** (Check-ins. ($\mathcal{C}$))**.** *A check-in is a tuple* $C_k = (R(v), L_j) \in \mathcal{C}$ *consisting a data-request pattern* $R(v) \in \mathcal{R}$ *triggered by* $v$ *and a location* $L_j \in \mathcal{L}$ *of a data-center capable of serving user requests.*

In other words, the check-in $C_k$ by a user $v$ at a location $L_j$ signifies a request for the data-items contained in $R(v)$ triggered from the data-center located at $L_j$. Note that a user can register multiple check-ins at the same location, and to better capture data-item – data-item and data-item – data-center associations, each such check-in is treated as different from the other. For example, two data-items $d(v_2)$ and $d(v_3)$ that are requested together seven times possess a stronger data-item – data-item association when compared to data-items $d(v_1)$ and $d(v_2)$ that co-exist in data-request patterns just twice. Similarly, the data-items in the data-request pattern of a user $v_5$ who visited Sydney five times possess a stronger data-item – data-center association with Sydney when compared to that of Frankfurt, which the user visited only once. To capture this, for each check-in by the user $v_5$ at Sydney there would be 5 different check-ins denoted as $C_1, \ldots, C_5$, each composed of the data-request pattern $R(v_5)$ and the location $L_4$ =Sydney. Moreover, this also substantiates the reason behind not indexing each user check-in uniquely using data-request patterns $R$ and locations $L_j$.

Building upon the example portrayed in Fig. 1 (Sec. 1), Fig. 2 showcases the data-request patterns corresponding to the check-ins registered by users $v_1, v_2$, and $v_3$. Let us denote the data-request patterns as $R(v_1), R(v_2)$, and $R(v_3)$ respectively, where $R(v_1) = \{d(v_2), d(v_3), d(v_4), d(v_5), d(v_6), d(v_7)\}$, $R(v_2) = \{d(v_1), d(v_3)\}$, and $R(v_3) = \{d(v_1), d(v_2)\}$. Let us also label the data-center locations as $L_1 = Virginia, L_2 = California, L_3 = Frankfurt, L_4 = Sydney$, and $L_5 = Tokyo$. Recall that the user $v_1$ registered two check-ins: one in Virginia and the other in Tokyo. Similarly, $v_2$ registered a check-in in Frankfurt, while $v_3$ checked-in in Tokyo. Thus, in total there are four check-ins: two ($C_1$ and $C_2$) for the user $v_1$, and one each ($C_3$ and $C_4$ respectively) for users $v_2$ and $v_3$, where $C_1 = (R(v_1), L_1)$, $C_2 = (R(v_1), L_5)$, $C_3 = (R(v_2), L_3)$, and $C_4 = (R(v_3), L_5)$.

Having defined the basic concepts and their notations, we formally define the problem as:

**Problem 1.** *Given a set of n data-items* $\mathcal{D}$ *corresponding to the set of social network users* $V$, $\rho$ *user check-ins* $C_k = (R(v), L_j) \in \mathcal{C} \mid v \in V, L_j \in \mathcal{L}$ *representing the system workload, each comprising a data-request pattern* $R(v)$ *being originated from a data-center located at* $L_j$*, a set of l data-centers with locations in* $\mathcal{L}$*, with the per unit cost of outgoing traffic from each data-center* $\Gamma(L_j) \mid L_j \in \mathcal{L}$*, the per unit storage cost of each data-center* $\mathcal{S}(L_j) \mid L_j \in \mathcal{L}$*, the inter data-center latency (directed) for each pair of data-centers* $\kappa(L_j, L_{j'}) \mid L_j, L_{j'} \in \mathcal{L}$*, and the average number of data-centers accessed by the data-items requested in each request pattern* $R(v)$ *being* $\mathcal{N}(R(v))$*, perform data placement to minimize the optimization objective* $\mathcal{O}$*, which is defined as the weighted average of* $\Gamma(\cdot), \kappa(\cdot,\cdot), \mathcal{S}(\cdot)$*, and* $\mathcal{N}(\cdot)$*.*

## 4. SpeCH Framework

In this section, we present the SpeCH framework and provide a description of its core components. In this article, our previous work on data placement using spectral clustering (Atrey et al., 2018), which leveraged low-rank approximations (SpectralApprox) of the hypergraph laplacian to scale to large matrices, has been comprehensively extended as a scalable framework called SpeCH. In addition to the SpectralApprox algorithm, we incorporate the use of in-memory distribution offered by Spark to propose SpectralDist, which enables SpeCH to scalably perform data placement on large workloads, thereby also facilitating improvement in the efficiency-efficacy trade-off. We also provide insights about the way in which SpeCH facilitates development of effective data placement strategies. Fig. 3 portrays an architectural overview of the SpeCH framework. The building blocks of this framework are detailed next.

- **Hypergraph Construction (Construct Hypergraph):** uses the check-in history to construct a binary hypergraph incidence matrix, which captures the existence of different types of associations between data-items and data-centers. Two data-items that are requested together in a user check-in result in data-item – data-item association, while a data-item being requested at a data-center location based on a user check-in results in data-item – data-center association.
- **Hyperedge Weighting (Calculate Hyperedge Weights):** uses the check-in history and data-center characteristics to assign weights to the different hyperedges constructed in the previous step. The weights facilitate capturing of the extent of the associations between data-items and data-centers as well as appropriately managing the contribution of each hyperedge in accordance with the considered optimization objective.

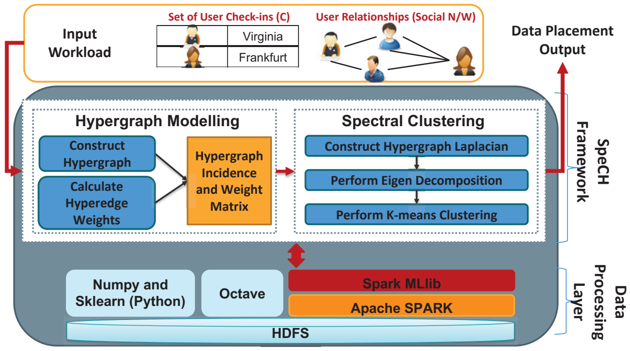

Input Workload
Set of User Check-ins (C)
Virginia
Frankfurt
User Relationships (Social N/W)
Data Placement Output
Hypergraph Modelling
Construct Hypergraph
Calculate Hyperedge Weights
Hypergraph Incidence and Weight Matrix
Spectral Clustering
Construct Hypergraph Laplacian
Perform Eigen Decomposition
Perform K-means Clustering
SpeCH Framework
Numpy and Sklearn (Python)
Octave
Spark MLlib
Apache SPARK
HDFS
Data Processing Layer

Figure 3: **Overview of the SpeCH Framework.**

- **Hypergraph Laplacian (Construct Hypergraph Laplacian):** is required for performing different analytical operations on the hypergraphs. Among multiple other applications(Wikipedia, 2018), it helps construct low dimensional embeddings, which find use in a variety of machine learning applications such as spectral clustering. The mathematical details of this step are provided in Sec. 5.
- **Eigen Decomposition:** is performed to identify the spectra: the eigen-values and eigen-vectors of the hypergraph laplacian constructed in the previous step. The eigen-vectors corresponding to the β smallest eigen-values encode important properties of the hypergraph similarity/affinity matrix, which are useful in spectral clustering. Eigen decomposition can be performed using one of the following tools/frameworks: Octave, Numpy (Python), or MLLib (Apache Spark).
- **K-means clustering:** allows for partitioning the data-items into $k$ data-centers. The eigen-vectors corresponding to the β smallest eigen-values are clustered into $k$ groups, thereby resulting in data placement of the data-items into geographically distributed data-centers. Performing k-means clustering on the eigen-vectors (spectra) of the hypergraph laplacian identified in the previous step is attributed as spectral clustering. K-means clustering can be performed using one of the following tools/frameworks: sk-learn (Python), or MLLib (Apache Spark).

### *4.1. Hypergraph Modeling*

As discussed in Sec. 1, hypergraphs provide the ideal representation to solve the data placement problem in geo-distributed data-centers owing to their capability of capturing multi-way relationships, thereby facilitating modeling of data-item – data-item and data-item – data-center associations (Atrey et al., 2018; Yu and Pan, 2017).

A hypergraph $H(V_H, E_H)$ is a more sophisticated graph construct and a generalization over a graph $G(V,E)$, where (hyper)edges are capable of capturing relationships between several vertices as opposed to just a pair of vertices in graphs. This ability of hyperedges to capture higher order relationships between data points facilitates the hypergraph model to manage both data-item – data-item and data-item – data-center associations.

In the context of our problem statement, every user check-in $C_k \in \mathcal{C}$ consists of a data-center location $L_j \in \mathcal{L}$ and a data-request pattern $R(v) \in \mathcal{R}$. Note that for a check-in by a user $v$, $R(v)$ is a set of data-items corresponding to all the friends of $v$, i.e. $\mathbf{Adj}(v)$, as portrayed by the social network $G(V,E)$. Since a data-request pattern involves data-items corresponding to multiple vertices of $G(V,E)$, hyperedges provide a better way to model data-item – data-item associations by linking/connecting multiple data-items via the same hyperedge. Additionally, hyperedges also facilitate modeling of data-item – data-center associations by connecting a data-item with a data-center location when a data-item $d(v_i)$ is requested from a data-center location $L_j$.

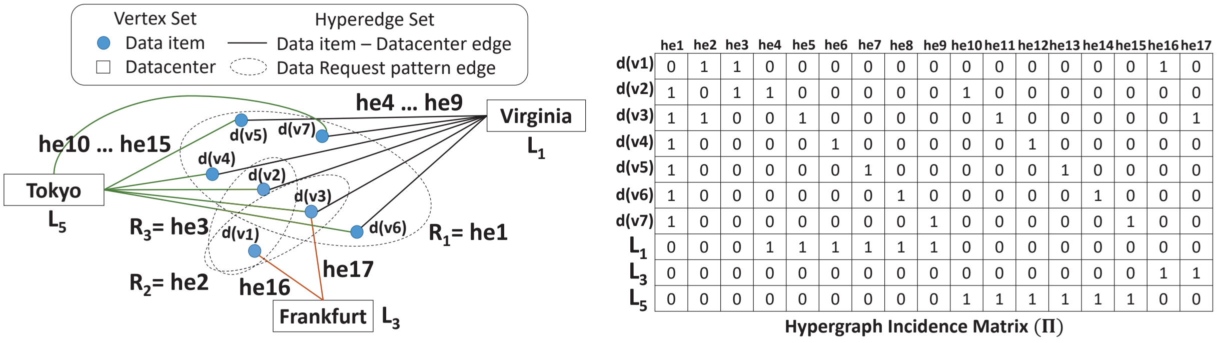
The hypergraph vertex set $V_H$ comprises of all the data-items $\mathcal{D}$ and the data-center locations $\mathcal{L}$. Thus, the number of vertices in the hypergraph are $|V_H| = n' = n + l$. Formally,

$$V_H = \mathcal{D} \cup \mathcal{L} \tag{1}$$

Let $\mathcal{R}_{\mathcal{L}} = \{R_{d(v_i),j} \mid \exists C_k = (R(v), L_j) \in \mathcal{C}, d(v_i) \in R(v), L_j \in \mathcal{L}\}$ denote the set of edges connecting data-items with data-center locations corresponding to all the data-request patterns $R(v) \in \mathcal{R}$ triggered by user check-ins $C_k \in \mathcal{C}$ at data-center locations $L_j \in \mathcal{L}$. The hypergraph edge set $E_H$ consists of hyperedges corresponding to all the data-request patterns $\mathcal{R}$ and all the data-item – data-center location edges $\mathcal{R}_{\mathcal{L}}$. Since the the number of data-items and distinct locations are $n$ and $l$ respectively, the number of data-item – data-center location edges are at most $n.l$. Thus, the number of hyperedges in the hypergraph are $|E_H| = m' = |\mathcal{R}| + n.l$. Formally,

$$E_H = \mathcal{R} \cup \mathcal{R}_{\mathcal{L}} \tag{2}$$

Fig. 4a portrays the hypergraph representation of the data-items and request patterns as presented in Fig. 2. The data-items $\{d(v_1), \ldots, d(v_7)\}$ and the data-center locations $\{L_1, L_3, L_5\}$ constitute the hypergraph vertex set. The hyperedges corresponding to the data-request patterns $\{R_1, R_2, R_3\}$ are labeled as $he1, he2, he3$ respectively and are denoted using a dashed ellipse, while the hyperedges connecting each data-item – data-center location pair are labeled as $he4, \ldots, he17$. Since $v_2, v_3, v_4, v_5, v_6, v_7$ are friends of $v_1$, the data-items $d(v_2), \ldots, d(v_7)$ belonging to the request pattern $R_1$ are connected by the hyperedge $he1$. Similarly, since $v_7$ is a friend of $v_1$, who registered two check-ins: one at Virginia ($L_1$) and the other at Tokyo ($L_5$), the hyperedge $he4 = (d(v_7), L_1)$ and $he10 = (d(v_7), L_5)$ represents the relationship between the data-item $d(v_7)$ and the data-center locations where it was requested, namely – Virginia ($L_1$) and Tokyo ($L_5$). Fig. 4b portrays the incidence matrix Π corresponding to the hypergraph $H$ presented in Fig. 4a. As can be seen, the rows of this matrix are the vertex set of the hypergraph, while the columns are the hyperedges. Moreover, if a vertex participates in a hyperedge then the row corresponding to it contains a 1 in the column corresponding to that hyperedge, and 0 otherwise. Since the hyperedge $he1$ (corresponding to the data-request pattern $R_1$) connects the data-



|  | he1 | he2 | he3 | he4 | he5 | he6 | he7 | he8 | he9 | he10 | he11 | he12 | he13 | he14 | he15 | he16 | he17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| d(v1) | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| d(v2) | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| d(v3) | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| d(v4) | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| d(v5) | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| d(v6) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| d(v7) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $L_1$ | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $L_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| $L_5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

(a) Hypergraph Representation

(b) Hypergraph Incidence Matrix

Figure 4: **Modeling the data-request patterns triggered by user check-ins as a Hypergraph.**

items $d(v_2),\dots,d(v_7)$ the entries in $\Pi$ corresponding to them are filled with 1 while all other entries are 0.

### *4.1.1. Calculating Hyperedge Weights*

Having constructed the hypergraph $H(V_H,E_H)$ and discussed its representation using a hypergraph incidence matrix $\Pi$, we next discuss ways to assign weights to hyperedges. There are two major types of hyperedges constructed in the representation discussed above: (1) data-request pattern hyperedges, and (2) Data-item – data-center hyperedges, and both of them capture different properties required by a data placement algorithm in geo-distributed data-centers. The weight of a data-request pattern hyperedge $W_{\mathcal{R}}$ is set using the request rate of that pattern, which is defined as the number of times a data-request pattern is triggered by a user check-in. The weight $W_{\mathcal{R}}$ facilitates prioritization of data-items that are usually requested together, to be placed together by the data placement algorithm, thereby helping optimize (minimize) $\mathcal{N}(R(v))$: the average number of data-centers accessed by a data-request pattern $R(v)$. On the other hand, the weights $(W^{\kappa}_{\mathcal{R}_{\mathcal{L}}},W^{\mathcal{S}}_{\mathcal{R}_{\mathcal{L}}},W^{\Gamma}_{\mathcal{R}_{\mathcal{L}}})$ corresponding to data-item – data-center hyperedges $(\mathcal{R}_{\mathcal{L}})$ facilitate minimization of inter data-center latency $\kappa(L_j,L_{j'})$, storage cost $\mathcal{S}(L_j)$, and cost of outgoing traffic $\Gamma(L_j)$ respectively, by giving priority to placing data-items at data-center locations from where they have been requested more frequently.

The resultant hyperedge weight matrix $\mathcal{W}_{\Pi}$ of $\Pi$ is a diagonal matrix of size $m'\times m'$, which is defined as the weighted sum of $W_{\mathcal{R}}$, $W^{\kappa}_{\mathcal{R}_{\mathcal{L}}}$, $W^{\mathcal{S}}_{\mathcal{R}_{\mathcal{L}}}$, and $W^{\Gamma}_{\mathcal{R}_{\mathcal{L}}}$. Mathematically,

$$\mathcal{W}_{\Pi}=\mathbb{W}\cdot(W_{\mathcal{R}},W^{\kappa}_{\mathcal{R}_{\mathcal{L}}},W^{\mathcal{S}}_{\mathcal{R}_{\mathcal{L}}},W^{\Gamma}_{\mathcal{R}_{\mathcal{L}}}). \tag{3}$$

where, $\mathbb{W}$ is the weight vector for deciding the priorities of the previously discussed hyperedge weighting strategies. The effect of different values of $\mathbb{W}$ on the data placement output is analyzed in detail in Sec. 7.

### *4.2. Spectral Clustering*

Spectral methods have been shown to be promising in a plethora of machine learning research areas: image segmentation (Shi and Malik, 2000; Meila and Shi, 2001), data clustering (Ng et al., 2001), web search (Gibson et al., 1998), and information retrieval (Deerwester et al., 1990). The power of these algorithms is that they possess both sound mathematical properties and strong empirical prowess. They are named spectral algorithms as they use the information manifested within the spectra (both eigen-values and eigen-vectors) of a similarity/affinity matrix. More fundamentally, for a graph $G$ represented using a similarity matrix containing node-node similarities, these methods use the spectra of the graph laplacian to understand the intrinsic data properties like structure, connectivity etc. Interestingly, the laplacian for hypergraph was derived in (Zhou et al., 2006), where it is shown to be analogous to the simple graph laplacian. This result facilitates application of spectral methods on hypergraphs.

Having discussed about the importance of spectral methods, we next describe the steps to perform spectral clustering on hypergraphs. The first step in spectral clustering on hypergraphs is to construct the hypergraph laplacian matrix $L_H$. The next step is to perform eigen-decomposition of the hypergraph laplacian matrix $L_H$, in order to identify its spectra: the eigen-values and eigen-vectors. The last step involves performing $k$-means clustering on the eigen-vectors $U$ of the hypergraph laplacian matrix $L_H$. The three step process: (1) Hypergraph Laplacian construction, (2) Eigen decomposition of the hypergraph laplacian, and (3) k-means clustering on the eigen-vectors, with added extensions and modifications constitutes the proposed scalable spectral clustering algorithm.

## 5. Data Placement Algorithms

To effectively solve the data placement problem, we propose a technique as outlined in Algorithm 1. Given the set of data items $\mathcal{D}$, and the set of user check-ins $\mathcal{C}$ comprising the set of data-request patterns $\mathcal{R}$ and their locations $\mathcal{L}$, we first construct a hypergraph (line 1). With the hypergraph incidence matrix $\Pi$ constructed, next, we partition the set of data-items $\mathcal{D}$ into $l$ parts corresponding to the $\mathcal{L}$ data-centers according to the desired storage distribution $\Phi$, using the proposed scalable spectral clustering algorithms (line 2).

Note that hypergraph construction was discussed in detail in Sec. 4.1. In this section, we provide a detailed and formal

**Algorithm 1** Data Placement Algorithm

```
Input: $\mathcal{D}$, $\mathcal{C}$, $\mathcal{R}$, $\mathcal{L}$, $G(V,E)$, $\Phi$
Output: Partitioning of the set of data-items $\mathcal{P}(\mathcal{D})$ into $l$ data-centers
(Π, $\mathcal{W}_\Pi$) ← ConstructHypergraph($\mathcal{D}, \mathcal{C}, G(V,E)$)
$\mathcal{P}(\mathcal{D})$ ← SpectralClustering($\Pi, \mathcal{W}_\Pi, l, \Phi$)
return $\mathcal{P}(\mathcal{D})$
```

description of spectral clustering, which is followed by a description of both the proposed algorithms – *SpectralApprox* and *SpectralDist*.

As stated in Sec. 4, the first step in spectral clustering is to construct the hypergraph laplacian matrix $L_H$, which is mathematically defined as follows.

The output of the hypergraph construction step is a $n' \times m'$ dimensional hypergraph incidence matrix $\Pi$ and a $m' \times m'$ dimensional diagonal hyperedge weight matrix $\mathcal{W}_\Pi$. As discussed previously in Sec. 4.1, the hypergraph incidence matrix $\Pi$ possesses $m'$ hyperedges, where each hyperedge is a $n'$-dimensional binary vector, which is formally defined as:

$$\Pi = [he_1, he_2, \ldots, he_{m'}]. \tag{4}$$

$$\forall i \in m', he_i = [he_{1,i}, he_{2,i}, \ldots, he_{n',i}]. \tag{5}$$

An entry $he_{j,i} = 1$ indicates that the $j^{th}$ vertex in the hypergraph vertex set is participating in the $i^{th}$ hyperedge, while $he_{j,i} = 0$ indicates otherwise.

Further, the hyperedge weight matrix is defined as:

$$\mathcal{W}_\Pi = diag([W_1, W_2, \ldots, W_{m'}]) \tag{6}$$

where, $diag(\cdot)$ denotes a diagonal matrix and $W_1, \ldots, W_{m'}$ are calculated as described in Eq. 3.

Constructing the hypergraph laplacian $L_H$ requires two additional operations on the hypergraph incidence matrix $\Pi$. We compute two diagonal matrices – the vertex degree matrix ($D_{v\Pi}$) and the hyperedge degree matrix ($D_{he\Pi}$) of dimensionality $n' \times n'$ and $m' \times m'$ respectively. The vertex degree matrix captures the number of hyperedges each vertex of the hypergraph is a part of, while the hyperedge degree matrix measures the number of vertices contained in each hyperedge. Mathematically,

$$D_{v\Pi} = diag(\sum \Pi). \tag{7}$$

$$D_{he\Pi} = diag(\sum \Pi^T). \tag{8}$$

where, $\sum X$ represents the row-wise sum of the input matrix $X$ and $X^T$ represents the transpose of the matrix $X$.

Note that similar to the graph laplacian (Ng et al., 2001; Wikipedia, 2018), the hypergraph laplacian was defined in (Zhou et al., 2006). With this, the hypergraph laplacian $L_H$ is mathematically defined as:

$$L_H = I - (D_{v\Pi}^{-1/2} \cdot \Pi \cdot \mathcal{W}_\Pi \cdot D_{he\Pi}^{-1} \cdot \Pi^T \cdot D_{v\Pi}^{-1/2}) \tag{9}$$

where, $I$ is a $n' \times n'$ identity matrix, $D_v$ is a $n' \times n'$ diagonal vertex degree matrix, $D_{he}$ is a $m' \times m'$ diagonal hyperedge degree matrix, and $\mathcal{W}_\Pi$ is a $m' \times m'$ diagonal hyperedge weight matrix. Thus, $L_H$ becomes a $n' \times n'$ matrix.

The next step is to perform eigen-decomposition of the hypergraph laplacian matrix $L_H$, in order to identify its spectra: the eigen-values and eigen-vectors. The eigen-decomposition of $L_H$ is written as:

$$L_H = U \Lambda V \tag{10}$$

where,

$$U = [u_1, \ldots, u_{n'}]. \tag{11}$$

$$\Lambda = diag(\lambda_1, \ldots, \lambda_{n'}). \tag{12}$$

$U$ is a $n' \times n'$ matrix and $\Lambda$ is a diagonal $n' \times n'$ matrix formed by the eigen-vectors and eigen-values of $L_H$ respectively. Since $L_H$ is a square symmetric matrix, $V = U^T$. The eigen-vectors $U$ and eigen-values $\Lambda$ collectively define the spectra for the hypergraph laplacian $L_H$.

As discussed in Sec. 1, despite their strong theoretical and mathematical properties, spectral methods are usually not scalable. This is mainly due to the complex operation of performing a full *eigen decomposition* of a large matrix, which is cubic $O({n'}^3)$ in the dimensionality of $L_H$ in the worst case. To this end, we employ the use of two different paradigms of algorithms: (1) approximation (SpectralApprox) and (2) distributed (SpectralDist), for scaling-up this operation to large matrices. Note that it is shown in the literature that it is not required to use all the eigen-vectors (Zhou et al., 2006) and thus, in practice one may work with just a small fraction of $n'$. To this end, in this article we work with a partial decomposition of $L_H$, thereby restricting the eigen decomposition to just calculate the $\beta$ smallest eigen vectors of $L_H$.

### 5.1. SpectralApprox

Alg. 2 presents the pseudo-code for *SpectralApprox*, the proposed approximate spectral clustering algorithm. As described previously, the first step is to compute the hypergraph laplacian $L_H$ of the hypergraph incidence matrix $\Pi$ (lines 1–2). To improve the efficiency of the eigen decomposition step for large matrices, we divide the computation into two phases: (1) identifying a small orthonormal matrix $Z$ such that $L_H \approx ZZ^T L_H$ (line 3), and (2) use $Z$ to compute the eigen decomposition of $L_H$ (lines 4–6).

We employ the use of random sampling methods to identify $Z$ (Halko et al., 2011). Specifically, for finding the $\beta$ smallest eigen-values of the $n' \times n'$ hypergraph laplacian matrix $L_H$ we randomly sample a $n' \times 2\beta$ gaussian matrix $\Omega$. Using a power-iteration exponent $p$ and $\Omega$, we construct a matrix $X$, which is defined as:

$$X = (L_H {L_H}^T)^p L_H \Omega \tag{13}$$

Using $X$ we identify $Z$ as an orthonormal matrix satisfying $X \approx ZZ^T X$. Once we have identified a sufficiently small $Z$, performing eigen decomposition becomes a simple and efficient task. Given $Z$, we construct $Y = Z^T L_H$, and obtain the eigen decomposition of $Y$ as:

$$Y = \tilde{U} \Lambda V \tag{14}$$

**Algorithm 2** SpectralApprox Algorithm

```
Input: $\Pi$, $\mathcal{W}_\Pi$, l, $\Phi$
Output: Partitioning of the hypergraph vertex set $\mathcal{P}(V_H)$ into $l$ clusters
$D_v(\Pi) \leftarrow diag(\sum \Pi)$; $D_{he}(\Pi) \leftarrow diag(\sum \Pi^T)$
Compute hypergraph laplacian $L_H$ as described in Eq. 9
Identify a small orthonormal matrix $Z$ satisfying $L_H \approx ZZ^T L_H$
Obtain $Y \leftarrow Z^T L_H$
Perform eigen decomposition of $Y$: $Y \rightarrow \tilde{U}\Lambda V$
$U \leftarrow Z\tilde{U}$
Partition $U$ into $l$ clusters using k-means: $\mathcal{P}(V_H) \leftarrow KMeans(U, l, \Phi)$
return $\mathcal{P}(V_H)$
```

Finally, the $\beta$ smallest eigen values $\lambda_1, \ldots, \lambda_\beta$ and the corresponding eigen vectors $U$ of $L_H$ are computed as $U = Z\tilde{U}$. While the traditional eigen decomposition algorithm would require $O(n'^2\beta)$ time to compute the $\beta$ smallest eigen values and vectors, the time complexity of the randomized approach is $O(\beta^2 n')$, which is significantly faster when compared to the former. As will be explained later in Sec. 7, the randomized approach facilitates SpectralApprox to achieve superior efficiency and scalability, without any noticeable loss in the efficacy.

After identifying $U$, we perform $k$-means clustering to partition it into $l$ clusters, as we know the number of data-center locations a priori, which is $l = |\mathcal{L}|$. This operation partitions the vertex set $V_H$ of the hypergraph, and consequently the data-items $\mathcal{D}$, into $l$ different sets, thereby forming $\mathcal{P}(V_H)$ ($\mathcal{P}(\mathcal{D})$), which is used as the placement decision recommended by the proposed data placement algorithm.

### 5.2. *SpectralDist*

Alg. 3 presents the pseudo-code for the distributed spectral clustering algorithm – *SpectralDist*. While SpectralApprox scales-up eigen decomposition elegantly to large matrices, it requires that the hypergraph incidence matrix $\Pi$ and the hypergraph laplacian $L_H$ fit into memory, thereby rendering SpectralApprox impractical for big data applications. SpectralDist is capable of handling such cases by distributing the data and performing each matrix operation on a small sub-partition of the data rather than working with the complete data at one go. Moreover, in our current scenario, as opposed to SpectralApprox which uses approximate methods, SpectralDist performs spectral clustering by using exact eigen decomposition thereby not sacrificing on efficacy at all.

Specifically, SpectralDist leverages the in-memory distribution offered by Apache Spark to elegantly scale the matrix operations for large matrices. The hypergraph incidence matrix $\Pi$ is distributed across $\alpha$ machines in the compute cluster and the distributed linear algebra operations provided by Apache Spark are used to efficiently construct the hypergraph laplacian $L_H$ (lines 1–3). Later, we distribute the matrix and vector multiplication operations in the traditional (sequential) eigen decomposition algorithm to improve its efficiency (line 4). Finally, a distributed version of the $k$-means clustering algorithm is used to efficiently cluster the eigen vectors identified in the previous step (line 5), thereby producing the data placement output.

**Algorithm 3** SpectralDist Algorithm

```
Input: $\Pi$, $\mathcal{W}_\Pi$, l, $\Phi$, $\alpha$
Output: Partitioning of the hypergraph vertex set $\mathcal{P}(V_H)$ into $l$ clusters
1: Distribute $\Pi$ into $\alpha$ machines
2: Compute sum of each sub-part of $\Pi$ on the $\alpha$ machines to obtain $D_v(\Pi) \leftarrow diag(\sum \Pi)$; $D_{he}(\Pi) \leftarrow diag(\sum \Pi^T)$
3: Compute hypergraph laplacian $L_H$ per Eq. 9 using distributed matrix and vector multiplications
4: Distribute the matrix and vector multiplications for computing eigen decomposition: $DistEig(L_H, \alpha) \rightarrow U\Lambda V$
5: Partition $U$ into $l$ clusters using distributed k-means: $\mathcal{P}(V_H) \leftarrow DistKMeans(U, l, \Phi, \alpha)$
6: return $\mathcal{P}(V_H)$
```

Note that the $k$-means clustering algorithm used in both SpectralApprox and SpectralDist possesses the following modifications. First, to ensure *load balancing* we modify the objective function of $k$-means to honor the desired storage distribution $\Phi$, which provides information about the expected capacity of each data-center location. Second, we employ the use of $k$-means++ initialization as proposed in (Arthur and Vassilvitskii, 2007) and parallelization to scale-up the clustering algorithm to very large datasets.

### 5.3. *Handling Replication*

Since replication may be important in real-world settings for ensuring fault-tolerance and load-balancing, in this section we discuss an extension over SpeCH to allow for the scenarios with replication. To this end, we use either of the proposed data placement algorithms[2] (SpectralApprox or SpectralDist) to obtain a placement without replication, however, we change the capacity of each data-center from $s_{L_j} \sim \Phi$ to $s_{L_j}/r$, where $r$ is the desired replication factor. Reducing the capacity of each data-center by a factor of $r$ ensures that each data-item is equally-likely to be replicated $r$ times. Having obtained an initial placement, we execute the proposed data placement algorithm for an additional $r-1$ rounds (a total of $r$ rounds) with a small change in the ordering of data-centers. Specifically, let $\mathcal{L} = L_1, L_2, \ldots, L_l$ represent an initial ordering of the data-centers, for each round we obtain a different permutation of $\mathcal{L}$ and then perform placement using the proposed data placement algorithms. Executing the data placement algorithms with different permutations of $\mathcal{L}$ facilitates data-items to be assigned to different data-centers in each round, thus, ensuring proper replication. Specifically, this procedure allows the same data-item to be stored (in the expected sense) on $r$ data-centers, thereby meeting the desired replication factor. Note that some items might be placed multiple times on the same data-center (of course, in that case we only keep a single copy), and hence, each data-item would be replicated at most $r$ times. With this, the SpeCH framework and its associated algorithms are extended to handle scenarios where replication is allowed as well.

---

[2] Note that the proposed extension works with both SpectralApprox and SpectralDist.

Table 2: **(a) Traffic and Storage Costs, and (b) Inter data-center Latency based on Geo-distributed Amazon Clouds.**

| **Region** | **Storage ($/GB-month)** | **Outgoing Traffic ($/GB)** |
|---|---|---|
| **Virginia** | 0.023 | 0.02 |
| **California** | 0.026 | 0.02 |
| **Oregon** | 0.023 | 0.02 |
| **Ireland** | 0.023 | 0.02 |
| **Frankfurt** | 0.025 | 0.02 |
| **Singapore** | 0.025 | 0.02 |
| **Tokyo** | 0.025 | 0.09 |
| **Sydney** | 0.025 | 0.14 |
| **Sao Paulo** | 0.041 | 0.16 |

(a) Costs (in $)

| **Region** | **Virginia** | **California** | **Oregon** | **Ireland** | **Frankfurt** | **Singapore** | **Tokyo** | **Sydney** | **Sao-Paulo** |
|---|---|---|---|---|---|---|---|---|---|
| **Virginia** | 0.0 | 72.738 | 86.981 | 80.546 | 88.657 | 216.719 | 145.255 | 229.972 | 119.531 |
| **California** | 71.632 | 0.0 | 19.464 | 153.202 | 166.609 | 174.010 | 102.504 | 157.463 | 192.670 |
| **Oregon** | 88.683 | 19.204 | 0.0 | 136.979 | 159.523 | 161.367 | 89.095 | 162.175 | 182.716 |
| **Ireland** | 80.524 | 153.220 | 136.976 | 0.0 | 19.560 | 239.023 | 212.388 | 309.562 | 191.292 |
| **Frankfurt** | 88.624 | 166.590 | 159.542 | 19.533 | 0.0 | 325.934 | 236.537 | 323.483 | 194.905 |
| **Singapore** | 216.680 | 173.946 | 161.423 | 238.130 | 325.918 | 0.0 | 73.807 | 175.328 | 328.080 |
| **Tokyo** | 145.261 | 102.523 | 89.157 | 212.388 | 236.558 | 73.785 | 0.0 | 103.907 | 256.763 |
| **Sydney** | 229.748 | 157.843 | 161.932 | 309.562 | 323.152 | 175.355 | 103.900 | 0.0 | 322.494 |
| **Sao Paulo** | 119.542 | 192.700 | 181.665 | 191.559 | 194.900 | 327.924 | 256.665 | 322.523 | 0.0 |

(b) Latency (in ms)

## 6. Evaluation Setup

### *6.1. Geo-distributed data-centers*

To simulate a real-world geo-distributed cloud environment, we employ the use of $l = 9$ geo-distributed data-centers based on the regions provided by AWS global infrastructure (Amazon, 2017). Note that the AWS infrastructure evolves continuously and for the sake of standardization and reproducible comparison with previous work (Yu and Pan, 2015), we only chose the 9 oldest and prominent regions, namely: Virginia, California, Oregon, Ireland, Frankfurt, Singapore, Tokyo, Sydney, and Sao Paulo, for our experimental setup. Our experimental setup closely mirrors the actual AWS setup, as the costs involved for storage and outgoing traffic as indicated in Table 2a are as advertised by Amazon. Moreover, the inter-datacenter latencies(Wang, 2016) are also measured by the packet transfer latency between the chosen regions using the Linux *ping* command. Table 2b presents the average inter-datacenter latency values (in ms) between the 9 chosen data-center regions. It is evident from the values portrayed in Table 2 that the properties exhibited by data-centers vary significantly with the region, and hence, any data placement strategy should incorporate this knowledge while performing placement decisions.

### *6.2. Data*

The dataset used in our experiments is a trace of a large scale location-based online social network – *Gowalla*[3], available publicly from the SNAP (Leskovec and Krevl, 2014) repository. The Gowalla dataset has been used extensively (Yu and Pan, 2015, 2017) for data placement research in geo-distributed cloud services. The social network contains 196591 vertices and 950327 edges. The vertices are the users in the social network, while the edges represent friend relationship between two users. The trace provides 6442890 user check-ins logged over the period of February, 2009 to October, 2010. As described in Sec. 3, each user check-in consists of a data-request pattern and a location. In continuation to our discussions in Sections 1 and 3, a request (indicated by a check-in) by a user $v$ would involve retrieving the data of all his/her friends. Thus, the data-request pattern corresponding to a check-in by a user $v$ is the set of data-items corresponding to all the friends of $v$, i.e. $R(v)$. The location field of a user check-in are the GPS coordinates of the place from where the check-in was triggered. These GPS coordinates were mapped to the closest (in terms of distance) data-center region to identify the source location of a data-request pattern, and can thus, be one of the 9 data-center regions as described in Sec. 6.1. With this pre-processing, we obtain a use-case scenario consisting of 196591 data-items and 102314 data-request patterns.

Moving ahead, we present dataset statistics. Fig. 5 portrays the probability distribution of user check-ins across the 9 data-center regions discussed above. It is evident that Virginia and Frankfurt register the highest ($\approx 40\%$) and the second highest ($\approx 30\%$) number of user check-ins. On the other hand, Tokyo, Sydney, and SaoPaulo get the fewest ($\approx 10\%$ combined) number of user check-ins. This clearly shows a huge disparity in the check-in distribution. Based on this, we extract the storage size distribution $\Phi$ of the data-center regions, which is dependent upon both the number of check-ins registered in a region and the size of data-request pattern triggered by each check-in. Mathematically, the storage size for each data-center region $\forall L_j \in \mathcal{L}$ is calculated as $S_j = \sum |R(v)| \mid \exists C_k = (R(v), L_j), L_j \in \mathcal{L}$. Let $S = \sum_{j=1}^{l} S_j$ be the total storage size, then the data-center storage size follows a multinomial distribution and is calculated as: $\Phi \sim [\frac{S_1}{S}, \frac{S_2}{S}, \ldots, \frac{S_l}{S}]$. As is clear from Fig. 6, the desired storage distribution portrays a similar trend as that of the check-in distribution. Note that this storage distribution $\Phi$ also serves as an input to the data placement algorithm, thereby facilitating load-balancing among the 9 data-center regions. More fundamentally, the load-balancing factor is calculated as the expected storage size at each data-center region using the storage distribution $\Phi$.

### *6.3. Algorithms Benchmarked*

We compare the data placement algorithms proposed under the SpeCH framework for effectiveness, efficiency, and scalability against a number of baselines – *Random* and *Nearest*, and the state-of-the-art hypergraph partitioning technique (Yu and Pan, 2015, 2017). All the benchmarked algorithms were implemented in C++. To perform hypergraph partitioning for the technique proposed by (Yu and Pan, 2015) and reproduce their results we use the *PaToH* (Catalyurek, 2011) toolkit. Next, we give brief descriptions of the compared techniques:

- **Random**: partitions the set of data-items $\mathcal{D}$ randomly into $|\mathcal{L}| = l$ data-centers. To distribute the data-items according to the data-center storage size distribution $\Phi$, we ensure that

---

[3] `http://snap.stanford.edu/data/loc-gowalla.html`

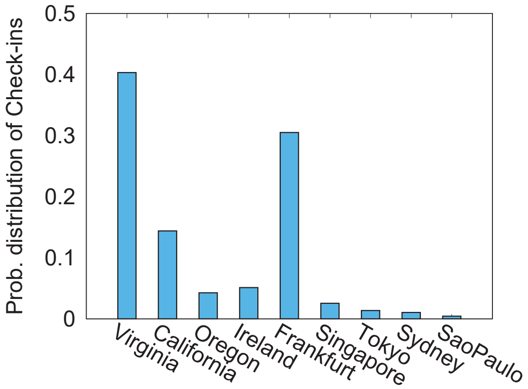


Figure 5: **Probability distribution of user check-ins across geo-distributed data-centers for the Gowalla dataset.**

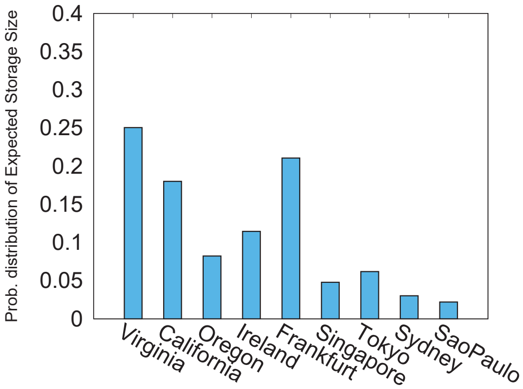


Figure 6: **Probability distribution of expected data-center storage size across geo-distributed data-centers for the Gowalla dataset.**

random partitioning samples data-items based on $\Phi$ thereby ensuring load-balancing.

- **Nearest**: assigns each data-item to the data-center from where it has been requested the highest number of times. Similar to random, to ensure load-balancing this technique follows the data-center storage distribution $\Phi$. Thus, once the data-center with the highest number of requests for a particular data-item has reached its capacity, we randomly choose a data-center location capable of serving new requests.
- **Hypergraph Partitioning**: is the data placement algorithm proposed by (Yu and Pan, 2015, 2017). After the hypergraph modeling step discussed in Sec. 4, it uses the hypergraph partitioning algorithms available in the PaToH toolkit. The partitioning algorithms maintain the data-center storage size distribution $\Phi$, thereby ensuring load-balancing.

### *6.4. Parameters*

As discussed in Sec. 4, different hyperedge weights $(W_{\mathcal{R}}, W^{\kappa}_{\mathcal{R}_{\mathcal{L}}}, W^{\mathcal{S}}_{\mathcal{R}_{\mathcal{L}}}, W^{\Gamma}_{\mathcal{R}_{\mathcal{L}}})$ facilitate optimization of different objectives. As stated in Eq. 3, a weight vector $\mathbb{W}$ facilitates prioritization of these objectives based on the assigned weights. In our study, we incorporate the use of specific weight vector $\mathbb{W}$ settings: $\mathbb{W}_1 : \{100, 1, 1, 1\}$, $\mathbb{W}_2 : \{1, 100, 1, 1\}$, $\mathbb{W}_3 : \{1, 1, 100, 1\}$, and $\mathbb{W}_4 : \{1, 1, 1, 100\}$, which represent different preferences or importance of the considered evaluation metrics, such as, higher priority of collocating the associated data-items thereby minimizing the data-center span $\mathcal{N}(\cdot)$, lower inter-datacenter traffic $\Gamma(\cdot)$, lower inter-datacenter latency $\kappa(\cdot)$, and lower storage cost $\mathcal{S}(\cdot)$ respectively. Note that in all the weight-vector settings, the value 100 is just used to indicate higher relative importance of the corresponding metric. The results portrayed are not dependent on the specific value of 100, rather the weight-vectors can work with any value as long as it is $>> 1$. Further, both SpectralApprox and SpectralDist employ the use of $\beta = 100$ smallest eigen-vectors of $L_H$ for spectral clustering.

### *6.5. Evaluation Metrics*

We consider two categories of evaluation metrics. The first type is concerned with the execution performance of the studied algorithms, while the other is concerned with their efficacy.

- **Efficiency:** We evaluate the efficiency of the methods using their execution time, i.e., the time required to produce the data placement output.
- **Efficacy:** of the studied methods is measured across the following metrics.
  - **Span ($\mathcal{N}(\cdot)$):** of a data-request pattern $R(v)$ is defined as the average number of data-centers required to be accessed to fetch the data-items requested in $R(v)$. Further, the span for the entire workload is calculated as the average of the data-center spans of each request pattern $R(v) \in \mathcal{R}$.
  - **Traffic ($\Gamma(\cdot)$):** The total traffic cost of a data-request pattern $R(v)$ is defined as the sum of outgoing traffic prices of the data-centers involved in outgoing requests for the data-items in $R(v)$. Further, the traffic cost of the entire workload is calculated as the sum of traffic costs of each request pattern $R(v) \in \mathcal{R}$.
  - **Latency ($\kappa(\cdot)$):** The inter-datacenter latency of a data-request pattern $R(v)$ is calculated as the sum of access latencies required to fetch all the data-items requested in $R(v)$ from the data-center where they are placed to the data-center from where the request was triggered. Further, the latency of the workload is calculated as the sum of the latencies of each request pattern $R(v) \in \mathcal{R}$.
  - **Storage ($\mathcal{S}(\cdot)$):** The sum of the total cost on storing all of the data-items corresponding to every data-request pattern $R(v) \in \mathcal{R}$ in data-centers $\mathcal{L}$ prescribed by the data placement algorithm.
  - **Balance:** is calculated as the pearson's correlation coefficient between the expected storage size distribution $\Phi$, and the actual storage size distribution obtained after performing data placement. If the value is close to 1, it means that the two distributions are highly similar, while they are dissimilar if the value is close to $-1$.
  - **Objective. (Obj.):** is defined as the weighted sum of the considered performance metrics, where the weights are described using the weight vector $\mathbb{W}$.

Note that the results portrayed in Sec. 7 corresponding to each evaluation metric (barring Balance) have been normalized in the scale of $[0,1]$ by dividing each value by the highest observed value in that particular metric. More specifically, let $nmax = max_{\forall R(v)\in\mathcal{R}}(\mathcal{N}(R(v)))$ be the highest observed span value, then the span for each data-request pattern $R(v)$ is normalized as: $\mathcal{N}(R(v))/nmax \mid \exists R(v) \in \mathcal{R}$. A similar operation is performed for other evaluation metrics as well. Normalization ensures that all the values lie in a common range ($[0,1]$), thereby facilitating joint analysis of all the considered evaluation metrics for all the algorithms. Moreover, normalization also ensures equal and fair contribution of each evaluation metric towards *Obj*, which as explained previously is calculated as the weighted sum of all the considered metrics.

## 7. Evaluation Results

Experiments for the sequential algorithms were done using code written in C++ on an Intel(R) Xeon(R) E5-2698 28-core machine with 2.3 GHz CPU and 256 GB RAM running Linux Ubuntu 16.04, while the distributed algorithms were executed using PySpark on an Apache Spark cluster consisting of 20 worker nodes, each comprising of 48 cores (2.3 GHz) and 256 GB RAM running Redhat Enterprise Linux 6.9. Owing to their non-deterministic nature, results for the random and hypergraph partitioning methods are averaged over 10 runs. Since the problem formulation in this study is concerned with the minimization of the evaluation metrics, the smaller the portrayed values the better the performance is. The replication factor was set to 3. Additionally, note that the results for the balance evaluation metric are close to 1 for all the techniques considered in this study. This is because every technique possesses the capability to honor the desired storage size distribution Φ.

The first set of experiments are concerned with identifying a Spark configuration that results in fast execution of the *SpectralDist* algorithm. The two parameters in the Spark configuration that have the highest impact on the execution time of a job are: (1) the number of executors, and (2) the number of cores per executor. To understand their impact on the overall execution time of *SpectralDist*, we perform an experiment as presented in Fig. 7. It is evident that increase in the number of executors facilitates decrease in the execution time at first, however beyond a certain number, the performance starts to deteriorate and the execution times increase. This behavior is due to distribution overhead and is expected for any distributed system, where after one point further distribution provides no more benefit. Moreover, this can be clearly observed for all the three settings presented in Fig. 7, i.e., with the number of cores per executor being 12, 16, and 20 respectively.

It is also clear from Fig. 7 that in a majority of the cases, the execution times obtained using a Spark configuration with executors possessing fewer cores per each executor instance, i.e. *tiny executors*, is better when compared to that of a configuration with more cores per each executor instance or *fat executors*. For instance with 12 executors, the execution time (15.05 minutes) for 16 cores per executor is lesser than the execution time (16.4 minutes) for 20 cores per executor. Similarly, with 20 executors, the execution times (in minutes) for 12, 16, and 20 cores per executor are $12.22 < 15 < 16.67$ respectively.

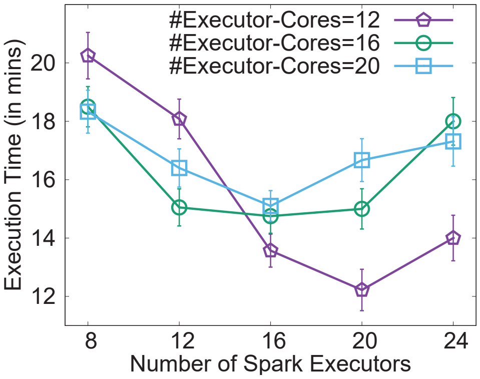


Figure 7: **Analyzing the variation in the execution time of spectral clustering with different Spark configurations.**

To better understand this observed behavior, we perform an experiment by keeping the total number of cores in the configuration fixed to 240, and exploring different possible combinations of the number of executors and number of cores per executor. The results are presented in Fig. 8. Note that since each machine in our cluster is comprised of 48 physical cores, it is impossible to obtain a configuration of 240 total cores with 4 executors. Thus, the total number of cores in the first entry (45,4) in Fig. 8 is 180 (as close as possible to 240), corresponding to a configuration with 4 executors, each with 45 cores. It is evident from Fig. 8 that an increase in the number of executors with a simultaneous decrease in the number of cores per executor (thereby keeping the total number of cores as 240), results in an improvement in the execution time of the *SpectralDist* algorithm till a certain point (6,40), i.e. with a configuration of 40 executors, each possessing 6 cores. Beyond this point, the execution time starts to increase and the performance deteriorates.

This result illustrates that neither less number of fat nor large number of tiny executors result in optimal runtime performance, rather the optimal configuration exists somewhere in between the two extremes. For the *SpectralDist* algorithm the best execution time of 10 minutes is achieved using a Spark configuration of 40 executors possessing 6 cores each. Note that all the results corresponding to *SpectralDist* in the rest of the paper are based on this configuration.

Having identified the optimal configuration of *SpectralDist* for the scenario under evaluation, we next analyze the efficacy of different spectral clustering methods. The eigen decomposition of the hypergraph laplacian (Sec. 5) is the most important step in spectral clustering as the quality of the obtained clusters is dependent on the quality of the spectra of the hypergraph laplacian. To this end, we compare the efficacy of the eigen decomposition step using three different methods, namely – (1) A sequential exact eigen decomposition method implemented in Octave, (2) An approximate eigen decomposition method implemented using the FBPCA library in Python, and (3) A distributed exact eigen decomposition method implemented in

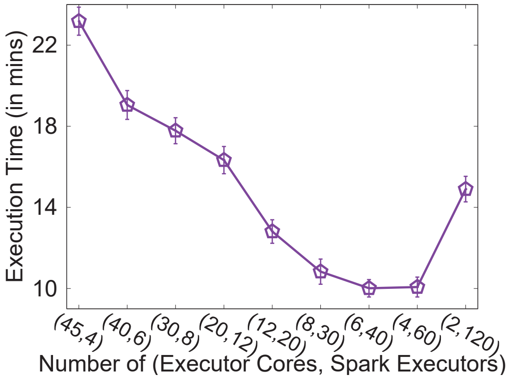


Figure 8: **Analyzing the effect of fat vs tiny executors on the execution time of spectral clustering.**

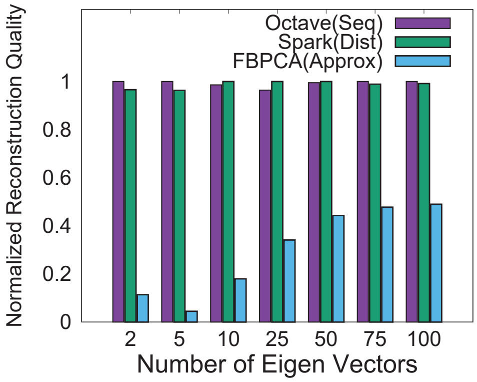


Figure 9: **Analyzing the difference in quality between exact and approximate eigen decomposition methods.**

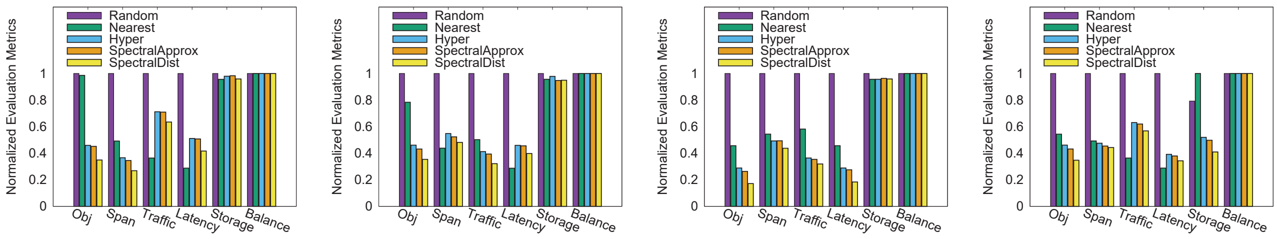
PySpark. As discussed in Sec. 5, eigen decomposition of a matrix $A$, decomposes it into three matrices $U$, $\Sigma$, and $V$, which can further be used to reconstruct the original matrix $A'$. The matrix reconstruction quality measured as $1 - MSE(A, A')$, where MSE stands for mean squared error, is used to measure the efficacy of the eigen decomposition step.

Fig. 9 presents the matrix reconstruction quality with varying eigen vectors for the three methods described above, normalized by the quality of the most efficacious method. Therefore, 1 represents the highest efficacy while 0 the lowest. Since methods (1) and (3), implemented in Octave and PySpark respectively, are exact in nature they possess the highest quality, which is also evident (values closer to 1) from Fig. 9. On the other hand, the approximate method can be 2-6 times worse when compared to the exact methods. In terms of their execution times, for obtaining a decomposition with top 100 eigen vectors, the exact eigen decomposition in Octave requires $\approx 1.5$ hours, in Spark $\approx 8$ minutes, while the approximate decomposition in FBPCA requires $\approx 30$ seconds for the Gowalla dataset used in this article. Since the execution time of Octave is impractically large, it is excluded from any further analysis in this article. The distributed exact eigen decomposition method in Spark is used in *SpectralDist*, while the approximate method in FBPCA is used in *SpectralApprox*.

Moving ahead we analyze the efficacy of both *SpectralApprox* and *SpectralDist* on the considered evaluation metrics. Fig. 10a presents the results corresponding to the weight-vector setting $\mathbb{W}_1$, where minimizing the data-center span holds the highest priority in the optimization objective. It is evident that the proposed spectral clustering algorithms (SpectralApprox and SpectralDist), and the state-of-the-art hypergraph partitioning algorithm (Hyper) achieve a low value on the overall optimization objective (Obj), while being significantly better than the random and nearest methods. This is because of their capability to preferentially minimize the data-center span, which possesses the highest priority in the optimization under $\mathbb{W}_1$. Note that Nearest outperforms Hyper, SpectralApprox, and SpectralDist on the traffic and latency metrics, as they have lower weights in the optimization objective under $\mathbb{W}_1$. However, SpectralApprox, SpectralDist, and Hyper are still significantly better than the Random method. The underlying eigen decomposition step in SpectralDist is exact while that in SpectralApprox is approximate, thus, the downstream efficacy of the former is much better (lower value on the considered metrics) than the latter.

A similar behavior is observed in Figs. 10b, 10c, and 10d corresponding to the other three weight vector settings $\mathbb{W}_2$, $\mathbb{W}_3$, and $\mathbb{W}_4$ respectively. SpectralApprox, SpectralDist, and Hyper outperform the other methods significantly on the overall optimization (Obj), while also being significantly better on the corresponding evaluation metric that the weight-vector setting is tuned to optimize. More fundamentally, in addition to being better on Obj., SpectralApprox, SpectralDist, and Hyper outperform the other methods in minimizing inter-datacenter traffic cost $\Gamma(\cdot)$, inter-datacenter latency $\kappa(\cdot)$, and storage cost $\mathcal{S}(\cdot)$, when a higher preference is given to these metrics under the weight-vector settings $\mathbb{W}_2$, $\mathbb{W}_3$, and $\mathbb{W}_4$ respectively.

The main limitation of Nearest is that it tries to assign each data-item to a data-center with the highest number of accesses to that data-item, thereby aiming to minimize (on an average) the geographical distance between the data-item and the source location of the data request oblivious to the fact that the actual traffic or storage costs might not be proportional to the distance. The main advantage of SpectralApprox, SpectralDist, and Hyper over Nearest is that owing to their higher-order modeling capabilities they are capable of better addressing multi-objective optimizations, and possess the capability to adapt their performance based on different weight-vector settings. This is evident from the results portrayed in Figs. 10a–10d. To further emphasize on the capability to adapt the optimization based on different weight-vector settings, we discuss the results portrayed in Fig. 10d. It is clear that according to $\mathbb{W}_4$, the optimization objective gives more preference towards minimizing the storage cost. Note that storage cost and other parameters like inter-datacenter traffic and latency might be inversely related to each other, i.e., a lower storage cost might lead to higher latencies or traffic cost. This behavior is also evident from Fig. 10d, where SpectralApprox, SpectralDist, and Hyper achieve lower storage costs, thereby also achieving better performance on Obj, however, suffer slightly on other metrics.



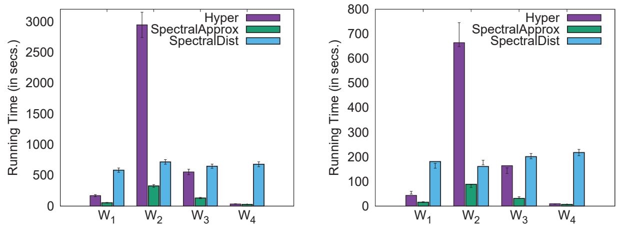
(a) (b) (c) (d)

Figure 10: **Analyzing the variation in the evaluation metrics under different weight-vector settings. SpectralDist results in reducing the (a) data-center span $\mathcal{N}(\cdot)$ by $\approx 27\%$ when compared to Hyper with $\mathbb{W}_1 = \{100, 1, 1, 1\}$; (b) inter data-center traffic $\Gamma(\cdot)$ by $\approx 22\%$ when compared to Hyper with $\mathbb{W}_2 = \{1, 100, 1, 1\}$; (c) inter data-center latency $\kappa(\cdot)$ by $\approx 36\%$ when compared to Hyper with $\mathbb{W}_3 = \{1, 1, 100, 1\}$; (d) storage cost $\mathcal{S}(\cdot)$ by $\approx 20\%$ when compared to Hyper with $\mathbb{W}_4 = \{1, 1, 1, 100\}$.**

Thus, methods like Nearest would find difficulty in handling such cases, while both the spectral clustering algorithms (SpectralApprox and SpectralDist), and Hyper possess the capability to adapt the optimization based on the weight-vector.

Figs. 10a–10d show that the performance of both SpectralApprox and Hyper are quite similar on the evaluated metrics. While SpectralApprox is only marginally better in efficacy when compared to Hyper, SpectralDist is 30-50% better when compared to both SpectralApprox and Hyper. The major advantage of both SpectralApprox and SpectralDist over Hyper comes from their capability to scale gracefully and efficiently to large datasets. It is intuitive that scalability is a paramount property for any data placement algorithm, since the scale of real-world social networks or for that matter any real-world data-intensive services is humongous. Fig. 11 shows the performance of the proposed algorithms on the Gowalla and Brightkite[4] datasets. It is clear from Fig. 11a that SpectralApprox is up to *10 times* ($\approx$ 3–4 times on average) *faster* when compared to Hyper, while also being slightly better in all evaluated metrics for the majority of the considered weight-vector settings. On the other hand, although SpectralDist is $\approx$ 2–3 and $\approx$ 6–8 times slower (on average) than Hyper and SpectralApprox respectively, it is $\approx$ 30–50% better in efficacy. Since Brightkite is much smaller in size when compared to Gowalla, the results presented on Gowalla should allow the readers a good insight into the performance of the proposed techniques. Nevertheless, to show that the proposed techniques can be applied similarly to other datasets, we present results on the Brightkite dataset in Fig. 11b. It is clear that Fig. 11b portrays trends similar to those observed in Fig. 11a. Thus, the SpeCH framework facilitates to significantly improve the efficiency-efficacy trade-off for the data placement problem using the two spectral clustering algorithms, SpectralApprox and SpectralDist.

To summarize, through extensive experiments we verify that both the spectral clustering algorithms, SpectralApprox and SpectralDist, proposed under the SpeCH framework are *efficient, scalable, and effective*. Although SpectralApprox and SpectralDist are not always the best on every evaluated metric, they serve to be the most effective technique in terms of improving the overall weighted sum of evaluation metrics (Obj), which is the main target of our multi-objective optimization. Additionally, they possess the capability to adapt to the change in weight vector settings $\mathbb{W}$, which facilitates handling of a variety of real-world scenarios as described by different weight vectors.



(a) Gowalla (b) Brightkite

Figure 11: **Comparing the execution times of the proposed spectral clustering algorithms with hypergraph partitioning algorithm proposed in (Yu and Pan, 2015) on the (a) Gowalla, and (b) Brightkite datasets**.

## 8. Conclusions and Future Work

In this article, we have addressed the problem of data placement of data-intensive services into geo-distributed clouds. We identified the need for specialized methods to perform data placement for data-intensive services, as contrary to MapReduce style workloads, these workloads require access to multiple datasets within a single transaction, thereby rendering traditional methods of hash based partitioning inadequate. Consequently, we devised a scalable framework, *SpeCH*, to perform data placement using spectral clustering for hypergraph partitioning. Under the SpeCH framework, we proposed two algorithms, namely – *SpectralApprox* and *SpectralDist*, which facilitated spectral clustering to scale to large workloads. While *SpectralApprox* improved the efficiency of spectral clustering by obtaining low-rank approximations to the hypergraph matrix with bounded error guarantees, *SpectralDist* employed the use of recent hardware and big data technologies to scale up spectral clustering by distributing the computation to several machines without compromising on quality. Moreover, this also

---

[4] `https://snap.stanford.edu/data/loc-Brightkite.html`

enables SpectralDist to scale to workloads that cannot fit in a single machine. Our empirical studies on a real-world trace-based social network dataset portrayed the effectiveness, efficiency, and scalability of the proposed algorithms. The *SpectralApprox* algorithm portrayed superior efficiency and was up to 10 times faster when compared to the state-of-the-art, while being as good in efficacy on the evaluated metrics. On the other hand, the *SpectralDist* algorithm portrayed superior efficacy on the evaluated metrics being up to 50% better, however, was 5-6 times slower when compared to the *SpectralApprox* algorithm. In summary, the *SpectralApprox* and *SpectralDist* algorithms proposed under the SpeCH framework provided a better *efficiency-efficacy trade-off*. When efficacy is of higher priority, *SpectralDist* offered a better solution while keeping execution times practical. On the other hand, *SpectralApprox* is the algorithm of choice when keeping execution times low was important, while keeping efficacy on the evaluated metrics as good as the state-of-the-art.

Currently, SpeCH and its algorithms learn a data placement strategy from a historical snapshot of the social network trace. In future work, we aim to make SpeCH and its algorithms adaptive for managing updates in the data and dynamically changing the data placement strategy accordingly. Additionally, the notion of replicas will be included directly in the data placement problem formulation.

## Acknowledgements

This research is partly funded by VLAIO, under grant number 140055 (SBO Decomads).

## References

Agarwal, S., Dunagan, J., Jain, N., Saroiu, S., Wolman, A., Bhogan, H., 2010. Volley: Automated Data Placement for Geo-distributed Cloud Services, in: NSDI, pp. 1–16.

Amazon, 2017. AWS Global Infrastructure. URL: `https://aws.amazon.com/about-aws/global-infrastructure/`. accessed: 5 Feb 2019.

Arthur, D., Vassilvitskii, S., 2007. k-means++: The Advantages of Careful Seeding, in: SODA, pp. 1027–1035.

Atrey, A., van Seghbroeck, G., Volckaert, B., Turck, F.D., 2018. Scalable data placement of data-intensive services in geo-distributed clouds, in: CLOSER, pp. 497–508.

Catalyurek, U.V., 2011. PaToH (Partitioning Tool for Hypergraphs). URL: `http://bmi.osu.edu/umit/PaToH/manual.pdf`. accessed: 5 Feb 2019.

Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozdag, D., Heaphy, R., Riesen, L.A., 2007. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations, in: IPDPS, pp. 1–11.

Chervenak, A., Deelman, E., Livny, M., Su, M., Schuler, R., Bharathi, S., Mehta, G., Vahi, K., 2007. Data Placement for Scientific Applications in Distributed Environments, in: Grid.

Deerwester, S.C., Ziff, D.A., Waclena, K., 1990. An Architecture for Full Text Retrieval Systems, in: DEXA, pp. 22–29.

Ebrahimi, M., Mohan, A., Kashlev, A., Lu, S., 2015. BDAP: A Big Data Placement Strategy for Cloud-Based Scientific Workflows, in: BigDataService, pp. 105–114.

insideBIGDATA Editorial Team, 2017. The Exponential Growth of Data. URL: `https://insidebigdata.com/2017/02/16/the-exponential-growth-of-data/`. accessed: 5 Feb 2019.

Ferdaus, M.H., Murshed, M., Calheiros, R.N., Buyya, R., 2017. An algorithm for network and data-aware placement of multi-tier applications in cloud data centers. JNCA 98, 65 – 83.

Gibson, D., Kleinberg, J., Raghavan, P., 1998. Inferring Web Communities from Link Topology, in: Hypertext, pp. 225–234.

Golab, L., Hadjieleftheriou, M., Karloff, H., Saha, B., 2014. Distributed Data Placement to Minimize Communication Costs via Graph Partitioning, in: SSDBM, pp. 1–12.

Guo, W., Wang, X., 2013. A data placement strategy based on genetic algorithm in cloud computing platform, in: Web Information System and Application Conference, pp. 369–372.

Halko, N., Martinsson, P., Tropp, J.A., 2011. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. SIAM Review 53, 217–288.

Han, S., Kim, B., Han, J., K.Kim, Song, J., 2017. Adaptive Data Placement for Improving Performance of Online Social Network Services in a Multicloud Environment, in: Scientific Programming, pp. 1–17.

Huguenin, K., Kermarrec, A.M., Kloudas, K., Taïani, F., 2012. Content and Geographical Locality in User-generated Content Sharing Systems, in: NOSSDAV, pp. 77–82.

Jiao, L., Li, J., Du, W., Fu, X., 2014. Multi-objective data placement for multi-cloud socially aware services, in: INFOCOM, pp. 28–36.

Kayaaslan, E., Cambazoglu, B.B., Aykanat, C., 2013. Document Replication Strategies for Geographically Distributed Web Search Engines. Information Processing & Management 49, 51–66.

Leskovec, J., Krevl, A., 2014. SNAP Datasets: Stanford large network dataset collection. URL: `https://snap.stanford.edu/data/`. (accessed: 5 Feb 2019).

Li, X., Zhang, L., Wu, Y., Liu, X., Zhu, E., Yi, H., Wang, F., Zhang, C., Yang, Y., 2018. A Novel Workflow-Level Data Placement Strategy for Data-Sharing Scientific Cloud Workflows. IEEE TSC PP.

Liu, X., Datta, A., 2011. Towards Intelligent Data Placement for Scientific Workflows in Collaborative Cloud Environment, in: IPDPSW, pp. 1052–1061.

Meila, M., Shi, J., 2001. Learning Segmentation by Random Walks, in: NIPS, pp. 873–879.

Ng, A.Y., Jordan, M.I., Weiss, Y., 2001. On Spectral Clustering: Analysis and an Algorithm, in: NIPS, pp. 849–856.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V., 2013. Scaling Memcache at Facebook, in: NSDI, pp. 385–398.

Quamar, A., Kumar, K.A., Deshpande, A., 2013. SWORD: Scalable Workload-aware Data Placement for Transactional Workloads, in: EDBT, pp. 430–441.

Rochman, Y., Levy, H., Brosh, E., 2013. Resource placement and assignment in distributed network topologies, in: INFOCOM, pp. 1914–1922.

Schultz, J., 2017. How Much Data is Created on the Internet Each Day? URL: `https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/`. accessed: 5 Feb 2019.

Shabeera, T., Kumar, S.M., Salam, S.M., Krishnan, K.M., 2017. Optimizing VM allocation and data placement for data-intensive applications in cloud using ACO metaheuristic algorithm. International Journal of Engineering Science and Technology 20, 616 – 628.

Shankaranarayanan, P.N., Sivakumar, A., Rao, S., Tawarmalani, M., 2014. Performance Sensitive Replication in Geo-distributed Cloud Datastores, in: DSN, pp. 240–251.

Shi, J., Malik, J., 2000. Normalized Cuts and Image Segmentation. PAMI 22, 888–905.

Wang, Z., 2016. Latency Between AWS Global Regions. URL: `http://zhiguang.me/2016/05/10/latency-between-aws-global-regions/`. (accessed: 5 Feb 2019).

White, T., 2012. Hadoop: The Definitive Guide. O'Reilly Media, Inc.

Wikipedia, 2018. Laplacian Matrix. URL: `https://en.wikipedia.org/wiki/Laplacian_matrix`. accessed: 5 Feb 2019.

Yu, B., Pan, J., 2015. Location-aware associated data placement for geo-distributed data-intensive applications, in: INFOCOM, pp. 603–611.

Yu, B., Pan, J., 2016. Sketch-based data placement among geo-distributed datacenters for cloud storages, in: INFOCOM, pp. 1–9.

Yu, B., Pan, J., 2017. A Framework of Hypergraph-based Data Placement among Geo-distributed Datacenters. IEEE TSC PP, 1–14.

Yu, T., Qiu, J., Reinwald, B., Zhi, L., Wang, Q., Wang, N., 2012. Intelligent database placement in cloud environment, in: ICWS, pp. 544–551.

Yuan, D., Yang, Y., Liu, X., Chen, J., 2010. A data placement strategy in scientific cloud workflows. FGCS 26, 1200 – 1214.

Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., 2016. Apache spark: A unified engine for big data processing. CACM 59, 56–65.

Zhang, J., Chen, J., Luo, J., Song, A., 2016. Efficient location-aware data placement for data-intensive applications in geo-distributed scientific data centers. Tsinghua Science and Technology 21, 471–481.

Zhao, Q., Xiong, C., Wang, P., 2016a. Heuristic data placement for data-intensive applications in heterogeneous cloud. Journal of Electrical and Computer Engineering 2016, 1–8.

Zhao, Q., Xiong, C., Zhang, K., Yue, Y., Yang, J., 2016b. A data placement algorithm for data intensive applications in cloud. International Journal of Grid and Distributed Computing 9, 145–156.

Zhou, D., Huang, J., Schölkopf, B., 2006. Learning with Hypergraphs: Clustering, Classification, and Embedding, in: NIPS, pp. 1601–1608.