On Exploiting Location Flexibility in Data-intensive Distributed Systems

by

Boyang Yu
B.Sc., Nankai University, China, 2006
M.Sc., Nankai University, China, 2009

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Boyang Yu, 2016
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.
On Exploiting Location Flexibility in Data-intensive Distributed Systems

by

Boyang Yu
B.Sc., Nankai University, China, 2006
M.Sc., Nankai University, China, 2009

Supervisory Committee

Dr. Jianping Pan, Supervisor
(Department of Computer Science)

Dr. Kui Wu, Departmental Member
(Department of Computer Science)

Dr. Wu-Sheng Lu, Outside Member
(Department of Electrical and Computer Engineering)
ABSTRACT

With the fast growth of data-intensive distributed systems today, more novel and principled approaches are needed to improve the system efficiency, ensure the service quality to satisfy the user requirements, and lower the system running cost. This dissertation studies the design issues in the data-intensive distributed systems, which are differentiated from other systems by the heavy workload of data movement and are characterized by the fact that the destination of each data flow is limited to a subset of available locations, such as those servers holding the requested data. Besides, even among the feasible subset, different locations may result in different performance.

The studies in this dissertation improve the data-intensive systems by exploiting the data storage location flexibility. It addresses how to reasonably determine the data placement based on the measured request patterns, to improve a series of performance metrics, such as the data access latency, system throughput and various costs, by the proposed hypergraph models for data placement. To implement the proposal with a lower overhead, a sketch-based data placement scheme is presented, which constructs the sparsified hypergraph under a distributed and streaming-based system model, achieving a good approximation on the performance improvement. As the network can potentially become the bottleneck of distributed data-intensive systems due to the frequent data movement among storage nodes, the online data placement
by reinforcement learning is proposed which intelligently determines the storage locations of each data item at the moment that the item is going to be written or updated, with the joint-awareness of network conditions and request patterns. Meanwhile, noticing that distributed memory caches are effective measures in lowering the workload to the backend storage systems, the auto-scaling of memory cache clusters is studied, which tries to balance the energy cost of the service and the performance ensured.

As the outcome of this dissertation, the designed schemes and methods essentially help to improve the running efficiency of data-intensive distributed systems. Therefore, they can either help to improve the user-perceived service quality under the same level of system resource investment, or help to lower the monetary expense and energy consumption in maintaining the system under the same performance standard. From the two perspectives, both the end users and the system providers could obtain benefits from the results of the studies.
Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables ix

List of Figures x

Acknowledgements xii

Dedication xiii

1 Introduction 1
   1.1 Data-intensive Distributed Systems 1
   1.2 Location Flexibility in Data Flows 3
   1.3 Research Problems and Contributions 6
      1.3.1 Offline Data Placement 6
      1.3.2 Sketch-based Data Placement 8
      1.3.3 Online Data Placement 10
      1.3.4 Memory Cache Cluster Auto-scaling 11
   1.4 Dissertation Organization 13

2 Hypergraph Models for Data Placement 15
   2.1 Overview 15
   2.2 Related Work 16
   2.3 Modeling Framework 19
      2.3.1 Data Items and Nodes 19
      2.3.2 Data Placement 20
List of Tables

Table 2.1 Prices of Amazon clouds .................................................. 39
Table 2.2 Running time of each round (in Seconds) with different input scales 47
Table 3.1 Overhead comparisons ...................................................... 66
Table 5.1 Example: Obtain sub-solutions in dynamic programming .......... 109
List of Figures

Figure 2.1 Problem inputs: (a) request pattern set \( P \); (b) request rate set \( R \) .......................... 19
Figure 2.2 Hypergraph framework .................................................. 24
Figure 2.3 Logic flow of the scheme with replicas ............................ 34
Figure 2.4 Measured latencies of distributed datacenters ................. 39
Figure 2.5 Distribution of geo-distributed requests .......................... 40
Figure 2.6 Performance verification with random weight vectors  .......... 43
Figure 2.7 Performance comparison, \( W = 100 : 0 : 1 : 1 : 1 \) ............... 43
Figure 2.8 Performance comparison, \( W = 1 : 0 : 100 : 1 : 1 \) ............... 44
Figure 2.9 Performance comparison, \( W = 1 : 0 : 1 : 100 : 1 \) ............... 45
Figure 2.10 Performance comparison, \( W = 1 : 0 : 1 : 1 : 100 \) .............. 45
Figure 2.11 Iterative replica placement ............................................. 47

Figure 3.1 Illustration of the system and schemes ......................... 53
Figure 3.2 Workflow of data placement decisions ......................... 54
Figure 3.3 Performance comparison of different heuristics .............. 58
Figure 3.4 Interactive steps of constructing the sparsifier ............... 60
Figure 3.5 Sampling on a sliding window ..................................... 63
Figure 3.6 Counting on a sliding window .................................... 65
Figure 3.7 Performance comparisons ............................................ 68
Figure 3.8 Counting accuracy ..................................................... 68
Figure 3.9 Sampling performance ................................................ 69

Figure 4.1 Data storage system: metadata server and nodes .............. 73
Figure 4.2 Neural network deployed ............................................. 77
Figure 4.3 Production system ..................................................... 82
Figure 4.4 Accumulated reward: W-optimized ............................... 86
Figure 4.5 Accumulated reward: R-optimized ............................... 86
Figure 4.6 Read/write latency: W-optimized ................................ 87
Figure 4.7 Read/write latency: R-optimized ................................ 88
ACKNOWLEDGEMENTS

I would like to sincerely thank many people who helped me in completing this dissertation. The first is my supervisor, Professor Jianping Pan, for his advice not only on research but also on life. His continuous support and guidance helped me in all the time of my study. I am also thankful to my supervisory committee members Professor Kui Wu and Professor Wu-Sheng Lu, for their valuable suggestions, advice and feedback. My studies at University of Victoria have been made better by many students, group members, collaborators, professors and staff members, so my thanks also go to them. Besides, my thanks go to the funding agencies that provided financial support to me.
DEDICATION

Dedicated to Hongrui with my love, and to my parents and friends.
Chapter 1

Introduction

1.1 Data-intensive Distributed Systems

Along with innovations and advances of Internet technologies, the scale of data generated, stored and processed today is increasing drastically [36]. For example, Google processed about 24 petabytes of data per day in 2009 [13]. As of 2014, Facebook managed over 282 petabytes of storage for photos and videos uploaded by its users [13]. Most recently, Google announced that its photo service had taken about 13.6 petabytes of storage in 12 months, which was open to the public in 2015 [11]. As pointed out by [49], we are going to witness further that the size of our digital world is increasing in an exponential way in the following decades. The rising data scale offers great opportunities to answer the questions that people have not been able to address or even ask in the past due to the limitation of technologies. Meanwhile, it introduces various challenges in designing and implementing the systems for data management [22].

The discussed data-intensive distributed systems are a class of computing infrastructures that store a large volume of data in distributed nodes and rely on the network to move data between the storage location and computing location intensively. They are designed to address certain large-scale computation tasks or fulfill data service objectives. Typically, these systems are constituted by a large number of processors and storage nodes, which are organized as large-scale commodity computing clusters. The nodes in the system are connected by high-speed communication switches and network links, so application software can utilize multiple nodes together to provision the high service capacity or to address large-scale problems.
Multiple clusters can also be inter-connected, which offers opportunities to support geo-distributed services. This dissertation contributes to improving the cost-efficiency of data-intensive systems and therefore providing a higher service quality under the same resource investment.

The data-intensive applications running in these systems may have different behavior and characteristics, however, they share similar fundamentals when we focus on how data are managed by the system and how data move among nodes in the runtime. Using the systems for data analytics as an example, due to the dataset being huge scale, the full dataset is always partitioned into multiple segments and handled by different nodes. With data being stored in a distributed way, the movement of data happens intensively in the system to fulfill the computational or data retrieval tasks. Besides, network availability and performance become influential factors to the system efficiency, or could even become the system bottleneck. Typical application scenarios of data-intensive systems are listed as follows.

1. Content Retrieval Services: Most traditional data services such as web services, Video-on-Demand services and file storage services fall in this category. Their common characteristic is to provide users with the information through centralized clusters. In order to meet the rapidly growing scale of user groups, who are also distributed across regions, the geographically distributed service architecture appears and Content Distribution Network (CDN) becomes the main form of infrastructure for content retrieval services.

2. Online Social Networks (OSN): It is a new paradigm of web services, where more data and contents are generated by the users in any form such as posts and tweets. It significantly increases the scale of data stored in the computer clusters, followed by much higher frequency of data access than ever before. Besides, the friend relationship in OSN can affect the pattern of data access, i.e., the data requested are always based on the existing friendship, which is much different from traditional content retrieval services.

3. Data Analysis Systems: With the concept of “Big Data” or “Internet of Things”, people note that some underlying principles or knowledge can be discovered from analyzing or mining the large-scale data captured from the physical world. The algorithms or logical methods that help to reveal the nature underlying the data are important in these systems, but meanwhile, how to manage, store and provision the data efficiently is also meaningful and challenging.
Various data-intensive distributed storage systems have been developed to match with different application requirements. Hadoop/HDFS [5] implements the MapReduce architecture and is mainly utilized for data analytics. HDFS, included as a component of Hadoop, is implemented as a distributed file system, for the purpose of persistent storage. HDFS consists of a centralized metadata server and multiple data servers/nodes; the former is used to provision the metadata of the file system, mainly the mapping between data blocks and their storage locations, while the latter are used to store data blocks. Besides, HDFS provides redundancy and fault tolerance through block replications. Another typical storage system is Cassandra [14], also recognized as a NoSQL database. Cassandra maintains data segments with consistent hashing, and the data tables being stored are partitioned by columns.

In order to overcome the I/O bottleneck in disk-based storage systems, memory-based cache clusters are used to lower the data retrieving latency. Memcached [8] is the state-of-the-art and most popular implementation of these systems. It achieves horizontal scalability mainly through consistent hashing [62], where the key space is partitioned into segments and each server is responsible for one segment or sub-space of the whole space. Besides, it also achieves vertical scalability through the mirroring of one sub-space, i.e., assigning multiple servers to redundantly cover the same sub-space and distributing the incoming traffic to them through a load balancer. When caching takes effect, the time duration for data retrieval can be significantly decreased and the burden of disks can be relieved, because of the throughput and latency difference between disk and memory.

However, when provisioning these infrastructures for data-intensive services, people meet various challenges, ranging from user-perceived performance metrics to cost issues such as hardware investment, power consumption and carbon dioxide emissions [36]. To conquer the challenges, more novel and principled approaches are needed to improve the design of data-intensive distributed systems. During our studies, we found that existing systems or techniques either ignored the potential of performance improvement through an optimized design, or overlooked the specific requirements of data-intensive applications, which inevitably results in a sub-optimal outcome.

### 1.2 Location Flexibility in Data Flows

The studies in this dissertation focus on the data flows in the discussed systems. Each
data flow can be characterized by the flow source, path and destination. There exist flexibilities in fulfilling data flows, i.e., data storage locations and flow destinations are variable. Therefore, this dissertation proposes to exploit this flexibility to improve the user-perceived performance, e.g., service latency, and lower the system cost. For example, for a data block to be written into a distributed file system, any storage nodes in the system can serve as the storage location for that data block. Meanwhile, we may achieve a lower write finish time, if we choose a destination closer to the source of data flow. On the other hand, for most storage systems, each data request from users can only be fulfilled at the node holding the requested data or its replicas, so how to better exploit the limited flexibility of data storage locations needs careful design and discussions.

In most existing implementations, the flexibility of data locations is not emphasized or fully utilized. Specifically, most existing implementations use hash-based methods to distribute the destination locations of requests or the data storage locations. Fundamentally, with uniform hashing functions, the hash-based method uniformly distributes tasks to all available serving nodes, so the system can achieve load balancing and thus avoid the performance issues related to overwhelmed nodes. However, the skewed request patterns and limited network capacities, e.g., the bandwidth between the source and destination, may result in a system far from optimal.

In the literature, some recent work starts to observe this issue and propose possible improvement schemes by utilizing the location flexibility. Jalaparti et al. [59] proposed to place all the data used by a MapReduce job at the nodes in the same datacenter rack, so a large portion of inter-rack traffic can be avoided, which will make the job time-consuming if not properly handled. Eltabakh et al. [46] discussed the possibility of reducing the data shuffling costs and network overhead through effective data partitioning and proposed a generic mechanism that allows applications to control data placement at the file system level which subsequently helps to co-locate corresponding replicas on the same data nodes. Under the scenario of OSN, there are some breakthroughs on addressing how to improve the system with social-awareness [60] [83]. Compared with the related work focusing on some specific scenarios, our proposed schemes have broader applications, because of our abstraction of data flows, avoiding the dependence on some application-specific information. Various efforts that have been taken to advance the exploitation of location flexibility in data-intensive distributed systems will be presented in this dissertation, and the experiment results to validate their effectiveness will also be shown.
First, a general problem of how to reasonably place data items and replicas among the available distributed nodes in a networked system, with the requests for them varying across different regions, is raised and solved. With the proposed offline data placement [108, 110] for geo-distributed storage systems, we can achieve performance benefits through the control of data fulfillment locations, which normally only happened at the stage of request dispatching. The storage locations can affect the performance of the whole system, e.g., when a data item is requested from multiple datacenters, it is preferred to be placed at the datacenter with the highest request rate, so that the benefit of data locality is achieved to the greatest extent. The proposed scheme aims at utilizing the ideas and improving the related performance metrics. Also, its implementation could be improved by the proposed sketch-based data placement [112], which largely lowers the overhead of the scheme itself, but remains effective in terms of the performance improvement.

To cope with a more dynamic network environment, which could become a bottleneck of the system if not properly handled otherwise, an online data placement scheme [109] to dynamically determine the storage locations is needed. Noticing that if we change the data location at the time of data writing and re-writing, there is no extra cost compared with forced data migration, a generic scheme is presented, which helps to dynamically determine the storage location for each specific data item when it is going to be written or updated. The scheme is based on reinforcement learning and neural network techniques. It is able to deal with the dynamics of network conditions and request patterns, which cannot be addressed by typical offline schemes. The storage backend of the scheme could be built on either random-access memory (RAM) or hard disks.

Distributed memory caches are broadly used to solve the throughput and latency bottleneck of storage systems limited by the physical properties of hard disks, but are subject to additional cost. Noticing that the fulfillment location of data requests is constrained but still allowed to change in such services, we proposed auto-scaling of memory cache clusters [111]. The proposed scheme improves the energy efficiency by introducing elastic system provisioning. Specifically, the cluster scaling and the request dispatching in the system are dynamically controlled, so that the average energy cost of the system is lowered when compared with other schemes. Besides, the proposed scheme can make the tradeoff between the request latency, cache hit rate performance and system energy cost to meet the specific demands of memory cache cluster operators.
In summary, three essential problems, offline data placement, online data placement and auto-scaling of memory cache clusters have been formulated so as to improve the design of data-intensive distributed systems. Besides, the offline data placement is further improved by sketch-based data placement to lower the overhead introduced and the time complexity. Among the problems, offline data placement and sketch-based data placement aim at long-term solutions that take effect for days, while online data placement and auto-scaling of memory cache clusters are designed to handle workload dynamics, so they are short-term and dynamic solution. All schemes proposed in this dissertation in the application layer of the network protocol stack, so they can coexist with the mechanisms in lower-layer protocols, e.g., congestion control, flow scheduling, etc. Below, the problems addressed and their contributions are clarified with more details.

1.3 Research Problems and Contributions

1.3.1 Offline Data Placement

In geo-distributed storage systems, people are facing a general problem of how to reasonably place the data items and replicas among the distributed datacenters, with the requests for them varying across different regions. The storage locations of the data items matter to the user-perceived performance, such as service latency, because each data request from the users can be fulfilled only at the datacenter holding the requested data or its replicas. Meanwhile, the optimized data placement can help the service providers by lowering their cost with the improved system efficiency and ensuring the high system availability.

Storing the requested data closer to the users is the motivation of most existing work on data placement, which helps to reduce the latency perceived by the users and lower the relaying traffic among datacenters. For the network applications that can take advantage of geographically distributed datacenters, Content Distribution Network (CDN) \[85\] is broadly applied to facilitate the access of videos, photos and text. With the emerging techniques related to geo-distributed public clouds, more factors are considered in determining the storage location of each data item, e.g., the different storage prices at different locations and the different cost of inter-datacenter traffic. Besides, instead of replicating each content to every datacenter, content service providers may need to constrain the number of replicas allowed for each data item.
under the increasing scale of data items and the underlying pattern of data traffic. This leads to the problem of how to choose the proper datacenters to store the replicas.

The issue of multi-get hole [6] has not been paid enough attention to until recently, but it can also affect the decisions of placing data items to distributed locations. The multi-get hole issue means that when multiple data items are requested in one transaction, the span of serving such a request affects the system throughput. It has been figured out that using fewer nodes to fulfill such a request is better in terms of the system throughput [6], because the request dispatched to each node will introduce a certain overhead to the node regardless of the amount of data requested. To overcome the issue, a favorable paradigm is to place the strongly associated data items, those that are often requested together in the same transaction, at the same location. It has profound applications in Online Transaction Processing systems, where a transaction is fulfilled only after accessing multiple data tables, in Online Social Network (OSN) services, where the polling of news feeds involves the data of multiple users, and even in regular web services, since visiting a webpage actually needs to download multiple files, such as documents, images and scripts.

It is challenging to solve the data placement problem that combines these different aspects. Thus, we propose a general framework of hypergraph-based data placement. Starting with a simple scenario without replicas, we provide the fundamental methods in the framework of hypergraph-based data placement, including the hypergraph modeling and the hypergraph partitioning. We use hypergraph models to capture the performance metrics desired by the system or service operators. Due to our formulation of edges in the hypergraph, the metrics supported by the framework fall into two categories: a) the associations among data items, which consist of the system time from the perspective of running efficiency and the distributed execution overhead reflected by the network load, and b) the distances between data items and nodes, which can reflect the inter-datacenter traffic, sum of access latencies and cost of storage. Further, we consider the scenario with replicas where a certain number of replicas are allowed for each item. The existence of multiple replicas in different locations introduces the routing decision problem, because any of the corresponding replicas can be used to fulfill the request. A multi-round scheme that iteratively makes the routing decision and replica placement decision is proposed.

The state-of-the-art implementations in most distributed storage systems today, such as HDFS [5] and Cassandra [14], are mainly hash-based, where the storage locations are only determined by the hashing results. Among the related work that
discussed the managed data placement, some existing work either just focused on the distance between data and user, such as [87] and [107], or only addressed the co-location of associated data items, such as [80, 84, 86]. Some others discussed the issues under the scenario of OSN [60, 71, 83], where the considered association of stored entities is only based on pairs of users (each pair consists of a user and one of the friends of the user), which overlooks the fine-grained relationship between exact data items as well as the association that involves more than two items. Agarwal et al. [15] proposed a scheme of automated data placement, which is the work most related to ours in the problem definition, but it solved the problem using some heuristics that are hard to be proved as optimal. To the best of our knowledge, our data placement scheme is advanced in terms of jointly considering the objectives from the two categories and without a relaxation in modeling. Besides, the developed methods to support replicas are also novel and make the scheme more comprehensive.

1.3.2 Sketch-based Data Placement

Over the years, the term “Big Data” has been used to describe datasets that were believed to be too large to be efficiently processed and utilized through traditional techniques. Among numerous challenges arising from the implementation of big data, it is important to deal with how to intelligently and efficiently determine the placement of each item in the big dataset among the available geo-distributed datacenters.

With the booming of distributed storages, the topic of data placement has been studied in the literature from different perspectives [15, 60, 108]. Existing schemes for data placement are typically implemented as follows: on each geo-distributed datacenter or site handling end-user access, capture the logs of requests; on a central controller, gather the logs from all these sites, process them to extract the access frequency of each requested itemset, and feed the extracted information to an algorithm which finally makes the placement decisions. These schemes are intuitively correct in design, but would meet some practical challenges when applied to a large-scale system, especially in the situations where the number of data items processed by the algorithm is huge and the request traffic is high. Specifically, we can expect a high cost on the storage and transfer of logs, and a long running time by the algorithm.

For these challenges, we propose the Sketch-based Data Placement (SDP), trying to lower the overhead introduced by data placement while still keeping its benefits. In SDP, sketches of the request traffic are maintained at distributed sites, as a sub-
stitute to the lengthy logs mentioned above. Sketches are data structures that can approximately characterize certain properties of a stream of events, using a sublinear space. A sketch usually supports two kinds of operations: update, which is applied when processing an incoming event in the stream and updates the data structure; and query, which extracts the properties captured by the data structure so far. In SDP, two kinds of sketches are maintained at each site and updated by each event in the stream. They are sampling sketch, providing the uniform sampling of events in the stream, and counting sketch, capturing the frequency of different events in the stream. Further, they are both designed to work on the sliding window over time, so that the system can make data placement decisions based on the recent traffic.

The controller of SDP will utilize the information stored in sketches to construct a sparsified hypergraph, termed as hypergraph sparsifier, and then apply the hypergraph partitioning algorithm on it to finally obtain the data placement decisions. It has been shown that a hypergraph can model the request traffic at distributed sites and the desired performance metrics of data placement, which accordingly facilitates the data placement. We would further show that the hypergraph sparsifier can approximately satisfy the need of data placement with a lower overhead. We propose a randomized heuristic to construct the sparsifier based on a formulated hypergraph, and also present the scheme which constructs the sparsifier through sketches in a distributed way. The challenges on aggregating the unsynchronized information from sketches representing distributed sites (or streams equivalently), are also addressed by the proposed interactive protocol between the controller and sites.

Overall, to conquer the efficiency issues in making data placement decisions, we propose a unified scheme utilizing multiple recent advances in the related fields, including graph sparsifiers, sampling sketches and counting sketches. In terms of the contributions, 1) our work first proposes to improve the data placement through sketches; 2) we justify the effectiveness of data placement based on hypergraph sparsifier; counting and sampling sketches are jointly utilized in constructing the sparsifier and a novel method of coordinating distributed sketches is proposed to avoid the synchronization issue; 3) some valuable numerical results about applying sketches on the practical problem are obtained, which are missing in the literature.
1.3.3 Online Data Placement

It has been witnessed that the time consumed when moving data in a distributed system could be the main bottleneck in terms of job finish time [46]. Several schemes have been proposed, trying to lower the impact of data transfer in job execution. For example, [90] proposed that by placing data items used in a job at the nodes in the same rack, inter-rack traffic can be largely reduced and thus the job finish time is shortened. In most traditional schemes, the relationships between the performance metrics and the factors that affect them are used to design a model manually, which is used to guide the data placement. However, these fixed models would be less effective if some hidden factors are missed in consideration, e.g., unreliable links, or the discovered relationship is changed because the user request patterns or system configurations evolve in the future.

Different from existing methods, we adopt a generic and simple system model and propose a solution that needs fewer assumptions about the system properties. Our objective is to create an intelligent scheme that automatically learns the optimal locations for storing data items or replicas through trials and feedback. Specifically, when a data item is to be written into the distributed storage system, the storage locations for the item or its replicas are dynamically determined by our proposed scheme **DataBot**, which is based on reinforcement learning fundamentally. The decisions help to lower the read/write latency and are only based on the end-to-end measurements for data flows between each pair of nodes.

When the data storage system is treated as a complex environment, **DataBot** can be considered as an agent interacting with this environment. The agent continuously makes actions of choosing the storage location for each data item, and collects the feedback from the environment, including the current state of request patterns and network conditions, and the resultant read/write latencies due to some actions. Through this process, the agent can learn how to make a better choice on data placement. We adopt the Q-learning method [79] to model the process of reinforcement learning. Our formulation may result in a large state space to maintain, so neural network techniques are introduced to lower the space complexity of the proposed scheme and make the convergence faster.

In this work, we also discuss how the system should be designed from the perspective of implementation, so less overhead would be introduced by the proposed scheme itself. Specifically, we divide the system into two components: the production and
training system. By decoupling the two components, the time taken in the training process will not affect how fast a read/write request can be served by the metadata server. This changes the traditional workflow of reinforcement learning, which updates the model after each decision, but it is still shown to be effective according to the experiment results. Besides, we also address how the state information should be maintained for a higher efficiency of the system. Several emulation-based experiments are conducted to validate the scheme. When we set the objective as focusing on optimizing the write performance, the average write latency can be decreased up to 55.7%, compared with the standard hash-based implementation. When the read performance is emphasized, the read latency can be decreased up to 31.9%.

The contributions of this work are summarized as follows: 1) a generic scheme of dynamic data placement at the time of data writing or updating is presented, which introduces no overhead of data migration and is robust even if the correlation model between the performance and the affecting factors, such as request patterns and network conditions, keeps changing; 2) the reinforcement learning method is adopted in addressing the problem, its deficiency in delaying the decision process is overcome by our implementation design; 3) the work makes pioneering attempts in solving the resource management problem with the machine intelligence, and it is suggested that the approach can potentially be applied to other problems in the field of resource management and control for distributed systems.

1.3.4 Memory Cache Cluster Auto-scaling

Today distributed memory cache clusters are broadly used in different large-scale networked systems to deal with the heavy workload. It has become an important building block of most cloud-based services today and therefore has been offered as a product by the cloud providers, e.g., Amazon ElastiCache [2] and Memcached [8]. Technically, the distributed memory cache service provides temporary key-value storage in memory and joins the memory of distributed servers as a whole, functioning as a unified cache covering the entire key space.

To lower the energy cost of the memory cache clusters is important and necessary. On one hand, the memory cache clusters have been broadly used in different Internet applications, and they are always large in scale, e.g., it was shown in [58] that the largest memory cache cluster in 2012 has already contained more than 800 servers. On the other hand, with the growth of scale, the cost of powering these servers becomes
a burden to service providers. Stated in [20], the power cost may take up to 50% of the three-year total cost of owning a server. Different from the work [20, 25, 69] trying to lower the power cost through utilizing the low-power embedded CPUs, our work is from another perspective, i.e., how to schedule the cluster to avoid the over-provisioning of resources and therefore save energy. With our efforts, not only the service providers could gain monetary saving on resource allocation, but also the whole society can earn some benefits, such as the lower carbon dioxide emission through the energy-efficient design.

The reasonable control of cluster scaling, which leads to a tradeoff between the energy cost and performance goals (which include the service latency and cache hit rate), is the main objective of this work, termed as dynamic server provisioning. The idea here can be simply explained as to dynamically consolidate the workload of the discussed cache service to fewer servers. We adopt the stochastic network optimization framework [78] to ensure the queue stability while taking the other performance goals (i.e., energy cost and cache hit rate) into account. Note that our work is not designed to purely earn energy benefit from performance sacrifice, instead, we make the joint considerations of both performance and cost in the method design, which results in that the solution can support diversified preferences of different operators and therefore is more practical.

Request dispatching is another issue that will be addressed. We will discuss where to dispatch a request if there are multiple servers available and what is the favorable amount of content to be batched in a request, with the queue backlogs at that time slot as the input. These decisions will also contribute to the queue stability and system efficiency because they can affect the workload directed to the servers. Since the applied framework only requires the information of the current queue backlogs when making a decision, the scheme proposed in this work does not rely on the prediction of the future workload, which makes it easy to apply and less vulnerable to the unpredictability of the workload.

There exist some studies on improving the resource provisioning and task scheduling in distributed systems, either using the framework of stochastic network optimization [74, 102, 116] or through some other approaches [45, 113, 118]. Compared with them, ours is characterized by its specific modeling and scheme design towards the distributed cache services. For example, in the cache services, a request can only be dispatched to some specific servers because consistent hashing is applied and a higher cache hit rate is favored. Another issue is that the change of server on/off
status can influence the responsible key space of servers still being active as well as the resultant cache hit rate, which is crucial to the effective throughput of the system. Meanwhile, the diminishing overhead in batching requests was observed before, but was not considered in the cluster scaling problem.

In this part of the work, the distributed cache service is firstly modeled as the DHT-based server groups with the consideration of cache hit rate, diminishing overhead and cache warm-up time. Then we formulate a stochastic network optimization problem, which aims at optimizing the queue stability, energy cost and cache hit rate through the control of cluster scaling and request dispatching. It is transformed to a minimization problem given the queue backlogs at each discretized time slot, which is further addressed through the proposed online algorithm, while dynamic programming is utilized to lower the computational complexity. Besides, in the proposed algorithm based on dynamic programming, the time complexity of obtaining the solution is ensured to be polynomial.

1.4 Dissertation Organization

The outline of this dissertation is as follows.

Chapter 1 contains a statement of the background and problems addressed in the dissertation followed by an overview of the structure of the dissertation itself.

Chapter 2 presents a generic scheme of offline data placement in geo-distributed storage systems. Exploiting the proposed hypergraph models, it improves various performance metrics and lowers the system costs. Through the scheme, more data requests are fulfilled at the source locations of data flows, and fewer storage nodes will be used to fulfill a data request involving multiple data items.

Chapter 3 describes how the hypergraph-based scheme for data placement can be improved through sketch-based algorithms. With overheads in implementing the hypergraph-based scheme considered, the method to construct hypergraph sparsifiers under streaming models is proposed. It is validated that the overhead of the scheme itself is largely lowered while a close approximation is maintained.

Chapter 4 presents how to dynamically adjust the storage locations of data items in a distributed storage system. The proposed scheme is based on reinforcement
learning, aiming at lowering the average data read/write latencies by continuously interacting with the system, i.e., the control of storage locations and the measurement of resultant performance metrics.

**Chapter 5** describes the problem of dynamic provisioning and request dispatching in distributed memory cache clusters. The clusters can effectively improve system performance, but may introduce some unnecessary running cost. We suggest to enhance the system by auto-scaling the cache cluster with the proposed algorithm, following the stochastic optimization framework.

**Chapter 6** contains a restatement of the claims and results of the dissertation. It also enumerates possible future work about the proposed schemes and their applications.
Chapter 2

Hypergraph Models for Data Placement

2.1 Overview

Large-scale data-intensive applications mostly need to address a common problem of how to properly place the set of data items in geo-distributed storage nodes. Traditional techniques use the hash-based method to achieve load balancing among nodes such as those used in Hadoop [5] and Cassandra [14], but are not efficient for the requests reading multiple data items in one transaction, especially when the source locations of requests to the same data item are also distributed. Some recent papers [83, 96, 99] proposed managed data placement schemes for online social networks, but have a limited scope of application. In this chapter, a general framework of hypergraph-based data placement is proposed, which improves the data locality and lowers the system cost.

Starting with a simple scenario without replicas, the fundamental methods in the framework of hypergraph-based data placement will be presented, including hypergraph modeling and hypergraph partitioning. The hypergraph modeling is about the methods to convert the optimization objectives into hypergraph models and the hypergraph partitioning is used to efficiently partition the set of data items and place them in distributed nodes. Due to our formulation of edges in the hypergraph, the metrics supported by the framework fall into two categories: a) the associations among data items, which consist of the system time from the perspective of running efficiency and the distributed execution overhead reflected by the network load, and b) distances
between data items and nodes, which can reflect the inter-datacenter traffic, sum of access latencies and cost of storage. Further, we consider the scenario with replicas where a certain number of replicas are allowed for each item. The existence of multiple replicas in different locations introduces the routing decision problem, because any of the corresponding replicas can be used to fulfill the request. A multi-round scheme that iteratively makes the routing decision and replica placement decision is proposed. Through extensive experiments based on trace-based datasets, we evaluate the performance of the proposed framework and demonstrate its effectiveness.

2.2 Related Work

Due to the availability of geo-distributed datacenters, there exist multiple choices in selecting the location to store data or the destination to fulfill user requests. Besides the system cost related issues, such as the capital expense [1] or electricity price [72], the logical distance between the user and serving node is among the most important considerations. In [15], Agarwal et al. presented the automatic data placement across geo-distributed datacenters, which iteratively moves a data item closer to clients and the other data items with which it is associated. In [93], the replica placement in distributed locations was discussed with the Quality of Service considerations. In [87], Rochman et al. investigated how to place the contents or resources in a distributed system to serve more requests locally for a lower cost. Xu et al. [107] solved the workload management problem, in order to maximize the total utility of serving requests minus the cost, which is achieved through the reasonable request mapping and response routing. Corbett et al. [39] proposed a scheme for globally distributed data storage, where applications can specify constraints to control which datacenter holds which data, and how far the data is from its users (to control latencies). Wu et al. [106] also tried to balance the cost and latency of data/replica access by storing data in geo-distributed clouds. Huguenin et al. [56] crawled a large dataset related to the access patterns of YouTube and proposed a scheme to proactively place videos close to the expected requests. Shankaranarayanan et al. [88] presented a model that helps application developers optimize the latency perceived by application users through determining the number and locations of replicas as well as the underlying consistency parameters.

Besides fulfilling the request at a location near the user, there are other factors that affect the system performance, such as the multi-get hole effect, first discussed
in [6]. It introduces the problem of how to co-locate the strongly correlated data items through managed data placement. Raindel et al. [86] proposed to create more replicas of data items to increase the chance of serving more requested items in one node. Nishtala et al. [80] mentioned that frequent request patterns can be discovered through trace analysis, and each group of items frequently accessed can be treated as a whole in the distributed storage. Want et al. [104] discussed how to co-locate the related file blocks in the Hadoop file system. Golab et al. [51] considered the challenges of extensive data migrations in distributed object stores and studied the problem of determining the data placement strategies that minimize the data communication costs incurred by join-intensive queries. Quamar et al. [84] proposed an improved solution to partition the data items into sets, exploiting a way to more efficiently partition hypergraphs [65] through compression. Eltabakh et al. [46] discussed the possibility of reducing the data shuffling costs and network overhead through the effective data partitioning and proposed a generic mechanism that allows applications to control data placement at the file system level which subsequently helps to co-locate corresponding replicas on the same data nodes. The aforementioned work did not consider the importance of fulfilling the request locally and was unaware of the difference among locations, which resulted in many requests inefficiently served at a remote node.

Recent research has studied the data and replica placement in OSN, which favors to place data items from close friends together. Pujol et al. [83] showed the necessity to co-locate the data of a user and the friends of that user and proposed a dynamic placement scheme. Turk et al. [99] discussed how the user data and replicas in OSN could be partitioned to reduce the query space while maintaining load balance. Liu et al. [71] stated that different data items from the same user may be requested with a heterogeneous rate or pattern, and the method to determine the proper number of replicas for each data item was given. Traverso et al. [96] exploited the social information to selectively replicate the data of a user to the followers of the users, in order to lower the wide area network traffic while meeting the quality of experience constraints. In [60], Jiao et al. summarized the relationships of entities in the OSN system and proposed the multi-objective data placement scheme. The friendship-based one-to-one data relationship discussed in these papers can be considered as a special case of the multi-data association in our modeling.

The utilization of fixed vertices in the hypergraph model is one of the most important methods utilized in the proposed solution. A general solution of hypergraph
partitioning with fixed vertices was proposed in [23]. It has various applications in different fields or problems, e.g., load balancing [30] among processors or load rebalancing through migration [32, 34]. In terms of the utilization of hyperedges in the hypergraph models, it features on modeling the queries that span multiple data instances, and has been used in text retrieval systems [29] and web crawling [98]. Compared with such existing work, although the same hypergraph function is also utilized in our work, the problem that we are trying to address introduces some new challenges, such as considering the heterogeneities of different locations, which at least include the costs, the performance and the surrounding users of different geodistributed datacenters. In [33], the heterogeneity was considered in satisfying user preferences, but not exploited to optimize the overall performance of the system.

We also discuss multi-replica placement, which is extended from the single replica scenario. In the literature, Kayaaslan et al. [66] discussed how to improve the search quality, query latency or workload in a geo-distributed search system, by replicating the data from its local storage to other locations. Replica placement is also discussed in [54], but it is based on an existing data placement and tries to improve it through adding extra replicas. In [60], the replicas considered are also treated as masters and slaves. Differently, our work does not try to replicate a data item that has been previously stored in one location to other locations; instead, we try to simultaneously find multiple storage locations for the same item.

In general, the problem that we formulate is under the setting of cloud-based geo-distributed infrastructures. It brings new perspectives to the system design, at least including: each datacenter or region could provide nearly unlimited resources to a single cloud user; the datacenters of different regions are heterogeneous in performance and cost; the distribution of end users around the datacenter becomes an important factor in choosing data storage locations. Also, since our design target is a general-purpose data storage system, the problem has a different form of inputs from the existing work and requires the designated outputs. Therefore, although the hypergraph models have been broadly utilized as a tool for solving many location-related problems, the work done here is still meaningful and useful because of the contemporary and intricate problem formulated.
Figure 2.1: Problem inputs: (a) request pattern set $P$; (b) request rate set $R$

### 2.3 Modeling Framework

#### 2.3.1 Data Items and Nodes

We use a set $X$ of $M$ data items to represent the data stored in the system. Depending on the actual type of data storage, the data items can be files, tables, fragments or segments in practice. We generally assume homogenous size of data items below, if the discussed metric is not related to the data size; for the metrics where data size matters, we will discuss the impact of data size accordingly. Each single transaction/request from users would involve no more than $I$ different items from the set $X$. We denote the space of request patterns by $\mathcal{P} = \{X, \emptyset\}^I$ and denote a request pattern or an itemset by $p$. The requests in a practical system actually fall in a subset of the space, denoted by

$$P \subset \mathcal{P} = \{X, \emptyset\}^I.$$  

(2.1)

As illustrated in Fig. 2.1h, the example system contains 5 different data items and 3 request patterns, which are $(1, 2, 3)$, $(1, 4)$ and $(4, 5)$. The paradigm of accessing multiple data items in one transaction has many applications. For example, news feed updates in OSN involve the data of multiple users. In data analysis systems, the output is made through combining or processing multiple data files, each extracted from a different data source, possibly remotely distributed.

We consider the scenario that data items are stored in geographically distributed datacenters, represented by a set $Y$ of $N$ nodes. Compared with the centralized storage, the distributed storage can improve the data access latency and achieve a higher level of fault tolerance. We term a datacenter as a node or a location interchangeably.
in the following. As illustrated in Fig. 2.1b, we place 5 data items in 2 datacenters. There are 3 data request patterns: (1, 2, 3), (1, 4) and (4, 5). The links between datacenters and request patterns represent which datacenter will have which request patterns. The figure does not illustrate where data are placed, which is to be discussed later. The estimated request rate of each request pattern from each datacenter is assumed to be known in advance and used as the input of our scheme. In practice, the request rates can be predicted from the history records with the method of Exponential Weighted Moving Average (EWMA) [57]. In this chapter, we will not discuss which specific node in the datacenter will be utilized for storage, which will be considered in Chapter 4.

2.3.2 Data Placement

Initially, we consider each data item \( x \in X \) is stored at a unique location \( y \in Y \). Thus the data-to-location mapping function is defined as

\[
D : x \rightarrow y ,
\]

(2.2)

which specifies the storage location \( y \) of each item \( x \). Fundamentally, our work focuses on designing the placement scheme that provides a reasonable solution of \( D \). Besides, we use \( D_y \) to represent the set of data items stored in node \( y \) after the data placement decision. Data de-duplication is an effective way to lower the storage cost assuming that a lot of data items are same or similar. It is not considered in our current design, but by properly modeling the associations among data items in the resultant dataset after data de-duplication, our scheme can be applied to further lower the system cost.

In the state-of-the-art implementation of such a function, the hash-based methods are widely adopted, such as those in HDFS and Cassandra. This is due to the fact that at the time of their design, the main concern was to achieve load balancing across nodes. Although some other policies affect data placement, such as avoiding placing the replicas of a data item in the same datacenter in order to improve the fault tolerance, the schemes can still be understood as random placement, as claimed in [46]. Obviously the hash-based schemes did not pay enough attention to the system performance affected by the data locations and ignored the potential performance improvement through managed data placement. To overcome this deficiency, especially under the settings of geo-distributed storage nodes, we present a general hypergraph-based framework for data placement, which not only considers the storage balance,
but also improves various performance metrics achievable through the managed data placement.

2.3.3 Workload Modeling

We assume the request from a client is always directed to the datacenter closest to the location of the client, termed as access datacenter, which in practice is usually achieved by the geographic support of DNS lookups in CDNs. The client can be located outside or inside the datacenters that store the data. When a human user is using the service, the user is always from locations outside. Another case is that the running jobs or processes on the machines in the datacenter can also request data items. We ignore the performance and cost issues along the path between a client and its access datacenter in this work. Without loss of generality, we consider the access datacenter as the source location of the request. So datacenters or nodes in our modeling have two roles simultaneously: the source location of requests and the destination location holding the stored data. To differentiate the role of a node \( y \) being the source or destination of flows, we may use \( s \) or \( d \) below, respectively.

The workload or request rate of each pattern \( p \in P \) from the requesting node \( y \in Y \) can be measured, denoted by \( R_{py} \). We use the predicted request rates as the input of our scheme to make the data placement decision. There exist mature methods in predicting the rates from a series of rate measurements in the previous time slots, such as EWMA \([57]\). The practical request rates may deviate from the prediction, but we can only rely on the prediction to optimize the system at the current stage, which mostly covers the general case. Below, the notations used for the predicted rates are the same as the measured ones for simplicity. We denote the workload or request rate set as

\[
R = \{ R_{py} | p \in P, y \in Y \}. \tag{2.3}
\]

In the example of Fig. 2.1b, the request rate set \( R \) is illustrated as a bipartite graph, where datacenters and request patterns are the two sets of vertices, and the edges between these two sets are weighted by the rate \( R_{py} \).

To facilitate the proposed hypergraph-based data placement framework, two types of refined request rates can be defined based on \( R \). One is the total request rate to a data item \( x \) from a source node \( y \). Formally, for each data item \( x \), we can calculate
its total request rate at each source node $y$ by

$$R_{xy} = \sum_{p \in P} R_{py} 1(x \in p) .$$

(2.4)

where $1(x \in p)$ indicates whether the data item $x$ is a member of the pattern $p$, returning 1 if true or 0 otherwise. Another is the total request rate to a pattern $p$ regardless of the source location. Formally, for each pattern $p$, we calculate its total request rate by

$$R_p = \sum_{y \in Y} R_{py} .$$

(2.5)

Note that below we denote the set of $R_{xy}$ and $R_p$ by $\mathcal{R}_{xy}$ and $\mathcal{R}_p$, respectively.

### 2.3.4 Problem Formulation

In the considered storage system, with the knowledge of the workload, we can make the data placement decisions to achieve certain objectives on the system cost, performance and efficiency. The formulation of data placement problem can be generalized as: given the workload $I = \{P, R\}$, find the optimal placement solution of $D : x \rightarrow y$ that minimizes an objective function defined on $I$ and $D$, subject to the balance constraint or the expected storage size distribution. The balance constraint ensures that the worst-case recovery time upon site failure is constrained. Besides, it helps to avoid hotspots in the distributed storage system. For the expected storage size distribution, it considers the unbalanced case and its detail is clarified in Section 2.4.1.

A well-defined objective function should reflect the preferences of the system operators. Under our current framework, the objective function is defined as a linear function, where the placement decisions $D$ are the variables and the workload $I$ determines the coefficients. A lot of metrics that help to evaluate the overall performance of a data placement scheme can be captured by the objective function, such as improving the system time of fulfilling requests or lowering the inter-datacenter traffic. If there are multiple metrics to be optimized simultaneously, they can all be considered through defining the objective function as a weighted sum of the metrics. We will discuss the considered metrics under our proposed framework in Section 2.4.2.
2.4  Hypergraph-based Data Placement

2.4.1  Hypergraph-based Framework

(a)  Hypergraph Model

We start by showing that without the considerations of data replicas, the optimization of data placement can be formulated as an $N$-way hypergraph partitioning problem. Note that in the existing work that also used hypergraph to model the relationship among multiple items, e.g., [84], the exact data location was not addressed or emphasized because of the less attention to the difference between locations.

A hypergraph $H(V, E)$ is a further generalization of a graph, i.e., the hypergraph allows each of its hyperedges to involve multiple vertices while the edge of an ordinary graph can only involve two vertices at most. This feature can be used to model the association among a group of entities, such as the friends of a user in OSN. In the scheme, we set up the vertex set $V$ with all the data items and all the nodes in the considered system, such as

\[ V = X \cup Y . \]  

The hyperedge set $E$ represents all the request patterns and all the pairs between each node and each data item. Therefore, it is defined as

\[ E = \{ e_p | p \in P \} \cup \{ e_{xy} | x \in X, y \in Y \} . \]  

For each request pattern hyperedge, it involves multiple data items and that is the main reason for introducing hypergraph. Each hyperedge $e \in E$ is assigned a weight to capture the desired performance metrics of data placement. The setting of weights should be in a reasonable way so that the solution of data placement based on the hypergraph model is meaningful, whose details are shown in Section 2.4.2.

An example of formulating the problem as a weighted hypergraph is illustrated in Fig. 2.2. In the hypergraph, there are two types of vertices: storage node (square) and data item (circle), which represent the fundamental entities in our system modeling. There are two types of edges, the request pattern hyperedge (dashed circle) and the data-node hyperedge (solid line), which represent the relationships between entities. We may use the term edge to refer to hyperedge below. For the request pattern hyperedge (dashed circle), it consists of multiple vertices and reflects the connection among a group of data items, e.g., news feed updates in OSN involve the data of
multiple users. For the data-node edge, it reflects how frequently a data item will be requested from a source node.

(b) Hypergraph Partitioning

An $N$-way hypergraph partitioning is to partition the vertices into $N$ output sets, such that each vertex only belongs to one of the $N$ output sets. The cost of the partitioning is defined as the sum of the cut sizes of all edges in the hypergraph, denoted by $H$. The objective of hypergraph partitioning is defined as to minimize $H$. A hyperedge $e$ would introduce some cost to the partitioning if its vertices fall into more than one output sets or partitions; the cost or cut size of edge $e$ is denoted by $H_e$ and counted as $H_e = (t-1)w_e$ if its attached vertices fall into $t$ sets, where $w_e$ is the weight of edge $e$. This is the connectivity metric defined in the classical hypergraph partitioning.

The overall objective of the $N$-way hypergraph partitioning is to minimize the cost of the partitioning. Note that the cost is related to not only the defined weights of hyperedges, but also the partitioning results, which are indeed the data placement results here. So if we properly set the weight of hyperedges, the purpose of achieving the performance objectives on data placement could be equivalent to the minimization of the cost of the partitioning. We have two kinds of edges in the formulated hyperedges, and in Section 2.4.2 we show how to set their weights respectively, through which the hypergraph model captures the desired performance metrics.

The $N$-way hypergraph partitioning has been shown to be NP-Hard, but different heuristics have been developed to solve the problem approximately, because of the wide applications of the hypergraph partitioning, such as in VLSI, data mining and bioinformatics. The PaToH tool [31] is what we use to partition the formulated hypergraph. The general steps of the algorithm in PaToH [31] are as follows: 1)
simplify or compress the initial hypergraph into smaller and smaller scales gradually; 2) solve the partitioning problem on the smallest scale graph; 3) gradually recover the partitions into the larger scale graph with refinements. The simplification process may eliminate the chance of placing some vertices in the same set, but people try to mitigate the negative effect through well-designed heuristics. Because our designed scheme is based on the hypergraph partitioning tool \cite{31}, so it is non-deterministic and can be suboptimal.

(c) Fixed-location Vertices

Supported by existing hypergraph partitioning tools, we can preassign some vertices to the $N$ output partitions before applying the hypergraph partitioning algorithm, i.e., they are fixed-location vertices. We denote the fixed location set by $F$ below. In the scheme, each of the $N$ nodes is preassigned to a different set before the partitioning, but the locations of data items are flexible. Besides, each data vertex is connected to all $N$ nodes to reflect that all nodes are available to store the data item. With these settings on the inputs of the partitioning, after the partitioning, we obtain where to place data directly from the $N$ output sets or partitions, because each node and the data items stored in it would fall in the same set.

(d) Storage Size Balance

In hypergraph partitioning, a balance ratio $\epsilon$ can be set as an input parameter, in order to control the balance of the total weight of vertices falling into the resultant partitions. We can easily utilize this feature to achieve the balance of the number of items stored in different locations. Given the input parameter $\epsilon$, the partitioning heuristic \cite{31} will try its best to ensure that the number of items stored in each location will be in the range of $[(1 - \epsilon)h_a, (1 + \epsilon)h_a]$, where $h_a$ is the average number of items for each datacenter among all datacenters, unless it is unachievable.

(e) Satisfying Specific Storage Distributions

Sometimes the administrators of storage systems may have different preferences on setting the distribution of storage size at different geographical locations. For example, we found that in a location-based Internet service shown in Section 2.6, the user request rates from different regions of the world are much skewed, which in turn may request for a skewed storage size distribution among the datacenters or regions, e.g.,
the storage size distribution could be set as similar to the request distribution. In such a case, a general solution for our scheme is that it accepts a ratio of expected storage sizes at different regions as input, denoted by $\Theta = \{\theta_1, ..., \theta_N\}$. Then the scheme tries to place data items according to the ratio, i.e., the final ratio of the numbers of items stored at different locations would be similar to $\Theta$. To do that, we would pre-set a constraint to the expected total weights of partitions. The weight of a partition is calculated as the total weight of vertices falling into the partition. Specifically, we set the weight of all vertices as 1, and add the constraint of partition weights so that the weight of each partition is proportional to its ratio in $\Theta$.

2.4.2 Considered Metrics

Data placement can affect the distributed storage systems in both the system-level performance metrics and user-perceived service qualities. Various performance and cost metrics can be incorporated into the proposed hypergraph-based framework. We focus on a read-intensive scenario, so only read-related performance and cost issues are supported and discussed under our current framework. Furthermore, depending on the preferences of cloud system operators, multiple metrics can be optimized simultaneously through properly setting the weights among them.

Because the data placement scheme among geo-distributed datacenters is designed to optimize some long-term performance objectives by offline algorithms, so the modeling is based on the average value of metrics. To handle the small variation of request patterns or the dynamics of performance affecting factors such as the end-to-end latency, certain dynamic flow scheduling schemes are more effective, and thus can be a supplement to the discussed offline data placement. Below we show what metrics are considered under the current framework and how they can be related to the hyperedge weight in the hypergraph model.

(a) Associations among Data Items

System time [S]. The system efficiency is characterized by the necessary system time to fulfill the given workload. According to the observation of [6], in distributed systems, the average system time of a request is not only related to the amount of information accessed, but also related to the number of distributed nodes involved due to the processing overhead at each node. Denote the span of a request $p$ by $S_p$, representing the number of items requested by it, and denote the number of items in
request \( p \) that are fulfilled at node \( y \) by \( S_{py} \), since a single node may not be able to provide all the items in \( p \). Their relationship is given by \( \sum_{y \in Y} S_{py} = S_p \). Note that \( S_{py} \) is a variable determined by the mapping function \( D \) and \( S_p \) is a constant. We model the necessary system time to partially or fully fulfill a request \( p \) at node \( y \) as

\[
t_{py} = S_{py} + \lambda \cdot 1(S_{py}) ,
\]

(2.8)

where the 0-1 function \( 1(S_{py}) \) indicates whether \( S_{py} \geq 1 \). The idea behind it is that the system time consists of the part proportional to \( S_{py} \) and the constant session overhead of handling the request, denoted by \( \lambda \). The latter is introduced by the routine operations in handling a request, such as the TCP connection establishment. Note that the metric here is more important to the small size data items due to the comparative difference between \( S_{py} \) and \( \lambda \cdot 1(S_{py}) \). With the request rates of different patterns, the total system time of fulfilling all the requests is

\[
\sum_{p \in P} R_p \sum_{y \in Y} t_{py} ,
\]

(2.9)

Minimizing (2.9) helps to lower the cost of the service provider, because under the pay-as-you-go payment model of clouds, the operational expense depends on the system time. A lower value of (2.9) can be achieved by the co-location of strongly associated data. For instance, in the extreme case, when placing all the items in a pattern \( p \) to the same location, the system time on \( p \) will be at the lower bound \( R_p(S_p + \lambda) \). Because \( \sum_{y \in Y} \sum_{p \in P} R_p \cdot S_{py} \) in (2.9) is a constant for any given workload, we can summarize the metric as

\[
A^{[S]} = \sum_{y \in Y} \sum_{p \in P} R_p \cdot \lambda \cdot 1(S_{py}) .
\]

(2.10)

By setting the edge weight of a hyperedge \( e_p \) through

\[
w^{[S]}_p = \lambda R_p ,
\]

(2.11)

to minimize the cut weight of hypergraph partitioning is equivalent to minimizing (2.10). Formally, the equivalence is defined as the theorem below.

**Theorem 1.** For any input \( I = \{P, R\} \), we can formulate it as a hypergraph \( (V, E) \), where the edge weights are set according to (2.11). From the optimal solution of par-
Partitioning the hypergraph with parameters \( \{P, N, \epsilon\} \), we can obtain the data placement \( D \). Denoting the cut size of the partitioning by \( H \), it satisfies that \( H = A^{[S]} - B \), where \( B \) is a constant regardless of the partitioning, and \( A^{[S]} \) is from (2.10) and based on \( D \). Meanwhile, \( H + B \) is the minimum achievable value on (2.10) for any data placement.

**Proof** The cut size of a request pattern edge \( e_p \) is denoted by \( H_p \). According to the definition of the cut of hyperedges, \( H_p = \left[ \sum_{y \in Y} 1(S_{py}) - 1 \right] \lambda R_p \). With (2.10), we can obtain \( A^{[S]} - \sum_{p \in P} H_p = \sum_{p \in P} \lambda R_p = B \), where \( B \) is a constant. Therefore, \( H = \sum_{p \in P} H_p = A^{[S]} - B \).

**Distribution overhead** \([D]\). For a lot of Map-Reduce transactions on distributed storage systems, the amount of information that needs to be transmitted over the network (due to the distributed storage of data) is related to the number of nodes involved. For example, a user may want to obtain the top \( K \) frequently visited web-links from the logs of requests distributed among nodes. Here the logs involved in a transaction can be considered as a request pattern \( p \) and each log can be considered as a data item \( x \). Assume there are \( N_p \) nodes necessary to be used, each holding some of the logs. To generate the overall top \( K \) links, each of the nodes first generates the top \( K \) links locally as a list. Note that here we assume the same size of outputs, which makes this metric only approximate. Denote the length of a list by \( \beta \). Then the logs are transmitted to the destination possibly with merging operations in the middle. During the process, at least \( N_p - 1 \) lists should be transferred. For such transactions, the storage system may have the objective of minimizing the necessary data transmission in the whole system,

\[
A^{[D]} = \sum_p R_p \cdot \beta \cdot (N_p - 1) .
\] (2.12)

For such an objective, we set the weight of hyperedge \( p \) by

\[
w^{[D]}_p = \beta R_p .
\] (2.13)

In fact, the objective (2.12) is equivalent to (2.10), ignoring the difference of parameter \( \lambda \) and \( \beta \). Because for each pattern \( p \), \( \sum_{y \in Y} 1(S_{py}) = N_p \), and the difference between (2.12) and (2.10) is a constant. Therefore, Theorem 1 also applies to the metric of distribution overhead.
(b) Distance between Data and Node

The data location might also be critical to some other performance metrics of the system, such as the traffic cost, access latency and storage cost. Such metrics can be generalized as some cost functions defined on the data storage location and data access location. Our initial work [108] only considered the difference in terms of whether the requested data is stored in the same location of the request or not. Here the consideration is extended to the distance model from the initial boolean model. As a result, we need to model the unit cost when the data storage location is \(d\) and the data access location is \(s\).

We can define a data access flow by its source-destination location pair \((s,d)\), where \(s \in Y\) and \(d \in Y\). Consider a general cost metric \([\ast]\) due to the difference between \(s\) and \(d\), a unit cost parameter \(C_{sd}^{[\ast]}\) is defined at first for any cost metric type \([\ast]\). Then we denote \(O_{xy}^{[\ast]}\) as the cost of placing a data item \(x\) at location \(y\), which can be obtained for any considered cost metric \([\ast]\), given the parameters of \(C_{sd}^{[\ast]}\) and data placement decisions. Depending on the metrics discussed, we can obtain a series of optimization objectives in the form of

\[
L^{[\ast]} = \sum_{x \in X} \sum_{y \in Y} 1_{xy} O_{xy}^{[\ast]},
\]

subject to the constraint that data \(x\) is placed at exactly one of the nodes in \(Y\). It indeed can be further decomposed to minimize \(\sum_{y \in Y} 1_{xy} O_{xy}^{[\ast]}\) for each \(x\), due to the independence of data items for such an objective. Here our main problem is to determine the proper weight \(w_{xy}\) for the data-node edges \(e_{xy}\), so that the partitioning cut is equivalent to the data placement objective. For each data item \(x \in X\) in the system, we can obtain an equation set as follows:

\[
\begin{align*}
O_{x1}^{[\ast]} &= w_{x2} + w_{x3} + \ldots + w_{xN} \\
O_{x2}^{[\ast]} &= w_{x1} + w_{x3} + \ldots + w_{xN} \\
&\quad \ldots \ldots \\
O_{xN}^{[\ast]} &= w_{x1} + w_{x2} + w_{x3} + \ldots + w_{xN-1}
\end{align*}
\]

(2.15)

Each equation for node \(y\) calculates what the overhead is if we place data \(x\) in node \(y\). With the equation set (2.15), the weight between item \(x\) and node \(y\) can be calculated
as
\[ w_{x_d}^{[*]} = \sum_{y \in V} O_{xy} / (N - 1) - O_{xd} . \]  

(2.16)

Before showing the metrics that can be incorporated by the generalized data-node edge weight formula (2.16), we show that the cost of a partitioning by setting edge weights according to (2.16) is equivalent to the minimization objective of data placement (2.14).

**Theorem 2.** For any input \( \mathcal{I} = \{P, R\} \), we can formulate it as a hypergraph \((V, E)\), where the edge weights are set according to (2.16). From the optimal solution of partitioning the hypergraph with parameters \( \{P, N, \epsilon\} \), we can obtain the data placement \( \mathcal{D} \). Denoting the cost of the partitioning by \( H \), it satisfies that \( H = L^{[*]} \), and \( L^{[*]} \) is from (2.14) and based on \( \mathcal{D} \). Meanwhile, \( H \) is the minimum achievable value on (2.14) for any data placement.

**Proof** The cut size of a data-node edge \( e_{xy} \) is denoted by \( H_{xy} \). For any data item \( x \), it was connected to all the nodes in the formulated hypergraph. After the partitioning, it can be connected to only one node, because otherwise some sets in the partitioning result would be connected by \( x \), considering that we have assigned each node to a different set. Assuming the item \( x \) is finally connected with node \( f_x \), the sum of the cut sizes of data-node edges related to \( x \) is \( \sum_{y \in Y} H_{xy} = \sum_{y \in Y/f_x} w_{xy}^{[*]} \).

After placing the data items according to the partitioning, with (2.14), we obtain the metric \( L^{[*]} = \sum_{x \in X} \sum_{y \in Y/f_x} w_{xy}^{[*]} \). Therefore, \( H = \sum_{x \in X} \sum_{y \in Y} H_{xy} = L^{[*]} \).

**Inter-datacenter Traffic \([T]\).** When the requested data item is not placed at the source or requesting node, inter-datacenter traffic occurs. Denote the unit cost of the inter-datacenter traffic from node \( s \) to node \( d \) as \( C_{sd}^{[T]} \), which is set by cloud providers, relating to the distance and the capacity of the communication path. To minimize the total cost of the inter-datacenter traffic, we are to minimize

\[ L^{[T]} = \sum_{x \in X} \sum_{y \in Y} 1_{xy} O_{xy}^{[T]} , \]  

(2.17)

where

\[ O_{xd}^{[T]} = \sum_{s \in Y/y} C_{sd}^{[T]} R_{xs} . \]  

(2.18)

By applying \( O_{xd}^{[T]} \) to (2.16), the corresponding edge weights are determined. Note that if \( C_{sd}^{[T]} \) is homogenous for any pair of nodes, we would have \( w_{xd}^{[T]} = R_{xd} \), which
degenerates to the metric of localization that we proposed in [108]. To handle the heterogeneous sizes of data items, $R_{xs}$ in the modeling should be divided by the data item size.

**Sum of Access Latencies [L]**. Consider that we are to optimize the sum of latencies on retrieving all the requested data items. We assume that the retrieving latency from node $s$ to node $d$ is a constant $C_{sd}^{[L]}$, as the propagation delay dominates the latency of long-distance transmissions under the setting of geo-distributed datacenters. For all network-based metrics considered in this scheme, we ignore the occasional fluctuation of network performance, e.g., link contention, because the scheme is designed to optimize the average data placement performance for a long time. The metric is defined as to minimize the total latency

$$L^{[L]} = \sum_{x \in X} \sum_{y \in Y} 1_{xy} O_{xy}^{[L]},$$  \hspace{1cm} (2.19)

where

$$O_{xd}^{[L]} = \sum_{s \in Y} C_{sd}^{[L]} R_{xs}.$$  \hspace{1cm} (2.20)

Similar to the case of minimizing the inter-datacenter traffic, to minimize (2.19), the edge weight $w_{xd}^{[L]}$ should be calculated through applying $O_{xd}^{[L]}$ to (2.16).

**Cost of Storage [C]**. Considering the different storage cost of the same data at different locations, we model the unit storage cost of an item $x$ at location $d$ by $C_{xd}^{[C]}$. Therefore, to minimize the total storage cost

$$L^{[C]} = \sum_{x \in X} \sum_{y \in Y} 1_{xy} O_{xy}^{[C]},$$  \hspace{1cm} (2.21)

where

$$O_{xd}^{[C]} = C_{xd}^{[C]}.$$  \hspace{1cm} (2.22)

we would calculate the edge weight $w_{xd}^{[C]}$ through applying $O_{xd}^{[C]}$ to (2.16). To handle the heterogeneous sizes of data items, $C_{xd}^{[C]}$ in the modeling should be divided by the respective data item size.

**(c) Multiple Objectives**

When the system considers multiple objectives, we can set the weight of each edge in the hypergraph to the weighted sum of considered metrics. First, we clarify that we
optimize a multi-objective performance metric

\[ C = \mathbf{W} \cdot (A^S, A^D, L^T, L^L, L^C)^T \]

(2.23)

where \( \mathbf{W} \) is the weight vector of the five metrics. Under the multi-objective scenario, we would set the hyperedge weights as the weighted sum of individual metrics, such as

\[
\begin{align*}
\mathbf{W} \cdot (w_p^S, w_p^D, 0, 0, 0)^T, & \quad \text{for each hyperedge } e_p \\
\mathbf{W} \cdot (0, 0, w_{xy}^T, w_{xy}^L, w_{xy}^C)^T, & \quad \text{for each edge } e_{xy}
\end{align*}
\]

(2.24)

Setting weights according to (2.24) will minimize the (2.23), which will be proved in Theorem 3.

Note that through (2.16), the calculated weight value for some data-node edges is possibly negative, however, almost all the hypergraph partitioning algorithms only support non-negative hyperedge weights. In such a case, for all the data-node edges, we can increase all of the calculated weights by a certain drift \( \xi \) to make all the edge weights positive, such that \( \xi = \min \{w_{xy}|x \in X, y \in Y\} \). This will not affect the optimality of the formulation, because before the partitioning, we add an edge from each data item to each node, and after the partitioning, exactly one edge will be kept. That means no matter what the partitioning results are, the obtained cost is always drifted by a constant value, which will be proved in Theorem 3. So we have

\[
\begin{align*}
\mathbf{W} \cdot (w_p^S, w_p^D, 0, 0, 0)^T, & \quad \text{for each hyperedge } e_p \\
\mathbf{W} \cdot (0, 0, w_{xy}^T, w_{xy}^L, w_{xy}^C)^T + \xi, & \quad \text{for each edge } e_{xy}
\end{align*}
\]

(2.25)

Last, we conclude the optimality of our solution in the theorem below.

**Theorem 3.** For any input \( \mathcal{I} = \{P, R\} \), we can formulate it as a hypergraph \((V, E)\), where the edge weights are set according to (2.25). From the optimal solution of partitioning that hypergraph with parameters \( \{P, N, \epsilon\} \), we can obtain the data placement \( D \). Denoting the cost of the partitioning by \( H \), it satisfies that \( H = C - B \), where \( B \) is a constant, and \( C \) is from (2.23) and based on \( D \). Meanwhile, \( H + B \) is the minimum achievable value on (2.23) for any data placement.

**Proof** We first prove that under the settings of (2.24), the minimization of the formulated hypergraph partitioning is equivalent to the minimization of (2.23). For the multi-objective function (2.23), it is a linear combination of the considered metrics.
Because of the linear property of the function, we can use the linear sum of the weights for different metrics as the hyperedge weight in the formulated hypergraph. Besides, the cost of the partitioning is the sum of the cutsizes of all edges, so the minimization of the formulated hypergraph partitioning is equivalent to the minimization of (2.23). Then we prove that, when we increase all the data-node edges involved in a data item \( x \) by \( \xi \), as shown in (2.25), there is no impact on the equivalence above. The root cause is that at the beginning the data item \( x \) is connected to each node \( y \in Y \) and after the partitioning, it is connected to only one of them. Under such a change, both \( H \) and (2.23) will be increased by \((N - 1)M\xi\), where \( M \) is the number of data items and \( N \) is the number of nodes. Therefore the minimization of \( H \) and the objective (2.23) are still equivalent.

\[ \square \]

### 2.5 Placement of Replicas

The data placement problem is more challenging if the replication of data items is allowed. Here we will not differentiate the data item itself and its replicas, so both of them are treated as replicas below. The allowed number of replicas for each item could be given from a separate process [71], so the effect of replica number setting is not discussed in our scheme. Below we assume the number of replicas for each data item is \( k \), but the scheme still applies when the number is different for different items. Meanwhile, as the scheme is only designed to determine the storage datacenters for data items, we assume that for each data item, the number of replicas is no greater than the number of datacenters.

Because the locations of replicas need to be determined, the data-to-location mapping function is updated to

\[ D : x \rightarrow \{ y_1, y_2, ..., y_k \} \quad (2.26) \]

Meanwhile, we have to face the routing decision problem, since the request for an item \( x \) can be fulfilled at any location holding a replica of \( x \). We adopt the deterministic routing and represent the routing as a mapping function

\[ M : (x, p, y) \rightarrow y_d \quad (2.27) \]

which can give the routing destination \( y_d \in Y \) for each item \( x \) in a pattern \( p \) requested...
Algorithm 1 Data Placement with Replicas

1: $\mathcal{D} \leftarrow \text{Phase}1(\mathcal{I}) \quad \triangleright \text{Initial replica placement}$
2: $\mathcal{C} \leftarrow 0$
3: repeat
4: $\mathcal{M} \leftarrow \text{Phase}2(\mathcal{D}) \quad \triangleright \text{Routing decision}$
5: $R' \leftarrow \text{Func}R(\mathcal{M}, \mathcal{I}) \quad \triangleright \text{Obtain workload to replicas}$
6: $\mathcal{I}' \leftarrow \{R', P\} \quad \triangleright \text{Inputs in the replica space}$
7: $\mathcal{D} \leftarrow \text{Phase}3(\mathcal{I}') \quad \triangleright \text{Hypergraph partitioning}$
8: $\mathcal{C}_{\text{last}} \leftarrow \mathcal{C}$
9: $\mathcal{C} \leftarrow C(\mathcal{M}, \mathcal{D})$
10: until $\mathcal{C} - \mathcal{C}_{\text{last}} < \gamma$
11: return $\mathcal{M}, \mathcal{D}$

Note that the solution for the scenario with replicas should include both $\mathcal{D}$ and $\mathcal{M}$. Indeed, the routing decision should be made based on a given placement of replicas, however after the routing decision is made for all requests, the previously generated placement might be sub-optimal. This makes the problem more complicated and we will present our solution in the following.

To overcome the issues introduced by the replicas, we design the multi-round scheme as shown in Fig. 2.3. In phase (1), we address the initial placement of data replicas by a simple greedy method. In phase (2), the local routing decision is made for each request pattern from each node, considering the existence of replicas. Then the request pattern $p$ (involving a group of data items) attached with each request rate $R_{py}$ is refined towards a group of specific replicas. In phase (3), based on the refined request rates towards replicas, we can make the replica placement decision in the space of replicas. Phase (2) and (3) are applied repeatedly until the improvement is smaller than a threshold. The general steps of the whole scheme are shown in Algorithm 1. Details of each phase are given as follows.
2.5.1 Initial Placement

First we present a greedy method of generating the initial replica placement, which is illustrated as phase (1) in Fig. 2.3. For each data $x \in X$, we obtain the set $W_x = \{R_{xy} | y \in Y\}$, representing the request rates of data $x$ from different locations, and sort it in descending order. Based on the allowed number of replicas for data $x$, which is $k$ in our model, we choose $k$ nodes with the highest rate in $W_x$ to store the replicas of item $x$. This initial placement at least ensures that the resultant total relaying traffic of the system is minimized, which is better than a random initial placement. Although other metrics of data placement are not considered in this phase, the final performance of the scheme is not affected, since all metrics take effect in the later steps.

2.5.2 Routing Decision

A main problem introduced by allowing replicas is to make the optimal routing decision based on the current status of the replica placement, which is shown as phase (2) in Fig. 2.3. For a requested pattern $p$ at a source node $s$, we can optimize the replicas used to fulfill all the items requested in the pattern $p$. We formulate a binary optimization problem:

$$
\min \quad C^A \sum_{y \in Y} X_y + \sum_{x \in p} \sum_{y \in Y} X_{xy} C^L_y
$$

s.t. \[
\sum_{y \in Y} X_{xy} \cdot 1(x \in D_y) = 1, \forall x \in p
\]

$$
X_{xy} \leq X_y, \forall x \in p, \forall y \in Y
$$

$$
X_{xy} \in \{0, 1\}, \forall x \in p, \forall y \in Y
$$

$$
X_y \in \{0, 1\}, \forall y \in Y
$$

(2.28)

where $C^A$ is a constant $W \cdot (\lambda, \beta, 0, 0, 0)^T$ under the current placement and $C^L$ is also a constant $W \cdot (0, 0, C^T_{xy}, C^L_{xy}, 0)^T$. The optimal solution of (2.28) ensures the minimized value of (2.23) under any given replica placement. The binary variable $X_{xy}$ is used to represent whether an item $x \in p$ will be routed to the node $y$. The binary variable $X_y$ indicates whether the node $y$ is utilized (or active) in the routing of $p$. The constraints ensure that each item $x \in p$ is actually routed to one node holding the replica of $x$ and being active. The objective is to minimize the cost incurred by
the fulfillment of request $p$ from node $j$. Its first part is related to the number of nodes involved. The second part is related to the inter-datacenter traffic and latencies in fulfilling $p$. Indeed they would contribute to the objective \[2.23\].

If we consider the first part of the objective independently, the formulation is equivalent to the set-cover problem, being NP-Complete. Otherwise, if we consider the second part independently, the solution is quite trivial, such that for each item $x$ we can simply choose the node $y$ which minimizes $C_{sy}^L$. Due to the difficulty of the first objective, we borrow the idea of addressing the set-cover problem through linear programming relaxation. Specifically, we relax all the variables to real numbers in the range of $[0, 1]$. The number can be considered as the probability that the corresponding variable will be set to be 1 in the final solution.

Our method is designed as follows. First, with the relaxation, we obtain the solution variables in the form of probabilities. Note that the linear programming problem can be solved in polynomial time. Then, for each data $x \in p$, we choose its serving node by $\arg \max_{y \in Y} X_{xy}$, which can be considered as choosing the node that has the highest probability of serving $x$. Note when there is a tie among top values, a random one from them will be chosen. Compared with the traditional set-cover problem where only $X_y$ is used in obtaining the final solution, our method further considers the second part in the formulated objective function.

Representing the complexity of the problem by the number of variables, the complexity of the formulated integer linear programming problem is in $O(MN)$, where $M$ is the number of data items and $N$ is the number of datacenters. Because in our scenario the number of items is very large, the formulated integer linear programming problem cannot be solved in reasonable time. Therefore, we follow the relaxation-based method in our implementation. The tool that we use to solve the linear programing problem is \texttt{lp.solve} [7].

### 2.5.3 Placement Decision

The placement decision is obtained by extending the solution for the case without replicas. We denote the set of replicas by $Z$ and denote a replica by $z$.

After phase (2), because the routing decision $\mathcal{M}$ is obtained, we can determine the request rate to each replica. We refine the workload set from $R = \{R_{py}\}$ to $R' = \{R'_{py}\}$, which is shown as $FuncR$ in Algorithm 1. The difference between $p$ and $p'$ is that $p'$ is obtained in the replica space. Formally, $p' \in \{Z, \emptyset\}^I$. Specifically, $p$
can only indicate whether a data item $x$ is in the pattern $p$, but $p'$ indicates which specific replica of each item $x \in p$ is actually involved in fulfilling the request.

Then in phase (3), with the obtained workload in the replica space, we make the replica placement decision by extending the hypergraph formulation for the scenario without replicas. Still, a hypergraph should be formulated. The vertices in the hypergraph should be the union of the replica set and node set. In the edge set, the data-node edges are substituted by the replica-node edges, such that we set up an edge between each node and each replica. The weights of edges are set according to

$$
\begin{align*}
W \cdot \left( w_p^{[S]}, w_p^{[D]}, 0, 0, 0 \right)^T, & \text{ for each hyperedge } e_p' \\
W \cdot \left( 0, 0, w_y^{[T]}, w_y^{[L]}, w_y^{[C]} \right)^T + \xi, & \text{ for each edge } e_{zy}
\end{align*}
$$

(2.29)

Based on the above formulation, we can still apply the hypergraph partitioning algorithm as in the scheme without replicas. The time complexity of the $N$-way hypergraph partitioning algorithm we used is $O\left( (|V| + |E|) \log N \right)$, so the time complexity of our whole scheme is no more than $O\left( (|P|N + kMN) \log N \right)$.

We also find a simplification to (2.29) in setting the weights of all edges in the form of $e_{zy}$. For each replica $z$, we only take into consideration the edge $e_{zy}$ with the highest weight in the set of $\{e_{zy}| y \in Y\}$. This kind of weight setting gives a higher preference to not cutting the edge with the highest weight in the data-node edge set related to a replica. This method is not optimal, but it can give a good performance as shown in the later experiment results. It also helps to largely lower the number of edges in the formulated hypergraph, which largely reduces the running time.

After applying the hypergraph partitioning, we obtain the replica placement result, which in fact is the input of routing decision making in the next round. After rounds of iterations of the routing decision and placement decision, the best performance among solutions that have been explored reaches the steady state, and we would stop the iteration after the improvement is less than a threshold $\gamma$. Finally, the placement and routing decision in the last round are transmitted to the nodes in the system. With the deterministic routing decision $M$, we can obtain a hash mapping function for each node, whose input is a request pattern and output is the routing destination of each item in the pattern. Such a function ensures the dispatch of any requests can be made in a very short time which is important for a practical system. Note that the data placement decisions including the final solution allow more than one replica of each data item placed at the same location.
2.6 Performance Evaluation

2.6.1 Experiment Settings

(a) Geo-distributed Datacenters

We conduct some experiments based on realistic settings and consider the heterogeneity of datacenters. We assume there are $N=9$ datacenters, mimicking the 9 available regions of Amazon clouds [1]. Besides, we use the currently advertised prices of storage and outgoing traffic in Amazon clouds as the parameters in our experiments. The locations and prices are listed in Table 2.1. Furthermore, we measured the access latencies of inter-datacenter traffic and show the average results in Fig. 2.4, which are used in the experiments as the parameters of latencies. We recorded the prices and measured latencies in Jun. 2015, by small instances of Amazon clouds. From both Table 2.1 and Fig. 2.4 we observe that the prices and latencies can be different among different regions. For the latencies, they were measured through the ping command between each pair of regions in Amazon cloud. As illustrated in Fig. 2.4, the latencies of different pairs of regions are different, justifying that data placement can affect the user-perceived data access latency when the requested data item is relayed between datacenters.

(b) Workload

The workload we use was extracted from the trace of a location-based online social network [37]. The data trace provides us with the social network of users and the check-in logs of users on various locations. Data were collected between Feb. 2009 and Oct. 2010. In the social graph, there are 196,591 nodes and 950,327 edges. The dataset is helpful in terms of providing us with the exact location of each user check-in, which is treated as a user request in our simulation.

We first analyze the check-in locations and frequencies in the trace. By analyzing the trace of user check-ins, the geo-distribution of user requests and their intensities are illustrated in Fig. 2.5. We observe that the distribution of user access is not uniform around the world, and the intensities are in fact not proportional to the distributions of the Amazon cloud locations.

We formulated the request patterns used in the simulations through the following procedure: for each considered user, we extract the friends of the user first from the trace and then consider that a request of news update from the considered user
would involve the pulling of the data of all friends, which forms the request pattern of the considered user. The request rates and source locations used in the simulation are generated according to user check-in behaviors, such that the source node of a request is chosen as the datacenter nearest to the check-in location in distance, and the frequencies of the requests are related to the number of the check-ins by the user in the dataset. After pre-processing the dataset, we obtain a scenario with 196,591 data items and 60,948 request patterns, and the average number of data items in each request pattern is 26.34.
(c) Compared Schemes

In the experiments, four types of data placement are compared. As the name implies, Random randomly chooses the datacenters to store replicas; Nearest prefers to choose the datacenters with higher request rates as the locations to store the replicas of each data item; Multiget can be considered as the current state of the intra-datacenter data placement, i.e., with only the data association considered and without the awareness of the locations among geo-distributed datacenters; and finally, Hyper, which is the framework proposed, supports the aforementioned various performance metrics.

For the compared scheme Nearest and Random, we implement them in a way so that the size of their output partitions can also satisfy the specified storage balance ratio Θ. In Random, for each data item or replica to be placed, we randomly sample an integer in \([1, N]\) according to the distribution of Θ, as its storage location. In Nearest, first we obtain the number of items stored at different locations. Then we greedily place a data item or a replica of it at the location with the highest request rate to it, among the ones that the data item has been placed at. If the chosen location is full in terms of the occupied storage capacity, we randomly choose another location that is available, regardless of the storage capacity, and whether there has been a replica already or not. Note that the design of these compared schemes does not have to strictly follow the constraints defined in our problem formulation. In the implementation of the proposed Hyper, to overcome the performance variation of the hypergraph partitioning heuristic, we run the heuristic 5 times in each round and choose the best one as the final result for replica placement decision. The heuristic
is non-deterministic, so multiple trials improve the result.

(d) Parameter Settings

In the experiments, we set up a multi-objective optimization function in the form of Eq. (2.23). The weights of metrics are extensively tested to show the possibility of customizing the objective according to the practical requirements of service operators. In the comparisons, we test 34 random weight settings, and 4 specific settings, in terms of the weight vector $\mathbf{W}$. The specific settings are $(100, 0, 1, 1, 1)$, $(1, 0, 100, 1, 1)$, $(1, 0, 1, 100, 1)$ and $(1, 0, 1, 1, 100)$, which represent different preferences or importances of the considered evaluation metrics, such as, the higher priority of co-locating the associated data, lower inter-datacenter traffic, lower latencies and lower storage cost. In the weight vector, we ignore the second metric because its numerical result is always proportional to the first one.

In terms of the ratio of storage balance, we set the input ratio $\Theta$ the same as the request rates among the $N = 9$ locations. As shown in Fig. 2.5, the distribution of requests is actually skewed. For the service considered, South America has a lower request intensity, compared with North America and Europe. Therefore according to our setting of $\Theta$, there would be fewer data items placed in the region of Sao Paulo as a result. Besides, we assume that there are 2 replicas for each data, since when each data is replicated to 2 different regions, the data can be always available when only 1 datacenter is in the state of failure. Changing the number of replicas affects the tradeoff between availability and efficiency, and our scheme still works if the number is changed.

(e) Evaluation Metrics

Throughout all the experiments, we obtain the replica placement results from each compared scheme under the given workload. Based on the obtained placement results, we can compare the schemes with regard to different aspects. The data used for evaluation are the same as the scheme input, so that we can observe how much improvement can be made through the approximate solution.

The compared aspects are reflected by the evaluation metrics listed as follows.

**Objective. (Obj.)** It is the weighted sum of the considered performance metrics. We follow Eq. (2.23) to obtain this value to represent the overall performance of the system.
Co-location. (Col.) It is related to the node span of each request pattern, which is also the number of nodes involved by the requested itemset. Then for all the request patterns, we calculate the weighted average of node spans, with the request rates being the weights. The metric is related to the system time and distribution overhead mentioned in Section 2.4.2(a).

Traffic. (Traf.) The total inter-datacenter traffic can be calculated based on the obtained placement results. We calculate the traffic cost considering the different outgoing traffic prices in different regions, as shown in Table 2.1.

Latency. (Lat.) The sum of access latencies on obtaining all of the requested data items in the workload is also calculated. This metric not only favors the scheme serving data items at the source node, but also takes the different latencies of retrieving the data item in different regions into account.

Storage. (Stor.) The sum of the total cost on storing all of the replicas is also calculated. Note that different regions may have different storage cost as shown in Table 2.1.

Balance. (Bal.) We calculate the Pearson correlation between the expected storage size distribution and the actual storage size distribution. If the value is close to 1, it means that the two vectors are quite similar. Actually, for all the schemes tested, the results are close to 1, because the storage size distribution is considered in all the data placement schemes compared.

2.6.2 Experiment Results

(a) Performance Verification

With the schemes proposed and compared, we can obtain the data placement results. First, we randomly generate some weight vectors \( W \) (where the value of items in the vector are sampled from independent uniform random variables in \([0, 100]\)) and conduct 33 experiments each on a different weight vector. The comparison results on the overall objective are shown in Fig. 2.6. In the figure, the multiple experiments were sorted in descending order of cost reduction ratio, for the illustration purpose. The resultant value of optimization objective \( (2.23) \) by each scheme is shown by the stacked bars. Under all the experiments, the proposed scheme outperforms the others. This justifies that the proposed method is effective irrespective of the chosen weight vector. By comparing the result of Hyper with the best result of the other three, we can observe that the ratio of improvement in terms of cost reduction is in the
Figure 2.6: Performance verification with random weight vectors

Figure 2.7: Performance comparison, $W = 100 : 0 : 1 : 1 : 1$

range between 0.11 and 0.38. Under different weight settings, the second best scheme could be Nearest or Multiget, since both make some improvement compared with Random. Hyper makes the most comprehensive considerations in terms of the metrics supported, so it performs better than the others on the overall objective under all the weight vectors that we have tested.

(b) Further Insights

We further test the performance on 4 specific settings of weights to give more insights. All the evaluation metrics are compared. Note that for each metric category except Balance, the shown results are all normalized, i.e., each shown value has been
divided by the largest one on that metric among all schemes. Due to the formulated minimization objective, the smaller the values of metrics, the better the performance is.

Fig. 2.7 shows how the results compare when the system operator holds a higher preference for the optimization of system time or distribution overhead. We can observe that both Multiget and the proposed Hyper achieve a low objective value since their effective improvement on the metric of co-location. On the metrics of latency and traffic, although Hyper does not perform well due to their low weight in the overall objective, it is still not worse than Random on such metrics.

The advantage of Hyper over Multiget is that the former considers the metrics that are related to source locations of requests, which are ignored by the latter. This can be evidenced by the results shown in Fig. 2.8, where we observe that when we give a higher preference to the minimization of inter-datacenter traffic in the formulated optimization, Hyper achieves a much better result with regard to that metric and the overall objective, when compared with Multiget. Besides, Hyper shows a better performance than Nearest as the latter only tries to lower the geographical distance of data and its user, without considering that the actual traffic cost may not be proportional to the distance. Fig. 2.9 shows how the results compare when the preference is to optimize inter-datacenter latencies. The result on that weight vector shows a trend similar to Fig. 2.8.

When the preference among metrics is given to the storage cost, we obtain the performance results shown in Fig. 2.10. Due to the overall consideration of the pro-
posed framework, when we give a higher preference to the weight of storage cost in the objective function, the proposed Hyper will take a larger effort to lower the data storage cost when compared with those under other weight vectors.

In summary, with the comparison through experiments, we verified that the proposed Hyper can adapt to the adjustment of weight vector $W$ and reasonably covers the metrics considered. Although Hyper is not always the best on all metrics, it shows to be the most effective in term of improving (2.23), which is the main target of defined multi-objective optimization. In the real world, different Internet services have their own characteristics, e.g., being latency-sensitive or cost-sensitive. Whenever their requirements fall in the range of metrics considered in our framework, the
providers of such services could obtain some benefits through applying the proposed managed data placement.

(c) Iterative Replica Placement

To show the effectiveness of the iterative replica placement proposed, we plot the intermediate results during the iterative process in Fig. 2.11. It is under the setting of $W = 56 : 0 : 41 : 77 : 25$. In the experiment, we record the current overall objective and the lowest overall objective found after each round. In the figure, we use two consecutive points to represent one round in the iterative scheme: one is the performance after placement decision and another is the performance after routing decision. We can find that with routing optimization, the performance can always be improved compared with the result before doing that.

Due to the randomness of the hypergraph partitioning algorithm, there is some fluctuation of performance in the iterative process. Actually, although this may introduce some not-so-good results temporarily, it can extend the exploration space, noting that the scheme is searching in a large high-dimensional solution space using heuristics. The reason that the results of different rounds show a large fluctuation is that we have two kinds of decisions in each round, which have different problem formulations. According to our tests, we can conclude that the best obtainable result under our scheme will reach the steady state after 1–9 rounds. This means that even under the iterative design, the scheme does not need to run a large number of rounds to get the best result it can find. Note that here we show multiple rounds only for the purpose of illustrating the iterative process, and the algorithm can end its loop when the performance improvement is less than $\gamma$ after each round.

(d) Running Time Analysis

Last, we analyze the running time of the proposed scheme. Our experimentation hardware features Intel 8-core E5-2670 CPU and 16 GB memory. The statistics shown in Table 2.2 is the average running time, standard deviation of running time, average hypergraph partitioning time, and standard deviation of hypergraph partitioning time in each round of iteration. In this experiment, we also shrink the scale of the input data, by treating several data items with the same hash value as the same data item, so that the number of vertices is lowered. This method is effective in lowering the algorithm input scale, and thus lowers the algorithm running time in larger-scale
problems. As shown in the table, by increasing of the input data scale, the total running time and the hypergraph partitioning time for each round would increase. Under the full scale of the data, which was used in previous experiments, it takes about 41 seconds in each round to execute the proposed scheme. Considering less than 10 rounds are needed as illustrated in Fig. 2.11, the proposed scheme is able to obtain a solution in a reasonable time, which is sufficient for offline data placement among geo-distributed datacenters.

### 2.7 Conclusions

We studied the data placement problem for geographically distributed datacenters, with joint considerations of the data-node relationships, and the associations of multiple data items. By formulating the problem as a hypergraph model, we presented
the framework to obtain the optimized solution of data placement with the consideration of various metrics. We further addressed the placement problem under the replica scenario, by introducing the iterative process of routing and replica placement. Finally, we evaluated the proposed scheme through extensive simulations.
Chapter 3

Sketch-based Data Placement

3.1 Overview

With the increasing demand of big data applications, a variety of problems on how to operate the supporting infrastructures more intelligently and efficiently have attracted much attention in the literature. To optimize the data placement among distributed network locations is one of the fundamental problems, which aims at facilitating the data storage and access. The scheme proposed in Chapter 2 and some other existing studies such as [13, 60], tried to overcome the deficiencies of the traditional hash-based random placement through managing the storage locations. They improved the system-level efficiency and the performance of data operations. However, there are some challenges on the running time and the overhead introduced due to the increasing scale of datasets. Therefore, in this chapter, the Sketch-based Data Placement (SDP) is proposed, trying to lower the overhead introduced by data placement while still keeping the benefits. We first justify the effectiveness of applying the hypergraph sparsification on the data placement problem, and then present the method of constructing sparsifiers through sketches of request traffic. Besides, the scheme supports aggregating distributed sketches to make the decision and capturing the pattern of recent traffic through sliding windows. Finally, we obtain numerical results through simulations, which confirm that the proposed scheme can place data effectively while reducing the introduced overhead in terms of algorithm running time, space and network traffic.
3.2 Related Work

Due to the availability of geo-distributed storage sites, there exist flexibilities in choosing the data storage location. Various data placement schemes have been proposed. In [15], the automatic data placement across geo-distributed datacenters was presented, which iteratively moved each data item closer to its end-users and other related data items. In [60], the social relationships in the Online Social Networks (OSN) were utilized to facilitate data placement, and a scheme with multiple objectives was proposed using the technique of graph cuts. In [108] and Chapter 2, we showed that hypergraphs can model and address the data placement problem, capturing relationships between data items and nodes. In this chapter, we argue that the overhead of the centralized design is significant under the increasing scale of datasets and propose an approximate solution through sketches.

The idea of the hypergraph sparsification proposed here is to randomly sample the edges in the hypergraph model proposed in Chapter 2. It is related to the random edge sampling presented in the pioneering work of Karger [63], which proposed a randomized algorithm for computing the minimum cut of a graph by edge contractions. Karger [64] further summarized how the sparse graph or skeleton, obtained from random edge sampling, can accurately approximate the value of all cuts in the original graph with a high probability. Our scheme adopts the probabilistic sampling of edges, which is different from the methods based on heavy-hitters [54].

The streaming model [19] of algorithms has attracted much attention recently due to the increasing scale of datasets. It assumes that the input of algorithms is a stream of elements which arrive consecutively. The streaming model has been applied to graph algorithms, such as graph sparsification [16], considering that the size of graphs keeps increasing. Based on the streaming model, McGregor [76] surveyed the graph algorithms using the sketch-based methods. Meanwhile, some heuristics were also proposed, e.g., [97] discussed graph partitioning with a vertex-based streaming model and [89] also compared some heuristics for graph partitioning. The streaming model was also used in solving some network measurement problems, e.g., flow-based measurements [114], detecting super sources and destinations [115], etc.

In solving the data placement problem, our work fundamentally deals with the hypergraph sparsification based on sketches. About hypergraphs, Deveci et al. [44] discussed how to sparsify them by removing identical (or similar) vertices and edges, Kogan et al. [67] showed the feasibility of sketching cuts of hypergraphs with uniform
edge-degrees and made a theoretical proof, and Wang et al. [103] discussed the sparsification of a classical graph transformed from the hypergraph. Such methods are not suitable for the data placement problem formulated, because they are centralized algorithms. Besides, the work in this chapter is related to some papers studying the sketches for distributed streams [50] [41] or supporting sliding windows [26] [81], which do not focus on hypergraph sparsification, but can be partially utilized in our scheme.

3.3 Preliminaries

3.3.1 Data Placement through Hypergraph Models

We start with a brief summary to the problem of data placement among geo-distributed datacenters, which have been presented in Chapter 2. Assume there are $N$ geo-distributed sites or nodes available for data storage, denoted by $Y$. The set of data items is denoted by $X$. The flow of data requests can be generalized as: a user may request an itemset $p$, which is a subset of items in $X$, from a user-access node $y \in Y$; in the flow, $y$ is the source of the flow, and the nodes storing the items in $p$ are the destinations. Denote $P$ as the set of itemsets. Then the data placement problem is formulated as: on a centralized controller, with the logs of requests at all sites in the recent time window $W$, the request rate set $\mathcal{R}$ is obtained first; then the storage location of each data item $x \in X$, denoted by $\mathcal{D} : x \rightarrow y$, is to be determined to optimize an objective function $\mathcal{C}$ (defined on $\mathcal{R}$ and $\mathcal{D}$).

We have proposed in Chapter 2 that the hypergraph models help to address the data placement problem, optimizing a series of metrics. Two categories of metrics have been claimed to be supported by the proposed hypergraph models. In this chapter, the first category of metrics is equivalently abstracted as the metric of associated data collocation, which favors placing associated data items together, while the second category is simplified as the metric of localized data serving, which favors not incurring network traffic in data access. The latter metric can be considered as a boolean model, characterizing whether the requested data item is placed at the source node or not. It cannot fully quantify the exact cost or benefit of placing at different nodes, as claimed by the scheme proposed in Chapter 2 due to the limitations of the current sketch-based techniques. The adopted boolean model for the data-node relationship can be considered as a simplification to the distance model. For example,
for most metrics in the category of data-node distances, we achieve the most benefit if the requested data item is placed at the source node.

In the scheme, by aggregating the request logs on all sites, two types of request rates can be calculated. One is the system-level request rate for each type of itemset \( p \), denoted by \( R_p \). Another is the request rate to a specific item \( x \) from the node \( y \), denoted by \( R_{xy} \). Therefore, \( R = R_p \cup R_{xy} \), where \( R_p \subseteq \{R_p|p \in P \} \) and \( R_{xy} \subseteq \{R_{xy}|x \in X, y \in Y \} \). Then the optimization objective or the evaluation of a placement \( D \) is defined as

\[
C(D, \mathcal{R}) = C^A(D, R_p) + \alpha C^L(D, R_{xy}) ,
\]

where \( C^A \) is a linear function of \( R_p \) to encourage the collocation of associated data items (favoring fulfilling a requested itemset \( p \) using fewer nodes for a higher system efficiency), \( C^L \) is a linear function of \( R_{xy} \) to promote localized data serving (i.e., the requested item is stored in the source node), and \( \alpha \) is a tradeoff parameter between the two parts. Note that the optimization objective can be customized to satisfy some different purposes, which is not discussed here since the focus is on the algorithm framework.

In Chapter 2 to obtain the solution of optimizing such an objective, a hypergraph \( \mathcal{H}(V,E) \) is formulated and the technique of hypergraph partitioning is applied to the hypergraph to obtain the solution of data placement. That scheme is illustrated as (a) in Fig. 3.1. In the hypergraph \( \mathcal{H} \), the vertex set contains all the storage nodes and all the data items to be placed. The hyperedge set \( E \) is defined as

\[
E = \left\{ \{e_p|p \in P\}, \{e_{xy}|x \in X, y \in Y\} \right\}.
\]

Each hyperedge \( e \in E \) is attached with a weight \( w_e \). Due to the objective (3.1), the weights are set according to

\[
w_e = \begin{cases} R_p, & \text{for the request pattern hyperedge } e_p \\ \alpha R_{xy}, & \text{for the data-node hyperedge } e_{xy} \end{cases}.
\]

By feeding \( \{\mathcal{H}(V,E), F, \epsilon, N \} \) into the hypergraph partitioning algorithm \( \mathcal{P} \) as the input, the data placement results can be obtained from the partitioning results, denoted by

\[
\mathcal{P} (\mathcal{H} (V, E) , F, \epsilon, N) \to D .
\]
Here $F$ is the fixed-location vertex set where we set each node-type vertex to a different partition, and $\epsilon$ defines the ratio of storage balance among different nodes. Note the scheme can be extended to the scenario of replica placement, by iteratively repeating the replica placement decision step and routing decision step, as shown in Fig. 2.3. The discussion in this chapter focuses on the scenario without replicas, but a similar approach can be applied to obtain a sparsified hypergraph for the replica placement problem.

However, when implemented in a large-scale distributed system, we consider to lower the overhead of the hypergraph-based scheme itself, which consists of the space to store the traffic logs at each individual site, the network traffic to transfer these logs (or at least the extracted rates for all formulated edges in the current time window) to the centralized controller and the computation overhead of the partitioning algorithm. Indeed, the overhead is related to the scale of the hypergraph formulated. With the increasing data scale, sometimes even a polynomial space/computation complexity in (3.4) is unacceptable.

### 3.3.2 Overview of Sketch-based Data Placement

Considering the issues related to the scale of the hypergraph, we propose to simplify or sparsify the original hypergraph $\mathcal{H}$ before applying the hypergraph partitioning algorithm, which helps to make approximate decisions on data placement. An intuitive idea could be, to construct the hypergraph at the centralized controller first and then
sparsify it there, illustrated as (b) in Fig. 3.1 instead, our proposal is to construct the sparsified hypergraph directly from the sketches maintained at the distributed sites, which is shown as (c) in Fig. 3.1. Through the proposed scheme, we make improvements on the communication overhead between the sites and the central controller, the calculation overhead at the controller, and the space for the traffic measurement at individual sites.

The proposed scheme SDP is built on the recent progress on the streaming-based or sketching algorithms. However, the specific problem of sketch-based hypergraph partitioning has not been addressed yet in the literature. Our work fills that gap, and presents an approximate data placement solution for distributed storage systems.

The workflow of our scheme is illustrated in Fig. 3.2. Fundamentally, it has two components. First, the sketches, capturing the characteristics of the request traffic, are maintained at each site $y$. Intuitively, we can measure the request rates and store them in a vector $U_y$, with the IDs of requested itemsets as the index. Here, instead of maintaining the full vector $U_y$, each site will record the requests by sampling and counting sketches in a streaming fashion. Under the streaming model, each individual request is handled as a group of events. These events will be processed by the sketches consecutively. Note that the processing of events is through only one pass, which means that the requests will not be stored and processed again. Sketches are data structures capturing the statistical properties of the stream, and can be considered as a lossy compression of $U_y$. Only the important information that is to be utilized in the later stage is kept in sketches. Note that sketch can introduce some relative errors, as data are compressed. We denote the sketches by $S_y$. Obviously, the size of $S_y$ should be much smaller than $U_y$.

Then, when a data placement decision is to be made, the central controller pulls...
some information extracted from $S_1, ..., S_N$. Here the communication overhead is reduced, compared with pulling $U_1, ..., U_N$. Based on the information, the controller can construct a sparsifier of the original hypergraph $\mathcal{H}$, denoted by $\mathcal{H}_S$. Finally, the hypergraph partitioning algorithm (3.4) is applied on the obtained sparsifier $\mathcal{H}_S$. Here, the running time for hypergraph partitioning is also reduced because of the much smaller size of $\mathcal{H}_S$.

Due to the dynamics of the request traffic, the formulated sparsifier $\mathcal{H}_S$ for data placement needs to be updated to capture the characteristics of the recent traffic. For this purpose, a time-based sliding window is applied on the stream, which means that only the requests in the recent $W$ seconds are used to formulate the sparsifier. Since the set of nodes and items are usually stable, we assume that the vertex set in the formulated hypergraph is always known by the controller; therefore the discussed dynamics of request traffic can only affect the hyperedge set of $\mathcal{H}_S$.

Below we divide our work into two stages: in Section 3.4, we propose a centralized heuristic, Probabilistic Sampling, to construct the sparsifier $\mathcal{H}_S$, assuming $\mathcal{H}$ is known, and verify its effectiveness; in Section 3.5, we present the scheme SDP, which constructs $\mathcal{H}_S$ directly from sketches in a distributed way, following the design of Probabilistic Sampling.

### 3.4 Hypergraph Sparsification

#### 3.4.1 Problem and Metrics

Given a formulated hypergraph $\mathcal{H}(V, E)$, we can construct its subgraph or sparsifier, denoted by $\mathcal{H}_S(V, E')$, where $E' \subset E$. Formally, the scheme to construct the sparsifier can be defined by a sparsification function

$$\mathcal{F} : \mathcal{H} \rightarrow \mathcal{H}_S$$

and how it should be defined and implemented is the fundamental problem of this work. We expect that $\mathcal{H}$ can be well approximated by $\mathcal{H}_S$. To evaluate the quality of approximation through a scheme $\mathcal{F}$, we will compare the difference between certain measurements on $\mathcal{H}_S$ and $\mathcal{H}$. Literally we will define some measurement functions, in the form of $\mathcal{M}(\mathcal{H}) \rightarrow M$, whose input is a hypergraph $\mathcal{H}$, and output is a number $M$ representing the performance that can be obtained with the knowledge of $\mathcal{H}$.
For example, two metrics are defined here: the graph complexity and system efficiency (represented by the optimization objective (3.1)). The complexity of the hyperedge is determined by its hyperedge set $E$. Since hyperedges are involved, the number of vertices attached to each hyperedge is different. We define the complexity based on the total number of attached vertices on all the edges as the following:

$$M_C(H) = \sum_{e \in E} |V_e|,$$

where $V_e$ is the set of vertices attached to an edge $e$. Note that we have verified that when using the tool in [31] as the partitioning function $P$ in (3.4) to partition a sparsifier, the running time is strongly related to $M_C$. The obtained performance of system efficiency through the proposed data placement is represented by the optimization objective in (3.1), and we have shown in (3.4) that the objective is determined by the formulated hypergraph. Therefore, the obtained system efficiency based on the information of a hypergraph $H$, is defined as a measurement function

$$M_P(H) = C[P(V, E, F, \epsilon, N)],$$

where only $E$ is changing under the traffic dynamics.

In the literature, most existing studies focused on approximating the minimum cut or any cuts of classical graphs through sparsification; but here we approximate the cut of a balanced $N$-way hypergraph partitioning through sparsification. Our problem is more difficult to address for the following reasons. 1) Edge degree: Each edge of the hypergraph can be attached with more than 2 vertices, and the number of vertices attached to different edges could be different, which do not apply to classical graphs; 2) Number of partitions: We consider the $N$-way partitioning, which means that all the vertices are partitioned into $N$ non-overlapping subsets, while $N = 2$ is implicitly set in classical graph cuts; 3) Extra constraints: Due to the nature of the data placement problem, we have extra inputs and constraints: one is the balance of the number of vertices in different partitions; another is that some vertices have pre-determined placement among the partitions, i.e., those vertices representing datacenters, which are not supported by the discussion of graph cuts.

For the listed difficulties, we propose a simple yet effective heuristic following the existing designs on constructing sparsifiers for graph cuts and the characteristics of our problem. The quality of the approximation in the proposed scheme is verified by
the numerical results obtained from the experiments with real datasets.

### 3.4.2 Sparsification Heuristics

We discuss three heuristics to construct a hypergraph sparsifier assuming the formulated hypergraph for data placement is known in advance at the controller. For all of them, a parameter \( \rho \) is used to control the ratio of edges finally retained in the sparsifier.

**Random Sampling.** (RandS) This sampling-based heuristic constructs the sparsifier \( H_S \) by randomly choosing edges in the original graph \( H \). An adjustable parameter \( \rho \) defines the percentage of edges chosen from \( H \), i.e., the sampling rate. Given parameter \( \rho \), an edge \( e \in E \) is chosen and added into \( E' \) as \( e' \) with the probability \( \rho \) and we set the weight by \( w_{e'} = w_e \) in the sparsifier. Simply, the expectation on the complexity metric \( M_C \) is \( E[M_C(H_S)] = \rho M_C(H) \). This method is used as a baseline in the comparison.

**Heavy-hitter.** (HeavH) The second is a deterministic heuristic. We greedily choose the larger weight hyperedges in \( E \), since they have a larger impact on the resultant cut in the worst case. Formally, the cut weight that can be introduced by a hyperedge in the worst case is determined by two factors. One is its weight \( w_e \), and the other is the number of partitions connected to this edge in the worst case. Considering these two factors, we set the weight of sampling an edge

\[
\hat{w}_e = w_e \cdot \left( \min(|V_e|, N) - 1 \right), \tag{3.8}
\]

where \( |V_e| \) is the number of vertices attached to each edge \( e \in E \) and \( N \) is the number of partitions. In the implementation, HeavH will sort all hyperedges in \( E \) into a descending order based on \( (3.8) \), and then choose the top \( \rho \) percent of hyperedges as the edge set \( E' \) for the sparsifier.

**Probabilistic Sampling.** (ProbS) Last but not least, we present another heuristic which introduces randomness to the design, in the hope of achieving a good average or probabilistic performance. Similar to HeavH, we set the edge sampling weight according to \( (3.8) \). Differently, we choose each edge randomly according to the probability \( \rho \), with \( \hat{w}_e \) trials. In each trial, we obtain a sample \( r \) from the random variable following the uniform distribution \( U(0, 1) \). When \( r \leq \rho \) becomes true among any one of the \( \hat{w}_e \) trials, the hyperedge \( e \) will be chosen into \( E' \) as \( e' \) and its weight will be set by \( w_{e'} = w_e \).
3.4.3 Heuristic Comparisons

We test the performance using a dataset based on an Online Social Network (OSN) trace [68]. The dataset was previously used in [108] to discuss the performance of the hypergraph-based data placement. The dataset was obtained by crawling Facebook between Dec. 2008 and Jan. 2009. For its social graph, there are 63,731 users, connected by 817,035 links. The reason that here we choose a dataset different from Chapter 2 includes: i) the previously used dataset is about a location-based OSN, but the one used here provides the social graph of a general purpose OSN, i.e., Facebook, which avoids the potential side effect of location-related features; ii) under the dataset used here, we avoid the highly skewed distribution of requests among locations to effectively evaluate the support of aggregating multiple distributed streams in our scheme design.

By considering each user data as a data item, we place 63,102 data items into 10 sites. For each user, the data items of the friends of the user form an itemset. The request rate of each type of itemset is generated following the Zipf distribution [27]. The source site of each request is generated following the uniform distribution, assuming that users are uniformly distributed. By the previously mentioned method of hypergraph formulation, the hypergraph has 300,915 hyperedges and 63,112 vertices.

We test the performance of the mentioned heuristics and show the results in Fig. 3.3. With the changing parameter $\rho$, we obtain multiple test instances for each heuristic. The measurements for each instance are shown as a point $(M_C, M_P)$ in the figure. On the right-most of the figure, we can observe the best performance obtained.
without sparsification, in which case, all the edges are included in the sparsifier. For each heuristic, there is a general trend that when the sparsifier complexity $\mathcal{M}_C$ becomes higher, the resultant performance $\mathcal{M}_P$ is improving. It is reasonable that with more information discarded by sparsification, the performance will degrade.

When the complexity is low, $\text{ProbS}$ and $\text{HeavH}$ are both better than $\text{RandS}$. Although $\text{ProbS}$ and $\text{HeavH}$ each has own advantages, we prefer to choose the former here for two reasons: i) as shown in the comparison results, the performance of $\text{ProbS}$ is similar to $\text{HeavH}$, since in some cases the edges with a larger sampling weight are not more important to be included in the sparsifier; ii) $\text{ProbS}$ introduces randomness, so we can further improve the performance by choosing the best one among multiple trials (each returns a different sparsifier), which is not achievable by the deterministic heuristic. Therefore, we propose to use $\text{ProbS}$ as the basis of designing our data placement scheme $\text{SDP}$. So far, our discussion is based on the assumption that the original hypergraph is known in advance. Next, we are going to show how we implement $\text{ProbS}$ directly by using sketches from distributed sites, from which we lower the overhead of making data placement decisions.

### 3.5 Streaming-based Sparsifiers

#### 3.5.1 Scheme Overview

Below we implement $\text{ProbS}$ under the streaming model, with the support of distributed streams and sliding windows. In an intuitive way, we should capture the request traffic at all sites, and save each requested itemset with a timestamp into logs beforehand. Later when constructing a hypergraph sparsifier, the entries with timestamps in the current sliding window are collected from the saved logs at all sites, based upon which, we obtain the frequency of each itemset appeared in the collection, and finally construct the hypergraph. Compared with the intuitive way, the proposed scheme $\text{SDP}$ is more efficient, i.e., it does not rely on the request traffic logs, and some operations necessary for the sparsification are distributed to sites and done earlier at the moment of user request handling.

We divide the heuristic into two tasks: one is to obtain a sampled hyperedge subset $E'$ from the whole set $E$; another is to obtain the weight of each sampled edge in $E'$. For the two tasks, each site is designed to maintain two kinds of data structures: sampling sketches, which help to choose edges according to their sampling
weights, and counting sketches, which maintain the weights of edges using a space sublinear to the number of edges. Both of them are updated along with the streams of users requests.

In distributed storage systems with dynamic request traffic, the sketches should support two features. Because the sketches are distributed across sites, the controller should be able to aggregate them together for the support of distributed streams. Further, the time-based sliding window on streams should be supported. Under the sliding window, the deletions of requests in the distant past are implicit, i.e., events stored in sketches are deleted at the moment that they are expired. We define the window length as $W$ seconds. Assuming the current time is $t$, we expect to construct the sparsifier using the request traffic statistics in the time period of $[t - W, t]$.

The overall workflow of SDP is as follows. For each site, when a new request is coming, it updates the sampling sketch and the counting sketch, whose details will be presented below. For the centralized controller, when it decides to obtain the desired set of sampled edges and corresponding weights in order to construct a sparsifier, it interacts with the distributed sites, the steps of which are illustrated in Fig. 3.4.

**Round-trip 1:** With the start command, the controller requests the total sampling weight of all the requests in the sliding window, which is stored in a counter at each site $y$, denoted by $\omega_y$. Note that $\omega_y$ is implemented as a sliding-window-based counter, explained in Section 3.5.3.

**Round-trip 2:** With $\{\omega_1, ..., \omega_N\}$, the controller makes a virtual sampling locally to determine the number of samples to be returned by each site $y$, denoted by $\varphi_y$. Then each site $y$ subsamples $\varphi_y$ samples from its current sample set and returns them to the controller, denoted by $E_y$. The edge set of the constructed sparsifier is
obtained as $E' = \bigcup_{y=1}^{N} E_y$. The process of virtual sampling and subsampling, and how to maintain samples on sites, are explained in Section 3.5.2.

**Round-trip 3:** Then the controller pulls the weights of edges in $E'$ from the counting sketches at distributed sites. Specifically, the weight for $e \in E'$ is $w_e = \sum_y \omega_{ey}$, where $\omega_{ey}$ is returned from the counting sketch at each site $y$, with the input key $e$. Section 3.5.3 will explain the details of counting. With the returned sampled edges and their weights, the hypergraph sparsifier is finally constructed.

### 3.5.2 Sampling Sketches

We start with introducing the *classical reservoir sampling*, where $k$ samples in a stream with an unknown length can be uniformly sampled using $O(k)$ space. The method is to maintain an array $S$ of size $k$ representing the samples chosen currently. When the $i$th event in the stream arrives, it will replace one item in $S$ with the replacement probability $p_i = k/i$. The decreasing replacement probability with the increase of $i$ ensures the uniform sampling probability on all the events in the stream.

This method cannot be directly applied to our problem due to the requirements of distributed streams and sliding windows, but we still use it in the process of local subsampling, i.e., when we already have a set of samples and want to draw some samples uniformly from the set, in order to lower the number of samples.

For the requirements on supporting distributed streams and sliding windows, we considered the protocol and algorithms proposed in [41]. It is not completely applicable here due to the assumed synchronization between the controller and sites, which introduces too much latency and communication overhead to the system. Therefore, we only utilize its design locally to address the issue of sliding windows and then overcome the issue of distributed streams through the *virtual sampling* that we propose. Our method also lowers the number of samples maintained at distributed sites.

In our design, each site $y$ tries to maintain $K_y$ samples for the current sliding window at any moment. The parameter $K_y$ is pre-assigned by the centralized controller, based on the traffic history. The setting of $K_y$ is a tradeoff between the storage space for sketches and the maximum number of edges that can be recalled in the hypergraph sparsification. Besides, $\omega_y$ is maintained at each site $y$, representing the sum of accumulated sampling weight on all the edges, through the requests initiated from site $y$ in the sliding window, or equivalently the total number of events generated at site $y$ in the sliding window.
**Update and Query.** The maintained sampling sketch at each site can provide two operations, update and query. In terms of update, when receiving a request to itemset \( p \) at site \( y \), it will change the weight of edges in \( E_p = \{ e_{xy} | x \in p \} \cup \{ e_p \} \), therefore we calculate \( r_e \) for each edge in \( E_p \). For each edge \( e \in \{ e_{xy} | x \in p \} \), we set \( r_e = \alpha (\min(|V_e|, n) - 1) \), and for edge \( e_p \), we set \( r_e = \min(|V_e|, n) - 1 \). Then we create \( r_e \) update events (or 1 event with probability \( r_e \) if \( r_e < 1 \)) with the identifier \( e \) and make the events processed by the underlying \( K_y \)-sampler to be explained below. Besides, for each of the \( r_e \) events, we increment \( \omega_y \) by 1, so that the measured weight for edge \( e \) is increased by \( r_e \).

In terms of query, when a data placement decision is to be made, each site tells the controller the number of samples available in the \( K_y \)-sampler, denoted by \( \kappa_y \) (while always \( \kappa_y \leq K_y \)) and the total number of events processed in the sliding window, denoted by \( \omega_y \). The central controller collects \{\( \omega_1, ..., \omega_N \)\} and generates the number of events to be sampled from each site \( y \). To do that, the controller needs to do the virtual sampling, i.e., to virtually replay all the events on a classical reservoir sampler to obtain \( Z \) samples, which is the total number of samples to be recalled. Specifically, for each \( y \in [1, N] \), we let the reservoir sampler repeatedly process event \( y \) for \( \omega_y \) times. Then in the obtained \( Z \) samples, we calculate the frequency of each \( y \in [1, N] \), which is actually the number of events to be sampled from each site, denoted by \( \varphi_y \). Sometimes, e.g., when the amount of traffic is very low, it may happen that \( \varphi_y > \kappa_y \). In this case, we lower the parameter \( Z \) on the classical reservoir sampler and redo the virtual sampling until \( \varphi_y \leq \kappa_y \) for all sites. Then for each site, it can return \( \varphi_y \) samples through subsampling the \( \kappa_y \) samples available.

**\( K_y \)-Sampler.** Next we briefly introduce how each site \( y \) maintains \( K_y \) samples within the sliding window, where we simplify the protocol proposed in [41] to a local version. Specifically, a Level Sampling (\texttt{LevelS}) structure is maintained for each non-overlapping segment with length \( W \) (the sliding window length) along the timeline, and a Binary Bernoulli Sampling (\texttt{BinaryS}) structure is also maintained for the current time window. Assuming the current time is \( t \), the current sliding window \([t - W, t]\) can be decomposed into two parts, \([t - W, \gamma W] \) and \([\gamma W, t]\), where \( t - W \leq \gamma W \leq t \). We can obtain a superset with no fewer than \( K_y \) samples, by grouping the samples from the two sampling structures. We obtain some samples in \([t - W, \gamma W]\) through \texttt{LevelS} and some other samples in \([\gamma T, t]\) through \texttt{BinaryS}. Fig. 3.5 illustrates an example. The current time is \( t \), so \texttt{LevelS} maintains the samples for the previous full window \([0, W]\) and \texttt{BinaryS} maintains the samples since the end
of the last full window, i.e., \([W, t]\). Combining the results from \texttt{LevelS} for \([0, W]\) and and from \texttt{BinaryS} for \([W, t]\), we can obtain the samples for \([t - W, t]\).

By ensuring the two sets of samples obtained are based on the same sampling rate and representing the non-overlapping segments of the time window, they can be grouped together. In \texttt{BinaryS}, the current sampling rate is known, while in \texttt{LevelS}, multiple sampling sets with different sampling rates are maintained simultaneously to overcome the issue related to the expired samples. Since at least one of the two subsets contains \(K_y\) samples, when we put them together, we obtain no fewer than \(K_y\) samples as a superset. Then the superset is subsampled to obtain the desired number of samples. For the exact process and more details about \texttt{LevelS} and \texttt{BinaryS}, please refer to [41].

The design of [41] needs the synchronization among sites and the controller, to keep the same sampling rate among sites and when samples for \texttt{BinaryS} are updated. Differently, our proposal makes it easier, i.e., we only maintain the structures locally without any interactions between the controller and sites on the sampling updates, and rely on the virtual sampling as stated above, to coordinate the sampling rate of different sites. Besides no need for synchronization, another benefit is that we only maintain \(K_y\) samples at each site and only transfer \(\varphi_y\) samples to the controller, both of which are much smaller than \(Z\) in the existing design. The drawback is that sometimes we obtain fewer than \(Z\) samples as a compromise for \(\varphi_y > \kappa_y\). However, because the scheme focuses on a long-term data placement problem, the amount of traffic at each site is in a highly predictable range, resulting in a lower probability of the negative case.

### 3.5.3 Counting Sketches

Through counting sketches, we can count the total number of events \(\omega_y\) at each site \(y\) and the number of each specific-type event, denoted by \(\omega_{ey}\) for each event \(e\) at each site \(y\), in the current sliding window. The scheme in [81] has addressed the
counting of specific-type events with sliding windows by combining the method of exponential histograms and count-min sketches. Here we can use the sliding-window-based counter to count \( \omega_y \) directly.

In our design, each site \( y \) maintains a sliding-window-based counter for \( \omega_y \). For any request to itemset \( p \) initiated at site \( y \), we obtain the related edge set \( E_p \) and create \( r_e \) events for each edge \( e \in E_p \). For the purpose of counting, each event will increment the counter of \( \omega_y \) by 1. Note that the counter will be implicitly decreased with time passing by, due to their support of time-based sliding window. Besides, we maintain a count-min sketch structure to support the counting of any specific-type event \( e \). The structure can use a space sublinear to the types of events counted with approximations.

**Sliding-window-based Counter.** Datar et al. [42] proposed that exponential histograms can estimate the number of events in a sliding window. Specifically, multiple buckets are maintained representing the decomposition of the time window, denoted by \( C_1, \ldots, C_n \), where \( C_1 \) is the most recent one. Each bucket \( C_i \) is characterized by the timestamp of the most recent event in it, denoted by \( T_i \), and the number of events in it, denoted by \( N_i \). Only the buckets that still contain some unexpired events are maintained. Thus, only the last bucket \( C_n \) is possible to be partially filled with some expired events. Due to the possibility of expired events in \( C_n \), the total number of events in the time window is estimated according to \( N_n / 2 + \sum_{i=1}^{n-1} N_i \).

By setting the size of the maintained buckets properly, the relative error of estimation can be limited. There are two constraints on the sizes of buckets. First, the bucket sizes \( \{N_1, \ldots, N_n\} \) should increase monotonically, which means \( C_1 \) has the smallest size, and the size of a bucket should be in a power of two, i.e., \( \{1, 2, 4, 8, \ldots\} \). Second, the number of consecutive same-size buckets should be in the range \( [\phi, \phi + 1] \), where \( \phi \) is to tradeoff the space required and relative error.

The algorithm is designed as follows: whenever there is a new event, we create a new bucket with size 1, and set the timestamp of the bucket to the timestamp of the event; we will merge the buckets when necessary to satisfy the constraints (e.g., we may create a new bucket of size 8 by merging two buckets of size 4 when there are more than \( \phi + 1 \) buckets with size 4); it is possible that the merge of a pair will introduce another violation, so the merge will continue until the constraints are satisfied on all the unexpired buckets.

As an example illustrated in Fig. 3.6, there are five buckets at the moment \( t \). The window size \( W \) indicates that all events in \( C_5 \) and some events in \( C_4 \) have expired,
so the estimation of the counter at that moment is $N_4/2 + N_3 + N_2 + N_1$. Besides, whenever there is a new event, we create a new bucket, insert it to the right of $C_1$ and make necessary merging of buckets to satisfy the constraints. The relative error that can be ensured with a certain space requirement has been shown in [42].

**Counting Structure.** We can approximate the counting of specific-type events in any time-based sliding window through the structures proposed in [81], which jointly utilizes the count-min sketch [40] and the sliding-window-based counters [42]. The count-min sketch is designed as a matrix of $H \times L$ counters. A set of $H$ pairwise independent hash functions are also needed. Each of them is responsible for one of the $H$ rows in the matrix, and can output a value in the range of $[1, L]$. When we see a new event $e$, we hash $e$ with each hash function and get a counter on each row of the matrix. Then the returned counters are all incremented by 1. To query the frequency of any event $e$, we still hash $e$ with each hash function to obtain $H$ counters and then choose the minimum value returned from such counters. The count-min sketch uses a sublinear space to store the frequency of events. It may have some relative errors due to the conflict in hashing different events. As stated in [40], for a Zipf distribution with parameter $z$, the space required to answer point queries with error $\epsilon|a|_1$ with probability at least $1 - \delta$ is given by $O(\epsilon^{-\min(1, 1/x)} \ln 1/\delta)$, where $|a|_1$ is the sum of all frequencies.

3.6 Numerical Results

3.6.1 Evaluation Methodology

We implement an event-driven program to simulate and evaluate the proposed sketch-based data placement. The setting of experiments here is the same as evaluating the heuristics with the OSN trace introduced in Section 3.4.3. In order to evaluate the scheme for data placement, previously the request rates in a certain period are used as the input for model training and evaluation. Here, individual requests are generated
Table 3.1: Overhead comparisons

<table>
<thead>
<tr>
<th></th>
<th>Space</th>
<th>Traffic</th>
<th>Edges</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central</td>
<td>233.6 MB</td>
<td>12.6 MB</td>
<td>300,488</td>
<td>1,293,268</td>
</tr>
<tr>
<td>SDP, $K = 3k$, $L = 3k$</td>
<td>57.6 MB</td>
<td>1.1 MB</td>
<td>21,103</td>
<td>153,818</td>
</tr>
<tr>
<td>SDP, $K = 3k$, $L = 5k$</td>
<td>77.3 MB</td>
<td>1.1 MB</td>
<td>20,890</td>
<td>154,893</td>
</tr>
<tr>
<td>SDP, $K = 5k$, $L = 3k$</td>
<td>64.6 MB</td>
<td>1.6 MB</td>
<td>31,808</td>
<td>232,473</td>
</tr>
<tr>
<td>SDP, $K = 5k$, $L = 5k$</td>
<td>84.3 MB</td>
<td>1.6 MB</td>
<td>31,918</td>
<td>232,321</td>
</tr>
</tbody>
</table>

to simulate the request processing over streams. We assume the inter-request interval for each type of requests is exponentially distributed, because the requests for an individual document on a web server follow the Poisson process \([21]\). The average request rates are set as following the Zipf distribution. To be compared with SDP, a scheme that simply saves all traffic logs and constructs the hypergraph based on all logs in the sliding window is also implemented, denoted by Central. A random data placement is also implemented, denoted by Random. For the proposed scheme, we test different settings of parameters to show the tradeoff between the overhead and performance. The average request rate among all sites is set to 5,000 requests/sec. The time window parameter $W$ is 240 seconds, to avoid the long running time in simulating the distributed system by the standalone event-driven simulator. Besides, we set $H = 7$ and $\phi = 2$ in the counting sketches by default.

### 3.6.2 Scheme Comparisons

We execute the simulation utilizing the proposed scheme with different parameters, as well as Central. One parameter is the number of samples to be maintained at each site, denoted by $K$. Another is the width of the count-min sketch, denoted by $L$. $K$ and $L$ are both set to be the same for all sites, since the traffic amounts on different sites are similar in our experiments. Besides, we set $Z = N \cdot K$, which means that we always try to obtain a sparsifier as large as possible from the available samples at all sites.

#### (a) Overhead Comparison

At the moment of making data placement decisions, we roughly estimate the space usage at sites by counting the number of variables maintained in the program, where each timestamp, itemset ID, and raw counter is calculated as 8, 8 and 4 bytes, re-
spectively. Besides, the scale of the resultant sparsifiers is shown as the number of hyperedges and the total number of vertices attached to all edges. The results are shown in Table 3.1. It reveals that SDP consumes much less resource in terms of space and traffic compared with Central, while formulating a hypergraph of a much smaller scale in making data placement decisions. Meanwhile, the results show that when we increase $K$ or $L$, the resource consumption increases, but it is still much less than Central. The resource saving shown here is not proportional to the number of edges recalled, due to the setting of the window size. When the window size becomes larger, which is always true in practice, the benefit of the resource saving will be more significant.

(b) Performance Comparison

We further compare the data placement performance and show the results in Fig. 3.7. The compared metrics are Objective, Localization, Collocation and Balance. Localization represents the percentage of data that can be served locally upon requests. Collocation shows the average number of nodes involved in serving requests for itemsets. Balance shows whether the numbers of data items stored on different nodes are balanced. From the results, we see that SDP can obtain slight worse performance as compared with Central. Then compared with Random which places each data item to a random site, the advantage of SDP is significant. Between SDP and Central, we observe that SDP prefers to improve the metric Collocation with more effort. This is because the edges representing a requested data itemset tend to have a higher weight in the formulated hypergraph under the current dataset, so they are more likely to be chosen in sampling. It is as expected from the perspective of optimizing the overall optimization objective. However, in the scenarios that the metric of Localization is equally important, we should change the tradeoff parameter $\alpha$ in (3.1) to re-balance the two metrics when applying SDP. Exploring the effect of setting this parameter is left to the future study.

3.6.3 Further Discussions

(a) Counting Accuracy

To evaluate the counting sketches used, we calculate the Pearson correlation between the weights obtained from the counting sketches and from the logs on all the sampled
Pearson correlation characterizes the similarity between two vectors $X_1$ and $X_2$, by the calculation of $\frac{X'_1 X'_2}{|X'_1|_2 |X'_2|_2}$ where $X'_1 = X_1 - \text{avg}(X_1)$. Fig. 3.8 plots the results with varying $L$ and $H$ in the count-min sketches. When the space allocated to the count-min sketches is increased, we observe that the Pearson correlation approaches 1, which indicates that the vector obtained from sketches is quite similar to the vector obtained from logs. The results justify that with a proper setting of the parameters in count-min sketches, we can approximate the counting reasonably.
(b) Sampling Performance

We consider two metrics for the discussion of sampling performance: the optimization objective, and the actual number of edges recalled in constructed sparsifier. With $K$ varying between 2,000 and 6,000, we plot the measurement results of the obtained sparsifiers in Fig. 3.9. From the figure, we can observe that with the increase of $K$, the pre-assigned number of samples for each site, the data placement performance is improving. Meanwhile, with a larger $K$, a higher number of edges are finally included in the resultant sparsifier. Therefore, when applying SDP to some practical scenarios in the future, we can adjust $K$ to obtain the desired tradeoff between the complexity and performance. In addition, we observe that the total number of edges in the sparsifier is smaller than the total number of samples, which is $10^K$, due to the duplication of samples, i.e., some obtained samples represent the same edge. Estimating the duplication rate would be helpful to set $K$ in advance to obtain a certain number of edges, which is to be discussed in the future work.

3.7 Conclusions

Realizing the fast growing datasets in geo-distributed datacenters, we tried to overcome the inefficiency of making data placement decisions in traditional methods. We proposed a scheme based on simplifying or sparsifying the formulated hypergraph for data placement, which effectively reduces the space and network overhead. The
running time of the hypergraph partitioning algorithm is lowered, determined by the difference between the scale of formulated hypergraphs and hypergraph sparsifiers. By comparing different heuristics on hypergraph sparsification, we adopted a probabilistic method, which achieves a good balance between the performance and complexity. Further, we proposed the scheme to implement the heuristic in a distributed and streaming fashion, which also captures the characteristics of the recent traffic. Finally, by the numerical results, we verified the effectiveness of the proposed scheme.
Chapter 4

Dynamic Data Placement by Reinforcement Learning

4.1 Overview

In order to provision high serving capacity or solve complex analytical problems, various large-scale distributed systems have been implemented \[22\]. They are always constituted by commodity computers and network devices. In these systems, data items need to be moved between computing or storage nodes in the processing of distributed jobs, because data items are not always stored at the nodes where the computation happens. Exploiting the centralized metadata server in distributed storage systems, we can optimize the placement of numerous data items among storage nodes, improving various system performance metrics. Various attempts have been made in the literature \[15, 51, 93\], however, traditional schemes, which are delicately designed based on the refined traffic and performance models, are less effective in adapting to the change of environment. To this end, we present DataBot, a dynamic data placement scheme with the joint-awareness of network conditions and request patterns.

By DataBot, we are to select the storage location for a data item or its replicas at the moment that it is to be written or updated. It utilizes the techniques of reinforcement learning and neural networks to learn the optimal solution for data placement gradually. DataBot is online and on the granularity of storage machines, and thus can be considered as a supplement to offline geo-distributed data placement in Chapter 2 and Chapter 3. The exact storage medium used by DataBot can either
be traditionally disk-based or memory-based (which will be discussed in Chapter 5). Through the conducted evaluations, both the average write and read latency of the whole system are shown to be lowered by DataBot, which justifies the new approach well.

4.2 Related Work

Resource management in large-scale systems is an effective measure to lower the system cost and improve the system performance. Given the limited resources in distributed systems, researchers have tried to make optimizations from different perspectives, such as, task scheduling [104], VM consolidations [55], and resource allocation [92]. Among the methods, data placement improves the data locality to ensure a better data read/write performance in data-intensive systems.

Different from existing methods for data placement, we use machine learning to dynamically obtain a model that automatically learns and adjusts the strategy. In the literature, Jalaparti et al. [59] presented a scheme that jointly places data and tasks to significantly improve the network locality. Agarwal et al. [15] presented an automated data placement scheme for geo-distributed services where the geographical distribution of requests would determine the final data locations. Baev et al. [24] provided some approximation algorithms with provable performance bounds. Besides, our scheme makes data placement decisions at the moment of write, which is similar to the copy-on-write in [82].

We adopt the techniques of reinforcement learning and neural networks in addressing the formulated problems. Kaelbling et al. [61] surveyed the methods in reinforcement learning extensively. The neural network techniques have been used to approximate the Q-function method for reinforcement learning in many literatures, e.g., [48, 70, 79]. Deep reinforcement learning [77] has been proved to be able to play some video games using deep neural networks. Before our work, rare attempt has been made on solving the data placement problem for distributed systems by reinforcement learning and neural networks.
4.3 Network-aware Data Placement Problem

4.3.1 System Model

We would discuss how to dynamically optimize the storage locations of data items in distributed storage systems with intensive data flows. For the system, denote the set of machines or serving nodes in the system as $\mathcal{M}$ with a total number of $M$ nodes, and therefore each node has an index $m \in [1, M]$. Nodes are connected through a datacenter network. We illustrate our sample system in Fig. 4.1. Here the network is in a tree topology for simplicity, but our scheme can support any more complicated topology, e.g., Fat-Tree [17], because the scheme is only based on the measurement of the end-to-end network performance. Each node in the system has both the computation and storage function. For the storage function, data items are stored at nodes in a distributed way. For the computation function, distributed applications run on multiple nodes and may require the movement of data among nodes.

A centralized metadata server is deployed to manage the storage locations of data items. We denote the number of data items by $N$ and one data item by $n$. Depending on the specific scenario, the unit of managed data items could be a data block, table column or file. We assume the same size of data items for the ease of explanation, but the proposed scheme is still effective if data items have different sizes, with the modification of dividing all data-specific metrics by the data size. When a data item is to be written into the system, its storage node is assigned by the metadata server based on the hash tag, i.e., the hash output using the index of data item as the input. Inherently, a mapping between the hash tag and the storage node is maintained at the metadata server. When a node needs to read a data object, it first asks the metadata...
server where the storage node is by using the hash tag. Under this addressing scheme, the storage location of a data item is flexible to be changed, whenever the data item is to be written or updated. By this design, the storage location of a data item is unchanged unless it needs to be updated, so that no movement overhead is introduced even though the proposed scheme occasionally changes data storage locations.

4.3.2 Objectives

As claimed in [46], the data storage locations can affect the finish time of distributed computation tasks. This is mainly because that network becomes a bottleneck when data are intensively moved between computation nodes when fulfilling a job, and a higher network oversubscription ratio would exacerbate the situation. Various attempts have been made to resolve the issues. For example, it was proposed in [38] that the data write flow could always choose the node with low occupancy links in the writing path as the destination, to lower the job finish time. However, the ignorance of read operations in this work may adversely affect the future read-related performance. Here we go one step further, that we propose to design a scheme that takes both the read and write latency into account. Formally, we try to improve the time taken to read/write a data item from/to the storage location, termed as latency below. Considering that the relative importance between read and write may be different in various scenarios, a parameter $\omega$ is introduced to make the tradeoff. Then our problem is defined as: when a data item is to be written or updated, how can we make the choice of where it should be stored among available nodes in order to optimize the performance objectives.

Another feature of our work is that the decisions are only made on the basis of end-to-end measurements for data flows. Specifically, noticing that network performance can largely affect the job finish time, the historical latencies on read/write operations between each pair of nodes are exploited. Besides, because fulfilling data requests is the main cause of data movement, how frequently each data item is requested, defined as request patterns, is also utilized. The measurement of these end-to-end metrics could be easily done by the source nodes of data requests.

Data replication is important in distributed storage systems to overcome machine or network failures. Therefore, we also address the replica placement problem, i.e., which $K$ nodes should be used to store a data item if we need to replicate each data item to $K$ different nodes in the system. Below we start our work with a simple
scenario where each data item is stored in exactly one node. Then, we extend it to the solution for how to choose $K$ different nodes for each data item by the proposed heuristic method.

4.4 Solution Details

4.4.1 Background Knowledge

(a) Reinforcement Learning

There exists difficulty in making accurate modeling of the system, i.e., the random changing of network conditions and request patterns, and the uncertainty between such factors and the system performance. Therefore, we propose to use a generic method, reinforcement learning, to address the formulated problem. Reinforcement learning is inspired by behaviorist psychology, addressing hard optimization problems through a learning process. In reinforcement learning, an agent interacts with the environment and then learns the underlying model utilizing the feedback of its actions. Different from a lot of traditional methods aiming at obtaining the analytical models of physical environments, the adopted reinforcement learning chooses an alternative, which is to purely fit with the statistical patterns of the environment. Besides, reinforcement learning is a widely used for learning closed-loop control policies.

Reinforcement learning provides a new direction of performance optimization for distributed systems. It can potentially overcome the issue of slow reaction to changes in traditional modeling based methods. Meanwhile, as we are still in the early stage of this direction, there are some open questions to explore through follow-up studies, such as what is the highest performance that we can achieve based on this method.

(b) Q-learning

We adopt Q-learning method in reinforcement learning for our formulated problem, which is model-free and designed to find an optimal action-selection policy for any given Markov Decision Process (MDP). The fundamental design of Q-learning is captured by the maintained reward function or called Q-function:

$$Q : State(s) \times Action(a) \rightarrow Reward(r).$$ (4.1)

In Q-learning, an agent will choose an action aiming at maximizing the long-term
reward. After an action is taken, the current state will be changed, and $Q$-function would also be updated according to the feedback of environment.

In our system, $Q$-learning helps to make decisions of choosing write locations. Specifically, when a specific data item $n$ is going to be written in the system at time $t_1$, we first choose the location based on the current state $s_1$ and then execute the write to that location as an action $a_1$. Until we start to overwrite or update the same data $n$ for the next time at $t_2$, we can gather the effect or performance metrics of the last write of $n$ at $t_1$ and the following reads of $n$ between $t_1$ and $t_2$. The weighted sum of the mentioned performance metrics is used as the immediate reward value $r_1$, corresponding to the action $a_1$. Since the write operation at $t_2$, the system jumps to another state $s_2$. Due to the large space of possible states, we choose neural networks to approximately lower the scale of the state space, which is a common way in handling such an issue in $Q$-learning.

For the immediate reward $r_t$ at a time $t$ in $Q$-learning, it is assumed to be still having impact in the future, and discounted under a discount factor $\gamma$ for the future moments. According to [77], if assuming the optimal solution $Q^*(s, a)$ that maximizes the expected long term reward is known, it must satisfy the following condition,

$$Q^*(s, a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a') | s, a] , \quad (4.2)$$

and through the method of value iterations,

$$Q_{t+1}(s, a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q_t(s', a') | s, a] , \quad (4.3)$$

we can approximately converge to the optimal $Q$-function through the iterations of system samples following the state-action distributions. To implement the $Q$-function, the simplest way is to maintain a table representing the mapping relationship among states, actions and rewards. However, the space overhead would be very large and the algorithm would converge very slow, if the state space is very large. Therefore, existing studies have made a lot of attempts to find approximate methods for $Q$-learning, and neural network is the most broadly adopted way in applications of $Q$-learning.
4.4.2 Neural Network Design

(a) States in the $Q$-function (also being the input of the neural network)

To approximate the $Q$-function, a single neural network is maintained in our system, which is illustrated in Fig. 4.2. A neural network consists of connected layers of neurons. The variables involved in a neural network are the weights of the connections between the neurons. Approximating the $Q$-function by a neural network makes the solution suboptimal, but it is practical in term of balancing the achieved performance and overhead. We consider that three categories of information should be monitored by the scheme to make a decision on the storage location when serving a write request of data item $n$ sourced at node $m$.

The first category is about network conditions

$$\{L_{xy}^{[R]}, L_{xy}^{[W]}, \forall x, y \in \mathcal{M}\}, \quad (4.4)$$

which includes: (i) Average latency of read operations on each pair of nodes in the network, $L_{xy}^{[R]}$, where $x, y \in \mathcal{M}$ and they represent the source and destination, respectively; (ii) Average latency of write operations on each pair of nodes in the network, $L_{xy}^{[W]}$. We measure the information above considering the network performance has a large impact on the objective of improving read/write latency. In our design, we ignore the link-based metrics mainly for the reason that the discussed data placement happens on the application layer and is only able to choose the flow destination, but not the path or links of the flow. Besides, to measure and obtain the link-related metrics would introduce a large overhead compared with end-to-end ones. Note that our decisions on storage locations can coexist with the underlying link-based or path-based flow scheduling, because the decisions are on the application layer.
The second category is about request patterns, denoted by

\[ \left\{ F^{[R]}_m, F^{[W]}_m, \hat{F}^{[R]}_m, \hat{F}^{[W]}_m, \forall m \in \mathcal{M} \right\}. \] (4.5)

The features include: (i) Read rate or frequency to data \( n \) from source node \( m \), denoted by \( F^{[R]}_m \); (ii) Write rate to data \( n \) from source node \( m \), denoted by \( F^{[W]}_m \); (iii) Read rate to all data from source node \( m \), denoted by \( \hat{F}^{[R]}_m \); (iv) Write rate to all data from source node \( m \), denoted by \( \hat{F}^{[W]}_m \). With both the request patterns towards all data and the request patterns towards the requested data, the scheme would be able to make a better choice on data storage locations. Such information indicates how applications are generating workload to the system, which is the root cause of the network traffic. The measurement of latencies and rates is implemented in the production system, so its method will be presented in Section 4.4.3(a).

The third category is a 0-1 vector, representing the source location of the current write request. We use 0 or 1 to represent whether each node \( m \in \mathcal{M} \) is the source location of the current write operation. Always there is only one item being 1 in the 0-1 vector. This category is included because the source location of data flow would make a difference to the latency of writing to a specific node.

To determine the write destination locations, an input vector of the neural network, also being the current state, should be obtained, denoted by \( s \), which consists of the above features in 3 categories. We can potentially add more features into the input of the neural network, however, that would introduce larger overhead in terms of measurement and computation. Therefore we make a tradeoff between benefit and cost, and finally include the current ones. According to our design, given the node number \( M \), the length of input vector \( s \) is

\[ |s| = 2M^2 + 4M + M = O(M^2) . \] (4.6)

According to (4.6), the number of data items in the system will not affect the complexity of the neural network deployed.

(b) **Output of the Q-function (also being the output of the neural network)**

The output is a vector with length \( M \), denoted by \( o \). For the Q-learning framework, each output value in \( o \) represents the expected reward of the corresponding action,
or equivalently, the expected reward of writing the data item to the corresponding node. Therefore, in our scheme, to obtain a lower read/write latency, the action corresponding to the largest element in $Q$ should be selected mostly, unless a broader exploration in the solution space is needed. Sometimes, for the sake of searching the unexplored portion of the solution space, a random action will be selected. Therefore, an $\epsilon$-greedy method is applied where $\epsilon$ is the probability of choosing a random action.

(c) Minimization of the Squared Difference

Given a state $s$, an action $a$, and the current weight vector $\theta$ of the maintained neural network, we can calculate the output of the neural network, denoted by $Q(s, a, \theta)$. It is obvious that this result may not be the same as the target, which could be obtained by measuring the consequence of applying the action $a$. Therefore, we should try to update the weight vector $\theta$ in order to minimize the difference or loss. Formally, we define the target value as $y$, and then we are to minimize

$$D(\theta) = \mathbb{E}_{(s,a)} [(y - Q(s, a, \theta))^2].$$  \hspace{1cm} (4.7)

Theoretically, through calculating the gradient of $D(\theta)$, denoted by $\nabla D(\theta)$, we can update the weight vector so as to minimize \hspace{1cm} (4.7). Practically, rather than calculating the expectation in the above function, the stochastic gradient descent method is used, assuming that the distribution of the samples collected by the agent is the same as the underlying distribution of the environment that the agent interacts with.

There exists mature tools to implement formulated neural networks by following their programing interfaces. The tool that we use is Keras \cite{12}. Specifically, we program the structure of the neural network that we formulated first, and then feed the input and expected output for each training entry into the algorithm, making the weight variables updated by the optimization calculation of the tool.

4.4.3 Scheme Design

We design the DataBot scheme as two components, production system and training system. They work asynchronously, so that the training process will not affect the running of the production system. Traditional reinforcement learning is designed as updating the model after each decision, which is serial and therefore will delay the
next request handling. Our modification to the process is shown to be still effective according to the experiment results. In the production system, whenever there is a request to the metadata server that queries the write destination for a specific data, the neural network with weight vector $\theta$ is used to make the decision. Also, a tuple $(s_t, a_t, s_{t+1}, r_t)$ is stored for each request; the tuples for a certain period constitute the replay memory used by the training system, denoted by $R$. In the training system, a weight vector $\theta^+$ is calculated based on $R$. Note that the training system is not completely synchronized with the production system. When the training result $\theta^+$ is obtained, the weight vector $\theta$ in the production system will be set to $\theta^+$. Simply speaking, DataBot constructs the replay memory $R$ in the production system and trains the new weight vector $\theta^+$ in the training system based on $R$.

(a) Production System

Here we clarify the design details of the production system illustrated in Fig. 4.3. The main tasks of the production system consist of serving the requests of querying write locations or read locations, and maintaining the states and replay memory, which are shown as metadata server and state monitor in Fig. 4.3, respectively.

For the requests of querying write locations, the decision is made through the deployed neural network. It uses the information of the current running state and the trained weight vector $\theta$ as inputs, and its output is the vector $o$, each element in which represents the estimated benefit of writing to the corresponding location or node in the system. Based on $o$, we can simply choose the node with the largest value as the write destination, or apply some more complicated policies with other considerations, such as the storage capacity constraint. Then the decisions of write locations are returned to the requesters and stored in the module of storage location. For the request of querying read locations, the response answers would be simply obtained from the stored information.

A key issue in the production system is how the running logs are stored for the training system and for the input of the neural network in making decisions. According to our design, the captured and logged information is in two categories and below we specify how and where they are captured. First, each node captures the time taken for each data transmission, denoted by $l$. This information is not only used to reconstruct the network condition information $L_{xy}^{[R/W]}$, but also used to calculate immediate reward $r$. For the calculation of read/write latency $L_{xy}^{[R/W]}$ in the input layer
of the neural network, we used the Exponential Weighted Moving Average (EWMA) mechanism. Specifically, after a data read/write operation is done, we measure the latency \( l \), and use it to update \( L_{xy}^{[R/W]} \) with \( L_{xy}^{[R/W]} = \alpha l + (1 - \alpha) L_{xy}^{[R/W]} \). Here \( \alpha \) is the discount factor to lower the importance of previous requests. By this method, we only need \( O(1) \) space to maintain the estimated latency for each pair of nodes in the network.

Second, the metadata server captures the request pattern information, which is about when and where a read/write request is initiated to which object, as well as the location returned by the server. We can use such information to reconstruct the second category of information required by the neural networks, i.e., request patterns. The Discounting Rate Estimator mentioned in [18] can be applied here for each type of request rate. It maintains a counter for each item in (4.5), and discounts the counter by a ratio \( \alpha_r \) at the end of constant time intervals. The benefit of this method is that it can ensure the \( O(1) \) requirement of both space and update time for each maintained counter.

The format of log entries maintained in the state monitor module is defined as

\[
(TS, R/W, Src, Dst, Lat) ,
\]

where \( TS \) is the timestamp, \( R/W \) represents the operation type (read or write), \( Src \) and \( Dst \) are the source and destination location of the requests, and \( Lat \) is the time taken to read/write a data item. Actually, the metadata server has all the information by itself except \( Lat \), so only \( Lat \) is necessary to be reported by nodes in our system design. The input of the designed neural network and the replay memory used by the training system can all be obtained by processing the entries.

(b) Training System

The training system periodically obtains the records in the replay memory of the production system, denoted by \( \mathcal{R} \), and replays them to train an updated weight vector \( \theta^+ \) for the future data placement.

The pseudo code of the training system is shown in Algorithm 2. Fundamentally, we process each record \( (s_t, a_t, s_{t+1}, r_t) \) in \( \mathcal{R} \), which contains the information about the current state, the action taken, the next state, and the reward obtained, to calculate the new weight vector \( \theta^+ \), which minimizes the loss between the neural network output and expected result.
Algorithm 2 *Training Method*

1: $\theta^+ \leftarrow \theta$
2: for epoch $i$ in $1, \ldots, I$ do
3:   for each sample $(s_t, a_t, s_{t+1}, r_t)$ in $\mathcal{R}$ do
4:     $y_t \leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a', \theta)$
5:   end for
6: for each batch $B$ with size $B$ in $\mathcal{R}$ do
7:   Update $\theta^+$ to minimize (4.11)
8: end for
9: $\theta \leftarrow \theta^+$
10: end for
11: return $\theta^+$

First, the immediate reward $r$ is defined as the weighted sum of transmission rates measured during time $[t_b, t_e)$, where $t_b$ is the timestamp of writing a data item $n$ and $t_e$ is the timestamp of writing the same data item for the next time. So, representing the set of reading operations on the data item $n$ in $[t_b, t_e)$ by $J$, $r$ is calculated by

$$r = \omega \cdot \frac{1}{l[W]} + (1 - \omega) \frac{1}{|J|} \sum_{j \in J} \frac{1}{l[R]_j}, \tag{4.9}$$

where $\omega$ is a tradeoff between write and read performance, $l[W]$ is the write latency measured at time $t_b$ and $l[R]_j$ is the read latency for read operation $j \in J$. The latency information is maintained by the state monitor module of the production system.
We use the traditional back propagation method to update the weight vector $\theta$ of the neural network. Fundamentally, we define the target output as follows,

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a', \theta),$$

(4.10)

where $r_t$ and $s_{t+1}$ are stored in each record. Then the records in $\mathcal{R}$ are partitioned to subsets or termed as batches, each with size $B$. For each batch $\mathcal{B} \in \mathcal{R}$, we can update the weight vector $\theta^+$ with the gradient method, in order to minimize

$$E_{(s,a) \sim \mathcal{B}} \left[ (y - Q(s, a, \theta^+))^2 \right].$$

(4.11)

In the algorithm, we have multiple iterations on all records to converge faster. The number of epochs is denoted by $I$. Besides, we keep $\theta$ static before all records in $\mathcal{R}$ have been processed; then after a complete round of processing, $\theta^+$ is set to $\theta$. This variant in training makes the optimization objective more stable and therefore alleviates fluctuations in the calculation.

### 4.4.4 Replica Placement

For fault tolerance, many distributed storage systems replicate each data item to multiple storage nodes. Therefore the method above is extended to support replica placement. Note that the output layer of the formulated neural network is a vector $o$, each element of which represents the expected reward of storing a data item at the corresponding location. Here we can further use this vector to generate multiple locations for storing the replicas of the data item.

We make minor changes to the system design. In the production system, we follow the existing procedure to obtain the output vector, but here we choose $K$ nodes instead, which are corresponding to the $K$-maximum elements in the vector, and let them each store one of the $K$ replicas. When the metadata server receives a query of requesting read location from a node $n$, the server simply returns the $K$ locations of replicas to the requesting node. Then the node can choose one from them as the read destination, following some simple policies, such as closest-first. For the training process, because each data write results in replicating the same data to $K$ nodes at the $K$-replica scenario, we need to maintain a separate log entry for each of the $K$ nodes under the write operation. Specifically, $K$ entries will be added to the replay memory when a data write happens. Denote the entry set by $\{e_k, \forall k \in \{1, 2, ..., K\}\}$. 
For the $k$-th entry in the set, which is in the form of $(s_t, a^k_t, s_{t+1}, r^k_t)$, the definition of state $s_t$ and $s_{t+1}$ are still the same as the scenario without replica, action $a^k_t$ defines the node where the $k$-th replica is determined to be placed, and reward $r^k_t$ is the immediate reward we measure for the $k$-th replica.

## 4.5 Performance Evaluation

### 4.5.1 Experiment Settings

For performance evaluation, we emulated a system with 64 nodes and 1,024 data items, each with the same size 40,960 Bytes. Mininet \[9\] is used to emulate the datacenter network, which can create a virtual network running real Linux-based application program. The 64 nodes are in a tree topology with 3 levels. The core switch connects to 4 aggregation switches, each aggregation switch connects to 4 edge switches, and each edge switch connects to 4 nodes. The capacity of links is set to 100 Mbps. The emulation does not choose a large scale or set large values to parameters, due to the constraints of the emulation tool and its physical computation resources. It is still representative to reflect the performance issues in a network-based system, because we still emulated the hierarchical layers of datacenter network and the link bandwidth used in the emulation is properly set to reflect the shared nature of the network.

We implemented the client program that initiates data read and write requests at each node. The read/write request rates from a node to each data item are randomly generated following the Zipf distribution. The average read-write ratio is set to 4 : 1. The inter-request time to the same data item follows the exponential distribution, whose mean value is inversely proportional to the generated request rate. Memcached \[8\] is used as the end of a data flow and functions as the data store. Simply speaking, each node has a client being the source of requests and a Memcached process being the destination of requests.

A metadata server program is implemented to handle control flows, whose functions include state monitoring, write destination decision and neural network training. Using one of the experiments lasting 3,000 s as an example, there are totally 16,810 write requests and 67,017 read requests processed by the metadata server. We consider 2 scenarios in evaluation. One is write-optimized with $\omega = 1.0$, and another is read-optimized with $\omega = 0.2$, where $\omega$ is the tradeoff parameter between
the importance of write and read. The two scenarios are shown as W-optimized and R-optimized in the figures. For the purpose of comparison, a standard hash-based data placement widely used currently [10] is used as the reference. The reason that we did not choose other schemes with data locality consideration is that other existing schemes are based on different inputs and mostly with strong assumptions on the application scenario. In addition, the proposed scheme is not aiming at exceeding them at the current stage, which is left to the future after follow-up improvements and fine tuning of neural network models.

Keras [12] and Theano [3] are used as the front end and back end framework for neural network implementation, respectively. For the neural network that we used in all experiments, it has 8,512 features in the input layer, determined by (4.6), and there are 64 outputs, each representing a storage node in the system. Besides, there are two hidden layers which have 6,000 and 3,000 neurons, respectively, whose settings are empirical to make a tradeoff between the complexity and performance. Increasing the number of neurons can improve the decision performance, but consume more resources. For the number of layers, much less benefit will be obtained if there are more than 2 layers due to the training mechanism [79]. System operators can choose the setting that satisfies their preference by multiple trial runs. The adjacent layers are fully connected and the Rectified Linear Unit is applied to each neuron in the hidden layers, fulfilling the activation function.

4.5.2 Experiment Results

(a) Reward Results

We start by showing the experiment results about the accumulated reward of reinforcement learning in the 3,000 s running time of the system. Note that the time of 3,000 s is long enough to reach a steady performance improvement ratio, as shown in the later experiment results. The reward is obtained when a write or read request is fulfilled and the reward value is defined in (4.9). The parameter settings of the training system are as follows if not especially mentioned. For each round of training, the system obtains the last 1,500 samples from the replay memory and processes them with \( I = 6 \) epochs and \( B = 300 \) batch size, which takes 65 s typically until the new weight vector \( \theta^+ \) takes effect in the production system. There is a 100 s sleeping time between the training rounds to collect more samples.

For the write-optimized scenario, the results are shown in Fig. 4.4. It shows that
the accumulated reward of the proposed **DataBot** begins to outperform the compared reference around 700 s, which is the moment that the second trained model is applied to the production system. With the continuous training, the improvement ratio through **DataBot** will change from increasing to steady, shown as a linear increase in the figure. In Fig. 4.5, we show the results of the read-optimized scenario. Here the improvement is less significant because of less improvement space in the read performance, i.e., when multiple requests to the same data item may come from different source locations, only a portion of the requests can obtain benefit from the selected storage location. Still, we can observe some improvement compared with the reference, showing that the proposed scheme is still effective.
We also measure the average write or read latency in every 100 s of the experiment duration. The results for the write-optimized scenario are shown in Fig. 4.6. We observe that the average write latency is in the trend of decrease when time is passing. This is because the reinforcement learning applied would need several rounds of training with numerous samples to get the performance close to optimal. For each 100 s in the last 1,000 s, DataBot can reduce the average write latency by 39.7% on average and by 50.3% in the best case. The performance shows some variance in the figure, which is because the requests are not uniformly generated along the time. When the overall request rate in a short period is high in the system, the latency tends to increase. The reason that the curve shows some periodicity is that the congestion periodically appears and disappears due to the workload.

For the read-optimized case, we show the results in Fig. 4.7. Here for the last 1,000 s, the read latency improvement is reduced by 15.9% on average and by 34.3% in the best case. It is not as significant as the improvement of the write latency in the previous case. The reason could be explained as: each write decision will affect the performance of multiple following read requests from different source locations in the network; when network latency is mainly influenced by the difference between the source and destination location of flows, the average read latency of these following requests will have a limited space for improvement.
(c) Parameter Discussion

Some parameters in the training process can affect the resultant performance, so we further discuss the difference due to the parameter change. First, the number of training epochs, denoted by $I$ in Algorithm 2, could affect the performance, represented by the accumulated reward. We change $I$ to 2, 4, 6 and 8. The experiment results are shown in Fig. 4.8. When the epoch number is higher, generally there is a trend that the performance is better. That is because more rounds of training tend to rectify the difference between the reward expected and that given by the neural network. However, when the number is increased to 8, we observe the degradation of performance, which can be explained as over-fitting, i.e., even if the return of the loss
function (4.7) in neural network training is further decreased, the obtained model is not effective for the future test data.

Second, the batch size $B$ in Algorithm 2 is varied and tested. It represents how frequently the weight vector $\theta^+$ is updated in the training process. As shown in Fig. 4.9, when setting the batch size to 300, the result is better compared with the other settings. This suggests that a careful selection of the batch size parameter can also help to improve the achievable performance. With these discoveries, we are interested in finding a more systematic way to properly set the parameters and to avoid over-fitting in the future.

(d) Replica Scenario

We further check the performance for the scenarios with replicas. The replica number of each data item is set to 3 here. First, for the accumulated rewards, there is a significant improvement compared with the reference in the write-optimized case as shown in Fig. 4.10. Fig. 4.11 plots the accumulated rewards for the read-optimized case and it also justifies that the learning process takes effect with the continuous running of the system.

Fig. 4.12 plots the latency results for the write-optimized case, which shows that even though we need to write the same data item to 3 different nodes under the replication setting, the scheme can still effectively choose the nodes with lower write latencies. Here for the last 1,000 s, DataBot reduces the latency by 41.9% on average and 55.7% in the best case. The latency comparison for the read-optimized case is
plotted in Fig. 4.13, which shows the reductions as 16.0% on average and 31.0% in the best case. Similar to the case without replicas, the improvement extent of read is still less than write, because although here each data item is stored at 3 nodes, it still can only help a portion of read requests, which may come from any of the 64 nodes in our experiment. Besides, we observe that both the reference and the proposed scheme can obtain benefits from the fact that data are replicated to multiple locations, in terms of lowering the read latency, by comparing the read latencies shown in Fig. 4.13 and Fig. 4.7.
We show how the read/write latencies of data requests for the last 1,000 s are distributed for the replica scenario in Fig. 4.14 and Fig. 4.15. The latency distribution could reveal how the performance is changed for a certain percentage of requests, e.g., 90%, after introducing the scheme. Fig. 4.14 plots the distribution for the write-optimized case, where we can observe that more than 90% of the write requests are finished in less than 101.1 ms by the proposed scheme while being 178.6 ms for the reference. This justifies that the proposed scheme is also very effective in ensuring the worst-case write latency under a certain percentage threshold.

Fig. 4.15 plots the distribution for the read-optimized case. There seems to only
be a slight difference on the read latency between DataBot and the reference, but it is still meaningful considering the limited improvement space of the read latency. From the distribution, we can conjecture that the read latency is improved more because of the increased percentage of low latency requests. For over 20% of read requests, the latency is less than 30.1 ms by DataBot, which is 51.0 ms by the reference. This also suggests that when only a limited number of nodes hold replicas, a large portion of high latency read requests cannot be effectively improved by our current design due to the broad distribution of read request sources. This reminds us that extra mechanisms are needed for applications sensitive to the read latency, such as applying full duplications of replicas or introducing memory-based caches. Meanwhile, we can also observe some improvement on the write latency, but the extent is smaller than
what is shown in the write-optimized case.

4.6 Further Discussions

In this work, we exploit reinforcement learning to solve the data placement problem. We find it effective here and suggest that more resource management problems for distributed systems could be potentially solved using similar methods. For instance, towards the path selection problem in a multi-path datacenter networks [53], we can potentially model the state space as link states and take the path selection for each flow as an action, while treating the experienced path rates as rewards. We summarize several general requirements to the scenarios that reinforcement learning is suitable for as follows. First, the system should allow some random actions, taken without serious performance degradation, which is needed in exploring the optimal solution. Second, the decision problem to address should be to select a choice or a set of choices among a limited available set and it needs to deal with a dynamically changing system environment. Third, the network-related or application-specific metrics and objectives should be able to be converted to the states and rewards in modeling, e.g., a continuous value in a limited range is easier to handle.

We should also mention that there are still some limitations in applying the reinforcement learning approaches to some decision problems in resource management. First, the huge number of states in the system modeling would introduce the state space explosion problem. The problem will not only consume much resource in the training and decision process, but also slow the convergence time since more records or samples are needed to cover the huge space. Second, there exists a tradeoff between the exploration and exploitation in designing the scheme. We have to take some actions randomly sometimes in order to discover the potential better choices and accumulate more knowledges of the unknown space, which however, may adversely affect the performance. Therefore, more research and practices are needed to overcome such challenging issues, such as the innovations on neural network structures and random action policies.

4.7 Conclusions

We introduced machine intelligence into the data placement problem in this chapter. The problem formulated is to dynamically determine the storage location when a
data item in the storage system is to be written or updated. Due to the uncertainty of factors that affect the data placement performance such as network conditions and user request patterns, we proposed a generic solution based on Q-learning and neural networks to simplify the modeling process. The proposed solution was implemented as two components: production system and training system, in order to overcome the overhead in training. The experiment results showed that the proposed scheme can reduce the average write and read latency by up to 55.7% and 31.0%, respectively. In the future, we would attempt to apply different types of neural networks to the system and try to further improve the performance. Besides, a general framework that can formulate most decision problems in distributed systems into a learning process would be helpful and meaningful.
Chapter 5

Auto-scaling of Memory Cache Clusters

5.1 Overview

To alleviate the high traffic and workload coming into the storage systems, the memory-based caching subsystem is added to the system typically. The memory-based caching subsystem provides temporary key-value storage in memory and joins the memory of distributed servers in a cluster as a whole, functioning as a unified cache covering the entire key space. Memcached is the state-of-the-art and most popular implementation method. It achieves the horizontal scalability mainly through the consistent hashing, where the key space is partitioned into segments and each server is responsible for one segment or sub-space of the whole space. Besides, it also achieves the vertical scalability through the mirroring of one sub-space, i.e., assigning multiple servers to redundantly cover the same sub-space and distributing the incoming traffic to them through a load balancer.

The dynamic provisioning of servers in a memory cache cluster helps to improve the system efficiency, leading to a reduction of energy cost, but it is overlooked in the existing design. In this chapter, an auto-scaling scheme is proposed for the dynamic provisioning of memory cache clusters. We first measure the cache hit rate, request batching effect and cache warm-up time of the system through experiments, considering that they can affect the system performance and efficiency. Then we formulate a stochastic network optimization problem, which aims at achieving objectives on the queue stability, energy cost and cache hit rate simultaneously, through the dy-
namic control of cluster scaling and request dispatching. The problem is transformed into a minimization problem in each time slot, which is further addressed through the proposed efficient online algorithm based on dynamic programming. Finally, the proposed algorithm is evaluated through extensive experiments.

5.2 Related Work

The distributed memory cache system [95] is broadly used in different networked services today and consistent hashing technique [62] is applied to support the scalability. How Facebook leverages and improves Memcached, a typical implementation of the cache system, to support its social network was provided in [80]. Raindel et al. [86] considered the distributed memory storage system, and discussed how to serve the multi-get requests to achieve the maximum throughput. Byers et al. [28] discussed the method in achieving load balance among servers under consistent hashing through mapping a key to multiple servers. Gomaa et al. [52] introduced a novel analytical model for estimating the cache hit rate and proposed a frequency based cache replacement policy. Such work mostly tries to improve the scalability, throughput or cache hit rate, but the possibility of adjusting the server on/off status to balance performance and efficiency is overlooked.

The theory of stochastic network optimization [78] is developed based on the system capacity analysis [94] and Lyapunov optimization. It helps to make optimized decisions only with current observations, not relying on the knowledge of arrival rate distribution. Urgaonkar et al. [102] proposed that the throughput and energy of a data center can be optimized through the admission control and routing control. Zhou et al. [116] solved the tradeoff between the performance and cost in the Virtual Machine (VM) resource pool through the framework with specific considerations on the nonlinear energy consumption and the power budget. Maguluri et al. [74] focused on task scheduling under the cloud scenario, and also used the Lyapunov optimization as the main method to solve the problem. Under the setting of geo-distributed datacenters, Zhou et al. [117] proposed a scheme to achieve green cloud through load balancing, capacity right-sizing and server speed scaling. Wang et al. [104] used the technique to stabilize the Map-Reduce tasks where the data locality limits the set of feasible servers to run a task. Join-the-Shortest-Queue and MaxWeight are shown to be effective for the system stabilization in most of such work.

In this chapter, we discuss the dynamic control of memory cache clusters. The
dynamic control is important considering the dynamics of workload [101], which was not well discussed in the design of memory cache clusters. Compared with the other work under the topic of dynamic system provisioning [73] or task scheduling [45], the properties of memory cache clusters are taken into account in our work, e.g., the dependence between the provisioning status and request dispatching.

5.3 System Design

5.3.1 Consistent Hashing

In typical distributed memory cache systems, such as Memcached [8], the cache servers form a cluster to work cooperatively to fulfill the data item requests initiated by the clients. Consistent hashing [62] is applied in dispatching requests to different servers, which results in each single server only covering a sub-space or segment of the whole key space and is only responsible for serving the requests falling into the range it covers. Considering that the workload in requesting data items with specific keys is dynamic, the provisioning of the servers, i.e., turning them to be active or inactive, should be controlled dynamically to tradeoff the resultant service quality and energy cost.

The consistent hashing method [62] is adopted by most memory cache clusters for the purpose of scaling, Based on it, we formulate the system as follows. The whole key space is denoted by $S$. Each key is a non-negative integer between 0 and the space size. We simplify the consistent hashing scheme as equally dividing $S$ into $N$ segments, denoted by $S_1, ..., S_N$. Formally $\bigcup_{k=1}^{N} S_k = S$ and $S_i \cap S_j = \emptyset$ for any $i \neq j$. Then the segments form a directional circle based on their indexes, such as $S_k$ is the predecessor of $S_{k+1}$ and $S_N$ is the predecessor of $S_1$, which is similar to Chord [91], a typical DHT system.

5.3.2 Cluster Modeling

There is a pool of available servers, which can be physical machines or virtual machines. We can allocate a server from the pool and set its status as active, which increments the number of servers serving requests. Also we can turn an active server to inactive, and release it to the pool. Each active server is assigned to one of the $N$ groups, and each server in group $i$ owns the key segment $S_i$, where $1 \leq i \leq N$. 
Meanwhile, if a server is active, it might also help to serve the segments preceding $S_i$ in the circle when necessary as specified below. The maximum number of active servers in each group is assumed to be $M$. Totally, there are at most $N \times M$ servers considered in our scheme.

We model the time as discretized into slots. In each time slot $t$, the on/off status of a server can be changed. Accordingly, the on/off status of a server $j$ in group $i$ is denoted by $a_{ij}(t)$ and the value 1 or 0 is used to represent the status of being active or inactive, where $1 \leq j \leq M$. We adopt a linear model for server power consumption modeling, according to [47]. The dynamic power consumption of an active server is modeled as a linear function $P_{idle} + P_{work} \times CPURate$. Here our dynamic provisioning design focuses on saving the first part $P_{idle}$. Given the total amount of tasks in the system, the total CPU workload is almost a constant, resulting in a constant power consumption on the second part of the model, so only how to reduce the first part of the model for the whole system is discussed in this work. Comparatively, $P_{idle}$ is saved if turning a server to inactive in some time slots. For physical machines, the inactive status saves energy directly, while for virtual machines, the energy saving is indirect, but the saving of monetary cost is directly related to the cloud users due to the release of resources.

We further model the on/off of a server group $i$ by

$$a_i(t) = 1(\sum a_{ij}(t)),$$  \hspace{1cm} (5.1)

where $1(x)$ is an indicator function, with the return value 1 if $x > 0$, or 0 otherwise.
Then we obtain the *binary provisioning vector* \( Y(t) \), such as

\[
Y(t) = \{a_1(t), ..., a_N(t)\},
\]

which represents the status of server groups in slot \( t \). Fig. 5.1 shows an example with \( N = 4 \) and \( M = 3 \). Active servers or server groups are drawn with solid lines while inactive servers or server groups are drawn with dotted lines. As illustrated, the whole group 2 and some servers in group 1 and group 4 are set to be inactive. Here \( Y(t) = \{1, 0, 1, 1\} \).

Determined by the consistent hashing applied in dispatching requests to servers, the responsible key space of each active server group \( i \) includes its own key segment \( S_i \) and the key segments of one or multiple consecutively inactive groups preceding it in the key-space circle (shown with a dash-dot style in Fig. 5.1), denoted by

\[
\hat{S}_i = \bigcup_{k=f+1}^i S_k,
\]

where \( f \) is the first active server group preceding \( i \) in the circle. In the example illustrated by Fig. 5.1 the servers in group 3 will also be responsible for the segment owned by group 2 due to the inactive status of the latter. Meanwhile, the servers in the same group cover the same sub-space of keys, so there would be multiple candidates in serving requests to that sub-space. Changing the status of a server normally only impacts the workload of the other servers in the same group, but it can also influence the workload of the other groups if it changes the group on/off status, which introduces another dimension of control in the server provisioning.

Besides, we set another layer between clients and servers in the system, which is shown as *proxy* in Fig. 5.1. A proxy consists of logical relays responsible for each one of the \( N \) key segments. The requests falling into the same key segment, even if from different clients, will be aggregated at the same relay, leading to a higher performance due to the diminishing overhead effect to be explained later. Another advantage of the proxy is making the dynamic adjustment of server provisioning transparent to the clients, because the traffic from clients is relayed by the proxy to the servers.

### 5.3.3 Design Objectives

Intuitively, we aim at turning servers to inactive as much as possible when the incoming traffic of the system is low, since it will help to lower the energy consumption of
the cluster. However, other issues should also be considered to avoid degrading the user experience. For example, the service latency of each request should not be too high and the overall cache hit rate should not be too low. Therefore, how to make server provisioning and request dispatching decisions will be proposed in Section 5.4, which only relies on the queue lengths at the time of decision as the input of the algorithm.

5.3.4 Measurement Results

To optimize the server provisioning and request dispatching, some service-specific factors that affect the resultant performance are investigated. For the cache services we
discuss in the chapter, we will analyze the influence of the key space size, diminishing overhead by batching and the cache warm-up time, through the experiments either emulated on a real server with Intel E5-2670 CPU or simulated based on service access logs. The measurement results are going to be used in the design of the optimization algorithm. Note that although the empirical results obtained here are specific to the servers and traces we used, interested system operators can still simply follow the same procedure to obtain the results suitable for their own systems.

(a) Covered Key Space vs. Cache Hit Rate

For a lower average content retrieval latency, either the cache hit rate or cache access latency should be improved. Controlling the average queueing latency at cache servers helps to achieve an expected cache access latency, which will be discussed in Section 5.4.2. Meanwhile, increasing the number of active server groups can improve the cache hit rate, because more servers might serve a smaller key space. When the unequal distribution of workloads in different key segments is ignored, the expected cache hit rate $C_i$ of server group $i$ is a function of the ratio of its responsible space size over the whole space size, such as $C_i = C(|\hat{S}_i|/|S|)$. An experiment is conducted to determine that relationship, based on the trace of HTTP requests to Wikipedia [100]. We simulate 10 LRU-based cache servers, each with the same cache capacity. LRU is commonly used in existing system implementations because it balances between performance and overhead. Depending on the provisioning status, an active server might be responsible for $n/10$ of the whole space, where $1 \leq n \leq 10$. $n$ is adjusted
from 1 to 10 and the results are shown in Fig. 5.2. Under each cache size setting, 10 samples of cache hit rate are obtained, each with different covered space ratio \( n/10 \). We notice that either a smaller key space or a larger cache size can increase the cache hit rate.

Based on the samples obtained from measurement, we use the function-fitting tool in Matlab to approximately model the relationship between the covered space ratio \( x \) and the cache hit rate \( C(x) \). When each server can store 64,000 data items at most, we obtain \( C(x) \) as

\[
C(x) = -0.257x^3 + 0.586x^2 - 0.495x + 0.830 ,
\]

where \( 0 \leq x \leq 1 \).

(b) Diminishing Overhead by Request Batching

The diminishing overhead effect by batching multiple data items in requests was initially termed as the multi-get hole [6] in the distributed cache service. Specifically, when the average number of data items to fetch through each request increases, the request processing efficiency improves, enlarging the system capacity. The reason for the efficiency improvement can be explained as follows: each request has a certain overhead, such as the packet header or session maintenance; batching the retrieval commands of multiple data items to the same request lowers the overhead amortized to each object.

To take the effect into account in the dynamic server provisioning, some experiments are conducted with physical servers. We keep clients sending requests repeatedly, each of which is to obtain \( x \) data items by using the multi-get command in the protocol of memcached. With \( x \) varied between 1 and 10, we measured the average number of requests served per second, denoted by \( T(x) \). Then the average delay of any request containing \( x \) data items, normalized to that of any request containing 1 object, is calculated as \( M(x) = \frac{1/T(x)}{1/T(1)} = \frac{T(1)}{T(x)} \). Our experiment results on the normalized mean delay with varying \( x \) and the number of servers are shown in Fig. 5.3. Due to the parallel overhead, increasing the number of servers will increase the normalized latency to retrieve each data item. Based on the samples from measurement as shown in Fig. 5.3, we obtain a linear approximation

\[
M(x) = \sigma + \tau x ,
\]
which can be intuitively understood as for each request, no matter how many data items are in it, there is an initial overhead $\sigma$ in delay, and another part of the delay is determined by the number of data items in it, with a rate of $\tau$. In the experiment with only one server, $\sigma \approx 0.6$ and $\tau \approx 0.4$.

(c) Cache Warm-up Time

The cache warm-up time is analyzed through a Wikipedia trace [100]. The trace is captured between Sep. 2007 and Jan. 2008. In the trace, we are given lines of records, each containing the timestamp and the URL of the captured request. The individual request information provided in this trace is useful to our experiment. Because each of our experiments lasts 45 seconds, only a small portion of entries in the trace from the beginning time are actually used. Through re-playing the captured log on real servers we can estimate the cache warm-up time by measuring the relationship between cache hit rate and the timestamp change. After a certain time of running, the average cache hit rate would reach the steady state, and we can use the necessary time before the steady state as the cache warm-up time, which further helps to model the cache warm-up overhead. Since the trace does not provide the exact size of each object, here we assume the homogeneous 10 Kbyte size of each object. We make measurements with the varied memory size in the cache server, and the results are shown in Fig. 5.4. We notice that with the increase of memory size, generally the highest reachable cache hit rate increases, and the time to reach it also increases. According to the experiment results, we assume that 20 seconds are necessary to get the cache hit rate into the steady state for a cold-boot server, especially considering that commodity cache servers are always with a large-size memory. Based on this measurement, we can estimate the overhead of switching a server to active from inactive in the system design below.

5.4 Proposed Solution

5.4.1 Preliminaries

We will utilize the theory of stochastic network optimization (SNO) [78] to optimize the server provisioning and request dispatching in the discussed cache clusters. The theory is applicable to problems that optimize the time averages such as network throughput. One of its advantages is that it can make decisions without knowing
the future. In the theory of SNO, the dynamics of the system are modeled based on discrete time slots and its running status is represented by the dynamic queue backlogs \( Q(t) \). The sum of squares of queue backlogs in time slot \( t \) is defined as Lyapunov function \( L(t) \), a measure of system congestion. Under the theory, we should greedily minimize the drift \( \Delta(t) = L(t+1) - L(t) \) in two adjacent slots, in order to make the queues less congested, which results in the queue stability. In addition, the drift-plus-penalty method was proposed to jointly optimize the time-averaged queue length and the corresponding penalties determined by the applied decisions.

5.4.2 Optimization Problem

Based on the modeling presented in Section 5.3, the incoming requests are queued respectively at each relaying queue based on the key of the data item to fetch. Each client request may involve multiple queues, and the destination queue is determined by the hashing result of the index of requested item. There are \( N \) queues in total, corresponding to the \( N \) segments of the whole key space. \( H_k(t) \) is defined to represent the queue backlog at relay \( k \), also corresponding to the segment \( S_k \), where \( k = 1, 2, ..., N \). When the time goes from \( t \) to \( t + 1 \), the queue backlog of relay \( k \) is updated by

\[
H_k(t + 1) = \max(H_k(t) - d_k(t), 0) + A_k(t) \tag{5.6}
\]

where

\[
0 \leq d_k(t) \leq d_{\text{max}} \tag{5.7}
\]

\( A_k(t) \) denotes the number of data items to retrieve by the newly arrived requests at time \( t \), whose keys are in segment \( S_k \). \( d_k(t) \) denotes the number of requested data items planned to be dispatched from the queue \( k \) to cache servers at time \( t \), constrained by the maximum rate \( d_{\text{max}} \). In our theoretical problem formulation, we assume that \( d_k(t) \) is completely controllable, but in the solution shown below, its value is only required to be set to the maximum rate or zero in each time slot. Therefore, its control can be easily implemented as whether allowing the departure of each queue in each time slot. Because the actual number of departing data items is limited by the number of existing data items in the queue, we use \( \hat{d}_k(t) \) to denote the actual number of data items fulfilled at relay \( k \) in the time slot \( t \), such as

\[
\hat{d}_k(t) = \min(d_k(t), H_k(t)) \tag{5.8}
\]
For the cache servers, the queue backlog at server \( j \) in group \( i \) is denoted by \( Q_{ij}(t) \). It decreases by \( l_{\text{max}} \) at most in each time slot if the server is in the active status, i.e., \( a_{ij}(t) = 1 \). So the server queue backlog is updated by

\[
Q_{ij}(t + 1) = \max(Q_{ij}(t) - a_{ij}(t)l_{\text{max}}, 0) + A_{ij}(t) ,
\]

where \( A_{ij}(t) \) is the number of data items newly arrived at the server in the time slot \( t \).

The dispatching of requests from the relays to the cache servers is determined by the active status of server groups. An active server group \( i \) will be responsible for its own segment \( i \) as well as the segments of its inactive predecessors. So the exact arrival \( A_{ij}(t) \) at a server is determined by both the provisioning vector \( Y(t) \) and corresponding departure amounts \( \hat{d}_k(t) \) in the related relays. If we consider that the departure \( \hat{d}_k(t) \) at relay \( k \) can be distributed to multiple servers in the same group \( i^* \), which will be determined later, we have

\[
\hat{d}_k(t) = \sum_{j=1}^{M} d_{kj}(t) ,
\]

where \( d_{kj}(t) \) is the sub-amount dispatched to server \( j \) among \( \hat{d}_k(t) \). Then the actual arrival amount to server \( j \) in group \( i \) could be represented by

\[
A_{ij}(t) = \sum_{k=f+1}^{i} M(d_{kj}(t)) ,
\]

where \( f \) is the first index before \( i \) satisfying \( a_f(t) = 1 \). The function \( M(x) \) is from the modeling of diminishing overhead in (5.5). Serving a request to retrieve a larger number of data items can lower the latency shared by each data item in the request. So \( M(x) \) can be intuitively considered as there is a discount on the counted number of arrived data items at servers when \( x \) is larger than 1.

We have defined two types of queues in the above modeling, and then \( H \) and \( Q \) are used to represent the set of relay queues and server queues. In optimizing the system, our first objective is to stabilize the queues in \( H \) and \( Q \), which ensures that the time-averaged queue length is bounded, so that a certain cache access latency is ensured. Also we try to maximize a utility function in each time slot to achieve
certain other goals. The utility function is defined as

\[ U(t) = - \sum_i \sum_j a_{ij}(t) + \gamma \frac{\sum_{i=1}^N C_i(t)a_i(t)}{\sum_{i=1}^N a_i(t)}, \quad (5.12) \]

where \( C_i(t) \) is the cache hit rate of server group \( i \) at time \( t \), obtained from (5.4). In (5.12), the first item represents the energy consumption in keeping some servers active at time \( t \), and the second item represents the benefit of cache hit, modeled as the average hit rate of all the active server groups. Besides, \( \gamma \) is the weight of the cache hit rate performance which is to tradeoff the energy cost and cache hit rate.

In order to achieve the goals specified above in an optimal way, we are to design an online algorithm that controls two sets of variables, i.e., the cache server on/off status \( a_{ij} \), and the dispatched amount \( d_{kj} \) from the relay \( k \) to the cache server \( j \) in the corresponding server group. Following the stochastic network optimization [78], Lyapunov drift is used in quantifying the queue stability. First, we define the \textit{Lyapunov function} on the queue backlog of \( H \) and \( Q \), such as

\[ L(Q(t), H(t)) = \sum_i \sum_j Q_{ij}(t)^2 + \sum_k H_k(t)^2, \quad (5.13) \]

which is abbreviated as \( L(t) \) in the following. Then the \textit{conditional Lyapunov drift function} at time \( t \) is obtained as

\[ \Delta(Q(t), H(t)) = E\left\{ L(t + 1) - L(t) | Q(t), H(t) \right\}, \quad (5.14) \]

which represents the change of Lyapunov function conditioning on the known queue backlogs in a previous time slot.

Minimizing the drift function can potentially ensure the queue stability or system stability. With the other goals on the energy cost and cache hit rate considered, we are to minimize the \textit{drift-plus-penalty function}, defined as

\[ \Delta(Q(t), H(t)) - VE\left\{ U(t) | Q(t), H(t) \right\}, \quad (5.15) \]

where \( V \) is a parameter to tradeoff the importance of the queue stability and utility. Here we use the minus of the utility as the penalty in the drift-plus-penalty method [78].

With the queue backlog functions defined in (5.6) and (5.9) and based on the
stochastic network optimization, (5.15) is relaxed to a minimization problem in each time slot \( t \), such as

\[
\min \quad 2 \sum_i \sum_j Q_{ij}(t) [A_{ij}(t) - a_{ij}(t)l_{\text{max}}] \\
+ 2 \sum_k H_k(t) [A_k(t) - d_k(t)] - VU(t) \quad (5.16)
\]

s.t. (5.7) (5.8) (5.10) (5.11)

In our scheme, the problem is solved in every time slot to give instructions on how to control the server provisioning and request dispatching of the system. In the problem, note that \( Q_{ij}(t) \), \( H_k(t) \) and \( A_k(t) \) are known parameters in each time slot, and the values of other variables are to be determined.

### 5.4.3 Online Algorithm

Below we clarify how to obtain the optimal solution of (5.16) by the proposed algorithm. The pseudo-code of the algorithm is shown in Algorithm 3.

The binary provisioning vector \( Y(t) \) determines the destination server group when requests are dispatched from the relays and it affects the request arrival at each server group. Intuitively, we need to iterate all the 0-1 combinations of the vector \( Y(t) \) to search the best solution of (5.16); in each iteration, the vector of \( Y(t) \) is given, so we obtain the best conditional solution based on that specific \( Y(t) \); after the iterations, we will choose the minimum in all the conditional solutions.

In fact, to traverse all the 0-1 combinations of \( Y(t) \) could be improved through dynamic programming, which saves the intermediate results and avoids the repeated calculation of them. Obviously we can iterate all the cases where only one server group is active in \( Y(t) \) using \( O(N) \) time, because there are \( N \) cases.

Then we discuss the case that there are at least two server groups being active. Because the organized structure of servers is in the form of a circle, such as group \( N \) is the predecessor of group 1, the two server groups chosen to be active divide the circle into two sub-structures. We use \((l, r]\) and \((r, l]\) to represent the two sub-structures, where \( l \) and \( r \) are the index of the two groups known to be active. ( and ] are used to represent that the left side is not included and the right side is included in the two sub-structures, respectively. If the optimal solutions for server groups in \((l, r]\) and \((r, l]\) are both known, denoted by \( D^{(l,r]} \) and \( D^{[r,l]} \), the optimal solution of at least \( l \)
Algorithm 3 Server Provisioning Decision and Request Dispatching Decision

**Input:** queue length of each \( H \) and \( Q \)

**Output:** server provisioning and request dispatching

1: for \( len \) in \([0, N - 1]\) do
2:     for \( l \) in \([0, N - 1]\) do
3:         \( r \leftarrow (l + len) \mod N \)
4:         \( D[l,r] \leftarrow DSol(l, r) \)
5:     end for
6: end for
7: for \( l \) in \([0, N - 1]\) do
8:     for \( r \) in \([l, N - 1]\) do
9:         if \( l \) is equal to \( r \) then
10:            if \( D[l,r].obj < minVal \) then
11:                \( minVal \leftarrow D[l,r].obj \)
12:                \( solRet \leftarrow D[l,r] \)
13:            end if
14:        else
15:            \( solTmp \leftarrow \text{Merge}(D[l,r], D[r,l]) \)
16:            if \( solTmp.obj < minVal \) then
17:                \( minVal \leftarrow solTmp.obj \)
18:                \( solRet \leftarrow solTmp \)
19:            end if
20:        end if
21:     end for
22: end for
23: return \( solRet \)

and \( r \) being the active server groups can be obtained by merging the results of \( D[l,r] \) and \( D[r,l] \), assuming that they have been obtained in an earlier stage. Using this method, we can iterate all the \( \binom{N}{2} \) cases and obtain the best solution conditioned on at least two active server groups.

In the implementation of the proposed dynamic programming, we need to maintain a matrix structure \( D \) with size of \( N \times N \). In the first round, we iterate all its items \( D[l,r] \) satisfying \( r - l = 0 \). Further, in the \( k \)th round all the items \( D[l,r] \) satisfying \( r - l = k - 1 \) would be solved and filled. Note that the server groups form an circle, so we also need to solve \( D[l,r] \) in the \( k \)th round when \( r - l < 0 \) and \( r + N - l = k - 1 \).

In Algorithm 3, through lines 1 to 6, we obtain the optimal solution to the sub-problem \( D[l,r] \) through dynamic programming. The solutions to the sub-problems are saved in the matrix \( D \). Then through lines 7 to 22 in Algorithm 3, we choose the optimal solution in two categories of solutions. One category is that there is exactly
one active server group, which are done between lines 9 to 13; another is that there are at least two active server groups, which are done between lines 14 to 18. The function $Merge$ is used here, which is quite simple because the two sub-solutions to be merged represents two non-overlapping segments and thus are independent.

### 5.4.4 Sub-solution $D^{(l,r]}$

The sub-solution $D^{(l,r]}$ can be obtained through dividing $(l, r]$ into two sub-structures further. Consider we have $l < m < r$, where $m$ is an item in the vector $Y(t)$ with value 1 and with a position between $l$ and $r$. Then we can obtain the solution of $(l, m]$ and $(m, r]$, and merge their results as a candidate of the best solution for $(l, r]$. Obviously, we need to iterate all the $m$ between $l$ and $r$. Besides, an extra candidate that needs to be considered is that all the items between $l$ and $r$ are 0. This decomposition follows the principle of dynamic programming. Formally, we can obtain the solution of a specific range $(l, r]$ from the sub-solutions of the range, such as,

$$D^{(l,r]} = \min \left\{ \min_{l < m < r} \left\{ D^{(l,m]} \oplus D^{(m,r]} \right\}, I^{(l,r]} \right\},$$

where $I^{(l,r]}$ means the optimal solution for $(l, r]$, assuming that item $l$ and $r$ in $Y(t)$ are set to 1 and all the items between $l$ and $r$ are set to 0. Besides, the operation $D^{(l,m]} \oplus D^{(m,r]}$ merges two non-overlapping sub-solutions together, which is the same as the $Merge$ function specified above.

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$s_6$</th>
<th>$s_7$</th>
<th>$s_8$</th>
<th>$s_9$</th>
<th>$s_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D^{[3,8]}$</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(1.1)</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(1.2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(2.1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(2.2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(3.1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(3.2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>*</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>(4.1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>(4.2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(5)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1 gives an example of obtaining the sub-solution $D^{(l,r]}$ through iterating
and merging sub-structures. In the table, 1/0 means the server is active/inactive and * means the server can be active or inactive in the solution. Here we consider there are 10 groups of servers. At the moment, the sub-problem considered is $D^{(3,8)}$, i.e., $s_3$ and $s_8$ are known to be active. Under the problem, the first candidate is merged from $D^{(3,4)}$ and $D^{(4,8)}$ which are labeled as (1.1) and (1.2) in the table, respectively. The second is merged from $D^{(3,5)}$ and $D^{(5,8)}$, which are labeled as (2.1) and (2.2), respectively. Similarly the cases with $m = 6$ and 7 should be considered. Another case is, $I^{(3,8)}$ which assumes all the groups between $s_3$ and $s_8$ are inactive, and is labeled as (5). By iterating all these cases, the solution for $D^{(3,8)}$ could be obtained through choosing the one with the minimum objective value among the candidates.

In the pseudo-code, we obtain the sub-solution $D^{(l,r)}$ through function $DSol(l, r)$. In $DSol(l, r)$, we choose the optimal solution for the sub-problem $(l, r)$ among the candidate solutions in two categories. One category is by merging the two sub-problems of $(l, r)$, done at line 4 to 10. Another category is by assuming $r$ is the only active server group in the range of $(l, r)$, which is done at line 11 through function $ISol(l, r)$. 

---

**Function 1 Function $DSol$**

1: function $DSol(l, r)$
2:  $minVal \leftarrow \text{Double.max}$
3:  $minM \leftarrow -1$
4:  for $m$ from $l + 1$ to $r - 1$ in circle do
5:    $solTmp \leftarrow \text{Merge}(D^{[l,m]}, D^{[m,r]})$
6:    if $solTmp.obj < minVal$ then
7:      $minVal \leftarrow solTmp.obj$
8:      $minM \leftarrow m$
9:  end if
10: end for
11: $solI \leftarrow ISol(l, r)$
12: if $solI.obj < minVal$ then
13:   return $solI$
14: else
15:   return $\text{Merge}(D^{[l,m]}, D^{[m,r]})$
16: end if
17: end function
5.4.5 Sub-solution $I^{(l,r)}$

The sub-solution $I^{(l,r)}$ can be obtained following the analysis below. Note that $I^{(l,r)}$ is the optimal solution for the sub-structure $(l, r]$ assuming that only the server group $r$ is active among $(l, r]$. Besides, due to the dispatching rule, we only need to consider the relays in $(l, r]$. Below we show our theoretical analysis of optimal decisions for server provisioning and request dispatching, and clarify the implementation of function $ISol$, which helps to obtain $I^{(l,r)}$.

(a) Dispatching Destination

We determine the value of $d_{kj}(t)$ first, which is related to constraint (5.10) and (5.11) in problem (5.16). Assuming the dispatched amount $d_k(t)$ of all the considered relays and the status $a_{ij}(t)$ of all the considered servers are given (with $d_k(t)$ determined by $d_k(t)$), the objective of (5.16) would be simplified to the minimization of

$$2 \sum_{i} \sum_{j} Q_{ij}(t) A_{ij}(t) = 2 \sum_{i} \sum_{j} Q_{ij}(t) \sum_{k=f+1}^{i} M(d_{kj}(t)) . \quad (5.18)$$

The problem can be decoupled on $i$, so each active server group is considered respectively. On the left side of (5.18), $Q_{ij}(t)$s are in fact the coefficients of a sum, and $A_{ij}(t)$s are possible to be adjusted to make the sum smaller. First, it is more beneficial to dispatch the requests to a shorter queue, as the queue length is the weight in the sum. Second, there is no benefit to partition $d_k(t)$ into parts and dispatch each part to different servers in the destined group, due to the fact that $M(x_1) + M(x_2) = T(1)/T(x_1) + T(1)/T(x_2) \geq T(1)/T(x_1 + x_2) = M(x_1 + x_2)$ holds for any positive $x_1$ and $x_2$. So the value of $d_{kj}(t)$, given the requests from relay $k$ would be dispatched to the group $i^*$, should be set to

$$d_{kj}(t) = \begin{cases} \hat{d}_k(t), & \text{if } j = \arg \min_j Q_{i^*j}(t) \text{ with } a_{i^*j}(t) = 1 , \\ 0, & \text{otherwise} \end{cases} \quad (5.19)$$

which can be intuitively understood as Completely-Join-the-Shortest-Queue. Therefore, in terms of the sub-solution $I^{(l,r)}$, the departure of each relay in $(l, r]$ should all be dispatched to the shortest active server queue in group $r$. 

(b) Dispatching Amount

Then the solution of \( d_k(t) \) is to be determined. Here we still assume the status status \( a_{ij}(t) \) of servers in each group is already determined. With such a prerequisite, the objective function of (5.16) is simplified to minimize

\[
-2 \sum_k H_k(t)d_k(t) + 2 \sum_i \sum_j Q_{ij}(t)A_{ij}(t),
\]

where the order of items in the sum is kept the same as in (5.16). It can be further decoupled on \( H \), so in fact we are to minimize

\[
-H_k(t)d_k(t) + Q_{i^*j^*}(t)\sigma 1(\hat{d}_k(t)) + \tau \hat{d}_k(t), \tag{5.21}
\]

where \( i^* \) is the destined group of relay \( k \) determined by \( Y(t) \) and \( j^* \) is the server being active and with the shortest queue in the group. The indicator function \( 1(\hat{d}_k(t)) \) is used to reflect that the dispatching amount could be 0. It is obvious that if \( d_k(t) = 0 \), (5.21) is 0, therefore we set \( d_k(t) \) to be positive only if it makes (5.21) negative. First, we assume \( 1(\hat{d}_k(t)) = 1 \). Because of the min operation in (5.8), there are two cases when comparing between \( d_{\text{max}} \) and \( H_k(t) \):

- In the case of \( d_{\text{max}} < H_k(t) \), we need to minimize \(-H_k(t)d_k(t) + Q_{i^*j^*}(t)\tau d_k(t)\), so we set \( d_k(t) \) to the maximum value \( d_{\text{max}} \) if \( H_k(t) \geq \tau Q_{i^*j^*}(t) \), otherwise to 0.

- In the case of \( d_{\text{max}} \geq H_k(t) \), \(-H_k(t)d_k(t) + Q_{i^*j^*}(t)\tau \min(H_k(t), d_k(t)) \) is to be minimized. Here we obtain the objective value \(-H_k(t)d_k(t) + Q_{i^*j^*}(t)\tau H_k(t) \) if \( H_k(t) < d_k(t) \), and \(-H_k(t)d_k(t) + Q_{i^*j^*}(t)\tau d_k(t) \) if \( H_k(t) > d_k(t) \). For the former, the minimum value is obtained when we choose \( d_k(t) = d_{\text{max}} \), which is \( H_k(t)(-d_{\text{max}} + \tau Q_{i^*j^*}(t)) \). For the latter, the minimum value could be 0 with \( d_k(t) = 0 \) or \(-H_k(t)H_k(t) + Q_{i^*j^*}(t)\tau H_k(t) \) with \( d_k(t) = H_k(t) \). But due to the initial condition \( d_{\text{max}} \geq H_k(t) \), the latter could not be better than the former unless the minimum objective value is 0. So we set \( d_k(t) \) to the maximum value \( d_{\text{max}} \) if \( d_{\text{max}} \geq \tau Q_{i^*j^*}(t) \), otherwise to 0.

In summary, we have the generalized solution

\[
d_k(t) = \begin{cases} 
  d_{\text{max}}, & \text{if } \max(d_{\text{max}}, H_k(t)) \geq \tau Q_{i^*j^*}(t) \\
  0, & \text{otherwise}
\end{cases}, \tag{5.22}
\]
Note that we have assumed $1(\sum \hat{d}_k(t)) = 1$. We further check whether the obtained $d_k(t)$ through the analysis above makes (5.21) less than 0. If not, $d_k(t)$ should be changed to 0. In fact, since there are only two possible optimal answers for $d_k(t)$ in the above analysis, we can simply test each of them and choose the one that makes (5.21) smaller. In terms of sub-solution $I^{[l,r]}$, the dispatching amount decision is done at line 5 and 6 in the function $I\text{SolWithSh}$ to be specified below. In the function, we determine the value of $d_k(t)$ for each relay in $(l, r]$.

(c) Cache Server Status

The server on/off status $a_{ij}(t)$ is still not determined yet. In our theoretical solution based on SNO, an active server with a nonempty queue might be turned off after the status decision, but in a real scenario, the decision can be implemented as the server will be turned off after all the requests in its queue are fulfilled. Besides, our problem formulation ensures that there would be no requests dispatched to a server that is determined to be inactive in the solution.

Note that if a server is set to be active, there are two possible reasons: to dispatch requests to it from the corresponding relays or to make it serve the requests that are already in the queue. After decoupling (5.16) on $i$, then in any group considered, when the server with the shortest queue among the active servers is given, denoted by $s$, the variable part in the objective function is simplified to

$$-2 \sum_j Q_{ij}(t)a_{ij}(t)l_{\text{max}} + V \sum_j a_{ij}(t).$$  (5.23)

Because each server has a queue, below we represent a server by its queue. Now we make decisions about whether to keep a queue longer than $s$ active in the group. Intuitively, to keep a longer queue active has a larger potential to make (5.23) negative, because it implies the larger weight $Q_{ij}(t)$ in the sum. Then in the solution, by iterations, each of the queues is assumed be the shortest in turn; in each iteration, we further iterate all the queues longer than $s$ following the decreasing order of the queue length and set those servers that can still keep (5.23) negative to be active. For a given server group, the computational complexity of this method is $O(M^2)$, because of the two-level iterations.

In the function of $I\text{Sol}$, we iteratively assume a queue in the group $r$ as the shortest active one in that group, and make decisions under such an assumption through the
Function 2 Function ISol

1: function ISol(l, r)
2:  listSorted ← preSort({Q_r1, Q_r2, ..., Q_rM})
3:  for sh in [0, M - 1] do
4:    solTmp ← ISolWithShort(l, r, sh, listSorted)
5:    if solTmp.obj < minVal then
6:      minVal ← solTmp.obj
7:      solRet ← solTmp;
8:  end if
9:  end for
10: return solRet
11: end function

function ISolWithSh. Note that the relay dispatching decision and the server status decision for the sub-problem are in fact implemented in ISolWithSh. The former is done between lines 4 and 10 and the latter is done between lines 11 and 18. The data structure sol.HDecision stores the decisions about the dispatching of the related relay queue in H while the data structure sol.QDecision stores the decisions about the status, arrival amount and service amount of the related server queues in Q.

5.4.6 Status Changing Overhead

Previously, only the running cost of servers is considered in the design. However, the extra cost/overhead in changing the running status of servers is inevitable in practice. So we further consider the overhead of changing the on/off status of a server and try to avoid the unnecessary overhead in that part. Typically, after turning a server to active from inactive, a certain pending period is necessary before the server is fully ready to use. In detail, if a physical server is turned to active, we will boot it first and warm up the cache to an acceptable cache hit rate by the replicated flows (but the replies will not be actually returned to the clients during warm-up). Similar pending period happens to Virtual Machines (VM) and it will consume power but cannot provide meaningful service, and thus attracts our attention to discuss how to avoid the frequent changing of the server provisioning status.

First we model that there are two stages in such a period: 1) the booting time of the server, denoted by BootTime; 2) the warm-up time of the cache, which is specific to the cache service, denoted by CacheTime. For the former, according to the measurement in [75], the average user-experienced boot time for a small instance
Function 3  Function ISolWithSh
1: function ISolWithSh(l, r, shortest, list)  
2:    s ← getIndexSorted(list, shortest)  
3:    solRet ← new Solution()  
4:    for k from l + 1 to r in circle do  \ Relayed Departure  
5:        gain ← −Hkdmax + Qrs(σ + τ min(dmax, Hk))  
6:        if gain < 0 then  
7:            solRet.HDecision.addDepart(k, r, s, dmax)  
8:            solRet.QDecision.addArrival(r, s)  
9:        end if  
10:    end for  
11:    for j in [0, shortest − 1] do  \ Server Departure  
12:        j′ ← getIndexSorted(list, j)  
13:        if −2 · Qrj′ · lmax + V < 0 then  
14:            solRet.QDecision.addDepart(r, j′, lmax)  
15:        else  
16:            break  
17:        end if  
18:    end for  
19:    solRet.obj ← GetObj(l, r, solRet)  
20:    return solRet  
21: end function

of Virtual Machine (VM) from Amazon EC2 is around 96.9 seconds. The time for a
Type IV instance from Rackspace is better, but still more than 40 seconds. For the
latter, we have made some measurements based on the real data trace, and shown
that the cache warm-up can take more than 20 seconds, in Section 5.3.4(c).

Considering overhead in optimizing the decision, we need to add a new factor to
the utility function (5.12). As a result, the function is updated to

\[ U(t) = - \sum_i \sum_j p_{ij}(t) + \gamma \sum_{i=1}^{N} C_i(t) a_i(t) \sum_{i=1}^{N} a_i(t) , \] (5.24)

where \( p_{ij}(t) \) is defined as the power consumption at time \( t \) considering both the
running cost and server status changing overhead. Note that we should be more
conservative in turning active servers to inactive, so we define \( p_{ij}(t) \) as

\[ p_{ij}(t) = a_{ij}(t) + \frac{\theta}{t - \sup_{x=0}^{t-1} xa_{ij}(x)} \cdot a_{ij}(t - 1)[1 - a_{ij}(t)] , \] (5.25)
where the parameter $\theta$ is an empirical parameter related to the tradeoff between the running cost and the status changing overhead. If $\theta$ is larger, it means that we are more conservative to turn an active server to inactive. Through dividing it by $t - \sup_{x=0}^{t-1} x a_{ij}(x)$, $\theta$ is decaying along with the time, starting from the last time slot that the server is being inactive. This ensures that the running server will always have some chance to be turned to inactive. Compared to the initial utility function, (5.25) can be intuitively understood as there is an extra overhead if we are changing a server from active to inactive.

Since the newly modeled $p_{ij}(t)$ could be considered as a substitute to the $a_{ij}(t)$ in the previous modeling, we do not need to make much change to the existing algorithm. Note that previously in function $ISol$, we obtain a set of candidates for the sub-solution $I^{(l,r]}$, while for each item in the set, a different queue in group $r$ is assumed to be the shortest. Based on the decision results, we re-calculate the objective value of (5.16) for the related relays and servers in $(l,r]$, with $a_{ij}(t)$ substituted by $p_{ij}(t)$, and then we choose the one with the minimum value as the return value of function $ISol$.

### 5.4.7 Algorithm Analysis

In the proposed dynamic programming approach, there are $O(N^2)$ iterations to obtain and fill all the items of the matrix $D$, and in each iteration, the complexity is $O(N)$ because there are at most $N$ groups between $i$ and $j$, so the complexity in obtaining the matrix is $O(N^3)$. After that, we need to iterate all the possible cases that there are at least two items in the vector $Y(t)$ being 1, which consists of $\binom{N}{2}$ cases, therefore it has the $O(N^2)$ complexity. In each of the $\binom{N}{2}$ cases, such as $i$ and $j$ are selected to be 1 in $Y(t)$, we merge the solution of $D^{(l,r]}$ and $D^{(r,l]}$ in the matrix as the solution of that case. Then after iterating all the $\binom{N}{2}$ cases, we choose the one with the minimal objective value as the best solution among the cases with at least two groups being active. The complexity of this step is $O(N^2)$. So the total complexity in iterating all the possible cases of group status is $O(N^3)$. With the computation in determining the active servers in each group considered, which is $O(M^2)$, the total complexity of our scheme is $O(N^3 M^2)$.

Because the problem is formulated under the stochastic optimization framework, the performance of our scheme follows Theorem 4.2 in [78], considering that the defined $U(t)$ is bounded and the drift condition still holds in our formulation. The
performance bounds characterize the performance difference between the proposed algorithm and any other randomized schemes without the knowledge of queue backlogs in each time slot. First, the time-averaged sum of the lengths of all queue backlogs through the proposed algorithm is in the ratio of $O(V)$, compared with the randomized schemes. The average queue length implies the average service latency through Little’s Law. Second, the minus of time-averaged utility through the proposed algorithm is in the ratio of $O(1/V)$ compared with the randomized schemes.

As shown in the analytical performance bound $O(V)$, the algorithm can make the tradeoff between service latency and other metrics depending on the chosen value of $V$. When $V$ is larger, the service latency (reflected by the average queue length) will become larger; but meanwhile, the performance metrics related to the defined utility will improve. In fact, the theoretical performance bounds shown here are not very tight, so in Section 5.5, more meaningful performance evaluation results are shown through simulations.

**5.5 Performance Evaluation**

**5.5.1 Simulation Settings**

We simulated a memory cache cluster, with 20 server groups and 10 servers in each group. The implemented program simulates the cluster through maintaining the queue backlogs in discrete slots. At the beginning of a slot, the implemented algorithm takes the backlogs as inputs to make decisions on the server provisioning and request dispatching, whose results affect the departure on queues, i.e., the reduction of backlogs. Then a certain number of requests is generated under some distribution, being added to the backlogs. At the end of each slot, we measure the queue backlogs and other performance metrics.

By default, the control granularity is each slot and the length of each slot is 1 second in real time. The investigated performance metrics include: time-averaged number of active servers (which represents the power costs), time-averaged cache hit rate, time-averaged queue length (which can affect the queueing latency) and total objective (which considers the first three metrics as a whole). In the simulation, we measure the value of metrics in each time slot and then calculate the average. Besides the average performance, we sort the instantaneous measurement of above metrics in different time slots, and then obtain the values of the 5% and 95% percentile in the
The maximum serving capacity of servers and those of relays are set as follows: $l_{\text{max}} = 70$ thousand-items/second for servers and $d_{\text{max}} = 1,800$ thousand-items/second for relays, representing how many data items can be processed per second by each server or relay, respectively. Each experiment runs 500,000 time slots with $V$ set to 10,000 by default. The status changing overhead is set to 60, considering that 40 s and 20 s are typical BootTime and CacheTime through the discussion in Section 5.4.6. Speaking in another way, in the simulation, we will add an extra energy cost if turning an server to inactive from active, which is 60 times the cost in keeping a server active for one time slot.

The mean arrival rates of the 20 key segments, denoted by $\lambda_1, ..., \lambda_{20}$, are set based on the statistics of the Wikipedia trace \[4\]. The part in the trace that we use is the access logs for Dec. 2007. In the trace, we are given the number of requests to each web content in each hour. These hourly request rates can be directly used as the input of our scheme, although the information of each request was not provided in this trace. By hashing the name of each content into an integer in $[1, 20]$, representing the 20 segments, we obtain the number of requests to each segment in each hour, based on which, we generate the hourly request arrival rates used in the simulations. We will always keep the ratio $\lambda_1 : \lambda_2 : ... : \lambda_{20}$ the same as the hourly rates extracted from the trace, but also scale the values of $\lambda_1, ..., \lambda_{20}$ to a certain extent, in order to simulate different traffic intensities.

The number of requests falling to each segment in the first hour of the trace is
shown in Fig. 5.5. The largest one is at segment 3, because the requests to \texttt{Main\_Page} fall to that segment. In the simulation, the average of arrival rates, \( \bar{\lambda} = \sum_{k=1}^{20} \lambda_k / 20 \), is set to 700 thousand-items/second by default which makes the largest arrival rate \( \lambda_3 \) near the system capacity. In each slot \( t \), the actual amount of arrivals for segment \( k \), denoted by \( A_k(t) \) in (5.6), is randomly generated through sampling a uniformly distributed random variable with mean value \( \lambda_k \), i.e., \( A_k(t) \sim \mathcal{U}(0, 2\lambda_k) \).

### 5.5.2 Performance Insights

Fig. 5.6 plots the objective value of the obtained solution for (5.16) under different average arrival rates. For each arrival rate \( \bar{\lambda} \), an experiment was taken. For the
purpose of illustration, we slightly move the bars horizontally under each arrival rate setting to avoid the overlap between them. Each bar shows both the average value of the entire experiment and the variation. It is obvious that with the increase of arrival rate, the obtained objective value is decreased. This because the increased amount of departure statistically lowers the average queue length. Besides, the increase of $V$ results in the slight increase of the objective value, since the increased weight of the utility function would introduce more positive components to the objective value. To obtain some further insights, we take extra experiments to obtain several detailed metrics, such as the average queue length, the number of active servers and the resultant cache hit rate.

First, how the change of parameter $V$ in (5.15) influences the queue length at
servers and relays is investigated. We expect that with the increase of $V$, the average queue length would be increased, because it was shown in stochastic network optimization that the scheme ensures an $O(V)$-approximation of the queue length. Fig. 5.7 shows the results of the time-averaged queue length of servers, relays and active servers with the increasing value of $V$. The average queue length at relays is shown to reach the steady state, because under the shown range of $V$, the throughput capacity at servers is still higher than the actual arrival rate. Besides, the increase of $V$ results in fewer servers being active to save energy and makes the average queue length increased.

We varied the overall average arrival rate $\bar{\lambda}$ and compared the results. Fig. 5.8 shows the evaluation result on the time-averaged queue length and Fig. 5.9 shows the resultant number of active servers. The smaller value of $V$ leads to the shorter queue length, however it results in a larger number of servers being active. Besides, when the workload becomes higher, the active number of servers and the queue length will increase. We also noticed that when $\bar{\lambda}$ is comparatively larger, the effect of $V$ in lowering the number of active servers is weakened, since the arrival rate is close to the system capacity.

Fig. 5.10 gives the cache hit rate performance with varying $\bar{\lambda}$ and $\gamma$, the tradeoff between cache hit rate and energy cost. When the workload is comparatively low, we see the effect of parameter $\gamma$ in helping the cache hit rate to increase is stronger. In the extreme case, the cache hit rate is improved from about 69% to 80%. When the arrival rate is near the system capacity, the same extent of increasing $\gamma$ leads to a much less noticeable effect on the cache hit rate. The reason is that when the arrival
rate is higher, the queue length at servers in certain groups would be longer, and then those groups have to be kept active, which leads to a lower cache hit rate.

### 5.5.3 Handling Dynamic Traffic

Two extra experiments were conducted to verify the performance under the dynamic or even bursty changing workload. Here the used mean arrival rates $\lambda_1, \ldots, \lambda_{20}$ are not constants any more. They are varying in different slots following the patterns to be specified below.

The first experiment lasts 7,200 slots (120 minutes) and is about the bursty workload. We set the average of mean arrival rate $\overline{\lambda} = 200$ initially, but in the time period of $[2400, 4800]$, it was increased to 300 while keeping the same ratio among all segment rates, so that the workload was largely increased in that period. We monitored the average server queue length, number of active servers and number of active server groups in every 600 slots. Fig. [5.11] shows that although the bursty workload increases the average queue length during the bursty period, after the workload decreases, the queue length decreases in quite a short time. Besides, the increased number of active servers during the bursty period shows that the scheme can adapt to the dynamic workload. In addition, in Fig. [5.11] we can also observe the slight change of the number of active server groups controlled by the proposed algorithm.

We took another experiment based on the trace [4] in the first 3 days of March 2013. In the trace, we are given the hourly request counts for each web content, so that the used mean arrival rate $\lambda_k$ will be different on different hours. First, the hourly

![Figure 5.11: Handling bursty traffic](image_url)
request counts for all the items in the 3 days are shown in Fig. 5.12. We can observe the periodic workload change every 24 hours, which somehow justifies the necessity to dynamically adjust the server provisioning. Then we apply the proposed algorithm under the arrivals with mean rates changing hourly, and show the resultant metrics in Fig. 5.13. The results validate that the active number of servers is dynamically changed along with the change of workload, by the proposed algorithm. Besides, the average queue lengths of all the servers keep stable under the change of workload and server provisioning.
5.5.4 Performance Comparison

Finally, the performance of our scheme was compared with others. For all the schemes compared, we adjust parameters to make the resultant number of active servers similar, so the average queue length is the metric of comparison. In the other schemes compared, the ratio of the number of active servers in different groups is set to be proportional to the workload predicted by Exponentially Weighted Moving Averaging (EWMA) [57]. Then the exact servers to be active in each group can be selected randomly (to represent the hash-based solution) or by giving a higher priority to the servers with a larger queue backlog (MaxWeight). Besides, the dispatching from a relay to an active server can be random or by Join-the-Shortest-Queue (JSQ). The results after the first 5,000 slots are shown in Fig. 5.14. It shows that the proposed scheme outperforms the others when $\lambda$ is large. Besides, the performance of the JSQ+MaxWeight scheme is quite close to the proposed, especially when the mean arrival rate is low, because it fundamentally applies some similar approaches.

5.6 Conclusions

We investigated the dynamic server provisioning and request dispatching in the distributed memory cache services and proposed the online scheme, which aims at achieving the queue stability, low energy cost and high cache hit rate. The stochastic network optimization framework was applied in devising the online algorithm to obtain the optimized solution, and dynamic programming was used to lower the algorithm
complexity. Through extensive simulations, we evaluated the performance of the proposed scheme, showing that it can balance the considered objectives and adapt to the dynamic workload. The scheme proposed in this chapter would bring a higher cost efficiency to the memory cache clusters, which can be used as either a cache to enhance the traditional storage services or a memory-based store that manages the intermediate results of distributed computation.
Chapter 6

Conclusions

This dissertation proposed to exploit the data location flexibility to optimize the data-intensive distributed systems, aiming at improving the user-perceived performance and lowering the system cost. Realizing the importance and constraints of controlling the data locations in distributed systems, various attempts were made in order to achieve the desired performance and address the cost issues in a principled fashion, either through choosing a favorable storage location for each data item or by controlling the system provisioning following the existing data addressing rules.

The proposed hypergraph models for data placement, as described in Chapter 2, made the attempt to optimize multiple performance objectives that are related to or affected by the data storage locations. It especially modeled the differences among locations for complex data services where each data flow may involve multiple data items, when compared with the existing work. The treatment of the location differences is meaningful under the geo-distributed cloud environments, where each location or datacenter in the system can have significantly different properties, e.g., the rental costs and the nearby user communities.

Then some efforts were taken to overcome the overhead of the proposed scheme itself, realizing that the number of data items managed by the scheme could become much larger in the future with the development of data-centric services or products. The solution proposed in Chapter 3 tried to approximately construct the hypergraph models in a distributed way. By maintaining the compressed data structures termed as sketches at distributed sites and utilizing them to construct hypergraph sparsifiers, the space taken to maintain the history information, the traffic on transferring the necessary data to the controller, and the time taken by the hypergraph partitioning algorithm were all significantly reduced.
To overcome the potential network bottleneck in a distributed storage system, an online data placement scheme that intelligently changes the storage locations of data items was discussed in Chapter 4. Here the storage locations are dynamically determined at each moment when the data item is written or updated on the granularity of machines, therefore this online scheme is also an important supplement to the schemes under the geo-distributed setting, which are offline and on the granularity of datacenter or region. Besides, the proposed scheme made a pioneering attempt on solving the problem using reinforcement learning, instead of the typical analytical models.

The distributed cache services are commonly believed to be an effective enhancement to the distributed storage systems by lowering the workload of the latter. The auto-scaling scheme proposed in Chapter 5 brought an improved running efficiency to the memory cache clusters. The auto-scaling is needed considering that the workload to systems shows regular peak-valley periods. Through the introduced stochastic network optimization framework, the proposed scheme can make the tradeoff between the energy cost of the cluster and other performance metrics, such as the request queuing delay and cache hit rate.

In the future, we need further efforts to answer the general question: What will the future data-intensive distributed system look like? To answer it, it is necessary to speculate what the future trends of system infrastructures will be and how the demand of system users will become. It can be foreseen that the unit cost of hardwares will keep reducing while the system scales will continue to grow, so the designs from the perspective of the entire system will become more meaningful than those only for a single entity or portion of the system. Also, more digital contents today are generated by individuals than before. The consideration of user behavior patterns would become more helpful to the system efficiency. Schemes proposed in this dissertation have utilized the information about user behavior patterns, and in the future, more detailed information, such as how the popularity of user-generated contents evolves along with the time, is possible to be considered and utilized further.

For the near future, the schemes proposed in this dissertation can be extended to cover more application scenarios by considering more application-specific details. For the data placement problem, we model it with the measured user request rates. This makes it suitable for different application scenarios, but how the application-specific metrics could be reflected by the system metrics that we have formulated can be analyzed further. In addition, different models such as bipartite graphs can be
investigated about their applications in the same problem. Meanwhile, new methods can be tried to further enhance the benefits obtained from the proposed ideas. Data characteristics were mainly used to improve the application layer in our studies, and it can be further utilized to improve the lower layers with cross-layer design [105]. Classical neural networks have been used in dynamically determining the data write destination. We can potentially introduce different neural network structures to the proposed scheme. For example, convolutional neural networks [77] have been successfully applied to image recognition due to its support of high-dimensional data input, and maybe the high-dimensional input property can be used in our scheme to lower the complexity. Besides, people can also study how some existing knowledge about distributed systems and applications can be introduced to the designed reinforcement learning system, in order to further improve the system.

In the long run, we expect that the systems should be able to take autonomic control in terms of discovering the optimal settings of its controllable variables. Some work in this dissertation has validated that when data locations are considered as an important factor that can affect how systems perform, a reasonable control of them would bring many desired benefits, but still, more generic methods are needed to achieve these benefits in a more autonomic way. Besides, in the future, the capabilities of hardwares and infrastructures will advance while the focus of system optimizations may also evolve, therefore a middle-layer software that can abstract the objectives and the underlying system properties is needed, so that existing schemes can be easily adjusted and applied to the new scenarios.
Bibliography


[96] Stefano Traverso, Kevin Huguenin, Ionut Trestian, Vijay Erramilli, Nikolaos Laoutarlis, and Konstantina Papagiannaki. Social-aware replication in geo-


