Brokering Algorithms for Data Replication and Migration Across Cloud-based Data Stores

Yaser Mansouri

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

March 2017

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

Brokering Algorithms for Data Replication and Migration Across Cloud-based Data Stores

Yaser Mansouri
Principal Supervisor: Prof. Rajkumar Buyya

Abstract

Cloud computing provides users with highly reliable, scalable and flexible computing and storage resources in a pay-as-you-go manner. Data storage services are gaining increasing popularity and many organizations are considering moving data out of their in-house data centers to the so-called Cloud Storage Providers (CSPs).

However, reliance on a single CSP introduces challenges in terms of service unavailability, vendor lock-in, high network latency to the end users, and a non-affordable monetary cost to application providers. These factors are vital for the data-intensive applications which experience a time-varying workload, and the providers of these applications require to offer users storage services at an affordable monetary cost within the required Quality of Service (QoS). The utilization of multiple CSPs is a promising solution and provides the increment in availability, the enhancement in mobility, the decline in network latency, and the reduction in monetary cost by data dispersion across CSPs offering several storage classes with different prices and performance metrics. The selection of these storage classes is a non-trivial problem.

This thesis presents a set of algorithms to address such problem and facilitates application providers with an appropriate selection of storage services so that the data management cost of data-intensive applications is minimized while the specified QoS by users is met. The thesis advances this field by making the following key contributions:

1. Data placement algorithms that select storage services for replication non-stripped and stripped objects respectively, with the given availability to minimize storage cost and with the given budget to maximize availability.
2. A dual cloud-based storage architecture for data placement, which optimizes data management cost (i.e, storage, read, write, and potential migration costs) and considers user-perceived latency for reading and writing data as a monetary cost.
3. The optimal offline algorithm and two online algorithms with provable performance guarantees for data placement, which exploit pricing differences across storage classes owned by different CSPs to optimize data management cost for a given number of replicas of the object while respecting the user-perceived latency.
4. A lightweight object placement algorithm that utilizes Geo-distributed storage classes to optimize data management cost for a number of replicas of the object that is dynamically determined.
5. Design and implementation of a prototype system for empirical studies in latency evaluation in the context of a data placement framework across two cloud providers services (Amazon S3 and Microsoft Azure).

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Yaser Mansouri, 2 March 2017

Preface

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2- 6 and are based on the following publications:

- **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Data Storage Management in Cloud Environments: Taxonomy and Survey," *ACM Computing Surveys*, ACM Press, New York, USA, 2016 (under minor review).

- **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Brokering Algorithms for Optimizing the Availability and Cost of Cloud Storage Services," *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2013, IEEE CS Press, USA)*, Bristol, UK, Dec. 2-5, 2013.

- **Yaser Mansouri** and Rajkumar Buyya, "To Move or Not to Move: Cost Optimization in a Dual Cloud-based Storage Architecture," *Journal of Network and Computer Applications (JNCA)*, Volume 75, Pages: 223-235, ISSN: 1084-8045, Elsevier, Amsterdam, The Netherlands, November 2016.

- **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Cost Optimization for Dynamic Replication and Migration of Data in Cloud Data Centers," *IEEE Transactions on Cloud Computing (TCC)*, DOI:10.1109/TCC.2017.2659728, 2017.

- **Yaser Mansouri** and Rajkumar Buyya, "Dynamic Replication and Migration for Data Objects with Hot-spot and Cold-spot Statuses across Geo-distributed storage Data centers," *Journal of Parallel and Distributed Computing*, ElSEVIER, 2017 (under review).

Acknowledgements

I cherish the opportunity of pursuing my doctoral study under the supervision of Professor Rajkumar Buyya. I would like to express my sincere gratitude to him for continuous mentoring, monitoring, guidance, and technical advices during my PhD.

I would also like to thank the members of PhD committee: A/Prof. Egemen Tanin and Prof. Rao Kotagiri, for their constructive comments and suggestions on my work. I am also thankful to post-docs in the CLOUDS Laboratory: Dr. Rodrigo Calheiros and Dr. Amir Vahid Dastjerdi for their insightful discussion and guidance on the technical side.

I am fortunate to have collaboration with Dr. Adel Nadjaran Toosi who was always generous with his time to answer my questions for improving my work. I will never forget his kind and friendly attitude and his characters will continue to inspire me.

I would like to thank all members of the CLOUDS Laboratory. Special thanks to Dr. Deepak Poola, Dr. Chenhao Qu, and Safiollah Heidari for their sincere friendships and for the wonderful moments we shared in the tea time. To my friends, Dr. Sareh Fotuhi and Dr. Maria Rodriguez for their inputs on technical writing and presentations of my work. My thanks to fellow members: Dr. Mohsen Amini Salehi, Dr. Anton Beloglazov, Dr. Linlin Wu, Dr. William Voorsluys, Dr. Nikolay Grozev, Atefeh Khosravi, Jungmin Jay Son, Bowen Zhou, Farzad Khodadadi, Liu Xunyun, Caesar Wu, Minxian Xu, Sara Kardani Moghaddam, Muhammad H. Hilman, Redowan Mahmud, Muhammed Tawfiqul, Yali Zhao, and to visitors of the lab, Deborah Magalhaes, Tiago Justino, and Guilherme Rodrigues for their friendship and support.

I acknowledge the University of Melbourne and Australian Federal Government for providing me with scholarships to pursue my doctoral study. I am also thankful to Microsoft Azure for providing credits to use its resources for our experiments.

I would like to express my sincerest thanks to Reza Moini, Amir Salehi, and their family, who shared their time with me in Melbourne to forget the toughness of being far from home and family. Also I thank Dr. Mehran Garmehi for the fun moments and for preparing foods without salt when he visited CLOUDS laboratory.

My deepest gratitude goes to my mother, father, brothers, and sisters who deserve the credit for whatever success that I have attained in my life. I thank them for their continuous support, dedication, and guidance.

Yaser Mansouri
Melbourne, Australia
March 2017

Contents

1 Introduction **1**
1.1 Motivations . . . 3
1.2 Research Problems and Objectives . . . 5
1.3 Methodology . . . 6
1.4 Contributions . . . 8
1.5 Thesis Organization . . . 9

2 A Survey and Taxonomy of Data Storage Management in Cloud-based Data Stores **13**
2.1 Introduction . . . 13
2.2 Overview . . . 17
2.2.1 A Comparison of Data-intensive Networks . . . 17
2.2.2 Terms and Definitions . . . 17
2.2.3 characteristic Data-intensive applications . . . 20
2.2.4 Architecture, Goals, and Challenges of Intra-cloud Storage . . . 21
2.2.5 Architecture, Motivations, and Challenges of Inter-Cloud Storage . 26
2.3 Data model . . . 32
2.3.1 Data structure . . . 32
2.3.2 Data abstraction . . . 33
2.3.3 Data access model . . . 34
2.4 Data Dispersion . . . 36
2.4.1 Data replication . . . 36
2.4.2 Erasure coding . . . 41
2.4.3 Hybrid scheme . . . 46
2.5 Data Consistency . . . 48
2.5.1 Consistency level . . . 48
2.5.2 Consistency metric . . . 50
2.5.3 Consistency model . . . 51
2.5.4 Eventual consistency . . . 52
2.5.5 Causal and Causal+ consistency . . . 55
2.5.6 Ordering, Strong, and Adaptive-level consistency . . . 57
2.6 Data Management Cost . . . 57
2.6.1 Pricing plans . . . 58
2.6.2 Overview of storage classes . . . 59
2.6.3 Cost optimization based on a single QoS metric . . . 63
2.6.4 Cost optimization based on multi-QoS metric . . . 64

2.6.5 Cost trade-offs 65
2.7 Thesis Scope and Positioning 66
2.7.1 Thesis Scope 67
2.7.2 Thesis Positioning 67
2.8 Conclusions 77

3 QoS-aware Brokering Algorithms for Data Replication across Data Stores 79
3.1 Introduction 79
3.2 Minimizing cost with given expected availability 82
3.3 Maximum Expected Availability with Given Budget 86
3.3.1 Optimal Chunks Placement (OCP) Algorithm 89
3.4 Performance Evaluation 91
3.4.1 Simulation Setting 91
3.4.2 Algorithm 3.1: Minimum Cost Fixed Expected Availability 93
3.4.3 Algorithm 3.2: Maximum Expected Availability with a Given Budget 95
3.5 Summary 97

4 Cost Optimization in a Dual Cloud-based Storage Architecture 99
4.1 Introduction 99
4.2 System and Cost Model 102
4.2.1 System Model 102
4.2.2 Cost Model 103
4.3 Data Management Cost Optimization 105
4.3.1 Optimal Object Placement (OOP) Algorithm 105
4.3.2 Near-Optimal Object Placement (NOOP) Algorithm 107
4.4 Performance Evaluation 108
4.4.1 Experimental settings 109
4.4.2 Results 111
4.5 Summary 124

5 Cost Optimization across Cloud Storage Providers: Offline and Online Algorithms 125
5.1 Introduction 125
5.2 System Model and Problem Definition 128
5.2.1 Challenges and Objectives 128
5.2.2 Preliminaries 130
5.2.3 Optimization Problem 135
5.3 Optimal Offline Algorithm 136
5.4 Online Algorithms 138
5.4.1 The Deterministic Online Algorithm 139
5.4.2 The Randomized Online Algorithm 142
5.5 Performance Evaluation 147
5.5.1 Settings 147
5.5.2 Benchmark Algorithms 148
5.5.3 Results 148
5.6 Summary 157

6 Cost Optimization across Cloud Storage Providers: A Lightweight Algorithm 159
6.1 Introduction . . . 159
6.2 System Model, Cost Model, and Cost Optimization Problem . . . 162
6.2.1 System model . . . 162
6.2.2 Cost model . . . 163
6.2.3 Cost Optimization Problem . . . 171
6.3 Cost Optimization Solution . . . 172
6.3.1 Optimal Solution . . . 173
6.3.2 Heuristic Solution . . . 173
6.4 Performance Evaluation . . . 176
6.4.1 Experimental setting . . . 177
6.4.2 Results . . . 178
6.5 Empirical Studies in Latency Evaluation . . . 182
6.5.1 Data Access Management Modules . . . 182
6.5.2 Measurement of Data Migration Time . . . 186
6.6 Conclusions . . . 188

7 Conclusions and Future Directions 191
7.1 Summary of Contributions . . . 191
7.2 Future Directions . . . 194
7.2.1 Trade-off between Availability and Monetary Cost . . . 194
7.2.2 The Selection of Home DC . . . 195
7.2.3 Cost Optimization of Data Management in Quorum-based Systems 195
7.2.4 Cost Optimization across Multiple Storage Classes . . . 196
7.2.5 Fault Tolerance . . . 196
7.2.6 Cost Optimization of Using Database Instance Classes . . . 196

List of Figures

1.1 A simple system model used in the thesis 4
1.2 The methodology used in the thesis 7
1.3 The thesis organization 10

2.1 Data elements in cloud storage 15
2.2 Relational schema and NoSQL schema with a sample 19
2.3 NewSQL Schema 19
2.4 Inter-cloud storage architecture 27
2.5 Data model taxonomy 32
2.6 Data replication taxonomy 36
2.7 Erasure coding taxonomy 42
2.8 Data consistency taxonomy 48
2.9 Mapping data stores to each
pair of properties in CAP 50
2.10 PACELC classification and mapping several data stores to the classification 50
2.11 Weak consistency taxonomy 52

3.1 Cloud Storage Broker 81
3.2 Minimum cost of replication versus expected availability of objects 93
3.3 Minimum cost of replication versus expected availability for three types of QoS 94
3.4 Expected availability versus *TN* for three types of QoS 94
3.5 Expected failure (EF_Q) versus Expected Availability for three types of QoS 95
3.6 Expected availability versus Budget 96
3.7 Expected failure versus Budget 96
3.8 Expected availability for three types of QoS Versus Budget 97

4.1 A scenario of the dual cloud-based storage architecture in the European and Asia-Pacific regions. Parenthesis close to each DC's name shows the storage price (per GB per month) for standard storage, backup storage, and network price (per GB), respectively. 102
4.2 Allocated users to DCs (%). 110
4.3 Total data size in data centers. 110
4.4 Cost saving of OOP and NOOP algorithms for two Azure DCs: `AZ-USS` and `AZ-USC` as home DCs with data size factor 0.2 and 1. 113
4.5 Cost saving of OOP and NOOP algorithms for two Google DCs: `GO-USC` and `GO-USE` as home DCs with data size factor 0.2 and 1. 115

4.6 Cost saving of OOP and NOOP algorithms for three Amazon data centers: AM-USW(O), AM-USE, and AM-USW(C) as home data centers with data size factor 0.2 and 1. 116

4.7 Cost saving of OOP and NOOP algorithms for the home DC of Azure, Google and Amazon when the data size factor is varied. The first (resp. last) two legends indicate DC with maximum (resp. minimum) cost saving when they are paired with the home DC. 118

4.8 Cost saving of OOP and NOOP algorithms for the home DC of Azure, Google and Amazon when the latency cost weight is varied. The first (resp. last) two legends indicate DC with the maximum (resp. minimum) cost saving when they are paired with the home DC. 120

4.9 Cost saving of OOP and NOOP algorithms for the home DC of Azure, Google and Amazon when the write to read ratio is varied. The first (resp. last) two legends indicate DC with the maximum (resp. minimum) cost saving when they are paired with the home DC. 121

5.1 Object updating in Europe region and the object migration in Asia-pacific region. 133

5.2 Overview of systems's inputs and output. 135

5.3 The description of $P(\vec{\alpha}(t))$ calculation in Equ. (5.7 137

5.4 The description of Deterministic online algorithm. The residential cost of the object as if the requests on the object in slot $v = t$ are served by (a) the determined DCs in time slot $v - 1$ and (b) the determined DCs in time slot v. (c) The migration cost of the object between the determined DC in time slot $v = t_{m-1}$ and t_m. 140

5.5 Illustration of Fixed Reduced Horizontal Control 144

5.6 Cost performance of algorithms under tight and loose latency for objects with a replica. All costs are normalized to the local residential algorithm. The values in boxes show the CR of DOA and ROA in the worst case. . . . 149

5.7 Cost performance of algorithms under tight and loose latency for objects with two replicas. All costs are normalized to the local residential algorithm. The values in boxes show the CR of DOA and ROA in the worst case 150

5.8 Normalized cost of algorithms when the latency is varied. Legend indicates object size in KB for different algorithms. All costs are normalized to the local residential algorithm. 153

5.9 Normalized cost vs. read to write ratio under tight and loose latency for objects with one and two replicas. Legend indicates object size in KB for different algorithms. All costs are normalized to the local residential algorithm. 154

5.10 Normalized cost of the Randomized algorithm when the window size is varied. All costs are normalized to the local residential algorithm. Legend indicates replicas number and objects size in KB. 154

5.11 CDF of cost savings for objects due to their migration under tight and loose latency. All costs are normalized to the non-migration algorithm. Legend indicates replicas number and objects size in KB. 157

6.1 Replica creation via (a) home DC and (b)potential DCs D1, D2, and D3. . . 164
6.2 Put propagation policy. (a) Client DC first updates the server DC that serve it and the home DC. DCs hosting a replica are updated via (b) the client DC, (c) the home DC, and (d) the server DC that serve the client DC (i.e., DC D1). (e)The relayed propagation via DC D2 which is updated by the home DC. 166
6.3 An example of illustrating the replica migration between two consecutive time slots. 169
6.4 Cost saving of closest-, network-, and storage-based polices under tight (100 ms) and loose latency (250 ms). 179
6.5 Cost saving of closest-based policy vs. quantile volume 181
6.6 Cost saving of closest-based policy vs. latency 181
6.7 Cost saving of closest-based policy vs. read to write ratio 182
6.8 An overview of prototype 186
6.9 Web services components used in the prototype 187
6.10 CDF of data migration time (a) from Azure DC in Japan west to Amazon DC US west and from Azure DC in Europe north to Amazon US east, and (b) Amazon DC in US west (California) to Amazon DC in US west and Azure DC in US center south to Amazon US east. 187

List of Tables

2.1 Comparison between data-intensive networks in charachteristics and objectives. . . . 18
2.2 Intra-Cloud storage goals. . . . 22
2.3 Intra-Cloud storage challenges 24
2.4 Inter-Cloud storage motivation. . . . 29
2.5 Inter-Cloud storage challenges 30
2.6 Comparison between different storage abstractions. . . . 34
2.7 Comparison between different databases. . . . 35
2.8 Comparison between replication models. . . . 36
2.9 Comparison between full and partial replications. . . . 38
2.10 Consistency-latency tradeoff of different replication techniques. . . . 41
2.11 Comparison between Replication and Erasure Coding schemes. . . . 47
2.12 Comparison between the state-of-the-art projects using diferent redundancy schemes. . . . 47
2.13 The characteristics of different storage classes for the well-known Cloud providers: Amazon Web Service (AWS), Azure(AZ), and Google(GO) . . . 62
2.14 The Scope of Thesis. . . . 66
2.15 Summary of Projects with Monetary Cost Optimization 73

3.1 objective and constraint of the proposed algorithms 92
3.2 Data centers parameters 92

4.1 Summary of Simulation Parameters 112
4.2 Evaluation Settings for Figures and Tables. . . . 113
4.3 Cost saving of OOP and NOOP (shown in bracket), and the potential DCs pairing with four home DCs when the acess patterns on the objects are Normal and Random. . . . 122

5.1 Cloud storage pricing as of June 2015 in different DCs. . . . 127
5.2 Symbols definition 130
5.3 Average cost performance (Normalized to the local residential algorithm) 151
5.4 Running time of algorithms on 23 DCs (in Second) 157

6.1 Summary of key notations 163
6.2 The time complexity of Algorithms 6.1 - ch6:alg:CVRP. . . . 176
6.3 Summary of Simulation Parameters 178
6.4 Average cost performance normalized to the benchmark algorithm cost . 180
6.5 The modules used for data access management in AWS. . . . 184

6.6 The input parameters used in Modules of AWS. 184
6.7 The modules used for data access management in Microsoft Azure. 185
6.8 The input parameters used in Modules of Microsoft Azure. 186

Chapter 1

Introduction

CLOUD computing has gained significant attention form the academic and industry communities in recent years. It provides the vision that encompasses the movement of computing elements, storage and software delivery away from personal computer and local servers into the next generation computing infrastructure hosted by large companies such as Amazon Web Service (AWS), Microsoft Azure, and Google. Cloud computing has three distinct characteristics that differentiate it from its traditional counterparts: pay-as-you-go model, on-demand provisioning of infinite resources, and elasticity [31].

Cloud computing offers three types of resources delivery models to users [120]: (i) Infrastructure as a Service (IaaS) which offers computing, network, and storage resources, (ii) Platform as a Service (PaaS) which provides users tools that facilitate the deployment of cloud applications, and (iii) Software as a Service (SaaS) which enables users to run the provider's software on the cloud infrastructure.

One of the main components of IaaS offering by cloud computing is Storage as Services (StaaS). StaaS provides an elastic, scalable, highly available, and pay-as-you-go model, which renders it attractive for data outsourcing, both for the users to manipulate data independent of the location and time and for firms to avoid expensive upfront investments of infrastructures. The well-known Cloud Storage Providers (CSPs)–AWS, Microsoft Azure, and Google– offer StaaS for several storage classes which differ in price and performance metrics such as availability, durability, the latency required to retrieve the first byte of data, the minimum time needed to store data in the storage, etc.

The data generated by online social networks, e-commerce, and other data sources is doubling every two years and is expected to augment to a 10-fold increase between

2013 and 2020-from 4.4 ZB to 44 ZB.[1] The network traffic, generated from these data, from data centers (DCs) to users and between DCs was 0.7 ZB in 2013 and is predicated to reach 3.48 ZB by 2020.[2] The management of such data in the size of several exabytes or zettabytes requires capital-intensive investment; the deployment of cloud-based data stores (data stores for short) is a promising solution.

Moving the data generated by data-intensive applications into the data stores guarantees users the required performance Service Level Agreement (SLA) to some extent, but it causes concern for monetary cost spent in the storage services. Several factors contribute substantially to the monetary cost. First, the monetary cost depends on the size of the data volume that is stored, retrieved, updated, and potentially migrated from one storage class to another one in the same/different data stores. Second, it is subject to the required performance SLA (e.g., availability,[3] durability, the latency needed to retrieve the first byte of data) as the main distinguishing feature of storage classes. As the performance guarantee is higher, the price of storage classes is more. Third, the monetary cost can be affected by the need of data stores to be in a specific geographical location in order to deliver data to users within their specified latency. To alleviate this concern (i.e., monetary cost spending on storage services) from the perspective of application providers/users, it is required to design algorithms for appropriate selection of storage classes offered by different CSPs during the lifetime of the object regarding to the above-mentioned factors.

The use of multiple CSPs offering several storage classes with different prices and performance metrics brings a substantial benefit to users who seek the reduction of monetary cost in storage services, while respecting their QoS in terms of availability and network latency. In spite of some efforts in this direction, designing algorithms that take advantage of price differences across CSPs with several storage classes to reduce monetary cost on storage services for time-varying workloads remains as an open challenge. This thesis deals with this challenge and investigates how much the monetary cost can be saved with the help of multiple CSPs while respecting the QoS determined by users. The remaining parts of this chapter discuss motivation, research problem, evaluation methodology, the-

[1] International Data Corporation (IDC). https://www.emc.com/leadership/digital-universe/2014iview/index.htm.

[2] The Zettabyte EraTrends and analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html.

[3] Availability and durability are described in terms of *nines*.

sis contribution, and thesis organization.

1.1 Motivations

Migrating data into a single data store facilitates users performance SLA/QoS to some extent, but faces them with several limitations. Data store unavailability can confront users with inaccessible data if the data is stored in a single data store. This is counted as one of the top ten obstacles for cloud adoption [70]. Reliance on a single data store makes it difficult to migrate data from a data store to another in the face of price increment by the cloud provider, the emergence of a new data store with lower price, the mobility of users, and changes in workload that demands data migration. This is recognized as data lock-in and is listed as another main obstacle in regard to cloud services. Storing data in a single data store faces the fact that the read (Get) and write (Put[4]) requests are not served with adequate responsiveness. This is because the requests are issued by users who are located worldwide and, consequently users experience more network latency to retrieve/store data from/into a data store. Furthermore, the use of a single data store deprives users from the opportunity to exploit the pricing differences across CSPs. Therefore, storing data within a single data store can be inefficient in both performance SLA and monetary cost.

These factors make inevitable the use of multiple CPSs which improve availability, durability, and data mobility. The deployment of multiple CSPs also brings another benefits to users. (i) If the outage of a data store happens then the requests issued by users are directed to another data store. (ii) Users can have a wider selection of data stores, which results in reducing user-perceived latency as experimentally confirmed [178]. (iii) This deployment also allows application providers to select storage classes across CSPs based on the workload on data and the Quality of Service (QoS) specified by users to reduce monetary cost of storage and network resources.

The workload on data can be a determining factor for the selection of a storage class. Some data-intensive applications generate a time-varying workload in which as the time passes the rate of read and write requests on the data changes. In fact, there is a strong

[4] Read and Write are respectively interchangeable with Get and Put in this thesis.

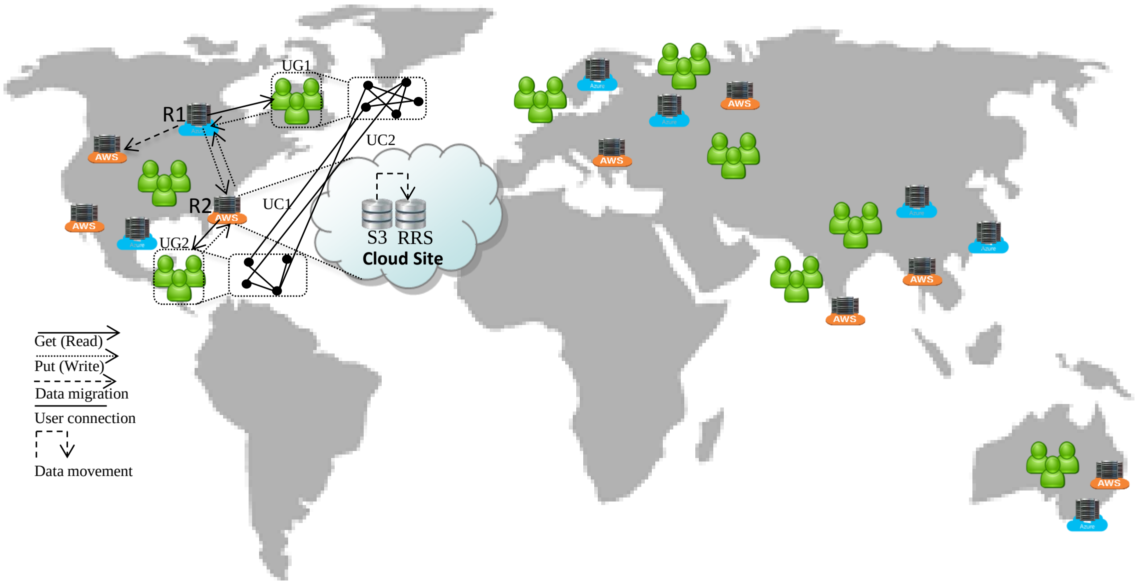


Figure 1.1: A simple system model used in the thesis

correlation between the age of data stored in a data store and data workload. For example, in Online Social Network (OSN) data initially receive many read and write requests and gradually these requests reduce [122]. Based on this change of requests rate, we define two statuses for data: *hot-spot* and *cold-spot*. Hot-spot data receive many read and write requests, while cold-spot data receive a few.

This demands for a suitable selection of storage classes throughout the lifetime of an object. Each storage class provided by the well-known CSPs can be suited for data with specific requirements. For instance, one class may be suitable for data that is frequently accessed. Another class may be designed to host data that is rarely accessed and required for a lower availability and durability.

Therefore, CSPs with a variety of storage classes with different prices and performance SLAs and data-intensive applications with time-varying workloads give us an incentive to design novel algorithms to optimize monetary cost. These algorithms work for any data to which workload transits from hot-spot to cold-spot and vice versa, as observed in OSN applications. Significant studies have been done in the area of cost optimization of OSN applications across CPSs, as conducted in SPANStore [177] and Cosplay [82]. Neither leverage different storage classes owned by different CSPs nor consider the object with hot- and cold-spot status, which result in migration of data across different data stores or movement of data between storage classes within a data store. This is the main focus of this thesis.

Fig. 1.1 illustrates a simple data placement across data stores (owned by different providers) to clarify our motivation. Among all available data stores, application providers, for example OSNs, select a subset of data stores to replicate data to serve their users. OSN users are typically assigned to the closest DCs and have a set of friends and followers who make network connections with them. These connections are between users who are assigned to the same DC as represented by a graph in rectangles (see Fig. 1.1) or different DCs (e.g., user connections UC1 and UC2). In this model, for example, a user in user group UG1 puts data in AWS DC named replica R1. To make this data available to his/her friends and followers, the data is also replicated in Azure DC as replica R2. These replicas stay in DCs until they receive many read and write requests. As time passes, one replica probably migrates to another DC or moves between storage classes (e.g., Simple Service Storage (S3) and Reduced Redundancy Storage (RRS) in AWS) within a DC as backup data.

1.2 Research Problems and Objectives

This thesis focuses on the cost optimization of data management across CSPs under QoS constraints. This cost includes the cost of storage and network resources. The data/objects[5] observe hot- and cold-spot statuses during their life-time, and can be owned by users in an OSN application. In respect to this aim, the following research challenges are investigated.

- **How to link the number of replicas of the object with the availability of the object?** Since the number of replicas of an object has direct impact on the availability of the object, it is necessary to define a metric for the availability of the object that can be used as a QoS constraint for cost optimization as an objective.
- **When to migrate objects?** Based on the read and write requests on the object, a decision on the object migration across data stores at appropriate times should be made. This object migration should be cost-effective and the requests submitted to

[5] Data and objects are interchangeably used in this thesis.

the object after its migration should be served within the defined QoS.

- **Where to place the objects selected for migration?** Make decision on the best placement of newly created objects by users or the objects selected for migration to other data stores is another key aspect that influences the cost optimization of the objects.
- **How to design algorithms for dynamic replication and migration of the objects and which features they should have?** The ideal deign is to find optimal placement of objects so that the monetary cost is minimized. Alternatively, a satisfactory design of algorithms is (i) to be competitive with optimal algorithms in monetary cost, and (ii) to be light in terms of time complexity since OSN applications host a huge number of objects.

To tackle the above challenges, the following objectives have been identified:

- Conduct a comprehensive survey and classification of data management in cloud-based data stores in several aspects; mainly data management cost in cloud storage to understand the existing gap in this area.
- Define a cost model and link it with the availability unit in the number of nines, and propose data placement algorithms.
- Propose a dual cloud-based storage architecture to provide insight into the cost saving derived from different CSPs which support several storage classes with different prices.
- Design optimal offline algorithm and online algorithms and conduct competitive analysis of online algorithms to understand their performance compared to the optimal offline algorithm.
- Design a novel lightweight algorithm with low time complexity, which is tailored for OSN applications hosting a huge number of users.

1.3 Methodology

This thesis endeavors to optimize cost of data placement across CSPs while satisfying the QoS defined by users. To this end, we use the following approach for each defined cost optimization problem as shown in Fig. 1.2.

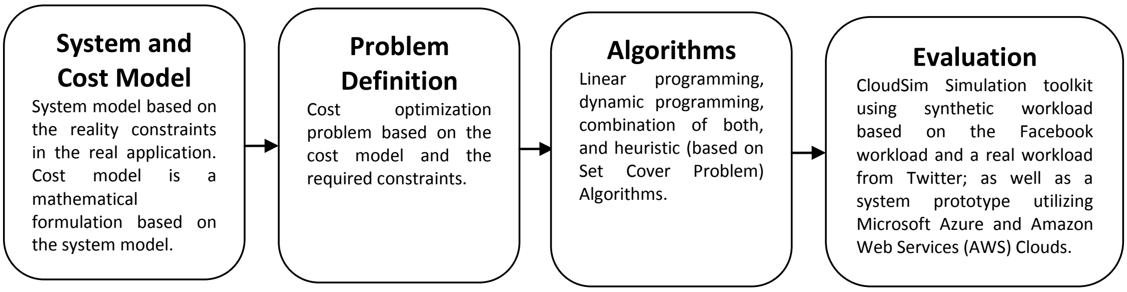

System and Cost Model
System model based on the reality constraints in the real application. Cost model is a mathematical formulation based on the system model.
Problem Definition
Cost optimization problem based on the cost model and the required constraints.
Algorithms
Linear programming, dynamic programming, combination of both, and heuristic (based on Set Cover Problem) Algorithms.
Evaluation
CloudSim Simulation toolkit using synthetic workload based on the Facebook workload and a real workload from Twitter; as well as a system prototype utilizing Microsoft Azure and Amazon Web Services (AWS) Clouds.

Figure 1.2: The methodology used in the thesis

1. **System and cost model.** We define a system model consisting of objects created by users, read and write requests submitted to the objects, and data stores with the measured network latency between each pair. Based on the system model, we mathematically define a cost model that consists of storage and network costs.
2. **Problem definition.** We formally define the cost optimization problem based on the cost model in the form of mathematical formula with the required QoS.
3. **Algorithms.** The algorithms used to solve the optimization problems are a linear programming, dynamic programming, combination of both, and a lightweight algorithm. We evaluate and analyse (in terms of time complexity) the proposed algorithms.
4. **Evaluation.** We evaluate the proposed algorithms via discrete-event simulation using the CloudSim simulation toolkit developed to support cloud-based data stores and the objects created by users [35]. We generate a synthetic workload based on the characteristics of the Facebook workload [16,21] and use it in an optimization problem in Chapter 5. We also perform the evaluation of algorithms using a real workload from Twitter [101]. With the help of the Google Maps Geocoding APIs,[6] the location of users in the workload is converted to Geo-coordination in order to assign each user to the closest data stores. The evaluation of the algorithms via simulation (i) makes it easy to conduct repeatable large-scale experiments and investigate the effect of all the parameters that can probably influence the performance of the algorithms. We also theoretically prove the performance of some of the algorithms in comparison to the optimal algorithm in terms of cost

[6] The Google maps geocoding API https://developers.google.com/maps/documentation/geocoding/intro

saving.

1.4 Contributions

The contribution of this thesis can be broadly categorized into: (i) A survey and taxonomy of the area, (ii) Proposed dynamic algorithms to optimally select data stores for stripped and non-stripped objects with the guaranteed availability, (iii) A dual cloud-based storage architecture exploiting the pricing differences of CSPs, (iv) The optimal offline algorithm, online algorithms, and a lightweight algorithm to replicate and migrate data across CSPs. The **key contributions** of the thesis are as follows:

1. A survey and taxonomy of data storage management in cloud-based data stores.

2. QoS-aware brokering algorithms for data dispersion across data stores:
   - A mathematical model for the DC selection problem in which the objective function, cost function, and constraints are clearly defined.
   - An algorithm to select a subset of given data stores to minimize the storage cost for objects when the expected availability is given.
   - A dynamic algorithm to select data stores optimally for storing objects that are split into chunks and each chunk is replicated a fixed number of times. This maximizes the availability of the striped data to the extent the users budget allows.

3. Cost optimization in a dual cloud-based storage architecture:
   - A system model and a formal cost model for data management in data stores.
   - The optimal algorithm that optimizes data management cost in the dual cloud-based architecture when the workload in terms of Gets and Puts on the objects is known.
   - A near-optimal algorithm that achieves competitive cost as compared to that obtained by the optimal algorithm in the absence of future workload knowledge.
   - A simulation-based evaluation and performance analysis of the algorithms using the real-world traces from Twitter [101].

4. Cost optimization across cloud storage providers: offline and online algorithms:
    - The optimal offline algorithm for data replication and migration across CSPs where the exact future workload is assumed to be known a priori:
    - Deterministic and randomized online algorithms for data replication and migration across CSPs where the future workload is unavailable or available for a limited time.
    - Competitive analysis and proof of the competitive ratios of online algorithms for data replication and migration across CSPs.
    - An extensive simulation-based evaluation and performance analysis of the proposed algorithms using the synthesized workload based on Facebook workload specifications [16][21].

5. Cost optimization across cloud storage providers: a lightweight algorithm:
    - A lightweight algorithm based on the set cover problem [48]. It yields a low time complexity which makes it suitable for the OSN applications hosting a huge number of users.
    - An extensive evaluation through simulation using a real workload from Twitter [101].
    - An implementation of a prototype using AWS and Microsoft Azure data stores to evaluate the duration of objects migration within and across regions.

1.5 Thesis Organization

The core chapters of this thesis are mostly derived from the publications made during the PhD candidature. Fig. 1.3 shows the structure of the thesis as described below:

- Chapter 2 presents a survey and taxonomy of data management in data stores, as well as the scope of this thesis and its positioning in the area. This chapter is partially derived from:
    - **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, “Data Storage Management in Cloud Environments: Taxonomy and Survey,” *ACM Computing Surveys*, ACM Press, New York, USA, 2016 (under minor review).

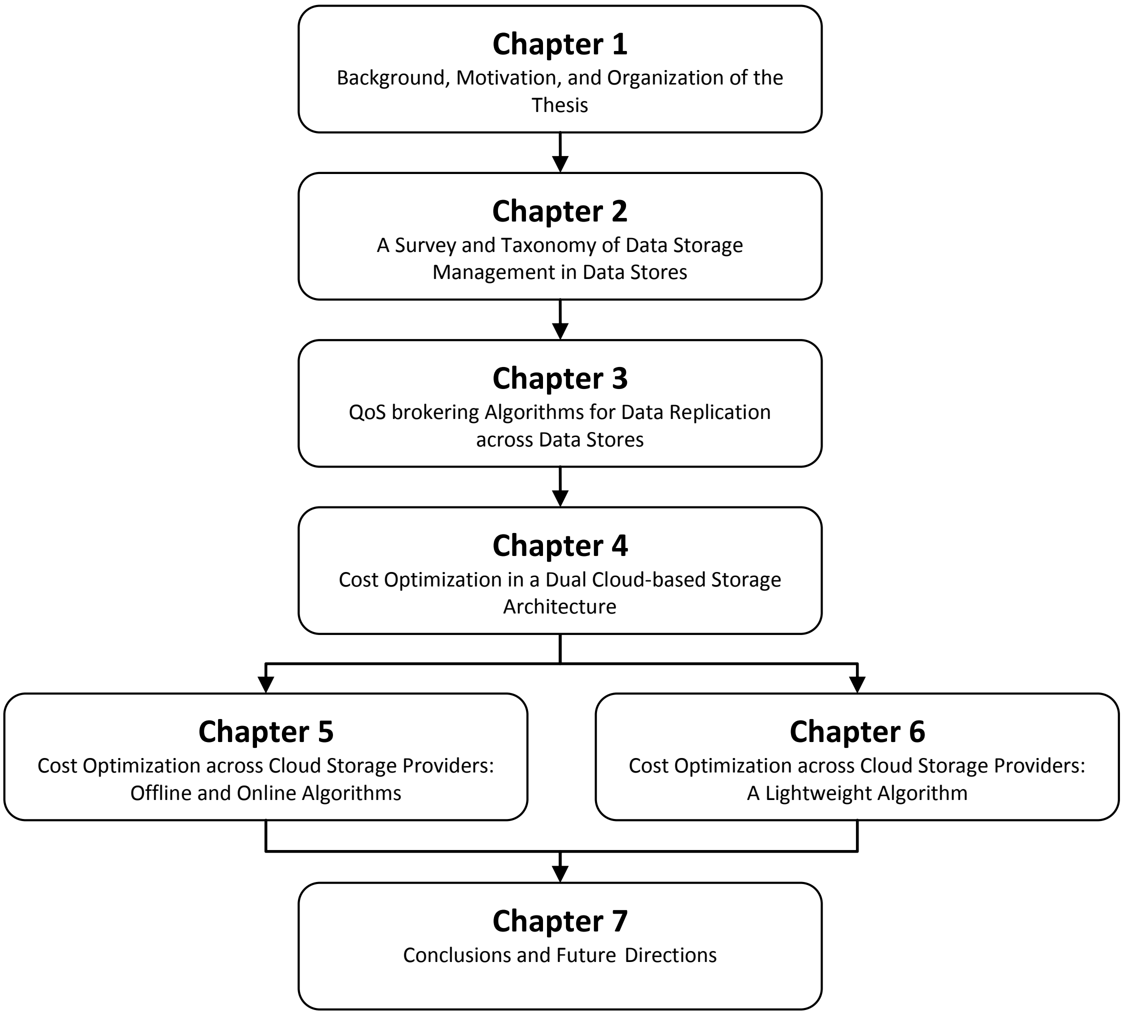

Chapter 1
Background, Motivation, and Organization of the Thesis
Chapter 2
A Survey and Taxonomy of Data Storage Management in Data Stores
Chapter 3
QoS brokering Algorithms for Data Replication across Data Stores
Chapter 4
Cost Optimization in a Dual Cloud-based Storage Architecture
Chapter 5
Cost Optimization across Cloud Storage Providers: Offline and Online Algorithms
Chapter 6
Cost Optimization across Cloud Storage Providers: A Lightweight Algorithm
Chapter 7
Conclusions and Future Directions

Figure 1.3: The thesis organization

- Chapter 3 proposes QoS-aware brokering algorithms for data replication across data stores. This chapter is derived from:
  - **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Brokering Algorithms for Optimizing the Availability and Cost of Cloud Storage Services," *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2013, IEEE CS Press, USA)*, Bristol, UK, Dec. 2-5, 2013.

- Chapter 4 proposes cost optimization in a dual cloud-based storage architecture. This chapter is derived from:

  - **Yaser Mansouri** and Rajkumar Buyya, "To Move or Not to Move: Cost Optimization in a Dual Cloud-based Storage Architecture," *Journal of Network and Computer Applications (JNCA)*, Volume 75, Pages: 223-235, ISSN: 1084-8045, Elsevier, Amsterdam, The Netherlands, November 2016.

- Chapter 5 describes cost optimization across cloud storage providers: offline and online algorithms. This chapter is derived from:

  - **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Cost Optimization for Dynamic Replication and Migration of Data in Cloud Data Centers," *IEEE Transactions on Cloud Computing (TCC)*, DOI:10.1109/TCC.2017.2659728, 2017.

- Chapter 6 proposes cost optimization across cloud storage providers: a lightweight algorithm. This chapter also proposes a prototype data placement framework across Amazon Web Service (AWS) and Microsoft Azure. It is derived from:

  - **Yaser Mansouri** and Rajkumar Buyya, "Dynamic Replication and Migration for Data Objects with Hot-spot and Cold-spot Statuses across Geo-distributed storage Data centers," *Journal of Parallel and Distributed Computing*, ElSEVIER, 2017 (under review).

- Chapter 7 concludes the thesis with a summary of the key findings and a discussion of directions for future work.

Chapter 2

A Survey and Taxonomy of Data Storage Management in Cloud-based Data Stores

Storage as a Service (StaaS) is a vital component of cloud computing by offering the vision of a virtually infinite pool of storage resources. It supports a variety of cloud-based data store classes in terms of availability, scalability, ACID (Atomicity, Consistency, Isolation, Durability) properties, data models, and price options. Application providers deploy these storage classes across different cloud-based data stores not only to tackle the challenges arising from reliance on a single cloud-based data store but also to obtain higher availability, lower response time, and more cost efficiency. Hence, in this chapter, we first discuss the key advantages and challenges of data-intensive applications deployed within and across cloud-based data stores. Then, we provide a comprehensive taxonomy that covers key aspects of cloud-based data store: data model, data dispersion, data consistency, and data management cost. We finally discuss the scope of the thesis and determine the position of thesis in regard to relevant work to provide a better understanding of the research problems addressed in the remaining chapters.

2.1 Introduction

THE The explosive growth of data traffic driven by social networks, e-commerce, enterprises, and other data sources has become an important and challenging issue for IT enterprises. This growing speed is doubling every two years and augments 10-fold between 2013 and 2020- from 4.4 ZB to 44 ZB.[1] The challenges posed by this growth of data can be overcome with aid of using cloud computing services. Cloud computing

This chapter is derived from: **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Data Management in Cloud Environments: Taxonomy and Survey," *ACM Computing Surveys*, ACM Press, New York, USA, 2016 (under minor review).

[1] International Data Corporation (IDC).https://www.emc.com/leadership/digital-universe/2014iview/index.htm

offers the illusion of infinite pool of highly reliable, scalable, and flexible computing, storage, and network resources in a pay-per-use manner. These resources are typically categorized as Infrastructure as a Service (IaaS), where Storage as a Service (StaaS) forms one of its critical components.

StaaS provides a range of cloud-based data stores (*data stores* for short) that differs in data model, data consistency semantic, and price model. A popular class of data stores, called Not only SQL (NoSQL), has emerged to host applications that require high scalability and availability without having to support the ACID properties of relational database (RDB) systems. This class of data stores – such as PNUTS [51] and Dynamo [59] – typically partitions data to provide scalability and replicates the partitioned data to achieve high availability. *Relational data store*, as another class of data stores, provides full-fledged relational data model to support ACID properties, while it is not as scalable as NoSQL data store. To strike a balance between these two classes, NewSQL data store was introduced. It captures the advantages of both NoSQL and relational data stores and initially was exploited in Spanner [52].

To take the benefits of these classes, application providers store their data either in a single or multiple data stores. A single data store offers the proper availability, durability, and scalability. But reliance on a single data store has risks like vendor lock-in, economic failure (e.g., a surge in price), and unavailability as outages occur, and probably leads to data loss when an environmental catastrophe happens [27]. Geo-replicated data stores, on the other hand, mitigate these risks and also provide several key benefits. First, the application providers can serve users from the best data store to provide adequate responsiveness since data is available across data stores. Second, the application can distribute requests to different data stores to achieve load balance. Third, data recovery can be possible when natural disaster and human-induced activities happen. However, the deployment of a single or multiple data stores causes several challenges depending on the characteristics of data-intensive applications.

Data-intensive applications are potential candidates for deployment in the cloud. They are categorized into transactional (ref. as *online transaction processing* (OLTP)) and analytical (ref. as *online analytical processing* (OLAP)) that demand different requirements. OLTP applications embrace different consistency semantics and are adaptable with row-

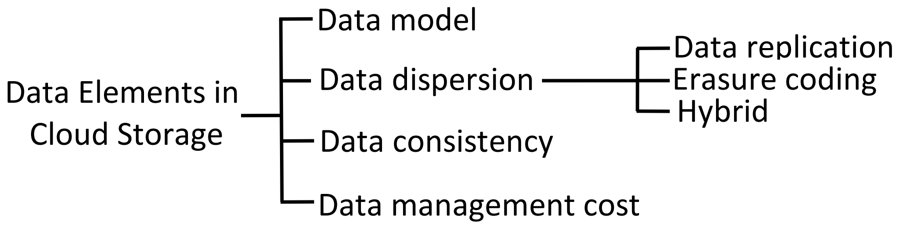


Figure 2.1: Data elements in cloud storage

oriented data model, while OLAP applications require rich query capabilities and compliance with column-oriented data model. These requirements are faced with several challenges, and they mandate that we investigate the key elements of data management in data stores as shown in Fig.2.1. The first five key elements are mature topics in the context of distributed systems and require how to apply them to data stores with possible modifications and adoptions if needed. The last element, *data management cost*, is a new feature for cloud storage services and it is important for users to optimize it while their Service Level Agreements (SLAs) are guaranteed.

The first element is *data model* that reflects how data is stored in and retrieved from data stores. The second element is *data dispersion* with three schemes. *Data replication* scheme improves availability and locality by moving data close to the user, but it is costly due to usually storing three replicas for each object in data stores. *Erasure coding* scheme alleviates this overhead, but it requires structural design to reduce the time and cost of recovery. To make a balance between these benefits and shortcomings, a combination of both schemes is exploited. We clarify these schemes and identify how they influence availability, durability, and user-perceived latency.

Other element of data management is *data consistency* that refers to coordination level between replicas within and across data stores. Based on CAP theorem [71], it is impossible to jointly attain $\underline{\text{C}}$onsistency, $\underline{\text{A}}$vailability, and $\underline{\text{P}}$artition tolerance (referred to the failure of a network device) in distributed systems. Thus, initially data stores provide *eventual consistency*– all replicas eventually converge to the last updated value– to achieve two of three these properties: availability and partition tolerance. Eventual consistency is sometime acceptable, but not for some applications (e.g., e-commerce) that demand strong consistency in which all replicas receive requests in the same order. To obtain strong consistency, the recent endeavours bring transactional isolation levels in

NoSQL/NewSQL data stores at the cost of extra resources and higher response time.

The last element is *cost optimization of data storage management* as the key driver behind the migration of application providers into the cloud that offers a variety of storage and network resources with different prices. Thus, application providers have many opportunities for *cost optimization* and *cost trade-offs* such as storage vs. bandwidth, storage vs. computing, and so on.

The main contributions of this chapter are as follows:

- Comparison between cloud-based data stores and related data-intensive networks,
- Delineation on goals and challenges of intra- and inter-cloud storage services, and determination of the main solutions for each challenges,
- Discussion on data model taxonomy in terms of *data structure*, *data abstraction*, and *data access model*; and comparison between different levels/models of each aspect,
- Providing a taxonomy for different schemes of data dispersion, and determining when (according to the specifications of workload and the diversity of data stores) and which scheme should be used,
- Elaboration on different levels of consistency and determination on how they are guaranteed,
- Discussion on the cost optimization of storage management, delineation on the potential cost trade-offs in data stores, and classifying the existing projects to specify the research venue for future.

This chapter is divided into seven sections. Section 2.2 compares cloud-based data stores to other distributed data-intensive networks and then discusses the architecture, goals and challenges of a single and multiple cloud-based data stores deployments. Section 2.3 describes a taxonomic of data model and Section 2.4 discusses different schemes of *data replication*. Section 2.5 elaborates on data consistency in terms of *level*, *metric*, and *model* and Section 2.6 presents the cost optimization of storage management. Section 2.8 finally concludes the chapter.

2.2 Overview

This section discusses a comparison between cloud-based data stores and other data-intensive networks (Section 2.2.1), the terms used throughout this survey (Section 2.2.2), the characteristics of data-intensive applications deployed in data stores (Section 2.2.3), and the main goals and challenges of a single and multiple data stores leveraged to manage these applications (Section 2.2.4).

2.2.1 A Comparison of Data-intensive Networks

Table 2.1 highlights similarities and differences in characteristics and objectives between cloud-based data stores and (i) Data Grid in which storage resources are shared between several industrial/educational organizations as Virtual Organization (VO), (ii) Content Delivery Network (CDN) in which a group of servers/datacenters are located in several geographical locations to serve users contents (i.e. application, web, or video) faster, and (iii) Peer-to Peer (P2P) in which a peer (i.e., server) shares files with other peers.

Cloud-based data stores share more overlaps with Data Grids in the properties listed in Table 2.1. They deliver more abstract storage resources (due to more reliance on virtualization) for different types of workloads in an accurate economic model. They also provide more elastic and scalable resources for different demands in size. These opportunities result in the migration of data-intensive applications to the clouds and cause two categories of issues. One category is more specific to cloud-based data stores and consists of issues such as vendor-lock in, multi-tenancy, network congestion, monetary cost optimization, etc.. Another category is common between cloud-based data stores and Data Grids and includes issues like data consistency and latency management. Some of these issues require totally new solutions, and some of them have mature solutions and may be applicable either wholly or with some modifications based on the different properties in cloud-based data stores.

2.2.2 Terms and Definitions

A *data-intensive application* system consists of applications that generate, manipulate, and analyze large amount of data. With the emergence of cloud-based storage services, the

Table 2.1: Comparison between data-intensive networks in charachteristics and objectives.

Property	Cloud-based Data Stores	Data Grids	Content Delivery Network (CDN)	Peer to Peer (P2P)
Purpose	Pay-as-you-go model, on-demand provisioning and elasticity	Analysis, generating, and collaboration over data	File sharing and content distribution	Improving user-perceived latency
Management Entity	Vendor	Virtual Organization	Single organization	Individual
Organization	Tree-based [173]† Fully optical Hybrid	Hierarchy Federation	Hierarchy	Unstructured Structured Hybrid [172]
Service Delivery	IaaS, PaaS, and SaaS	IaaS	IaaS	IaaS
Access Type	Read-intensive Write-intensive Equally of both	Read-intensive with rare writes	Read-only	Read- intensive with frequent writes
Data Type	Key-value, Document-based, Extensible record, and Relational	Object-based (Often big chunks)	Object-based (e.g., media, software, script, text)	Object/file-based
Replica Discovery	HTTP requests Replica Catalogue	Replica Catalog	HTTP requests	Distributed Hash Table†† Flooded requests
Replica Placement	See section 2.4.1	Popularity Primary replicas	A primary copy Caching	Popularity without primary replica
Consistency	Weak and Strong	Weak	Strong	Weak
Transaction Support	Only in relational data stores (e.g., Amazon RDS)	None	None	None
Latency Management	Replication, caching, streaming	Replication, caching, streaming	Replication, caching, streaming	Caching, streaming
Cost Optimization	Pay-as-you-go model (in granularity of byte per day for storage and byte for bandwidth)	Generally available for not-for-profit work or project-oriented	Content owners pay CDN operators which, in turn, pays ISPs to host contents	Users Pay P2P to receive sharing files.

† Tree-based organization has a flexible topology, while fully optical (consisting of a "pure" optical switching network) and hybrid (including switching network of electrical packet and optical circuit) organizations have a fixed topology. Google and Facebook deploy fat tree topology, a variant of tree topology, in their datacener architecture. †† Distributed hash table is used for structured organization and flooded requests for unstructured organization.

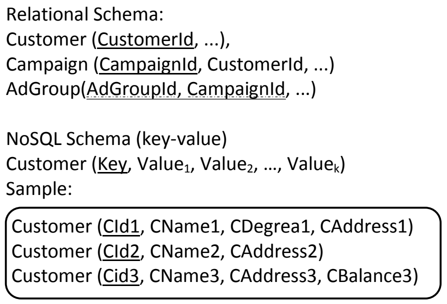

Relational Schema:
Customer (CustomerId, ...),
Campaign (CampaignId, CustomerId, ...)
AdGroup(AdGroupId, CampaignId, ...)
NoSQL Schema (key-value)
Customer (Key, Value$_1$, Value$_2$, ..., Value$_k$)
Sample:
Customer (CId1, CName1, CDegrea1, CAddress1)
Customer (CId2, CName2, CAddress2)
Customer (Cid3, CName3, CAddress3, CBalance3)

Figure 2.2: Relational schema and NoSQL schema with a sample

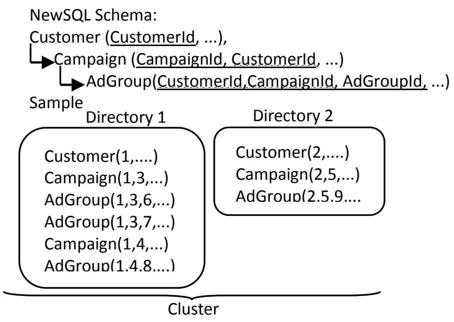


Figure 2.3: NewSQL Schema [155]

data generated from these applications are typically stored in *single data store* or *Geo-replicated data stores* (several data stores in different worldwide locations). The data is organized as *dataset* which is created and accessed across data centers (DCs) by *users/application providers*. *Metadata* describe the dataset with respect to the several attributes such as name, creation time, owner, replicas location, etc.

A dataset consists of a set of *objects* or *records* that are modeled in relational databases (RDBs) or NoSQL databases. The data store that manages RDBs and NoSQL databases are respectively called as *relational data store* and *NoSQL data store*. As shown in Fig. 2.2, RDBs have fixed and predefined fields for each object whereas NoSQL databases do not. NoSQL data stores use *key-value* data model (or its variations such as graph and document) in which each object is associated with a pair of *key* and *value*. Key is unique and is used to store and retrieve the associated value of an object. NewSQL data stores follow a *hierarchical data model*, which introduces a *child-parent* relation between each pair of table where the child table (`Campaign` in Fig. 2.3) borrows the primary key of its parent (e.g., `Customer`) as a prefix for its primary key. In fact, this data model has a *directory table* (e.g., `Customer` in Fig. 2.3) in the top of hierarchical structure and each row of directory table together with all rows in the descendant tables (e.g., `Campaign` and `AdGroup`) constructs a *directory*.

A *complete* or *partial* replica of the dataset is stored in a single data store or different data stores across Geo-distributed DCs based on the required QoS (response time, availability, durability, monetary cost). An operation to update an object can be initially submitted to a predefined replica (called *single master*) or to any replicas (*multi-master*) based on predefined strategy (e.g., the closest). Replicas are *consistent* when they have the same value for an object. Replicas are in *weak/strong* consistency status if replicas return (probably) *different/same* values for a read operation. A *transaction* is a set of reads and writes, and it is *committed* if all reads and writes are conducted (on all replicas of data); otherwise it is *aborted*.

2.2.3 characteristic Data-intensive applications

In respect to the cloud characteristics, two types of data-intensive applications can be nominated for the cloud deployment [2].

Online transaction processing (OLTP) applications must guarantee ACID properties and provide an "all-or-nothing" proposition that implies each set of operations in a transaction must complete or no operation should be completed. Deploying OLTP applications across data stores is not straightforward because achieving the ACID properties requires acquiring distributed locks, executing complex commit protocols and transferring data over network, which in turn causes network congestion across data stores and introduces network latency. Thus, OLTP should be adapted to ACID properties at the expense of high latency to serve reads and writes across data stores. *Online Analytical Processing* (OLAP) applications usually use read-only databases and often need to handle complex queries to retrieve the desired data for data warehouse. The updates in OLAP are conducted on regular basis (e.g., per day or per weak), and their rate is lower than that of OLTP. Hence, the OLAP applications do not need to acquire distributed locks and can avoid complex commit protocols.

Both OLTP and OLAP should handle a tremendous volume of data at incredible rates of growth. This volume of data is referred to as *big data* which has challenges in five aspects: *volume* refers to the amount of data; *variety* refers to the different types of generated data; *velocity* refers to the rate of data generation and the requirement of accelerating analysis; *veracity* refers to the certainty of data; and *value* refers to the determination of hidden values from datasets. Schema-less NoSQL databases easily cope with two of these aspects via providing an infinite pool of storage (i.e., volume) in different types of data (i.e., variety). For other aspects, NoSQL data stores are controversial for OLTP and OLAP. From the velocity aspect, NoSQL data stores like BigTable [39], PNUTS [51], Dynamo [59], and Cassandra [95] facilitate OLTP with reads and writes in low latency and high availability at the expense of weak consistency. This is a part of the current chapter to be discussed. From velocity, veracity and value aspects (the last two aspects are more relevant to OLAP), OLAP requires frameworks like Hadoop,[2] Hive [166] and Pig[3] as well as algorithms in big data mining to analysis and optimize the complex queries in NoSQL data stores. It is worth to mention these frameworks lack rich query processing on the cloud and change into the data model is a feasible solution.

2.2.4 Architecture, Goals, and Challenges of Intra-cloud Storage

This section describes a layered architecture of data store and discusses the key goals and challenges of deploying a single data store to manage data-intensive applications.

Architecture of Intra-Cloud Storage

Cloud storage architecture shares a storage pool through either a dedicated Storage Area Network (SAN) or Network Attached Storage (NAS).[4] It is composed of a distributed file system, Service Level Agreement (SLA), and interface services. The architecture divides the components by physical and logical functions boundaries and relationships to provide more capabilities [192]. In the layered architecture, each layer is constructed based on the services offered by its underneath layer.

(1) *Hardware layer* consists of distributed storage servers in a cluster that consists of several racks, and each of which has disk-heavy storage nodes. (2) *Storage management* provides services for managing data in the storage media. It consists of the fine-grained services like data replication and erasure coding management, replica recovery, load balancing, consistency, and transaction management. (3) *Metadata management* classifies the metadata of stored data in a global domain (e.g., storage cluster) and collaborates with different domains to locate data when it is stored or retrieved. For example, *Object Table* in Window Azure Storage [34] has a primary key containing three properties: *AccountName*, *PartitionName*, and *ObjectName* that determine the owner, location, and name of the table

[2] Apache Hadoop. http://wiki.apache.org/hadoop

[3] Apache Pig. http://hadoop.apache.org/pig/

[4] A NAS is a single storage device that operates on file system and makes TCP/IP and Ethernet connections. In contrast, a SAN is a local network of multiple devices that operates on disk blocks and uses fiber channel interconnections.

respectively. (4) *Storage overlay* is responsible for storage virtualization that provides data accessibility and is independent of physical address of data. It converts a logical disk address to the physical address by using metadata. (5) *User interface* provides users with primitive operations and allows cloud providers to publish their capabilities, constraints, and service prices in order to help subscribers to discover the appropriate services based on their requirements.

Table 2.2: Intra-Cloud storage goals.

Goals	Techniques (Examples)	Section(s)
Performance and cost saving	Combination of different storage services	2.6
Fault tolerance and availability	Replication (Random replication, Copyset, MRR) Erasure coding	2.4.1 2.4.2
Multi-tenancy	Shared table (saleforce.com [112]) Shared process (Pisces [154], ElasTras [57]) Shared machine (live VM migration techniques [108])	—
Elasticity and load balancing	Storage-load-aware data balancing (Dynamo, BigTable) Access-aware data balancing ([42])	—

Goals of Intra-Cloud Storage

Table 2.2 introduces the five main goals of data-intensive applications deployment in a single data store. These goals are as follows:

- *Performance and cost saving*. Efficient utilization of resources has direct influence on cost saving since different data stores offer the pay-as-you-go model. To achieve both these goals, a combination of storage services varying in price and performance yields the desired performance with lower cost as compared to relying on one type of storage service. To make an effective combination, the decision on when and which type of storage services to use should be made based on *hot*- and *cold*-spot statuses of data and the required QoS. Note that *hot-spot* data/nodes receive many read and write requests, while *clod-spot* data/nodes receive a few.
- *Fault tolerance and availability*. Failures arise from *faulty hardware* and *software* in two models. *Byzantine* model presents arbitrary behaviour and can be survived via $2f + 1$ replicas (f is number of failures) along with non-cryptographic hashes or cryptographic primitives. *Crash-stop* model silently stops the execution of nodes and can be tolerated via *data replication* and *erasure coding* in multiple servers located in different racks, which in turn, are in different domains. This failure model

has two types: *correlate* and *independent*. The random placement of replicas used in current data stores only tackles independent failures. Copyset [50] replicates an object on a set of storage nodes to tolerate correlated failures, and multi-failure resilient replication (MRR) [109] places replicas of a single chunk in a group of nodes (partitioning into different sets across DCs) to cope with both types of failure.

- *Multi-tenancy*. This allows users to run applications on shared hardware and software infrastructure so that their isolated performance is guaranteed. It causes *variable and unpredictable performance*, and *multi-tenant interference and unfairness* [154] which happen to different multi-tenancy models (from the weakest to the strongest): *shared table* model in which applications share the database tables, *shared process* model in which applications share the database process, and *shared machine* model in which applications only share the physical hardware, but they have independent database process and VM. These models have implication on the system's performance for which the shared process outperforms other models [57].
- *Elasticity and load balancing*. Elasticity refers to expansion and consolidation of servers during load changes in a dynamic system. It is orthogonal with load balancing in which workloads are dynamically moved from one server to another under skewed query distribution so that all servers handle workloads almost equally . A prevalent approach for load balancing is *storage-load-aware* which uses *key range* and *consistent hash-algorithm* techniques to distribute data across storage nodes. This approach, used by all existing cloud storage services, is not effective when data-intensive applications experience hot- and cold-spot statuses. These applications thus require the *access-load-aware* approach for load balancing [42]. Both these approaches use *stop and copy migration* technique (used in most key-value data store) and *live migration* technique which is better in availability and response time [65]

Challenges of Intra-Cloud Storage

Table 2.3 introduces what challenges application providers confront with the deployment of their applications within a data store. These challenges are as below:

- *Unavailability of services and data lock-in*. *Data replication* across storage nodes in an efficient way (instead of random replica placement widely used by current data

Table 2.3: Intra-Cloud storage challenges

Challenges	Solutions	Section(s)/References
Unavailability of services and data lock-in	**(i)** Data replication **(ii)** Erasure coding	2.4.1 2.4.2
Data transfer bottleneck	**(i)** Workload-aware partitioning (for OLTP) **(ii)** Partitioning of social graph, co-locating the data of a user and his friends along with the topology of DC (for social networks) **(iii)** Scheduling and monitoring of data flows	[93][85] [163][41][198] [7][55][152]
Performance unpredictability	**(i)** Data replication and redundant requests **(ii)** Static and dynamic reservation of bandwidth **(iii)** Centralized and distributed bandwidth guarantee **(iv)** Bandwidth guarantee based on network topology and application communications	[158] [154] [20][182] [80][20][129][73] [97]
Data security	**(i)** Technical solutions (encryption algorithms, audit third party (ATP), digital signature) **(ii)** Managerial solutions **(iii)** A combination of solutions (i) and (ii)	[153]

stores) and *erasure coding* are used for high availability of data. Using these schemes across data stores also mitigates data lock-in in the face of appearance of new data store with lower price, mobility of users, and change in workload that demands data migration.

- *Data transfer bottleneck*. This challenge arises when data-intensive applications are deployed within a single data store. It reflects the optimality of data placement that should be conducted based on the characteristics of the application as listed for OLTP and social networks in Table 2.3. To reduce more network congestion, data flows should be monitored in the switches, and they should be then scheduled (i) based on the data flow prediction [55], (ii) when data congestion occurs [7], and (iii) for integrated data flows [152] rather than individual data flow. In addition, data aggregation [54], novel network typologies [173], and optical circuit switching deployment [43] are other approaches to drop network congestion.
- *Performance unpredictability*. Shared storage services face the challenges due to multi tenancy, like *unpredictable performance* and *multi-tenant unfairness*, which results in degrading the response time of requests. In Table 2.3, the first two solutions solve challenge with respect to storage, where replicas placement and selection should be considered. The last two solutions cope the challenges related to network aspect, where *fairness* in allocated network bandwidth to cloud applications and *maximiz-*

ing of network bandwidth utilization should be taken into consideration. These solutions are: (i) static and dynamic reservation of bandwidth where the static approach cannot efficiently utilize bandwidth, (ii) centralized and distributed bandwidth guarantee where the distributed approach is more scalable, and (iii) bandwidth guarantee based on the network topology, and application communications which is more efficient [97]. Such solutions suffer from data delivery within deadline, and they thus are suitable for batch applications, but not for OLTP applications which require the completion of data flows within deadline [170].

- *Data security*. This is one of the strongest barriers in the adoption of public clouds to store users' data. It is a combination of (i) data integrity, protecting data from any unauthorized operations; (ii) data confidentiality, keeping data secret in the storage (iii) data availability, using data at any time and place; (iv) data privacy, allowing data owner to selectively reveal their data; (v) data transition, detecting data leakage and lost during data transfer into the storage; and (vi) data location, specifying who has jurisdiction and legislation over data in a transparent way. Many solutions were proposed in recent decades to deal with concerns over different security aspects except *data location* where data is probably stored out of users' control. These solutions are not generally applicable since unlike the traditional systems with two parties, the cloud environment includes 3 parties: users, storage services, and vendors. The last party is a potential threat to the security of data since it can provide a secondary usage for itself (e.g., advertisement purposes) or for governments.

  Solutions to relieve concerns over data security in the cloud context can be *technical*, *managerial*, or *both*. In addition to technical solutions as listed in Table 2.3, managerial solutions should be considered to relieve security concerns relating to the geographical location of data. A combination of both type of solutions can be: (i) designing location-aware algorithms for data placement as data are replicated to reduce latency and monetary cost, (ii) providing location-proof mechanisms for user to know the precise location of data (e.g., measuring communication latency and determining distance), and (iii) specifying privacy acts and legislation over data in a transparent way for users. Since these acts, legislation, and security and privacy requirements of users are a fuzzy concept, it would be relevant to design a fuzzy framework in conjunction of location-aware algorithms. This helps users to find storage services which are more secure in respect to rules applied by the location of data.

2.2.5 Architecture, Motivations, and Challenges of Inter-Cloud Storage

To overcome the above challenges in intra-cloud storage, application providers take the advantages of Inter-Cloud Storage deployment. This section describes the layered architecture of Inter-Cloud storage deployment along with its key benefits and challenges.

Architecture of Inter-Cloud Storage

The architecture of inter-cloud storage does not follow standard protocols. So far, several studies exploit multiple data stores diversity for different purposes with a light focus on the architecture of Inter-cloud storage (e.g., RACS [3] and ICStore [33]). Spillner et al. [157] focused more on the Inter-cloud storage architecture that allows user to select a data store based on service cost or minimal downtime. Inspired by this architecture, we pictorially clarify and discuss a layered inter-cloud architecture as shown in Fig. 2.4.

- *Transport layer* represents a simple data transport abstraction for each data store in its underneath layer and transfers data to data store specified in the upper layer by using its transport module. This layer implements a file system (e.g., the Linux Filesystem in Userspace[5]) to provide *protocol adaptor* for the data store.
- *Data management layer* provides services for managing data across data stores and consists of services such as load balance, replication, transaction, encryption, and decryption. Load balance service monitors the submitted requests to each data store and reports to replication service in order to store a new replica or write/read the object in/from data store with the lowest load. Replication and transaction services can be configured in the layered architecture and collaborate with each other to provide the desired consistency. Encryption and decryption services can be deployed to make data more secure and reduce the storage and network overheads.
- *Integration layer* allows user to access and manipulate the data through *client library*

[5] Filesystem in Userspace. https://en.wikipedia.org/wiki/Filesystem_in_Userspace

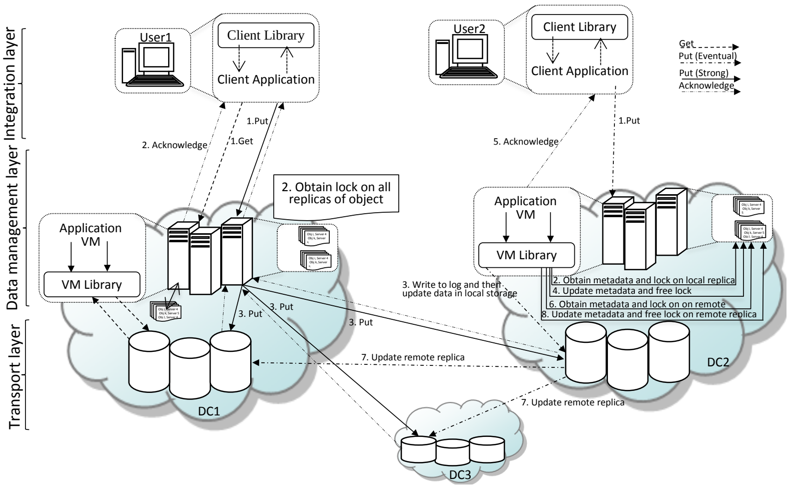


Figure 2.4: Inter-cloud storage architecture

(API), which is exploited by the client application to issue operations to data stores (often the closest one).

- *Metadata component* plays as an essential component in the architecture and collaborates with data management layer for writing (resp. reading) objects in (resp. from) data stores. It contains *object metadata* (e.g., in the form of XML) to determine the location of object replicas and is stored by all DCs in which the application is deployed.

As an example in Fig. 2.4, an issued Get (read) request from client library is usually directed to the closest DC (e.g., DC1), and the requested object is searched in metadata stored at DC. If the object exists, then the application code in VM retrieves the object from the local data store and returns it to the client. Otherwise, the DC sends the request to DC which hosts the object according to the metadata. For a Put (write) request, after finding the location of replicas, first the object is written in VM's application and the metadata in VM is changed, and then the VM library invokes the Put in client library to store the object in local data store via *transport layer*. Finally, the Put is propagated to other replicas to guarantee eventual consistency (Put in DC2) or, in a simple way, all replicas are locked and the Put is synchronously propagated to them to provide strong consistency (Put in DC1).

Motivations of Inter-Cloud Storage

Table 2.4 lists the key motivations of leveraging Geo-replicated data stores. These are as follows:

- *High availability, durability, and data lock-in avoidance.* These are achievable via data replication scheme across data stores owned by different cloud providers. A 3-way data replication provides availability of 7 nines [116] which is adequate for most applications. This is inadequate for protecting data against *correlated* failures, while two replicas are sufficient for guarding data against *independent* failures [49]. *Erasure coding* is another way to attain higher durability even though it degrades data availability in comparison to data replication.
- *Cost benefit.* Due to *pricing differences* across data stores and *time-varying workloads*, application providers can diversify their infrastructure in terms of vendors and locations to optimize cost. The cost optimization should be integrated with the demanding QoS level like availability, response time, consistency level, etc. This can be led to a trade-off between the cost of two resources (e.g., storage vs. computing) or the total cost optimization based on a single-/multi-QoS metrics.
- *Lowe user-perceived latency.* Application providers achieve lower latency through deploying applications across multiple cloud services rather than within a single cloud services. Applications can further decrease latency via techniques listed in Table 2.4. In spite of these efforts, users may still observe the latency variation which can be improved through caching data in memory [124]), issuing redundant reads/writes to replicas [179]), and using feedback from servers and users to prevent requests redirection to saturated servers [161]).[6]
- *High data confidentiality, integrity, and auditability.* Using *Cryptographic protocols with erasure coding and RAID techniques* on top of multiple data stores improves security in some aspects as deployed in HAIL [28] and DepSky [25]. In such techniques, several concerns are important: scalability, cost of computation and storage for encoding data, and the decision on where the data is encoded and the keys used for data encryption are maintained. A combination of private and public data stores

[6] It is worth to mention that the latency can be reduced via optimization in terms of viirtualization architecture design, VM introspection, and inter-domain communication between virtual machines [45].

Table 2.4: Inter-Cloud storage motivation.

Goals	Techniques/Schemes	Section(s)
High availability, durability, and data lock-in avoidance	**(i)** Data replication **(ii)** Erasure coding	2.4.1 2.4.2
Cost benefit	**(i)** Exploitation of pricing differences across data stores and time-varying workloads	2.6
Low user-perceived latency	**(i)** Placing replicas close to users **(ii)** Co-locating the data accessed by the same transactions **(iii)** Determining the location and roles of replicas (master and slave) in a quorum-based configuration [150]	—
High data confidentiality, integrity, and auditability	**(i)** Cryptographic protocols with erasure coding and RAID techniques	—

and applying these techniques across public data stores improve data protection against both insider and outsider attackers, especially for insider ones who require access to data in different data stores. Data placement in multiple data stores, on the other hand, brings a side effect since different replicas are probably stored under different privacy rules. Selection of data stores with similar privacy acts and legislation rules would be relevant to alleviate this side effect.

Challenges of Inter-Cloud Storage

The deployment of data-intensive applications across data stores is faced with the key challenges as listed in Table 2.5. These are discussed as below:

- *Cloud interoperability and portability.* Cloud interoperability refers to the ability of different cloud providers to communicate with each other and agree on the data types, SLAs, etc. Cloud portability means the ability to migrate application components and data across cloud providers regardless of APIs, data types, and data models. Table 2.5 lists solutions for this challenge.
- *Network congestion.* Operating across Geo-DCs causes *network congestion* which can be *time-sensitive* or *non time-sensitive*. The former, like interactive traffic, is sensitive to delay, while the latter, like transferring big data and backing up data, is not so strict to delay and can be handled within deadline [193] or without deadline [79]. The first solution for this challenge, listed in Table 2.5, is expensive, while the second solution increases the utilization of network. The last solution is Software-Defined Networking (SDN) [92]. SDN separates *control plane* that decides how to handle network traffic, and *data paths* that forwards traffic based on the decision

Table 2.5: Inter-Cloud storage challenges

Challenges	Solutions	Example	Section(s)
Cloud interoperabiliy and portability	Standard protocols for IaaS	n/a†	—
	Abstraction storage layer	CASL[75]	
	Open API	JCloud	
Network congestion	Dedicating redundant links across DCs	n/a	—
	Store and forward approach	Postcard [66], [175]	
	Software-Defined Networking (SDN)	B4 [79], [175]	
Strong consistency and transaction guarantee	Heavy-weight coordination protocols	—	2.5
	Contemporary techniques		

†n/a: not applicable,

made from control plane. All these solutions answer a part of this fundamental question: *how to schedule the data transfer so that it is completed within a deadline and budget subject to the guaranteed network fairness and throughput for jobs that processes the data*.

- *Strong consistency and transaction guarantee.* Due to high communication latency between DCs, coordination across replicas to guarantee strong consistency can drive users away. To avoid high communication latency, some data stores compromise strong consistency at the expense of application semantics violations and stale data observations. Others, on the other hand, provide strong consistency in the cost of low availability. To achieve strong consistency without compromising with availability and scalability, coordination across replicas should be reduced or even eliminated.

Challenges of Inter-Cloud Storage

The deployment of data-intensive applications across data stores is faced with the key challenges as listed in Table 2.5. These are discussed as below:

- *Cloud interoperabiliy and portability.* Cloud interoperabiliy refers to the ability of different cloud providers to communicate with each other and agree on the data types, SLAs, and etc. Cloud portability means the ability to migrate application components and data across cloud providers regardless of APIs, data types, and data models. A well-solution for cloud portability is standard protocols which are proposed more for IaaS, not for XaaS (X: Platform, Software, and Database). Therefore, there is a need of (i) an abstraction storage layer (e.g., CSAL [75]) or open API (e.g., JCloud[7]) to make applications portable across data clouds, and (ii) a framework (e.g., CDPort [10]) to ease data portability via a transparent layer on top of *various query interfaces*, *non-unified data types*, and *data models* provided by different data stores.

- *Network congestion.* Operating across Geo-DCs causes *network congestion*, which can be *time-sensitive* or *non time-sensitive*. The former is sensitive to delay. Interactive traffic, as an example of the time-sensitive traffic, can be happened in the case of guaranteeing strong consistency across replicas. The latter traffic is not so strict to delay, and it can be handled within deadline (e.g., Amoeba [193]) or without deadline (B4 [79]). Transferring big data and moving backup data across DCs fall in this category of traffic. A trivial solution for dealing with network congestion is dedicating redundant links across DCs, but it is expensive and wasteful. A better approach is conservatively using network bandwidth across DCs so that network utilization increases, while the cost and time of data transferring decrease. *Store and forward* is an example of such approach in which the data is split into chunks that are scheduled during long periods, over multiple paths and multiple intermediate DCs within a path. By deploying this approach, NetStitcher [96] increases network utilization, Postcard [66] minimizes the cost of data transferring, and Wu. et al. [175] maximize the network utilization while the transfer of data is guaranteed within the deadline.

  To make more effective reduction of network congestion and improvement of data transmission across DCs, traffic engineering techniques recently deployed Software-Defined Networking (SDN) [92]. SDN separates *control plane* that decides how to handle network traffic, and *data paths* that forwards traffic based on the decision made from control plane. For example, B4 [79] deployed by Google's Inter-DC WAN and the architecture studied the transfer of Big data across DCs [175], work based on SDN to improve the utilization of Inter-DCs WAN. These studies try to answer a part of this fundamental question: how to schedule the data transfer so that it is completed within a deadline and budget subject to the guaranteed network performance (e.g., fairness and throughput) for users or jobs that processes

[7] JCloud project. https://jclouds.apache.org/

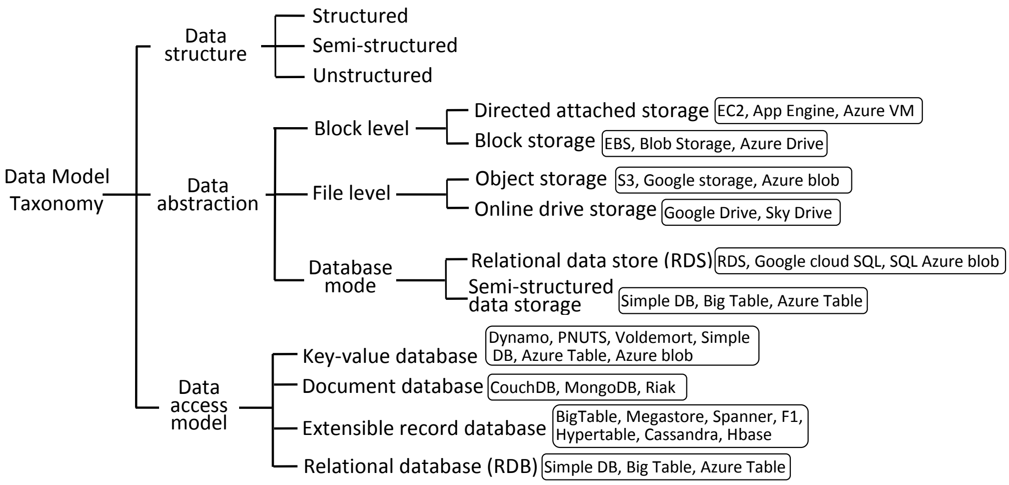

Data Model Taxonomy
Data structure
Structured
Semi-structured
Unstructured
Data abstraction
Block level
Directed attached storage
EC2, App Engine, Azure VM
Block storage
EBS, Blob Storage, Azure Drive
File level
Object storage
S3, Google storage, Azure blob
Online drive storage
Google Drive, Sky Drive
Database mode
Relational data store (RDS)
RDS, Google cloud SQL, SQL Azure blob
Semi-structured data storage
Simple DB, Big Table, Azure Table
Data access model
Key-value database
Dynamo, PNUTS, Voldemort, Simple DB, Azure Table, Azure blob
Document database
CouchDB, MongoDB, Riak
Extensible record database
BigTable, Megastore, Spanner, F1, Hypertable, Cassandra, Hbase
Relational database (RDB)
Simple DB, Big Table, Azure Table

Figure 2.5: Data model taxonomy

the data.

- *Strong consistency and transaction guarantee.* Due to considerably high communication latency between DCs, coordination across replicas to guarantee strong consistency can drive users away. To avoid such coordination, one class of data stores compromises strong consistency in the expense of application semantics violations and stale data observations. In contrast, another class of data stores provides strong consistency in the cost of low availability and scalability which are in conflict with the key objective of data stores. To achieve strong consistency without scarifying the objective of data stores (i.e., high availability and scalability), the need to coordination across replicas should be reduced or even eliminated (See Section 2.5).

2.3 Data model

Data model reflects that how data is logically organized, stored, retrieved, and updated in data stores. We thus study it from different aspects and map data stores to the provided data model taxonomy in Fig. 2.5.

2.3.1 Data structure

Data structure affects the speed of assimilation and information retrieving. It has three categories. (i) *Structured* data refers to data that defines the relation between the fields specified with a name and value, e.g., RDBs. It supports a comprehensive query and transaction processing facilities. (ii) *Semi-structured* is associated with the special form of structured data with a specific schema known for its application and database deployments (e.g., document and extensible DBs). It supports simple query (e.g., primitive operations) and transaction facilities as compared to structured data. (iii) *Unstructured* data refers to data that have neither pre-defined data model nor organized in a pre-defined way (e.g., video, audio, and heavy-text files). It takes the simplest data access model, i.e., key-value, that delivers high scalability and performance at the cost of sacrificing data consistency semantic and transaction support.

In these logical data structures, data is internally organized row-by-row, column-by-column closely related to database normalization, or combination of both schemes- called hybrid – within a block. *Structured data* can be organized in all schemes, *semi-structured* in row-by-row and hybrid schemes, and *Unstructured data* in a row-by-row scheme.

2.3.2 Data abstraction

This refers to different levels of storage abstraction in data stores. These levels are as below:

1. *Block-level* provides the fastest access to data for virtual machine (VM). It is classified into (i) *directed-attached storage* coming with a VM that provides highly sequential I/O performance with a low cost, and (ii) *block storage* that pairs with a VM as a local disk.
2. *File-level* is the most common form of data abstraction due to its ease of management via simple API. It is provided in the forms of (i) *object storage* that enables users to store large binary objects anywhere and anytime, and (ii) *online drive storage* that provides users with folders on the Internet as storage.
3. *Database mode* offers storage in the forms of *relational data store* and *semi-structured data storage* which respectively provide users with RDB and NoSQL/NewSQL databases. RDB exploiting the SQL standard does not scale easily to serve large web applications but guarantees strong consistency. In contrast, NoSQL provides horizontal scalability by means of shared nothing, replicating, and partitioning data over many servers for simple operations. In fact, it preserves BASE (Basically

Available, Soft state, Eventually consistent) properties instead of ACID ones in order to achieve higher performance and scalability. NewSQL– as a combination of *RDB and NoSQL*– targets delivering the scalability similar to NoSQL, meanwhile maintaining ACID properties.

Table 2.6: Comparison between different storage abstractions.

Ease of use†	Scalability	Performance††	Cost [Applicability]
File-level	File-level	Block-level	Block-level [OLAP applications]
Database mode	Block-level	Database mode	Database mode [OLTP with low latency queries]
Block-level	Database-mode	File-level	File-level [Backup data and web content static]

† The levels of storage abstract are listed from high to low for each aspect listed in each column. †† Performance is defined in terms of *accessibility*.

Table 2.6 compares different levels of storage abstractions in several aspects as well as their applicability. This comparison indicates that as the storage abstraction (ease of use) level increases, the cost and performance of storage reduce.

2.3.3 Data access model

This reflects storing and accessing model of data that affect on consistency and transaction management. It has four categories as below:

1. *Key-value database* stores keys and values which are indexed by keys. It supports primitive operations and high scalability via keys distribution over servers. Data can be retrieved from the data store based on more than one attribute if additional key-value indexes are maintained.
2. *Document database* stores all kinds of documents indexed in the traditional sense and provides primitive operations without ACID guarantee. It thus supports *eventual consistency* and achieves scalability via *asynchronous* replication, *shard* (i.e., horizontal partition of data in the database), or both.
3. *Extensible record database* is analogous to table in which columns are grouped, and rows are split and distributed on storage nodes [37] based on the primary key range as a *tablet* representing the unit of distribution and load balancing. Each cell of the table contains multiple versions of the same data that are indexed in decreasing timestamps order, thereby the most recent version can always read first [142]. This scheme is called NewSQL which is equivalent with *entity group*

in Megastore, *shard* in Spanner, and *directory* in F1 [155].

4. *Relational database* (RDB) has a comprehensive pre-defined scheme and provides manipulation of data through SQL interface that supports ACID properties. Except for *small-scope* transactions, RDB cannot scale the same as NoSQL.

Table 2.7: Comparison between different databases.

Database	Simplicity	Flexibility	Scalability	Properties	Data Structure	Application [Query] type
key-value	High	High	High	–	Unstructured	OLAP [Simple]
Document	High	Moderate	Moderate	BASE	Semi-structured	OLAP [Moderate]
Extensible record	High	High	High	ACID	Structured	OLTP [Moderate, repetitive]
Relational	Low	Low	Low	ACID	Structured	OLTP [Complex]

Table 2.7 compares NoSQL (i.e., the first three databases) and relational databases in several aspects and indicates which type of application and query can deploy these databases. NoSQL databases offer horizontal scalability and high availability compared to the relational databases by scarifying consistency semantic and query processing which respectively make them unsuitable for the deployment of OLTP and OLAP applications. To simultaneously achieve strong consistency and scalability for OLTP applications, it is vital to carefully partition data within and especially across data stores and to use the mechanisms that exempt or minimize the coordination across transactions. The rarely use of these databases has also several limitations relating to big data for OLAP applications. All these disk-based data stores cannot suitably facilitate OLAP applications in the concept of *velocity* and thus most commercial vendors combine them with in-memory NoSQL/relational data stores (e.g., Memcached[8], Redis[9] and RAMCloud [139]) to further improve performance. In terms of *variety*, OLAP applications receive data with different formats which require a platform to translate them into a canonical format. OLAP applications can combat the remaining limitations (i.e., veracity and value) in the face of large data volumes via designing indexes to retrieve data from data stores. It would be efficient to design such indexes with low time and space complexity using hash- and tree-based methods.

[8] Memcached project. https://memcached.org/

[9] Redis project. https://redis.io/

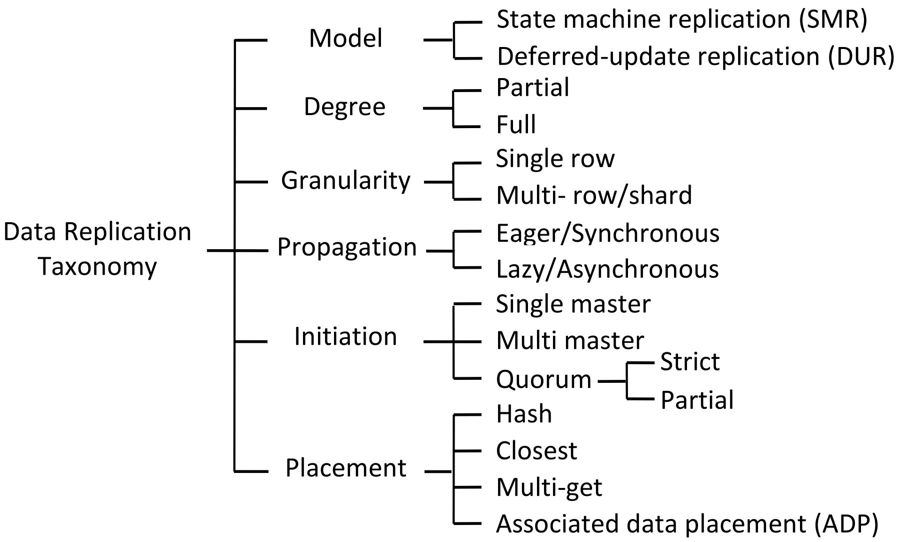


Figure 2.6: Data replication taxonomy

Table 2.8: Comparison between replication models.

Replication Models	Advantages	Disadvantages
State machine replication†	Failure transparency Abort-free	Low throughput for RW transactions Low scalability for RW transactions
Deferred-update replication††	High throughput and scalability for RO transactions.	Stale data Replica divergence

† It is also called *active replication*.

†† Sate machine replication and deferred-update replication provide *linearizability* and *non-linearizability* consistency semantic.

2.4 Data Dispersion

This section discusses data dispersion schemes as shown in Fig. 2.1.

2.4.1 Data replication

Data replication improves availability, performance (via serving requests by different replicas), and user-perceived latency (by assigning requests to the closest replica) at the cost of replicas coordination and storage overheads. This is affected by facets of data replication based on the taxonomy in Fig. 2.6.

Data replication model

There are two replication models for fault tolerant data stores [128]: The model is *state machine replication* (SMR) in which all replicas receive and process all operations in a *deterministic way* (in the same order) using *atomic broadcast* in which all replicas receive the same set of updates in the same order. This implies SMR is abort-free and failure transparency which means if a replica fails to process some operations those are still processed in other replicas. However, SMR has low throughput and scalability for read and write (RW) transactions since all servers process each transaction. Thus, the scalability and throughput are confined by the processing power of a server. Scalable-SMR (S-SMR) [26] solves this issue across data stores via (i) partitioning database and replicating each partition, and (ii) using cache techniques to reduce communication between partitions without compromising consistency. SMR and S-SMR are suitable for contention-intensive and *irrevocable transactions* that require abort-free execution.

The second model is Deferred-update replication (DUR). It resembles single-/multi master replication and scales better than SMR due to locally executing RW transactions on a server and then propagating updates to other servers. Thus, in DUR, the RW transactions do not scale with the number of replicas in comparison to the read-only (RO) transactions executed only on a server without communication across servers by using a *multiversion* technique. Scalable-DUR (S-DUR) [145] and Parallel-DUR (P-DUR) [126] allow update transactions to scale with the number of servers and the number of cores available for a replica respectively. In respect to pros and cons summarized in Table 2.8, the scalability and throughput of transactions can be improved through borrowing the parallelism in DUR and abort-free feature in SMR [89].

Data replication degree

Data replication can be either *partial* or *full*. In partial (resp. full) replication each node hosts a *portion* (resp. *all*) of data items. For example, in the context of Geo-DCs, *partial (resp. full) replication* means that certain (resp. all) DCs contain a replica of certain (resp. all) data items.

Table 2.9 shows that partial replication outperforms full replication in storage services

Table 2.9: Comparison between full and partial replications.

Properties	Comparison	Reason/Description
Scalability	Partial > Full	Due to access to a subset of DCs not all DCs
Complexity	Partial > Full	Due to the requirement for the exact knowledge where data reside
Storage Cost	Partial < Full	Due to data replication in a subset of data stores
Applicability	Partial > Full	If read requests come a specific DCs, or when objects are write-intensive
	Partial < Full	If read requests come from all DCs, or when transactions are global

cost due to access to a subset of data stores deployed across DCs. It is also better than full replication in scalability because full replication is restricted by the capacity of each replica that certifies and processes transactions. These advantages demand more complex mechanisms for consistency and transaction support in partial replication, which potentially degrades response time. Many partial replication protocols provide such mechanisms at the expense of communication cost same as full replication [15]. This is due to unpredictable *overlapping* transactions in which the start time of transaction T_i is less than the commit time of transaction T_j and the intersection of write set T_i and T_j is not empty. The deployment of *genuine partial replication* solves such issue and enforces a transaction to involve only the subset of servers/DCs containing the desired replicas for coordination. In terms of applicability, partial and full replication is more suitable for write-intensive objects (due to submitting each request to a subset of DCs [151]) and for execution of global (multi-shard) transactions respectively.

Therefore, the characteristics of workload and the number of DCs are main factors in making a decision on what *data replication degree* should be selected. If the number of DCs is small, full replication is preferable; otherwise, if global transactions access few DCs, partial replication is a better choice.

Data replication granularity

This defines the level of data unit that is replicated, manipulated, and synchronized in data stores. Replication granularity has two types: *single row* and *multi-row/shard*. The former naturally provides horizontal data partitioning, thereby allowing high availability and scalability in data store like Bigtable, PNUTS, and Dynamo. The latter is the first step beyond single row granularity for new generation web-applications that require to attain both high scalability of NoSQL and ACID properties of RDBs (i.e., NewSQL features).

This type of granularity has an essential effect on the scalability of transactions.

Update propagation

This reflects *when* updates take place and is either *eager/synchronous* or *lazy/asynchronous*. In eager propagation, the committed data is simultaneously conducted on all replicas, while in lazy propagation the changes are first applied on master replica and then on slave replicas. Eager propagation is applicable on a single data store like SQL Azure [36] and Amazon RDS, but it is hardly feasible across data stores due to response time degradation and network bottleneck. In contrast, lazy propagation is widely used across data stores (e.g., Cassandra) to improve response time.

Update initiation

This refers to *where* updates are executed in the first place. Three approaches for update initiation are discussed as below:

Single master approach deploys a replica at the closest distance to the user or a replica receiving the most updates as master replica. All updates are first submitted to the master replica and then are propagated either eagerly or lazily to other replicas. In single master with lazy propagation, replicas receive the updates in the same order accepted in the master and might miss the latest versions of updates until the next re-propagation by the master. Single master approach has advantages and disadvantages as listed in Table 2.10. These issues can be mitigated somehow by *multi-master* approach in which every replica can accept update operations for processing and in turn propagates the updated data to other replicas either eagerly or lazily. Thus, this approach increases the throughput of read and write transactions at the cost of stale data, while replicas might receive the updates in the different order which results in replicas divergence and thus the need for conflict resolution.

Quorum approach provides a protocol for availability vs. consistency in which writes are sent to a write set/quorum of replicas and reads are submitted to a read quorum of replicas. The set of read quorum and write quorum can be different and both sets share a replica as coordinator/leader. The reads and writes are submitted to a *coordinator* *replica* which is a single master or multi-master. This protocol suffers from the disadvantages in determining the coordinator location and the quorum of write and read replicas as addressed when workload changes [150]. Though this classical approach guarantees strong consistency, many Geo-replicated data stores like Cassandra, Dynamo, Riak[10], and Voldemort[11] achieve higher availability at a cost of weaker consistency via its adaptable version in which a read/write is sent to all replicas and is considered successful if the acknowledgements are received from a quorum (i.e., majority) of replicas. This adapted protocol is configured with write (W) and read quorum (R) in synchronous writes and reads. The configuration is determined in (i) *strict* quorum in which any two quorums have non-empty intersection (i.e., $W + R > N$, where N is the number of replicas) to provide strong consistency, and (ii) partial quorum in which at least two quorums should not overlap (i.e., $W + R < N$) to support weak consistency. Generally speaking, (i) a raise in $\frac{W}{R}$ improves the consistency, and (ii) a raise in W reduces availability and increases durability.

Replica placement

This is related to the mechanism of replica placement in data store and is composed of four categories. (1) *Hash mechanism* is intuitively understood as a random placement and determines the placement of objects based on the hashing outcome (e.g., Cassandra). It effectively balances the load in the system, however it is not effective for a transactional data store that requires co-located multiple data items. (2) *Closest mechanism* replicates a data item in the node which receives the most requests for this data item. Although closest mechanism decreases the traffic in the system, it is not efficient for a transactional data store because the related data accessed by a transaction might be placed in different locations. (3) *Multiget* [124] seeks to place the associated data in the same location without considering the localized data serving. (4) *Associated data placement* (ADP) [188] makes the strike between closest and Multiget mechanisms.

As discussed above, using each strategy differed on various aspects of replication can affect the response time, complexity of consistency and transaction, and monetary cost.

[10] Riak NoSQL Database. http://docs.basho.com/riak/kv/2.1.4/learn/why-riak-kv/
[11] Voldemort project. http://www.project-voldemort.com/voldemort/

Among these, the key challenge is the coordination between replicas without compromising response time (i.e., consistency-latency trade-off), which depends on update *initiation* and *propagation*, as discussed in Table 2.10.

Table 2.10: Consistency-latency tradeoff of different replication techniques.

Technique	Synchronous propagation	Asynchronous propagation
Single master	Case 1: • Latency is high especially across DCs and is constrained by the slowest DC. • Strong consistency is guaranteed, no matter reads are performed on which replica. • Latency source is the latency within (is very small) and across DCs. • This trades response time for consistency.	Case 2: • Latency is lower than that in Case 1. • If reads are submitted to the master replica, then consistency is guaranteed and latency increases. Otherwise, the trend for consistency and latency is reverse. • Latency source are: (i) routing reads to master replica, and (ii) waiting time in queue for serving reads and recovery time for probable failed master replica. • Both aspects of consistency-latency trade-off are achievable.
Multi master	Case 3: • It incurs latency the same as that in synchronous single master strategy, though to roughly lesser degrees. • Strong consistency is guaranteed, no matter where reads are submitted. • Latency source is the same as in Case 1 plus conflict detection and resolution time. • This trades consistency for latency.	Case 4: • Latency is less than that in Case 3, and roughly in Case 2. • The trade between consistency and latency is similar to that in Case 2. • Latency source is the same as that in Case 2.
Quorum-based	Case 5: • If reads are submitted to synchronous updated replica precipitated in quorum, then consistency is guaranteed but latency is high. • Latency source is similar to that in Case 2. • This trades consistency for latency.	Case 6: • If reads are submitted to asynchronous updated replica precipitated in quorum, then consistency is not guaranteed. • Latency source is the same as that in Case 2.

2.4.2 Erasure coding

Cloud file system uses *erasure coding* to reduce storage cost and to improve availability as compared to data replication. A (k, m)-erasure coding divides an object into k equally sized chunks that hold original data along with m extra chunks that contain data coding (parity). All these chunks are stored into $n = k + m$ disks which increases the storage

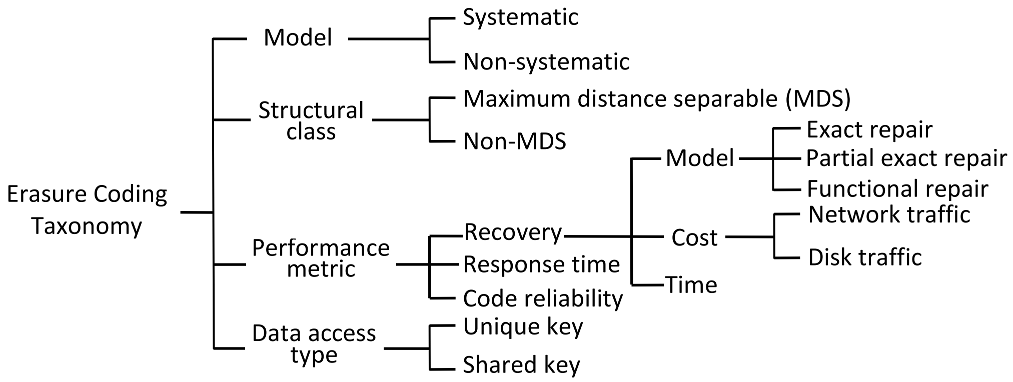


Figure 2.7: Erasure coding taxonomy

overhead by a factor of $1/r = k/n < 1$ and tolerates m faults, as opposed to $m-1$ faults for $m-$way data replication with a factor of $m-1$ storage overhead. For example, a (3,2)-erasure coding tolerates 2 failed replicas with a 2/3(=66%) storage overhead as compared to 3-way replication with the same fault tolerance and a 200% storage overhead. To use erasure coding as an alternative to data replication, we need to investigate it in the following aspects as shown in Fig. 2.7.

Model

Erasure coding has two models. *Systematic* model consists of *(original) data* and *parity* chunks which are separately stored in $n-m$ and m storage nodes. *Non-systematic* model includes the coded data (not original data) which are stored in n nodes. Systematic and non-systematic codes are respectively relevant for archival and data-intensive applications due to respectively low and high rate of reads. Systematic codes seem more suitable for data stores because they decode data when a portion of data is unavailable. Comparatively, non-systematic codes decode data whenever data are retrieved due to storing the coded data not the original data. This may degrade the response time.

Structural class

This represents the reconstruct-ability of code that is largely reliant on erasure coding rate (i.e., r), including two classes. The first class is (k,m)-maximum distance separable (MDS) code in which the data is equally divided into k chunks which are stored in $n = k + m$

storage nodes. A (k,m)-MDS code tolerates any m of n failed nodes and rebuilds the unavailable data from any k surviving nodes, known as MDS-*property*. A MDS code is optimal in terms of reliability vs. storage overhead while it incurs significant recovery overheads to repair failed nodes. The second class is non-MDS code. Any code that is not MDS code is called non-MDS code. This code tolerates less than m failed nodes and typically rebuilds the failed nodes from less than k surviving nodes. Thus, compared to MDS codes, non-MDS codes are (i) more economical in network cost to rebuild the failed nodes, which makes them more suitable for deploying across data stores and (ii) less efficient in the storage cost and fault-tolerance.

Performance metrics

Erasure coding is evaluated based on the following performance metrics that have received significant attention in the context of cloud.

(1) *Recovery* refers to the amount of data retrieving from disk to rebuild a failed data chunk. Recovery is important in the below aspects.

- *Recovery model* in which a failed storage node is recovered through survivor nodes has three models [160]. The first is *exact-repair* in which the failed nodes are exactly recovered, thus lost parity with their exact original data are restored. The second is *partial exact-repair* in which the data nodes are fixed exactly and parity nodes are repaired in a functional manner by using random-network-coding framework [62]. This framework allows to repair a node via retrieving functions of stored data instead of subset of stored data so that the code property (e.g., MDS-property) is maintained. The third is *functional repair* in which the recovered nodes contain different data from that of the failed nodes while the recovered system preserves the MDS-code property. Workload characteristics and the deployed code model determine which recovery model satisfies the application requirements. Exact- and partial exact-repair are appropriate for archival applications while functional repair is suitable for non-secure sensitive applications because it requires the dynamics of repairing and decoding rules that results in information leakage.
- *Recovery cost* is the total number of required chunks to rebuild a failed data chunk. It consists of *disk traffic* and *network traffic* affected by network topology and repli-

cation policy [197]. Recently, the trade-off between recovery cost and storage cost takes considerable attention in the context of cloud as discussed below.

A (k,m)-Reed-Solomon (RS) code contains k data chunks and m parity chunks, where each parity chunk is computed from k chunks. When a data chunk is unavailable, there is always a need of any subset of k chunks form $m + k$ chunks, as recovery cost, to rebuild the data chunk. This code is used in Google ColossusFS [69] and Facebook HDFS [122] within a DC (a XOR-based code–using pure XOR operation during coding computation– across DCs). In spite of the RS code optimality in reliability vs. storage overhead, it is still unprofitable due to high bandwidth requirements within and across data stores. Hitchhiker [136] mitigates this issue without compromise on storage cost and fault tolerance throughout the adapted RS code in which a single strip RS code is divided into two correlated sub-stripes.

Similar to MDS-code, *regenerating* and non-MDS codes [176] alleviate the network and disk traffics. Regenerating codes aim at the optimal trade-off between storage and recovery cost and come with two optimal options [135]. The first is the *minimum storage regenerating* (MSR) codes which minimize the recovery cost keeping the storage overheads the same as that in MDS codes. NCCloud [44] uses *functional* MSR, as a non-systematic code, and maintains the same fault tolerance and storage overhead as in RAID-6. It also lowers recovery cost when data migrations happen across data stores due to either transition or permanent failures. The second is the *minimum bandwidth regenerating* (MBR) codes that further minimize the recovery cost since they allow each node to store more data.

A $(k, l, r)-$ Local Reconstruction Code (LRC) [77] divides k data blocks into l local groups and creates a *local parity* for each local group and $r = \frac{k}{l}$ *global parities*. The number of failure it can tolerate is between $r + 1$ and $r + l$. HDFS-Xorbas [144] exploits LRC to make a reduction of 2x in the recovery cost at the expense of 14% more storage cost. HACFS [180] code also uses LRC to provide a *fast code* with low recovery cost for *hot* data and exploits Product Code (PC) [137]) to offer a compact code with low storage cost for *cold* data. Table 2.11 compares RS-code and LRC, used in current data stores, with data replication in the main performance metrics.

- *Recovery time* refers to the amount of time to read data from disk and transfer it within and across data stores during recovery. As recovery time increases, *response time* grows, resulting in notorious effect on the data availability. Erasure codes can improve recovery time trough two approaches. First, erasure codes should reduce network and disk traffics to which RS codes are inefficient as they read all data blocks for recovery. However, *rotated* RS [87] codes are effective due to reading the requested data only. LRC and XOR-base [136] codes are also viable solutions to decrease recovery time. Second, erasure codes should avoid retrieving data from hot nodes for recovery by replicating hot nodes' data to cold nodes or caching those data in dynamic RAM or solid-state drive.

(2) *Response time* indicates the delay of reading (resp. writing) data from (resp. in) data store and can be improved through the following methods. (i) *Redundant requests* simultaneously read (resp. write) n coded chunks to retrieve (resp. store) k parity chunks [147]. (ii) *Adaptive batching* of requests makes a trade-off between delay and throughput, as exploited by S3 (Simple Storage Service) and WAS [118]. (iii) *Deploying erasure codes across multiple data stores* improves availability and reduces latency. Fast Cloud [103] and TOFEC [104] use the first two methods to make a trade-off between throughput and delay in key-value data stores as workload changes dynamically. Response time metric is orthogonal to data monetary cost optimization and is dependent on three factors: scheduling read/write requests, the location of chunks, and parameters that determine the number of data chunks. Xiang et al. [181] considered these factors and investigated a trade-off between latency and storage cost within a data store.

(3) *Reliability* indicates the *mean time to data loss* (MTTDL). It is estimated by standard Markov model and is influenced by the speed of block recovery [144] and the number of failed blocks that can be handled before data loss. LRCs are better in reliability than RS codes, which in turn, are more reliable than replication [77] with the same failure tolerance. As already discussed, failures in data stores can be *independent* or *correlated*. To relieve the later failure, the parity chunks should be placed in different racks located in different domains.

Data access type

There are two approaches to access chunks of the coded data: *unique key* and *shared key* [104], in which a key is allocated to a chunk of coded data and the whole coded data respectively. These approaches can be compared in three aspects. (1) *Storage cost*: both approaches are almost the same in the storage cost for writing into a file. In contrast, for reading chunks, shared key is more cost-effective than unique key. (2) *Diversity in delay*: with unique key, each chunk, treated as an individual object, can be replicated in different storage units (i.e., server, rack, and DC). With shared key, chunks are combined into an object and very likely stored in the same storage unit. Thus, in unique (resp. shared) key, there is low (resp. high) correlation in the access delay for different chunks. (3) *Universal support*: unique key is supported by all data stores, while shared key requires advanced APIs with the capability of partial reads and writes (e.g., S3).

2.4.3 Hybrid scheme

Hybrid scheme is a combination of data replication and erasure coding schemes to retain the advantages of these schemes while avoiding their disadvantages for data redundancy within and across data stores. Table 2.11 compares two common schemes in performance metrics to which three factors contribute into when and which scheme should be deployed: Access rate (AR) to objects, object size (OS), price (Pr) and performance (Pe) of data stores.

These factors have a significant effect on the *storage overhead*, *recovery cost*, and *read/write latency*. As indicated in Table 2.11, replication incurs storage overhead more than erasure coding especially for large objects, while it requires less recovery cost due to retrieving the replica from a single server/data store instead of fragmented objects from multiple servers/data stores. Erasure coding is more profitable in read/write latency (i) for cold-spot objects since update operations require re-coding the whole object, and (ii) for large objects because the fragmented objects can be accessed in parallel from multiple server/data stores. For the same reasons, replication is more efficient for hot-spot objects with small size like metadata objects. Thus, cold-spot objects with large size should be distributed across cost-effective data stores in the form of erasure coding, and hot-spot objects with small size across performance-efficient data stores in the form of replication.

We classify the hybrid scheme into two categories. (i) *Simple hybrid* stores an object in the form of either replication or erasure coding (ROC) or replication and erasure coding (RAC) during its lifetime. (ii) *Replicated erasure coding* contains replicas of each chunk of coded objects and its common form is *double coding* which stores two replicas of each coded chunk of object. Compared to ROC, double coding and RAC increase storage cost two times, but they are better in availability and bandwidth cost due to retrieving the lost chunk of the object from the server which has a redundant copy. Table 2.12 summarizes projects using common redundancy or hybrid schemes. Neither workload characteristics nor data stores diversity (in performance and cost) are fully deployed in these projects using hybrid scheme. It is an open question to investigate the effect of these characteristics and diversities on the two categories of hybrid scheme.

Table 2.11: Comparison between Replication and Erasure Coding schemes.

Schemes	Availability	Durability	Recovery	Storage Overhead	Repair Traffic	Read/Write latency
Replication	High	Low	Easy	>1X	=1X	Low for hot-spot objects with small size
Erasure Coding	Low	High	Hard	<1X	>1X	Low for cold-spot objects with large size

Table 2.12: Comparison between the state-of-the-art projects using diferent redundancy schemes.

Projects	Redundancy Scheme	Contributing Factors	Objective(s)
DepSky [25]	Replication	AR†, Pr	High availability, integrity, and confidentiality
Spanner [177]	Replication	OS, AR, Pr	Cost optimization and guaranteed availability
CosTLO [179]	Replication	OS, AR, Pe	Optimization of variance latency
SafeStore [90]	Erasure coding	AR, Pr	Cost optimization
RACS [3]	Erasure Coding	AR	Cost optimization and vendor-lock in
HAIL [28]	RAID technique	n/a	High availability and integrity
NCCloud [44])	Network Codes	n/a	Recovery cost optimization of lost data
CDStore [100]	Reed-Solomon	n/a	Cost optimization, security and reliability
CAROM [114]	Hybrid (RAC)	AR	Cost optimization
CHARM [195]	Hybrid (ROC)	AR, Pr	Cost optimization and guaranteed availability
HyRD [117]	Hybrid (ROC)	OS, Pr, Pe	Cost and latency optimization

† n/a: (not applicable), OS: (object size), AR: (access rate), Pr: (price), and Pe (performance).

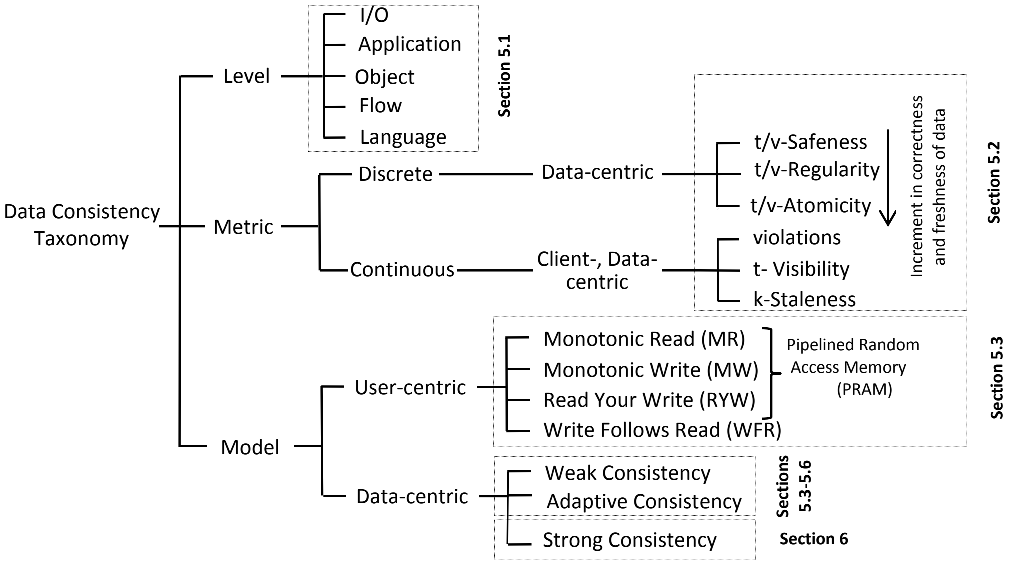


Figure 2.8: Data consistency taxonomy

2.5 Data Consistency

Data consistency means that data values remain the same for all replicas of a data item after an update operation. It is investigated in three main aspects, *level*, *metric*, and *model*, as shown in Fig. 2.8. This section first describes different consistency levels and their pros and cons (Section 2.5.1). Then it defines the consistency metrics to determine how much a consistency semantic/model is stronger than another (Section 2.5.2). Finally, it discusses consistency models from the user-perspective (Section 2.5.3) and from the data store perspectives– weak consistency (Sections 2.5.4 and 2.5.5) and adaptive consistency (Section 2.5.6).

2.5.1 Consistency level

Distributed data stores rely on different levels of data consistency [11], as shown in Fig. 2.8.

I/O-level consistency allows a clear separation between low-level storage and application logic. It simplifies the development of the application and the complexity of distributed programming. However, I/O-level consistency requires conservative assumptions like concurrent write-write and read-write on the same data, resulting in inefficiency. It should also execute writes and reads in a serial order due to its unawareness of the application semantics. *We focus on this level of consistency in this chapter.*

application-level consistency exploits the semantics of the application to ensure the concreteness of invariants[12] without incurring the cost of coordination among operations. Thus, it imposes a burden on the developers and sacrifices the generality/reusability of the application code.

Object-level consistency makes a trade-off between *efficiency* and *reusability* respectively degraded by I/O- and application-level consistency. It provides the convergence of replicas to the same value without any need of synchronization across replicas via Conflicted-free Replicated Data Types (CRDTs) [149] in which the value of objects can change in an *associative*, *commutative*, and *idempotent* fashion. Though object-level consistency removes concerns of the reusability of application-level consistency, it requires mapping the properties of the application to invariants over objects by developers.

Flow-level consistency is an extension of object-level consistency and requires a model to obtain both the semantic properties of *dataflow component* and the dependency between interacting components.[13] Some components are insensitive to message order delivery as a semantic property,[14] and they are *confluent* and produce the same set of outputs under all ordering of their inputs [12]. Flow-level consistency demands manual definition for confluent components, resulting in error-prone. But it is more powerful than object-level consistency and it has more applicability than object- and language-level consistency. Indigo [19] exploits flow-level consistency based on confluent components which contain the application-specific correctness rules that should be met.

Finally, *language-level* consistency integrates the semantics and dependencies of the application and maintains a long history of invariants to avoid distributed coordination across replicas. The CALM principle [13] shows a strong connection between the need of distributed coordination across replicas and *logical monotonicity*. *Bloom* language [13] deploys this principle and translates *logical monotonicity* into a practical program that is expressible via *selection*, *projection*, *join*, and *recursion* operations. This class of program,

[12] The term invariant refers to a property that is never violated (e.g., primary key, foreign key, a defined constraint for the application- e.g., an account balance $x \geq 0$).

[13] A component is a logical unit of computing and storage and receives streams of inputs and produces streams of outputs. The output of a component is the input for other components, and these streams of inputs and outputs implement the flow of data between different services in an application.

[14] The semantic property is defined by application developers. For example, developers determine confluent and non-confluent path between components based on analysis of a component's input/output behaviour.

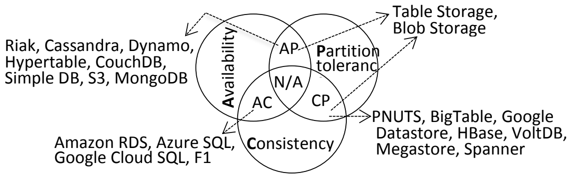


Figure 2.9: Mapping data stores to each pair of properties in CAP

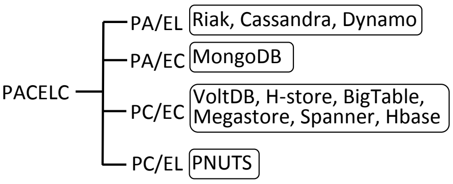


Figure 2.10: PACELC classification and mapping several data stores to the classification

called *monotonic program*,[15] provides output as it receives input elements, and thus guarantees eventual consistency under any order of inputs set. Unlike Bloom, QUELEA language [156] maps operations to a fine-grained consistency levels such as *eventual*, *causal*, and *ordering* and transaction isolation levels like *read committed* (RC), *repeatable read* (RR) [22], and *monotonic atomic view* (MAV) [17].

2.5.2 Consistency metric

This determines how much a consistency model is stronger than another and is categorized into *discrete* and *continuous* from data store perspective (i.e., data-centric) and user perspective (i.e., client-centric).

Discrete metrics are measured with the maximum number of time unit (termed by *t-metric*) and data version (termed by *v-metric*). As shown in Fig. 2.8, they are classified into three metrics [14]: (i) *Safeness* mandates that if a read is not concurrent with any writes, then the most recent written value is retrieved. Otherwise, the read returns any value. (ii) *Regularity* enforces that a read concurrent with some writes returns either the value of the most recent write or concurrent write. It also holds *safeness* property. (iii) *Atomicity* ensures the value of the most recent write for every concurrent or non-concurrent read with write.

Continuous metrics, shown in Fig. 2.8, are defined based on *staleness* and *ordering*. The former metric is expressed in either *t-visibility* or *k-staleness* with the unit of probability distribution of *time* and *version lag* respectively. The latter one is measured as (i) *the number of violations* per time unit from data-centric perspective and (ii) the *probability dis-*

[15] Non-monotonic program contains aggregation and negation queries, and this type of program is implementable via block algorithms that generate output when they receive the entire inputs set.

tribution of violations in the forms of MR-, MW-, RYW-, WFR-violation from client-centric perspective as discussed later.

2.5.3 Consistency model

This is classified into two categories: *user-* and *data-centric* which respectively are vital to application and system developers [162].

User-centric consistency model is classified into four categories as shown in Fig. 2.8. *Monotonic Read* (MR) guarantees that a user observing a particular value for the object will never read any previous value of that object afterwards [189]. *Monotonic Write* (MW) enforces that updates issued by a user are performed on the data based on the arrival time of updates to the data store. *Read Your Write* (RYW) mandates that the effects of all writes issued by users are visible to their subsequent readers. *Write Follows Read* (WFR) guarantees that whenever users have recently read the updated data with version n, then the following updates are applied only on replicas with a version$\geq n$. Pipelined Random Access Memory (PRAM) is the combination of MR-, MW-, and RYW-consistency and guarantees the serialization in both of reads and writes within a session. Brantner et al. [29] designed a framework to provide these client-centric consistency models and the atomic transaction on Amazon S3. Also, Bermbach et al. [23] proposed a middle-ware on eventually consistent data stores to provide MR- and RYW-consistency.

Data-centric consistency aims at coordinating all replicas from the internal state of data store perspective. It is classified into three models. *Weak consistency* offers low latency and high availability in the presence of network partitions and guarantees *safeness* and *regularity*. But it causes a complicated burden on the application developers and caters the user with the updated data with a delay time called *inconsistency window* (ICW). In contrast, *strong consistency* guarantees simple semantics for the developer and *atomicity*. But it suffers from long latency which is eight times more than that of weak consistency, and consequently its performance in reads and writes diverges by more than two orders of magnitude [164]. *Adaptive consistency* is switching between a range of weak and strong consistency models based on the application requirements/constraints like availability, partition tolerance, and consistency.

The consistency (**C**) model has a determining effect on achieving availability (**A**) and

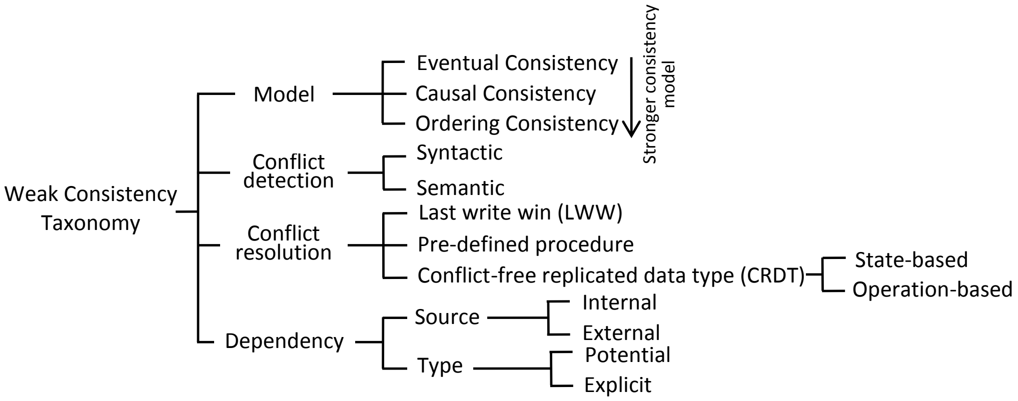


Figure 2.11: Weak consistency taxonomy

partition tolerance (**P**). Based on the CAP theorem [71], data stores provide only two of these three properties. In fact, data stores offer only CA, CP, or AP properties, where CA in CAP is a better choice within a data store due to rare network partition, and AP in CAP is a preferred choice across data stores (see Fig. 2.9). Recently, Abadi [1] redefined CAP as PACELC in order to include latency (**L**) that has a direct influence on monetary profit and response time, especially across data stores, where the latency between DCs might be high. The term PACELC means that if there is a network partition (**P**) then there is a choice between **A** and **C** for designers, else (**E**) the choice is between **L** and **C** (see Fig. 2.10).

2.5.4 Eventual consistency

In this section, we first define the eventual consistency model (Section2.5.4). Then, we discuss how this model is implemented and describe how the conflicts that arise from this model are solved (Section2.5.4).

Definition

Eventual consistency is defined as all replicas eventually converge to the last update value. It purely supports *liveness* which enforces that all replicas eventually converge based on the operations order, while lacking *safety* which determines the correct effects of operations] and leads to incorrect intermediate results. The safety property is assessed in terms *t-visibility* and *k-staleness* as *inconsistency window* (ICW) which is affected by the communication latency, system load, and replicas number. *Probabilistically Bounded Staleness* (PSB) predicts the expected ICW to measure how far data store's behaviour deviates from that of strongly consistent data store [18]. Based on PSB, eventual consistency under partial quorum replication is "good enough" in data freshness while providing a considerable reduction in latency; as confirmed for S3 [24] and Cassandra [134].

Implementation

Eventual consistency-based data stores employ optimistic/lazy replication in which (i) the operation is typically submitted to the closest replica and logged/remembered for the propagation to other replicas later, and (ii) replicas exchange the operation or the effect of operation among each other via *epidemic/gossip protocol* [61] in the background [141]. Operations are partially ordered by deploying vector clocks. This leads to data *conflicts* which happen as operations are simultaneously submitted to the same data in multi-master systems.

There are four approaches to deal with conflicts. (i) *Conflict detection* approaches strengthen the application semantic and avoid the problems arising from ignoring conflicts. These approaches are classified into *syntactic* and *semantic* [141]. The syntactic approach relies on logical or physical clock, whereas the semantic approach works based on the semantic knowledge of operations such as invariants, commuting updates (i.e., CRDTs), and pre-defined procedure. (ii) Conflicts *prohibition* is attainable via blocking or aborting operations and using a single master replica, which comes at the expense of low availability. (iii) Conflicts *ignorance* and *reduction* are achievable by the following conflict resolution techniques (Fig. 2.11) to guarantee *safety*.

(1) *Last Write Win (LWW)* [165] ignores conflicts, and the update with the highest timestamp is accepted (e.g., Riak (by default), SimpleDB, S3, and Azure Table[16]). It causes lost updates (i.e., updates with less timestamp as compared with winner update) and the violation of expected semantics.

(2) *Pre-defined procedure* merges two versions of a data item to a new one according to application-specific semantic as used in Dynamo. The merges must be *associative* and *commutative* for guaranteeing eventual/causal consistency. Albeit the pre-defined pro-

[16] Azure Table storage. `https://azure.microsoft.com/en-us/services/storage/`

cedure solves conflicts without the need of the total order, it is error-prone and lacks generality. Some data stores use *application-specific precondition* (i.e., a condition or predicate that must always be true just prior to the execution of other conditions) to determine *happened-before dependencies* among requests when the causal consistency model comes as a need.

(3) *Conflict-free replicated data types* (CRDTs) avoid the shortcomings of the above approaches and provide eventual consistency in the presence of node failure and network partition without cross replicas coordination. CRDTs enforce the convergence to the same value after all updates are executed on replicas, and they are either *operation-based* or *state-based* [149].

In *state-based* CRDT, the local replica is first updated and then the modified data is transmitted across replicas. State-based CRDT pursues a partial order $\leq_v$ (e.g., integer order) with *least upper bound* (LUB) $\sqcup_v$ (e.g., maximum or minimum operation between integer numbers) that guarantees *associative* (i.e., $(a_1 \sqcup_v a_2) \sqcup_v a_3 = a_1 \sqcup_v (a_2 \sqcup_v a_3)$), *commutative* (i.e., $a_1 \sqcup_v a_2 = a_2 \sqcup_v a_1$), and *idempotent* (i.e., $a_1 \sqcup_v a_1 = a_1$) properties, for each value of object a_1, a_2, a_3 (e.g., integer numbers). Such CRDT is called *Convergent Replicated Data Type* (CvRDT) and is used in Dynamo and Riak. CvRDT can tolerate out-of order, repeatable, and lost messages as long as replicas reach the same value. Thus, CvRDT achieves eventual consistency without any coordination across replicas, but it comes at the expense of monetary cost and communication bottleneck for transferring large objects particularly across data stores. Almeida et al. [8] addressed this issue by propagating the effect of recent update operations on replicas instead of the whole state; meanwhile all properties of CvRDT are maintained. As an example of CvRDT, consider Grow-only set (G-set) that supports only *union* operations. Assume a partial order $\leq_v$ on two replicas of G-set S_1 and S_2 is defined as $S_1 \leq_v S_2 \iff S_1 \subseteq S_2$ and *union* is performed as $S_1 \cup S_2$. Since the *union* operation preserves the mentioned three properties, G-set is a CvRDT.

In *operation-based* CRDT, first the update is applied to the local replica, and then is asynchronously propagated to the other replicas. Operation-based CRDT demands a reliable communication network to submit all updates to every replica in a delivery order $\leq_v$ (specified by data type) with *commutative* property [149], as utilized in Cassandra. If all concurrent operations are commutative, then any order of operations execution converges to an identical value. Such data type is called *Commutative Replicated Data Type* (CmRDT) and is more useful than CvRDT in terms of data transferring for applications that span write-intensive replicas across data stores. This is because CmRDT demands less bandwidth to transfer *operation* across replicas, as compared to CvRDT that transfers the effect of operation. For instance, G-set is also CmRDT because *union* is commutative. Similar to CvRDT, CmRDT allows the execution of updates anywhere, anytime, and any order, but they have a key shortcoming in guaranteeing integrity constraints and invariants across replicas.

2.5.5 Causal and Causal+ consistency

We first introduce a formal definition of causal and causal+ consistency models (Section 2.5.5), followed by a description of the source and type of dependencies found in this model (Section 2.5.5).

Causal consistency definition

Causal consistency maintains the merits of eventual consistency, while respecting to the causality order among requests applied to replicas. It is stronger and more expensive than eventual consistency due to tracking and checking dependencies. It defines Lamport's *"happens-before"* relation [72] between operations o_1 and o_2 as $o_1 \leadsto o_2$. Potential causality $o_1 \leadsto o_2$ maintains the following rules [4]. *Execution thread*: If o_1 and o_2 are two operations in the same thread of execution, then $o_1 \leadsto o_2$ if o_1 happens before o_2. *Read from*: If o_1 is a write, and o_2 is a read and returns the value written by o_1, then $o_1 \leadsto o_2$. *Transitivity*: if $o_1 \leadsto o_2$ and $o_2 \leadsto o_3$, then $o_1 \leadsto o_3$. Causal consistency does not support concurrent operations (i.e., $a \not\leadsto b$ and $b \not\leadsto a$). According to this definition, the write operation happens if all write operations having causal dependency with the given write have occurred before. In other words, if $o_1 \leadsto o_2$, then o_1 must be written before o_2. Causal+ is the combination of *causal* and *convergent conflict resolution* to ensure *liveness* property. This consistency model allows users locally receive the response of read operations without accessing remote data store, meanwhile the application semantics are preserved due to enforcing causality on operations. However, it degrades scalability across data stores for write operations because each DC should check whether the dependencies of these operations have been satisfied or not before their local commitment. This introduces a trade-off between *throughput* and *visibility*. *Visibility* is the amount of time that a DC should wait for checking the required dependencies among the write operations in the remote DC, and can be influenced by *network latency* and *DC capacity for checking dependencies*.

Dependency source and type

Dependencies between operations are represented by a graph in which each vertex represents an operation on variables and each edge shows the causality of a dependency between two operations. The source of dependencies can be *internal* or *external* [64]. The former refers to causal dependencies between each update and previous updates in the same session, while the latter relates to causal dependency between each update and updates created by other sessions whose values are read in the same session. COPS [110], Eiger [111], and Orbe [63] track both dependency sources. Dependency types can be either *potential* or *explicit* for an operation (as in Eiger and ChainReaction [9]) or for a value (as in COPS and Orbe). Potential dependencies capture all possible influences between data dependencies, while explicit dependencies represent the semantic causality of the application level between operations. The implementation of potential dependencies in modern applications (e.g., social networks) can produce large metadata in size and impede scalability due to generating large dependencies graph in the degree and depth. The deployment of explicit dependencies, as used in Indigo [19], alleviates these drawbacks to some extent, but it is an ad-hoc approach and cannot achieve the desired scalability in some cases (e.g., in social applications). This deployment is made more effective with the help of garbage collection, as used in COPS and Eiger, in which the committed dependencies are eliminated and only the nearest dependencies for each operation are maintained.

2.5.6 Ordering, Strong, and Adaptive-level consistency

We first define ordering consistency model and how it is provided. We then discuss projects that enable application providers to switch between a range of consistency models based on their requirements. As discussed earlier, eventual consistency applies the updates in different orders at different replicas and causal consistency enforces partial ordering across replicas. In contrast, *ordering consistency*–also called *sequential consistency*–provides a global ordering of the updates submitted to replicas by using a logical clock to guarantee monotonic reads and writes. In fact, *ordering consistency* mandates a read operation from a location to return the value of the last write operation to that location. For example, PNUTS provides ordering consistency per key by deploying a master replica which is responsible for ordering writes to an object and then propagating the updates to slave replicas. Another way to provide ordering consistency is deploying *chain replication* [171].

Strong consistency guarantees that all read and write operations receive a global timestamp using a synchronized clock within and across data stores. This ensures that every read operation on the data d returns the value corresponding to the most recent write request on the data d.

Adaptive-level consistency switches between weak and strong consistency models based on the requirements of application to reduce response time and monetary cost. This is because strong consistency is expensive in terms of monetary cost and performance, while weak consistency places a high burden on the developers in respect to the maintenance of application semantics.

2.6 Data Management Cost

This section first presents a background on the pricing plans and different storage services offered by the well-known cloud providers- AWS, Microsoft Azure, and Google. Then it discusses the optimization of data management cost with satisfaction of a single-QoS/multi-QoS metric and the cost trade-offs.

2.6.1 Pricing plans

Price is a new and important feature of data stores as compared to traditional distributed systems like cluster and grid computing. Data stores offer a variety of pricing plans for different storage services with a set of performance metrics. Pricing plans offered by data stores are typically divided in two categories [123]: *bundling price* (also called *quantity discount*) and *block rate pricing*. The first is observed in most data stores (e.g., Google Drive) and is recognized as a non-liner pricing, where unit price changes with quantity to follow *fixed cost* and *per-unit charge*. The second category divides the range of consumption into sub-ranges and in each sub-range unit price is constant as observed in Amazon. This category is a special form of the *multi-part tariffs* scheme in which the fixed price is zero. Note that the standard form of multi-part tariffs consists of a fixed cost plus multi-ranges of costs with constant cost in each range. One common form of this scheme is *two-part tariffs* that are utilized in data stores with a fixed fee for a long term (currently 1 or 3 years) plus a per-unit charge. This model is known a *reserved pricing model* (e.g., as offered by Amazon RDS and Dynamo) as opposed to an *on-demand pricing model* in which there is no fixed fee and its per-unit charge is more than that in the reserved pricing model. All pricing plans offered by the well-known cloud providers follow *concavity property* that implies as the more resources the application providers buy the cheaper the unit price is. The unit price for storage, network, and VM respectively are often GB/month, GB, and instance per unit time.

A cloud provider offers different services with the same functionality while performance is directly proportional to price. For example, Amazon offers S3 and RRS as online storage services but RRS compromises redundancy for lower cost. Moreover, the price of same resources across cloud providers is different. Thus, given these differences, many cost-based decisions can be made. These decisions will become complicated especially for applications with time-varying workloads and different QoS requirements such as availability, durability, response time, and consistency level. To do so, a joint optimization problem of resources cost and the required QoS should be characterized. Resources cost consists of: (i) *storage cost* calculated based on the duration and size of storage the application provider uses, (ii) *network cost* computed according to the size of data the application provider transfers out (reads) and in (writes) to data stores (typically data transfer into data stores is free), and (iii) *computing cost* calculated according to duration of renting a VM by application providers. In the rest of section, we discuss the cost optimization of data management based on a single QoS or multi-QoS metrics, and cost trade-offs.

2.6.2 Overview of storage classes

Well-known cloud providers such as AWS, Azure, and Google offer resources as Infrastructure as a Service (IaaS), where Storage as a Service (StaaS) is one of its main components. StaaS supports several classes of storage, which are differentiated in price and performance metrics.[17] These metrics are (i) durability, (ii) availability SLA, (iii) minimum object size: an object smaller than s kilobytes in size is charged for s kilobytes of storage, (iv) minimum storage duration: an object deleted less than d days after storing in storage incurs a minimum $d-$day charge, (v) retrieval first byte latency: the time taken to retrieve the first byte of an object. The storage classes supported by AWS, Azure, and Google are as follows.

AWS provides four classes of storage[18]: (i) Simple Storage Service (S3) is a highly reliable, available, and secure storage service for data frequently accessed; (ii) Reduced Redundancy Storage (RRS) offers users storage resources with lower cost at the expense of lower levels of redundancy as compared to S3. RSS is suitable for data which require less durability as compared to those stored in S3; (iii) Standard-Infrequent Access (S-IA) is optimized for data accessed less frequently, but needs a low retrieval time (e.g., backup data); (iv) Glacier storage is the cheapest AWS storage which is suited to data with very low access rates and without the need for rapid access.

Microsoft Azure supports four classes of storage services,[19] which are mainly distinguished based on the number of replicas of an object that are stored in a single or multiple DCs. These classes are: (i) Locally Redundant Storage (LRS) stores 3 synchronous replicas within a single DC; (ii) Zone Redundant Storage (ZRS) stores 3 asynchronous replicas across multiple DCs within or across regions; (iii) Geographical Redundant Storage (GRS)

[17] Key features of storage classes. https://aws.amazon.com/s3/storage-classes/.

[18] AWS storage classes. https://aws.amazon.com/s3/storage-classes/

[19] Microsoft Azure storage classes. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/

is the same as LRS, in addition to storing 3 asynchronous replicas in a secondary DC that is far away from the primary DC; (iv) Read-access GRS (RA-GRS) is the same as GRS with the added functionality of allowing users to access data in the secondary DC. All classes of Azure storage support five types of storage: Blob, Table, Queue, File, and Disk. Each type of storage is used for a specific purpose. Blob storage is specialized for unstructured object data, Table storage for structured data, Queue storage for reliable messaging between different components of cloud services, File storage for sharing data across the components of an application, and Disk (premium) storage for supporting data-intensive workload running on Azure virtual machines (VMs). Besides these classes and types of storage, Azure also provides Blob storage with two access tiers. These are *hot* and *cold* access tiers which are supported in three classes of storage: LRS, GRS, and RA-GRS. Hot (resp. cold) access tier is used for data that are frequently (resp. rarely) accessed. Hot tier access is more expensive than the cold one, and this allows users to save cost when they switch between these access tiers based on a change in the usage pattern of the object. This switch incurs additional charges, and thus users are required to select each access tier in the appropriate time during the lifetime of the object.

Google supports five storage classes[20]: (i) Multi-regional storage is appropriate for frequently accessed data. This class is Geo-redundant storage service that maintains an object in at least two regions; (ii) Regional storage enables users to store data within a single region. This class is suitable for Google Compute Engine (GCE) instances; (iii) Durable Reduced Availability (DRA) has a lower availability SLA with the same cost (apart from the cost of operations) compared to Regional storage; (iv) Nearline storage is suitable for data that are accessed on average once a month or less. Thus, this class is a good choice for data backup, archival storage, and disaster recovery; (v) Coldline storage is the cheapest Google storage, and it is a suitable option for data accessed at most once a year.

Based on the offered storage classes, we classify them into five tiers. (i) Very hot tier provides the highest levels of redundancy across multiple regions and allows users to access the data in the secondary DC as the primary DC faces faults. (ii) Hot tier stores data in multiple regions, but the redundancy level is lower than the first tier. (iii) Warm tier is

[20] Google storage classes. https://cloud.google.com/storage/

the same as hot tier in redundancy level, but less durable. (iv) Cold tier provides lower availability and durability as compared to the first three tiers and imposes restriction on metrics like minimum object size and minimum storage duration. (v) Very cold tier has the same durability and availability levels compared to the cold tier, but it has more minimum storage duration. The last two tiers impose retrieval cost and they are more expensive than the first three tiers (i.e., very hot, hot, and warm) in operations cost. Table 2.13 summarizes the characteristics of the storage tiers offered storage services by AWS, Azure, and Google based on the discussed tiers.

Although these tiers are the same in functionality, their performance is directly proportional to price. For example, AWS offers S3 (belongs to hot tier) and RRS (belongs to warm tier) as online storage services, but RRS compromises redundancy for lower cost. Moreover, the price of storage resources across cloud providers is different. Thus, given these differences, many cost-based decisions can be made. These decisions will become complicated especially for applications with time-varying workloads and different QoS requirements such as availability, durability, response time, and consistency level. To make satisfactory decisions for application providers, a joint optimization problem of resources cost and the required QoS should be characterized. Resources cost consists of: (i) *storage cost* calculated based on the duration and size of storage the application provider uses, (ii) *network cost* computed according to the size of data the application provider transfers out (reads) and in (writes) to data stores (typically data transfer into data stores is free), and (iii) *computing cost* calculated according to duration of renting a VM by the application provider. The first two costs relate to the cost of data storage management. In the following section we discuss the optimization problems in regard to the cost of data storage management.

Table 2.13: The characteristics of different storage classes for the well-known Cloud providers: Amazon Web Service (AWS), Azure(AZ), and Google(GO)

Storage Class	Durability	Availability	**Minimum Object Size**	**Minimum Storage Duration**	**Retrieval Fee**	**First Byte Latency**	Applicability	**Examples of Cloud Providers**		
								AWS	Azure	Googlec
Very Hot	N/A	6 Replicas	N/A	N/A	N/A	milliseconds	Data frequently accessed	N/A	GRS, RA-GRS	N/A
Hot	11 nines	3-3.5 nines	N/A	N/A	N/A	milliseconds	Data frequently accessed	S3	ZRS	Regional, Multi-Regional
Warm	N/A	4 nines: AM 2 nines: GO	N/A	N/A	N/A	milliseconds	Data frequently accessed	RRS	LRS	DRA
Cold	11 nines	2 nines	128 KB: AM N/A: GO	30 days	Per GB	milliseconds	Data accessed at most once a month	Standard - IA	GRS (Cool tier)	Nearline
Very Cold	11 nines	N/A: AM 2 nines: GO	N/A	90 days	Per GB	hours: AM milliseconds: GO	Data accessed at most once a year	AWS Glacier	LRS (Cool tier)	Coldline

† The abbreviation used in this table are:

GRS: Geographically Redundant Storage, RA-GRS: Read-Access Geographically Redundant Storage

ZRS: Zone Redundant Storage, LRS: Locally Redundant Storage

S3: Simple Storage Service, RRS: Reduced Redundancy Storage

Standard-IA: Standard- Infrequent Access, DRA: Durable Reduced Availability

2.6.3 Cost optimization based on a single QoS metric

Application providers are interested into the selection of data stores based on a single QoS metric so that the cost is optimized or does not go beyond the budget constraint. This is referred to as a cost optimization problem based on a single QoS metric and is discussed as below.

(1) *Cost-based availability and durability optimization*. Availability and durability are measured in the number of nines and achieved by means of usually triplicate replication in data stores. Chang et al. [38] proposed an algorithm to replicate objects across data stores so that users obtain the specified availability subject to budget constraint. Mansouri et al. [116] proposed two dynamic algorithms to select data stores for replicating non-partitioned and partitioned objects, respectively, with the given availability and budget. In respect to durability, PRCR [99] uses the duplicate scheme to reduce replication cost while achieving the same durability as in triplicate replication. CIR [102] also dynamically increases the number of replicas based on the demanding reliability with the aim of saving storage cost.

(2) *Cost-based consistency optimization*. While most of the studies explored consistency-performance trade-off (2.5), several other studies focused on lowering cost with adaptive consistency model instead of a particular consistency model. The *consistency rationing* approach [91] divides data into three categories with different consistency levels assigned and dynamically switches between them in run time to reduce the resource and the penalty cost paid for the inconsistency of data which is measured based on the percentages of incorrect operations due to using lower consistency level. Bismar [46] defines the consistency level on operations rather than data to reduce the cost of the required resources at run time. It demonstrates the direct proportion between the consistency level and cost. C3 [67] dynamically adjusts the level of consistency for each transaction so that the cost of consistency and inconsistency is minimized.

(3) *Cost-based latency optimization*. User-perceived latency is defined as (i) a constant in the unit of RTT, distance, and network hops, or (ii) latency cost that is jointly optimized with monetary cost of other resources. Latency cost allows the latency metric to be changed from a discrete value to continuous one, thereby achieving an accurate QoS in terms of latency constraint and easily making a trade-off between latency and other

monetary costs. OLTP (resp. OLAP) applications satisfy the latency constraint by optimization of data placement (resp. data and task placement). This placement has an essential effect on optimizing latency cost or satisfying it as a constraint as well as in reducing resources cost.

2.6.4 Cost optimization based on multi-QoS metric

Application providers employ Geo-replicated data stores to reduce the cost spent on storage, network, and computing under multi-QoS metrics. In addition, they may incur *data migration cost* as a function of the data size transferring out from data store and its corresponding network cost. Data migration happens due to application requirements, the change of data store parameters (e.g., price), and data access patterns. The last factor is the main trigger for data migration as data transits from *hot-spot* to *cold-spot* status (defined in 2.2) or the location of users changes as studied in "Nomad" [169]. In Nomad, the changes in users' location are recognized based on simple policies that monitor the location of users when they access an object. Depending on the requirements of the applications, cost elements and QoS metrics are determined and integrated in the classical cost optimization problems as linear/dynamic programming [53], k-center/k-median [78], ski-rental [146], etc. In the following, these features and requirements are discussed.

(1) For a file system deployment, a key decision is to store a data item either in cache or storage at an appropriate time while guaranteeing access latency. Puttaswamy et al. [131] leveraged EBS and S3 to optimize the cost of file system and they abstracted the cost optimization via a ski-rental problem. (2) For data-intensive applications spanning across DCs, the key decision is which data stores should be selected so that the incurred cost is optimized while QoS metrics are met. The QoS metrics for each data-intensive application can be different; for example, online social applications suffice causal consistency, while a collaborative document editing web service demands strong consistency. (3) For online social network (OSN), the key factor is replica placement and reads/writes redirection, while "social locality" (i.e., co-locating the user's data and her friends' data) making reduction in access latency is guaranteed. In OSN, different policies to optimize cost are leveraged: (i) minimizing the number of slave replicas while guaranteeing social locality for each user [130], (ii) maximizing the number of users whose locality can

be preserved with a given number of replicas for each user [168], (iii) graph partitioning based on the relations between users in OSN (e.g., cosplay [82]), and (iv) selective replication of data across DCs to reduce the cost of reads and writes [107]. (4) The emergence of content cloud platforms (e.g., AWS Cloud-Front[21] and Azure CDN[22]) help to build a cost-effective cloud-based content delivery network (CDN). In CDN, the main factors contributing into the cost optimization are replicas placement and reads/writes redirection to appropriate replicas [40].

2.6.5 Cost trade-offs

Due to several storage classes with different prices and various characteristics of workloads, application providers are facing with several cost trade-offs as below.

Storage-computation trade-off. This is important in scientific application workloads in which there is a need for the decision on either storing data or recomputing data based on the size and access patterns. Similar decision happens to the privacy preservation context that requires a trade-off between encryption and decryption of data (i.e., computation cost) and storing data [196]. The trade-off can be also seen in video-on-demand service in which video transcoding[23] is a computation-intensive operation, and storing a video with a variety of formats is storage-intensive. Incoming workload on the video and the performance requirement of users determine whether the video is transcoded on-demand or stored with different formats.

Storage-cache trade-off. Cloud providers offer different tiers of storage with different prices and performance metrics. A tier, like S3, provides low storage cost but charges more for I/O, and another tier, like EBS and Azure drive, provides storage at higher cost but I/O at lower cost [47]. Thus, as an example, if a file system frequently issues reads and writes for an object, it is cost-efficient to save the object in EBS as a cache, or in S3 otherwise. This trade-off can be exploited by data-intensive applications in which the generated intermediate/pre-computed data can be stored in caches such as EBS or memory attached to VM instances.

Storage-network trade-off. Due to significant *differences in storage and network costs* across

[21] AWS CloudFront. `https://aws.amazon.com/cloudfront/`.

[22] Azure CDN. `https://azure.microsoft.com/en-us/services/cdn/`.

[23] Video transacoding is the process of converting a compressed digital video format to another.

Table 2.14: The Scope of Thesis.

Characteristics	Thesis scope
Data model	Key-value
Application	Online Social Networks (OSNs)
System resources	Storage and network
Target system	The well-known Cloud Storage Providers (CSPs)
Goal	Cost optimization of data storage management
Workload	Twitter and Facebook

data stores and *time-varying workload* on an object during its lifetime, acquiring the cheapest network and storage resources at the appropriate time of the object lifetime plays a vital role in the cost optimization. Simply placing objects in a data store with either the cheapest network or storage for their whole lifetime can be inefficient. Thus, storage-network trade-off requires a strategy to determine the placement of objects during their lifetime, as studied in a dual cloud-based storage architecture [115]. This trade-off also comes as a matter in the recovery cost in erasure coding context, where *regenerating* and *non-MDS* codes are designed for this purpose.

Reserved-on demand storage trade-off. Amazon RDS and Dynamo data stores offer *on-demand* and *reserved* database (DB) instances and confront the application providers with the fact that how to combine these two types of instances so that the cost is minimized. Although this trade-off received attention in the context of computing resources [174], it is worthwhile to investigate the trade-off in regard to data-intensive applications since (i) the workload of these two is different in characteristics, and (ii) the combination of on-demand DB instances and different classes of reserved DB instances with various reservation periods can be more cost-effective.

2.7 Thesis Scope and Positioning

This section first describes the scope of this thesis in terms of data model, application type, system resources, target system, goal, and the used workload. Then the relevant studies focusing on the cost optimization of data management are discussed in detail.

2.7.1 Thesis Scope

This thesis investigates the cost optimization of data storage management across different CSPs so that the QoS specified by users/application providers is met. The main incentive for this investigation is that (i) CSPs offer several storage classes with different prices for different purposes, and (ii) applications generate time-varying workloads.

This work focuses on the well-known CSPs –AWS, Azure, and Google– which provide storage classes as summarized in Table 2.13. It considers these classes for key-value applications that generate time-varying workloads in the form of long-tail on the objects [21]. An instance of these applications is Online Social Network (OSN) [122] in which an object often receives many read and write requests during its initial lifetime, and then these requests reduce as time passes . In other words, the object is in a hot-spot status as frequently accessed and then it transits to a cold-spot status as rarely accessed.

To optimize the cost of data management for such applications, it is important to exploit the price differences of storage resources across CSPs facilitated different storage classes. According to the defined two statuses for objects, two storage classes are used: one for hot-spot status and one for cold-spot status. Hence, the main problem is to make a two-fold decision: which storage class of a CSP and in what time during the lifetime of object should be selected to optimize monetary cost while the required QoS is guaranteed.

In addition to the cost optimization and the specified QoS by users, data consistency is another factor that is important for users. Since this work focuses on OSN applications, eventual consistency is sufficient for users. To provide the strong consistency model for applications, we consider a single-master multi-slave data management environment which supplies users with the most updated data as their requests are submitted to the master replica. In summary, the scope of this thesis is summarized in Table 2.14.

2.7.2 Thesis Positioning

This section discusses some relevant studies that are close to the research direction in this thesis. These studies are different in one or more aspects as outlined below.

In Chapter 3, we first propose algorithms which trade-off availability against cost in distributing data across CPSs. Our algorithms help the user to select DCs with the minimal cost, honoring constraints such as data availability. Additionally, we aim to maximize the availability of the striped data to the extent the users budget allows. The closest work to this contribution was conducted by Chang et al. [38]. They proposed an algorithm to replicate objects across data stores so that users obtain the specified availability subject to budget constraint. Also a few studies improve availability within cloud-based data stores through data replication. Ford et al. [52] investigated object availability in Google's storage infrastructure, and analyzed availability of components, e.g., machines, racks and multiple racks in tens of Google storage clusters. In their analytical measurements, they predict object availability based on Markov chain modelling in a DC. Cidon et al. [50] proposed Copysets in which data is replicated across storage nodes within a data store in an efficient way instead of using random replica placement widely employed by current data stores. In contrast, our work focuses to improve availability across DCs.

In chapter 4, we propose a dual cloud-based architecture to save cost for a time-varying workload. This architecture mimics a hierarchical storage management (HSM)[24] when data automatically are moved between low- and high-cost storage media. FCFS [131] deployed a generalized form of HSM to reduce the cost of operating a file system in a single cloud storage. Unlike our work, they neither require to consider migration cost nor need to deal with the latency across DCs. Our approach is different from their proposed solutions as we consider the object migration cost between DCs.

In Chapters 5 and 6, we extend this architecture for Geo-replicated data stores to optimize the cost of data management across CSPs. To accurately determine the position of the work in Chapters 5 and 6, we are required to investigate existing work in the following main categories.

Using multiple cloud services. Clearly, reliance on a single cloud provider results in three-folds obstacles of: *availability of services*, *data lock-in*, and *non-economical use* [167]. To alleviate these obstacles, one might use multiple cloud providers that offer computing, persistent storage and network services with different features such as price and performance [98]. Being inspired by these various features, automatic selection of cloud providers based on their capabilities and user's specified requirements are proposed to

[24] Hierarchical storage management (HSM). https://en.wikipedia.org/wiki/Hierarchical_storage_management

determine which cloud providers are suitable in the trade-offs such as cost vs. latency and cost vs. performance [138].

Several previous studies attempted to effectively leverage multiple CSPs to store data across them. RACS [3] utilized *erasure coding* to minimize migration cost if either economic failure, outages, or CSP switching happens. Hadji [74] proposed various replica placement algorithms to enhance availability and scalability for encrypted data chunks while optimizing the storage and communication cost. SPANStore [177] optimized cost by using pricing differences among CSPs while the required latency for the application is guaranteed. It used a storage class across CSPs for all objects without respect to their read/write requests, and consequently it did not require to migrate object between storage classes. SPANStore also leveraged algorithms relying on workload prediction. Cosplay [82] optimized the cost of data management across DCs– belonging to a single cloud– through swapping the roles (i.e., master and slave) of data replicas owned by users in the online social network. None of these systems explore minimizing cost by exploiting pricing differences across different cloud providers with several storage classes when dynamic migration of objects across CSPs is a choice.

Online algorithms and cost trade-offs. A number of online algorithms have been studied to figure out different issues such as dynamic provisioning in DCs [113], energy-aware dynamic server provisioning [105] and load balancing among Geo-distributed DCs [106]. All these online algorithms are derived from ski-rental framework in order to determine when a server must be turned off/on to reduce energy consumption, while we focus on data management cost optimization which comes with different contributing factors such as data size and read/write rates. In FCFS [131], the same framework is used to optimize data management cost in a single data store that offers cache and storage with different prices. Different from this work, we utilize price differences among CSPs in a Geo-replicated system, and therefore its deployment strategy must account for inter-DCs latencies, data migration among DCs, and writing strategy with the minimum cost in the eventual (considered in Chapter 5) or strong–if data is read from master (home) DC– (considered in Chapter 6) consistency setting.

More importantly, the ski-rental deployed in the above studies is not applicable in our model because it makes a decision on *time* (e.g., when a server is turned off/on or when data are moved from storage to cache), while we need to make a two-fold decision (*time and place*) to determine *when* data should be migrated and to *which* storage class(es) owned by a CSP. To make this decision, we propose a deterministic online algorithm (Algorithm 5.2) that uses Integer Linear Programming (ILP) to optimize cost. Also, a randomized online algorithm (Algorithm 5.3) is proposed based on Fixed Receding Horizon Control (FRHC) [105, 194] technique to conduct dynamic migration of objects. In [194], the authors proposed online and offline algorithms to optimize the routing and placement of big data into the cloud for a MapReduce-like processing, so that the cost of processing, storage, bandwidth, and delay is minimized. They also considered migration cost of data based on required historical data that should be processed together with new data generated by a global astronomical telescope application. Instead, our work focuses on optimizing replicas placement of objects being transited from hot-spot to cold-spot. Our optimization problem takes different settings as compared to [194]. These settings are (i) replicas number, (ii) latency Service Level Objectives (SLO) for reads and writes, and (iii) variable workload (reads and writes) on different objects. These objects demand a dynamic decision on when replicas are migrated between two DCs, when they are moved between two classes of storage in a DC, or both. These differences in settings make our optimization problem different in the cost model (storage, read, write, and migration) and problem definition as well.

Some literature focused on trade-offs between different resources cost. The first is compute vs. storage trade-off that determines when data should be stored or recomputed, and can be applicable in video-on-demand services. Kathpal et al. [86] determined when a transcoding on-the-fly solution can be cost-effective by using ski-rental framework. They focused neither on Geo-replicated systems nor theoretical analysis on the performance in terms of competitive ratio (CR) computed for Algorithms in Chapter 5. The second trade-off is cache vs. storage as deployed in MetaStorage [23] that made a balance between latency and consistency. This study has a different goal, and furthermore it did not propose a solution for the cases in which the workload is unknown. FCFS [131] also made this trade-off as already discussed. The third trade-off can be bandwidth vs. cache as somehow simulated in DeepDive [125] that efficiently and quickly identifies which VM should be migrated to which server as workload changes.

This study is different in objective, contributing parameters, and even in the scope.

Computation and data migration. Virtualization partitions the resources of a single compute server into multiple isolated environments which are called *virtual machines (VMs)*. A VM can be migrated from one host to another in order to provide fault tolerance, load balancing, system scalability, and energy saving. VM migration can be either *live* or *non-live*. The former migration approach ensures almost zero downtime for service provisioning to the hosted applications during migration, whereas the latter one suspends the execution of applications before transferring a memory image to the destination host. There has been a spate of work on VM migration approaches. Interested readers are referred to survey papers [5, 119] for detailed discussion on VM migration techniques.

Similarly, data migration is classified into two approaches. The first approach is *live data migration* [169]. This approach allows that while data migration is in progress, the data is accessible to users for reads and writes. Although live data migration approach minimizes performance degradation, it demands precise coordination when users perform read and write operations during the migration process. Recently, live data migration approaches have been exploited for transactional databases in the context of cloud [58, 65].

The second approach is *non-live data migration* [169]. This approach is classified into *stop and copy* and *log-based* migration techniques. In both techniques, while the data migration is in progress, the data is accessible to users for *reads*. But, these techniques differ in their capability to handle *writes*. In stop and copy technique, the writes are stopped during data migration, while in log-based technique the writes are served through a log which incurs a monetary cost. Thus, stop and copy and log-based migration techniques are respectively efficient in monetary cost and performance criteria. Non-live data migration approach is often used in non-transantional data stores that does not guarantee ACID properties, e.g., HBase[25] and ElasTraS [56].

There are several factors affecting data migration: the changes to cloud storage parameter (e.g., price), optimization requirements, and data access patters. In response to these changes and requirements, a few existing studies discuss data migration from pri-

[25] Apache HBase. https://hbase.apache.org/book.html

vate to public cloud [132], and some study object migration across public cloud providers [121, 179]. In [179], authors focused on predicting access rate to video objects and based on this observation, dynamically migrate video objects (read-only objects). In contrast, our study in Chapters 5 and 6 attempts to use pricing differences and dynamic migration to minimize cost with or without any knowledge of the future workload for objects with read and write requests. Write requests on an object raise cost of consistency as a mater. Mseddi et al. [121] designed a scheme to create/migrate replicas across data stores with the aim of avoiding network congestion, ensuring availability, and minimizing the time of data migration. While we designed several algorithms to minimize cost across data stores with different classes of storage.

Deploying cloud-based storage services in CDN. With the advent of cloud-based storage services, some literature has been devoted to utilize cloud storage in a Content Delivery Network (CDN) in order to improve performance and reduce monetary cost. Broberg et al. [30] proposed MetaCDN that exploits cloud storage to enhance throughput and response time while ignoring the cost optimization. Chen et al. [40] investigated the problem of placing replicas and distributing requests (issued by users) in order to optimize cost while meeting QoS requirement in CDN utilizing cloud storage. Papagianni et al. [127] went one step further by optimizing replica placement problem and requests redirection, while satisfying QoS for users and considering capacity constraints on disks and network. In [143], there is another model that minimizes monetary cost and QoS violation, while guaranteeing SLA in a cloud-based CDN. In contrast to these works which proposed greedy algorithms for read-only workload, our work in Chapter 5 exploits the pricing differences among CSPs for time varying writable workload and proposes offline and online algorithms with a theoretical analysis on the CR. These algorithms determine the location of the object with a limited and fixed replicas. This makes them inappropriate for the object that demand a variable and high number of replicas during its lifetime. The reason behind of this demand is that the object receives time-varying workloads from a large set of DCs across the globe. Unlike algorithms in Chapter 5, we propose a lightweight heuristic algorithm in Chapter 6 to dynamically determine the number of replicas for any object based on its variable workload receiving from different DCs. These algorithms also demand low time complexity, thereby making them suitable for applications which host a large number of objects.

Table 2.15 summarizes the discussed studies in this section and Section 2.6.3. This table describes each work in four aspects: (i) Project specification (top to down: project name/authors, data store platform, data application type), (ii) cost elements, (iii) QoS metrics/constraints, (iv) Features (in order: solution techniques, cost optimization type, and key feature).

Table 2.15: Summary of Projects with Monetary Cost Optimization

Project specification	**Cost elements**					**QoS metrics/constraints**					**Features**
	Storage	Read	Write	VM	Migration	Latency	Bandwidth	Availability	durability	Consistency level	
Chang et al. [38] MDC Data-intensive	✓							✓			(i) Dynamic programming. (ii) Cost-based availability optimization. (iii) Maximizing availability under the budget constraint for non-partitioned objects.
PRCR [99] SDC Scientific workflows	✓								✓		(i) Proactive replica checking. (ii) Cost-based durability optimization. (iii) Reducing $\frac{1}{3}$ to $\frac{2}{3}$ storage cost as compared triplicate storage cost.
CIR [102] SDC Scientific workflows	✓								✓		(i) an incremental replication approach. (ii) Cost-based durability optimization. (iii) Incurring cost the same as that for triplicate replicas in long-term storage deployment.
Copysets [50] SDC Data-intensive	✓								✓		(i) Near optimal solution. (ii) Cost-based durability optimization. (iii) Achieving higher durability than widely used random replication.
Consistency rationing [91] SDC OLTP	✓									✓	(i) A general policy based on a conflict probability (ii) Cost-based consistency optimization. (iii) Providing *serializability*, *session*, and *adaptive consistency* between these two consistency models.
C3 [67] SDC Data-intensive	✓	✓	✓	✓						✓	(i) A selective solution based on defined rules. (ii) Cost-based consistency optimization. (iii) Providing *serializability*, *snapshot isolation*, and *eventual consistency*.

Table 2.4: Summary of Projects with Monetary Cost Optimization

Project specification	**Cost elements**					**QoS metrics/constraints**					**Features**
	Storage	Read	Write	VM	Migration	Latency	Bandwidth	Availability	durability	Consistency level	
Shankaranarayanan et al. [148] MDC OSN	✓	✓	✓			✓				✓	(i) Linear programming (ii) Cost-based latency optimization. (iii) Minimizing latency in quorum-based data stores and optimizing monetary cost under the latency constraint.
CosTLO [179] MDC Data-intensive	✓	✓	✓	✓		✓				✓	(i) A comprehensive measurement study on S3 and Azure. (ii) Cost-based latency optimization. (iii) Reduction in latency variation via augmenting read/write requests with a 15% increase in cost.
FCFS [131] SDC File system	✓	✓	✓								(i) Ski-rental problem. (ii) Storage-cache trade-off. (iii) Deploying EBS and S3 to reduce monetary cost for a file system.
Khanafer et al. [88] SDC File system	✓	✓	✓								(i) A variant of ski-rental problem. (ii) Storage-cache trade-off. (iii) Reducing the deploying cost of file system across S3 and EBS as assumed the average and variance of arrival time of reads/writes are known.
Jiao et al. [81] MDC OSN	✓	✓	✓			✓					(i) Using *graph cut* techniques to determine master and slave replicas. (ii) Cost optimization based on single QoS metric. (iii) Optimizing the mentioned costs plus carbon footprint cost.
Cosplay [82] MDC OSN	✓		✓			✓		✓		Eventual	(i) A selective solution (greedy). (ii) Cost optimization based on multi-QoS metric. (iii) Achieving cost reduction via role-swaps between master and slave replicas in a greedy approach.
Hu et al. [76] MDC CDN	✓	✓	✓			✓					(i) Lynapunov optimization technique. (ii) Cost optimization based on single QoS metric. (iii) Seeking the optimal solution without the future knowledge in regard to workload.
Chen et al. [40] MDC CDN	✓	✓	✓			✓					(i) A selective solution (greedy). (ii) Cost optimization based on single QoS metric. (iii) Achieving cost reduction via greedy algorithms in the case of offline and online (Dynamic and Static) replication.

Table 2.4: Summary of Projects with Monetary Cost Optimization

Project specification	Cost elements					QoS metrics/constraints					Features
	Storage	Read	Write	VM	Migration	Latency	Bandwidth	Availability	durability	Consistency level	
Papagianni et al. [127] MDC CDN	✓	✓	✓	✓		✓	✓				(i) Linear programming and graph partitioning heuristic algorithms. (ii) Cost optimization based on multi-QoS metric. (iii) Reducing cost by considering the optimized replica placement and distribution path construction.
COMIC [187] MDC CDN	✓	✓									(i) Mixed integer linear programming. (ii) Cost optimization based on Multi-QoS metric. (iii) Optimizing electricity cost for DCs and usage CDN cost (read cost) under the processing capacity of DCs and CDN as constraints.
Wu et al. [175] MDC social media streaming	✓	✓		✓	✓	✓	✓				(i) An integer programming and an online algorithm based on prediction. (ii) Cost optimization based on multi-QoS metric. (iii) Reducing cost per video as workload changes across DCs.
Qiu et al. [132] MDC (public and private) social media streaming	✓	✓		✓	✓	✓	✓				(i) Lynapunov optimization technique. (ii) Cost optimization based on multi-QoS metric. (iii) Reducing cost (migration cost is only between private and public DC) for video files while ensuring bandwidth constraint for the private DC.
Ruiz-Alvarez and Humphrey [138] MDC (Private and AWS) Data-intensive	✓	✓	✓	✓		✓	✓				(i) Integer linear programming. (ii) Cost optimization based on multi-QoS metric. (iii) Determining either private or AWS cloud to run application based on the budget and job turnaround time constraints.
SPANStore [177] MDC Data-intensive	✓	✓	✓	✓		✓		✓		Eventual Strong	(i) Linear programming. (ii) Cost optimization based on multi-QoS metric. (iii) VM cost optimization only for writes propagation.
Chiu and Agrawal [47] SDC Data-intensive	✓	✓	✓	✓							(i) A comprehensive scenario conducted on S3, EBS and EC2 instances. (ii) Storage-cache trade-off. (iii) A reduction cost via using S3 vs. EBS and cache owning by different EC2 instance.

Table 2.4: Summary of Projects with Monetary Cost Optimization

Project specification	**Cost elements**					**QoS metrics/constraints**					**Features**
	Storage	Read	Write	VM	Migration	Latency	Bandwidth	Availability	durability	Consistency level	
Yuan et al. [190,191] SDC Scientific workflows	✓			✓							(i) Using intermediate data dependency graph or Cost Transitive Tournament Shortest Path (CTT-SP)-based to decide whether to store data or recompute later in run time. (ii) Storage-computation trade-off. (iii) Significant reduction in cost by using AWS cost model.
Jokhio et al. [84] SDC Video Transcoding	✓			✓							(i) A selective solution utilizing estimation of storage and computation costs and the popularity of transcoded video. (ii) Storage-computation trade-off. (iii) Significant reduction in cost by using AWS cost model.
Byholm et al. [32] SDC Video Transcoding	✓			✓							(i) Using utility-based model which determines when and for how long each transcoded video should be stored. (ii) Storage-computation trade-off. (iii) Decision of model is based on the storage duration t, the average of arrival request during t, and the popularity distribution of video.
Deelman et al. [60] SDC data-intensive	✓	✓	✓	✓							(i) Just measuring the incurred cost for data-intensive application. (ii) Storage-computation trade-off. (iii) storing data for long term is more cost-effective than recomputing it later (about next two years- for this specific application).
Triones [159] MDC data-intensive	✓	✓	✓		✓						(i) Non-linear programming and geometric space. (ii) Storage-computation trade-off. (iii) Improving fault-tolerance (2 times) and reducing access latency and vendor lock-in at the expense of more monetary cost.

2.8 Conclusions

This chapter discussed the key advantages and disadvantages of data-intensive application deployed within and across cloud-based data stores. In addition, the chapter investigated the key aspects of cloud-based data stores: data model, data dispersion, data consistency, and data cost optimization as the research direction of the thesis. The chapter finally discussed the related work in greater details to identify the gaps in regard to cost optimization of data management across cloud-based data stores. This helps us to gain a deeper understanding of research problems solved in the remaining chapters.

Chapter 3

QoS-aware Brokering Algorithms for Data Replication across Data Stores

Using multi-cloud broker is a plausible solution to remove single point of failure and to achieve very high availability. Since highly reliable cloud storage services impose enormous cost to the user, and also as the size of data objects in the cloud storage reaches magnitude of exabyte, optimal selection among a set of cloud storage providers is a crucial decision for users. To solve this problem, we propose an algorithm that determines the minimum replication cost of objects such that the expected availability for users is guaranteed. We also propose an algorithm to optimally select data centers (DCs) for striped objects such that the expected availability under a given budget is maximized. Simulation experiments are conducted to evaluate our algorithms, using failure probability and storage cost taken from real cloud storage providers.

3.1 Introduction

CLOUD storage is a novel paradigm for storing user objects[1] on a remote location in large scale. During recent years, some of the cloud storage companies, such as Amazon, Rackspace, Google, etc. have provided on-line mass storage to cloud users. Since every storage service belongs to a different company, they offer services in different performance Service Level Agreements (SLAs) and costs.

A typical performance SLA articulates precise levels of the services such as availability of the services which are in operation. In the context of intense economic competition, different cloud storage providers supply a variety of services with different SLAs, which

This chapter is derived from: item **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Brokering Algorithms for Optimizing the Availability and Cost of Cloud Storage Services," *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2013, IEEE CS Press, USA)*, Bristol, UK, Dec. 2-5, 2013.

[1] Data and object are used interchangeably in this chapter.

are proportional to the cost. That is, users interested in more reliable SLA must pay more. Moreover, as the total size of user objects in the cloud storage reach up to several exabyte (2^{60} bytes), it can impose enormous cost on users. Therefore, optimal selection of cloud storage providers in terms of higher availability and lower cost is a crucial decision to users.

Availability of service as an imperative criterion in the SLA is listed as one of the top ten obstacles to the growth of cloud computing [70]. Although the most well known cloud storage providers such as Amazon, Rackspace and Google, etc. warranty availability of services in high level, software bugs, user errors, administrator errors, malicious insiders, and natural catastrophes endangering availability are inevitable and unpredictable [90]. This is why some well-known cloud providers have experienced outages in their data centers (DCs) [70], and the number of vulnerability incidents has doubled from 2009 to 2011 [140]. Availability of services is defined as the ratio of the total time that the storage services of a cloud provider is accessible during a given interval (e.g., one year) to the length of the interval. The metric which we use for availability is number of nines [38]. For example, if the availability of the system is 99.9% then we refer it as three nines. The system with three nines availability is expected to have 8.75 hours downtime per year.

One simple way to get the desired availability is to replicate objects in multiple DCs. This approach is costly because as the number of replicas increases, the storage cost of the object raises. Therefore, minimizing cost with the aim of achieving desired availability as a required Quality of Service (QoS) is a key decision to user, which has not been studied very well.

Data Lock-in is another main problem among the top ten obstacles in regard to cloud computing. This is undesirable for users because they are vulnerable to rise in price, to decrease in availability and even to the cloud provider's bankruptcy [70]. Users also lose a chance to migrate from a cloud storage provider to another when new cloud providers emerge with better services or with lower price in the market. This is because some cloud storage providers charge the users for download service, which imposes heavy cost on users especially when one requires a large storage volume of a particular cloud storage provider. Moreover, transferring large objects from one cloud storage provider to another through the network is time consuming and most often is impossible.

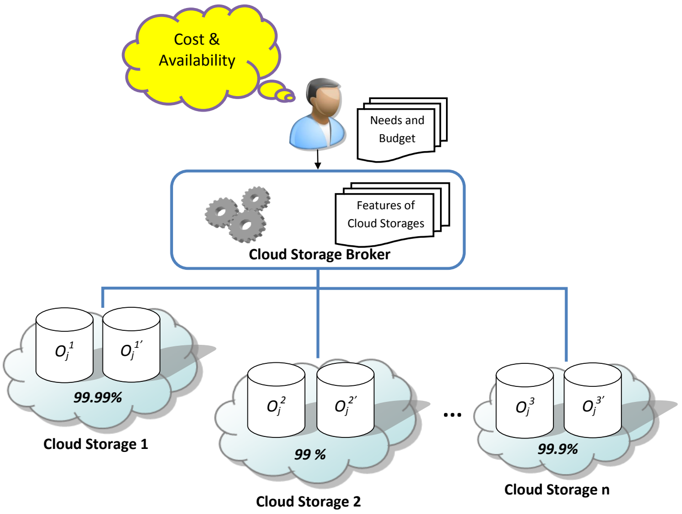


Figure 3.1: Cloud Storage Broker

One solution to mitigate the data lock-in and to allow users to migrate quickly from a cloud provider to another in reaction to any provider changes is placing an object at a fine granularity rather than a coarse one [3]. That is, the object, for example a database table or an archival object, is split to chunks, and stored in different cloud storages. Therefore, we need an algorithm to find the optimal placement of chunks according to the user's needs and budget. This algorithm can be used in a *cloud storage broker*, which provides transparent object access among several cloud storage providers [3]. The broker, as illustrated in Fig. 3.1, gathers all features of cloud storage providers and assists users in finding the right cloud storage based on their required QoS.

In this chapter, we propose algorithms that can be embedded in a cloud storage broker. These algorithms help the user to find a suitable placement of objects according to the required QoS. The first problem we focus on is to minimize the storage cost of objects while a given expected availability is met. The second problem is how to select the optimal placement of each strip of object such that the expected availability under a given budget is maximized. Due to the growing number of cloud storage providers with different characteristics and price, both problems are challenging and important.

The main **contributions** of this chapter are:

- a mathematical model for the DC selection problem in which the objective function, cost function and constraints are clearly defined,
- an algorithm to select a subset of a given DCs to minimize the storage cost for objects when the expected availability is given, and

- a dynamic algorithm to select cloud providers optimally for storing objects that are split into chunks and each chunk is replicated a fixed number of times, such that the expected availability under a given budget is maximized.

The rest of this chapter is organized as follows: Section 3.2 describes an algorithm employed to calculate the minimum storage cost for storing objects subject to a given expected availability. Section 3.3 presents the dynamic algorithm proposed to determine optimal DCs for storing striped objects to maximize expected availability with a given budget. Section 3.4 reports the experimental results of our simulation. Finally, conclusions and future works are stated in section 3.5.

3.2 Minimizing cost with given expected availability

We present an algorithm to find subsets of DCs to store replicas of all objects such that the replication cost of them is minimized, and expected availability of objects as the user's QoS is satisfied. Note that the subset of DCs for each object can be different. From the user perspective, one type of object may be popular, whilst another type may be seldom used (e.g., archival objects) and they can tolerate lower expected availability as QoS. So, it is reasonable to allow popular objects to be stored in more available DCs at higher cost, and guarantee lower QoS for non-popular objects at lower cost. Our algorithms guarantee this criterion for objects. That is, the expected availability of objects is met according to the priority attached to their type. In other words, as the priority of a type of objects increases, the objects of that type are replicated in DCs with more availability. In order to introduce our algorithm, we first present the following notations and definitions.

Consider a set of independent DCs represented by $D = \{d_1, d_2, ..., d_n\}$, where $d_i \in D$ $(1 \leq i \leq n)$ is an individual DC. Assume that a pair of characteristics for each DC: a weight $s(d_i)$ representing the *storage cost* of an object in the DC d_i, and $f(d_i)$ the *failure probability* of d_i. Also, suppose that the replica set denoted by δ is a subset of DCs in D such that each d_i hosts a replica of the object. Let M be the number of objects with equal size, and TN is the total number of nines for M objects. Assume that t is the average number of nines per object requested by a user as QoS. The value of TN either can be computed by $M \times t$ or can be a arbitrary value (that is the value of TN) which is determined by user. We define three types of QoS for objects, in descending order of priority: *Gold (G)*, *Silver (S)*, and *Bronze (B)*. A priority parameter P_Q is assigned to each QoS type Q such that the sum of priority parameters for three types of QoS is 1. That is, $\sum_Q P_Q = 1$, where $Q = \{G, S, B\}$ $(P_G \geq P_S \geq P_B)$. Note that the value of P_Q is used in a general sense in our algorithms. It can be interpreted as different priority measures such as the access probability of objects on average for a type of QoS, or the importance of objects to the user. Also, let all objects of QoS type Q be denoted by set J_Q, and EA_Q and EF_Q be the expected availability and expected failure of objects in J_Q, respectively.

Definition 3.1. *(Objects Placement): Let $\Phi = \bigcup_{j=1}^{M}\{\Phi_j\}$ be a placement of objects, where Φ_j indicates a subset of DCs (i.e, $\Phi_j \subseteq D$) that contains the jth object with r replicas. Therefore, for all Φ_j, $|\Phi_j| = r$ and $|\Phi| = M$.*

Definition 3.2. *(Replication Cost of Objects): Assume that the cost of object j is the sum of the replication cost of the object in a set of DCs Φ_j. Thus, the replication cost of Φ, referred to the cost of replicas for M objects, is defined as:*

$$C(\Phi) = \sum_{j=1}^{M} C(\Phi_j), \tag{3.1}$$

where $C(\Phi_j) = \sum_{d_l \in \Phi_j} s(d_l)$ is the cost of storing object j with duplication factor r.

Definition 3.3. *(Expected Availability of Objects): Let X_j be a discrete random variable with numerical values of $\{0, 1\}$ which shows whether the object j is available under the set Φ_j ($X_j = 1$) or not ($X_j = 0$). Thus, we have $E(X_j) = \sum_{x_j \in \{0,1\}} x_j P(X_j = x_j) = P(X_j = 1)$. Since X_j is referred to the availability of the jth object under Φ_j, the value of X_j is 0 only when all DCs in Φ_j fail. Otherwise, the value of X_j is 1 when at least one of DCs in set Φ_j containing object j is available. As a result, $E(\Phi_j) = E(X_j) = 1 - \prod_{d_i \in \Phi_j} f(d_i)$.*

The expected availability for M objects is equal to the sum of the expected availability of each object j because the expected availability of the sum of independent random variables is the sum of the expected availability of these random variables. Thus, we have:

$$E(\Phi) = \sum_{j=1}^{M} E(\Phi_j) = \sum_{j=1}^{M} (1 - \prod_{d_i \in \Phi_j} f(d_i)). \tag{3.2}$$

In this sense, $E(\Phi)$ can be viewed as the expected availability of M objects under $\Phi = \sum_{j=1}^{M}\{\Phi_j\}$, and expressed as a number of nines.

In the rest of this section, we formally define the object placement problem, and then a dynamic algorithm to tackle this problem is proposed. Based on the above definitions, the problem can be defined as follows. Assume that a set of DCs D, M objects with duplication factor r, the QoS requirement for user $E(\Phi)$, measured in the number of nines, and each QoS type associated with P_Q where $Q = \{G, S, B\}$ are given. The objective is to find a subset of DCs $\Phi_j \subseteq D$ for each object j so that $E(\Phi)$ is satisfied, the replication cost of objects $C(\Phi)$ is minimized, and the expected availability for each type of objects is proportional to the priority parameter of that type. This is translated to:

$$\min \;\; C(\Phi) = \sum_{j=1}^{M} C(\Phi_j), \tag{3.3}$$

$$s.t. \;\; E(\Phi) = \sum_{j=1}^{M}(1 - \prod_{d_i \in \Phi_j} f(d_i)) \geq TN, \tag{3.4}$$

$$\forall J_Q, \;\; E(J_Q) \propto P_Q, \;\; Q = \{G, S, B\}, \tag{3.5}$$

where $E(J_Q)$ is the expected availability of objects which belong to QoS type Q.

In order to get a maximization problem, we mathematically redefine (3.3) as follows:

$$\max \;\; 1/\sum_{j=1}^{M} C(\Phi_j), \tag{3.6}$$

while the constraints are (3.4) and (3.5).

Before we propose our algorithm, let us express the feasibility of the problem. The sum of nines of all DCs $d_i \in \Phi$ must not be less than TN, if there exists a feasible solution for the above problem. That is, $E(\Phi) \geq TN$.

We propose a dynamic algorithm called *Minimum Cost Fixed Expected Availability* to solve the aforementioned objects placement problem. With no loss of generality, and assuming all objects have equal priority (that is, QoS type for objects is ignored), recursive equations of the dynamic algorithm can be obtained as explained below.

The first step of the dynamic algorithm is to define a recursive solution for two cases. *Case 1*: $j > 1$. This solution is calculated to $MC[j][tn]$, which means the minimum cost for the jth object[2] $(1 < j \leq M)$ with a given number of nines tn $(1 \leq tn \leq TN)$. To obtain the best placement for the jth object, we first find all combinations of r distinct DCs that can be chosen from D. Assume that each combination is denoted by δ $(|\delta| = r)$. Since the availability of jth object is $E(\delta)$, we have the expected availability $tn - E(\delta)$ for the $j-1$ objects. It means we should consider all possible cases with $tn - E(\delta)$ for the first $j-1$ objects, assuming δ is a set of DCs, which contains the jth object. Therefore, if we place the jth object in the set δ of DCs, then the minimum cost of replication j objects in DCs equals to the minimum cost of replication $j-1$ objects with the expected availability $tn - E(\delta)$ plus the cost of DCs in the set δ. This mathematically translated into:

$$MC[j][tn] = 1/(\max_{\delta}(1/MC[j-1][tn - E(\delta)]) + C(\delta)). \tag{3.7}$$

Case 2: $j = 1$. All the possible subsets $\delta \subseteq D$ so that $tn = E(\delta)$ are considered, and then the subset δ with the minimum cost is selected. In other words, $\Phi_1 = \delta$, where δ is a subset of DCs with the minimum cost of replication for the first object. Thus, the recursive function for $j = 1$ with fixed tn can be obtained as:

$$MC[1][tn] = \max_{\delta}(\frac{1}{C(\delta)}, MC[1][tn]). \tag{3.8}$$

The second step of the algorithm is the termination conditions. First, if $tn - E(\delta) < 0$, then $MC[j][tn] = 0$, which means the subset $\delta \subseteq D$ should not be considered. Second, if $j = 1$ and there is not a subset of DCs such that $tn = E(\delta)$, $MC[1][tn]$ also is assigned to zero. Third, the value of MC should be infinity when j and tn are both zero. According to the above discussion, the proposed algorithm is outlined in Algorithm 3.1.

In order to consider constraint (3.5), we slightly revise Algorithm 3.1. First, $E(J_Q)$ is computed by $\lfloor tn \times P_Q \rfloor$ where $1 \leq tn \leq TN$. To guarantee constraint (3.5) accurately, $E(J_Q)$ is sorted decreasingly by dif_Q where $0 \leq dif_Q = tn \times P_Q - E(J_Q) < 1$, and then the first $tn - \sum_Q E(J_Q)$ of $E(J_Q)$ is increased by one. Second, having calculated $E(J_Q)$ for all objects $j \in J_Q$, it is sufficient to replace TN with $E(J_Q)$ in Algorithm 3.1.

[2] In this section, we henceforth consider each object has r replicas, unless otherwise mentioned.

Algorithm 3.1: Minimum Cost Fixed Expected Availability

```
Input  : $D, M, TN, r, f(d_i), s(d_i)$
Output: $\frac{1}{MC[M][TN]}$
for $tn \leftarrow 1$ to $TN$ do
    $MC[0][tn] \leftarrow +\infty$
end
for $tn \leftarrow 1$ to $TN$ do
    $MC[j][0] \leftarrow 0$
    for $tn \leftarrow 1$ to $TN$ do
        $MC[j][tn] \leftarrow 0$
        forall combination $\delta \in (D, r)$ do
            if $((tn - E(\delta)) \geq 0)$ then
                if $(j = 1)$ and $(tn = E(\delta))$ then
                    $MC[1][tn] \leftarrow (\frac{1}{C(\delta)}, MC[1][tn])$
                end
                if $(j > 1)$ then
                    if $(MC[j-1][tn - E(\delta)] = 0)$ then
                        $MC[j][tn] \leftarrow 0$
                    else
                        $C \leftarrow 1/MC[j\text{-}1][tn\text{-}E(\delta)] + C(\delta)$
                        $MC[j][tn] \leftarrow \max(\frac{1}{C}, MC[j][tn])$
                    end
                end
            end
        end
    end
end
```

3.3 Maximum Expected Availability with Given Budget

One way to prevent object lock-in in the cloud storage is to store the object at a fine granularity rather than in coarse one [3]. Due to this advantage, in this section, our aim is to introduce a dynamic algorithm to provide the best placement for chunks of an object across the cloud providers so that the expected availability is maximized under a given budget. In the rest of the section, we define some preliminaries and definitions, and then we present the optimization problem in details.

In addition to notations in the previous section, we assume that each object is split to m chunks with the same size and replicated with duplication factor r. Since it is assumed that each replica of chunks of each object is placed in a separate DC, the number of DCs,

Algorithm 3.2: Maximum Expected Availability with a Given Budget

```
Input  : $D, M, m, B, r, f(d_i), s(d_i)$
Output: $E[M][B]$
for $b \leftarrow 0$ to $B$ do
    $E[0][b] \leftarrow 0$
end
for $j \leftarrow 1$ to $M$ do
    $E[j][0] \leftarrow -\infty$
end
for $j \leftarrow 1$ to $M$ do
    for $b \leftarrow 1$ to $B$ do
        $E[j][b] \leftarrow E[j][b-1]$
        forall combination $\delta \in (D, m \times r)$ do
            if $(b - C(\delta)) \geq 0)$ then
                $E(\delta) \leftarrow Call\ \ OCP(\delta, r, m)$ $e \leftarrow E[j-1][b - C(\delta)] + E(\delta)$
                $E[j][b] \leftarrow max(E[j][b], e)$
            end
        end
    end
end
```

n, must be at least $m \times r$ ($n \geq m \times r$). In fact, storing each object requires at least $m \times r$ independent DCs to gain maximum performance of striping [3].

Definition 3.4. *(Chunks Placement): Assume that* $\Phi_j = \bigcup_{k=1}^{m} \{\varphi_{j,k}\}$ *is a placement set for chunks of object j, where* $\varphi_{j,k}$ *represents a subset of DCs (i.e,* $\varphi_{j,k} \subset D$*) that containing r replicas of kth chunk. Therefore, for all* $\varphi_{j,k}$ *and* Φ_j*, we have* $|\varphi_{j,k}| = r$ *and* $|\Phi_j| = m \times r$*, respectively.*

Definition 3.5. *(Replication Cost of Chunks): Let* $C(\varphi_{j,k})$ *denote the cost of the kth chunk with r replicas of object j. Since cost per object in DC* d_i *is* $s(d_i)$ *and each object consists of m chunks,* $C(\varphi_{j,k}) = \sum_{d_l \in \varphi_{j,k}} \lceil s(d_l) \rceil / m$*. Therefore, the total replication cost of m chunks with duplication factor r of object j is given by:*

$$C(\Phi_j) = \sum_{k=1}^{m} C(\varphi_{j,k}). \tag{3.9}$$

The total cost of M objects can be written as:

$$C(\Phi) = \sum_{j=1}^{M} \sum_{k=1}^{m} C(\varphi_{j,k}). \tag{3.10}$$

Definition 3.6. *(Availability of Chunks): Suppose that* X_j *is a random variable as defined in*

Definition 3.3. Since object j is split to m chunks, we recalculate $E(X_j)$ as follows. The kth chunk with r replicas of object j is not available if all DCs $d_l \in \varphi_{j,k}$ fail. Thus, the failure probability of the kth chunk with r replicas is $\prod_{d_l \in \varphi_{j,k}} f(d_l)$. As a result, the kth chunk of object j is available if at least one replica of that is available, which results in $1 - \prod_{d_l \in \varphi_{j,k}} f(d_l)$ as availability of the kth chunk with r replicas. Obviously, the jth object can be retrieved, if all m chunks are available. Therefore, the expected availability of object j consisting of m chunks with duplication factor r under set Φ_j can be calculated as:

$$E(X_j) = E(\Phi_j) = \prod_{k=1}^{m}(1 - \prod_{d_l \in \varphi_{j,k}} f(d_l)). \tag{3.11}$$

Similar to the previous section, the expected availability of M objects termed by $E(\Phi)$ is the sum of $E(\Phi_j)$. Thus, we have:

$$E(\Phi) = \sum_{j=1}^{M} E(\Phi_j) = \sum_{j=1}^{M}(\prod_{k=1}^{m}(1 - \prod_{d_l \in \varphi_{j,k}} f(d_l)). \tag{3.12}$$

Now, we express the optimization problem, which lies in the above definitions and notations. Assume that a DC set D, M objects consisting m chunks with duplication factor r and a budget B are given, and also suppose that each QoS type Q is associated with P_Q. The objective is to find a subset $\Phi_j \subseteq D$ for each object j such that $E(\Phi)$ is maximized whilst $C(\Phi) \leq B$ and the expected availability of each type of objects is proportional to the priority parameter of that type. This is translated into:

$$\max \;\; E(\Phi) \;\; s.t. \;\; C(\Phi) \leq B \;\; and \;\; E(J_Q) \propto P_Q. \tag{3.13}$$

In order to solve Equ. (3.13), we propose a dynamic algorithm called *Maximum Expected Availability with a Given Budget*, in which the *Optimal Chunks Placement (OCP)* algorithm is called to provide optimal placement of chunks of an object. Algorithm 3.2 is presented without considering the constraint $E(J_Q) \propto P_Q$. That is, all objects have the same priority from the user perspective, and then this constraint is applied to the proposed algorithm.

In the proposed dynamic algorithm, let $E[M][B]$ be the expected availability for M

objects with a given budget B. In the first step, we obtain a recursive formulation for $E[j][b]$, where $1 \leq j \leq M$ and $0 \leq b \leq B$. In order to store the jth object in the set D of DCs, all possible δs of D ($|\delta| = m \times r$) are checked, and then $C(\delta)$ by using (3.9) and $E(\delta)$ based on the OCP algorithm are calculated. Since the replication cost of the jth object is $C(\delta)$, we have the budget $b - C(\delta)$ to consider for storing the $j-1$ objects. Thus, considering all possible δs ($\delta \subseteq D$) and all possible cases with budget $b - C(\delta)$ for $j-1$ objects, $E[j][b]$ is calculated as follows.

$$E[j][b] = \max_{\delta}((E[j-1][b - C(\delta)]) + E(\delta)). \tag{3.14}$$

In the second step, terminal conditions are considered. clearly, if $b - C(\delta) < 0$ then the subset δ is ignored and $E(\delta)$ is set to negative infinity. Also, if $b = 0$ and $j = 0$, $E[j][b]$ is initialized to zero. The proposed algorithm is outlined in Algorithm 3.2.

3.3.1 Optimal Chunks Placement (OCP) Algorithm

In this section, we discuss the OCP algorithm and its objective. Based on the Definition 3.4, an object has m chunks and each of them is replicated in r separate DCs. Without any special policy to select DCs for storing the chunks of an object, it might be some replicas of a chunk placed in more reliable DCs (that is DCs with less failure probability) whilst other replicas of another chunks are stored in less reliable ones. As a results, Equ. (3.11) is not maximized. In order to maximize that, we should maximize availability of each chunk, that is $(1 - \prod_{d_l \in \varphi_{j,k}} f(d_l))$, which is between 0 and 1. Therefore, the availability of all chunks should be close to each other as much as possible. Ideally, the availability of all chunks should be equal to each other. If it is feasible, $\forall k \neq k', f_{c_k} = f_{c_{k'}}$, where f_{c_k} is the failure of kth chunk with r replicas.

Since n is a small constant [70] and the number of replicas, r, is 2 or 3 at most [68], it is possible to search all the problem space in order to find the optimal placement of chunks. Thus, we present the OCP algorithm to find the optimal placement for the replication of chunks as follows.

The way the OCP algorithm works is by computing two functions, namely CA and PCA, each with two entries. One entry for the kth chunk and another one for all possible

Algorithm 3.3: Optimal Chunks Placement

```
Input  : $\delta, r, m$
Output: $CA[k][S]$
$S = C(\delta, r)$
Procedure OCP($S, r, m$)
forall ($\varphi \in S$) do
    $PCA[0][\varphi] \leftarrow 1$
end
for $k \leftarrow 1$ to $m$ do
    $CA[k][S] \leftarrow 0$
    forall ($\varphi \in S$) do
        $P \leftarrow \forall \varphi' \in S | \forall d_i, d_i \in \varphi' \wedge d_i \notin \varphi$
        $CA[k][\varphi] \leftarrow A(\varphi) \times PCA[k-1][P]$
        if ($k == 1$) then
            $PCA[k][P] \leftarrow \max_{\varphi' \in P}(CA(\varphi'))$
        end
        if ($k > 1$ and $kr < mr$) then
            $PCA[k][P] \leftarrow$ OCP($P, r, k-1$)
        end
        $CA[k][S] \leftarrow max(CA[k][\varphi], CA[k][S])$
    end
end
```

combinations of size r from δ, denoted by S ($|S| = \binom{|\delta|}{r}$), where δ is a qualified set of DCs which is determined by Algorithm 3.2. $\varphi \in S$ refers to any element of S, which is an r-combination of δ ($|\varphi| = r$). In more details, $CA[k][\varphi]$ is the maximum availability of kth chunk with r replicas if its replicas are stored in all $d_i \in \varphi$. Associated to each $\varphi \in S$, P is a subset of S including all elements $\varphi' \in S$, such that φ' does not include any DC $d_i \in \varphi$. $PCA[k][P]$ denotes the maximum availability of k chunks with r replicas that are replicated in P.

We derive a general recursive equation for $CA[k][\varphi]$. First, we enumerate all possible $\varphi \in S$ that could store the kth chunk with r replicas, as if we were placing the kth chunk. Second, we consider all possible placement of the first $(k-1)$ chunks with r replicas, which are placed into P. Thus, if set φ is considered for kth chunks, the availability of k chunks will be the multiplication of the maximum availability from the first $(k-1)$ chunks with r replicas, which are placed into P (i.e., $PCA[k-1][P]$), and the availability of the kth chunk, $A(\varphi)$, when we use φ. Thus, we have:

$$CA[k][\varphi] = A(\varphi) \times PCA[k-1][P]. \tag{3.15}$$

The computation for $PCA[k][P]$ is a recursive approach as follows. If ($k > 1$ and $kr < mr$), it is assumed that the first chunk with r replicas placed in $\varphi \in S$, and the remaining $(k-1)$ chunks, with duplication factor r should be optimally placed into P. Thus, OCP algorithm called recursively with appropriate parameters. This means OCP($P, r, k-1$). Otherwise, if($k = 1$), assumed as the terminal condition of the OCP algorithm, $PCA[1][P]$ is maximum availability of all $\varphi' \in P$ as if the chunk with r replicas is replicated in $\varphi' \in P$. Therefore, the recursive equation for PCA can be obtained as follows.

$$PCA[k][P] = \begin{cases} \max\limits_{\varphi' \in P}(A(\varphi')), \text{if } k = 1 \\ \text{OCP}(P, r, k-1) \ \text{ if } k > 1 \text{ and } (k \times r < m \times r) \end{cases}$$

Thus, from the derived recursive equations for CA and PCA, Algorithm 3.3 gives the pseudo-code for the OCP problem.

Similar to the previous section, we apply constraint $E(J_Q) \propto P_Q$ to Algorithm 3.2. First, budget B is allocated to each QoS type Q proportional to P_Q. That is, $b(J_Q) = \lfloor b \times P_Q \rfloor$, where $b(J_Q)$ is the budget allocated to QoS type Q. To guarantee constraint $E(J_Q) \propto P_Q$, $b(J_Q)$ is sorted decreasingly by $difb(J_Q)$, where $0 \leq difb(J_Q) = b \times P_Q - b(J_Q) < 1$, and then the first $B - \sum_Q b(J_Q)$ of $b(J_Q)$ is increased by one. Second, we substitute B with $b(J_Q)$ in Algorithm 3.2 to hold the constraint.

3.4 Performance Evaluation

3.4.1 Simulation Setting

We performed several experiments to assess the performance of our algorithms. Table 3.1 summarizes the objective and constraint of the proposed algorithms. We first present the parameters setting of cloud providers used in the performance evaluation. Although the failure probability of most cloud providers are not disclosed, some of them have revealed this parameter. For example, Amazon S3 provides two level of storage services: *Standard Storage* has eleven nines as durability and four nines as availability whilst the other ser-

Table 3.1: objective and constraint of the proposed algorithms

Algorithm	Replication Availability (Expected Availability, EA)	Replication Cost (Budget, B)
Algorithm 3.1	Limited by EA	Minimize
Algorithm 3.2	Maximize	Limited by B

Table 3.2: Data centers parameters

DC#	FP	CPO	DC#	FP	CPO
d_1	0.0001	48	d_6	0.004	12
d_2	0.0002	36	d_7	0.01	6
d_3	0.0004	30	d_8	0.04	4
d_4	0.001	24	d_9	0.1	2
d_5	0.002	18			

vice, *Reduced Redundancy Storage (RRS)*, provides four nines (99.99%) for availability and durability. Storage cost for both services depends on the region of the cloud provider and the level of service. In all regions the storage cost of standard storage is more than that of RRS. Google, another well-known cloud provider, has not disclosed failure probability; however, researchers [69] have done extensive studies on Google's main storage infrastructure and they have measured failure probability, which is about 0.045(that is, 3 nines) on average. Rackspace guarantees 3 nines (99.9%) availability within its SLA. The other providers such as Nirvanix, EMC Atoms, etc. do not disclose failure probability, but they consider credits to compensate availability violation as noted in the SLA.

According to the above description, we determine a set of DCs with two parameters, *Failure Probability (FP)* and *Cost Per Object (CPO)* for our simulation as shown in Table 3.2. Since we have the failure probability of Amazon S3, Rackspace and Google's storage infrastructure, we use them as baseline, and add 6 DCs with the assumption that as availability of service is increased, the storage cost of object is raised [38]. It is also assumed that the cost and the failure probability reported in Table 3.2 remains constant during the simulation.

In our simulation, we set the number of objects to 100, while duplication factor r is fixed to 2 since the number of replicas is a small constant in practice (e.g., 2 or 3) [68].

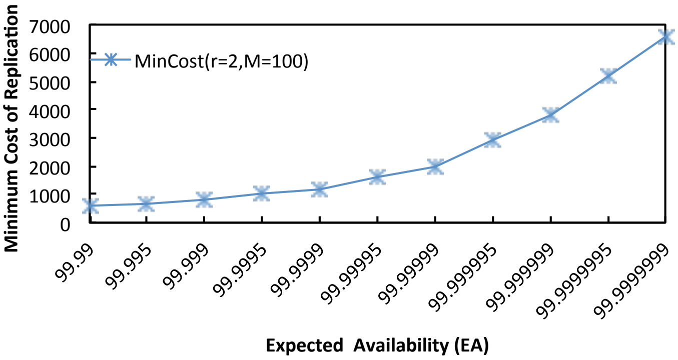


Figure 3.2: Minimum cost of replication versus expected availability of objects

3.4.2 Algorithm 3.1: Minimum Cost Fixed Expected Availability

We have performed experiments to evaluate this algorithm in order to measure the minimum cost of replication whilst the expected availability in the form of number of nines is satisfied. In the first experiment, we relax the constraint (3.4) and assume that all objects have the same priority from the user's perspective.

The result of this experiment are depicted in Fig. 3.2. We can say that if the user wants to have the expected availability with 4 *nines*, the minimum cost of replication is imposed to the user is about 600. Thus, if the user randomly chooses an object among the objects, he is able to access that object with the probability of 99.99%. Fig. 3.2 also demonstrates that the minimum cost of replication experiences almost two stages of significant increment. We observed that the algorithm explores most DCs in the range from d_9 to d_6 to provide 4 *nines* and 5 *nines* as expected availability. As the value of EA is increased from 5 to 7 *nines*, the algorithm dynamically switches to the more expensive DCs, where the first stage of increment in the minimum cost of replication happens. The second stage of increment incurs when the value of *EA* is augmented from 7 *nines* to 9 *nines*, which results in three times increment in the minimum cost of replication.

To evaluate Algorithm 3.1 with constraints (3.4) (i.e., the revised Algorithm 3.1), it is assumed that the user asks to store Gold, Silver and Bronze objects in DCs subject to, for example, $\text{P}_G = 50\%$, $\text{P}_S = 30\%$ and $\text{P}_B = 20\%$. Furthermore, the number of objects in each type of QoS is set to 100. With the above assumptions, the revised Algorithm 3.1 is run and its results are illustrated in Fig. 3.3.

This figure demonstrates as the *EA* value is increased, the minimum cost of replication of Gold objects significantly grows compared to that of two other types. For example,

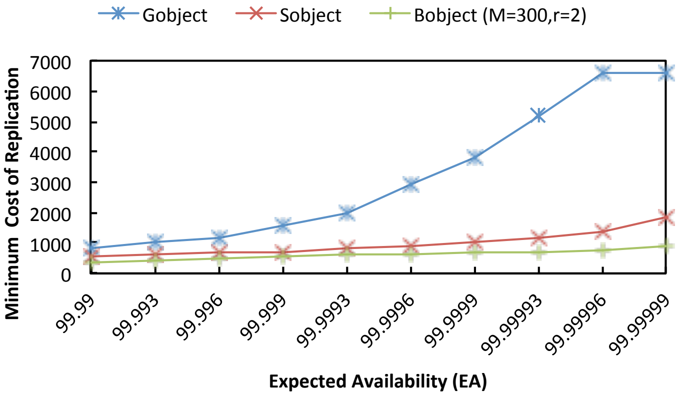
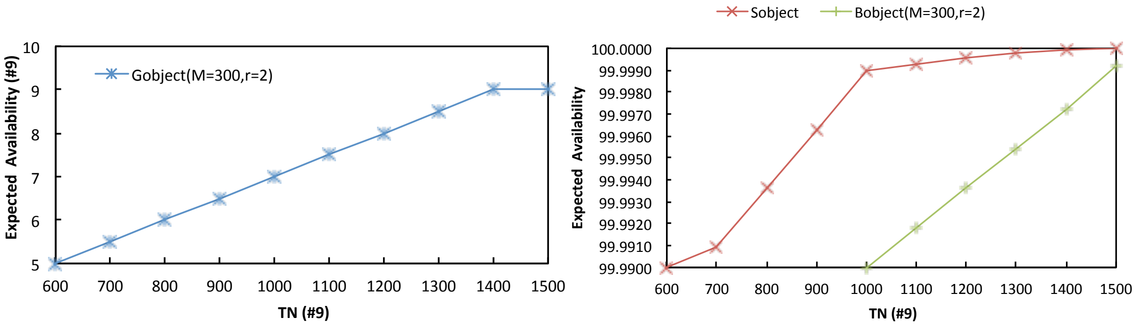


Figure 3.3: Minimum cost of replication versus expected availability for three types of QoS



(a) Expected availability versus *TN* for Gold objects (b) Expected availability versus *TN* for Silver and Bronze objects

Figure 3.4: Expected availability versus *TN* for three types of QoS

with *EA*=7, the minimum cost of replication for Gold objects is approximately three and seven times more than that of the required for Silver and Bronze objects, respectively. This is because the revised Algorithm 3.1 explores more reliable DCs, which in turn are more expensive ones, to host Gold objects in comparison with two other QoS types.

In Fig. 3.4, as expected, the hierarchy between the QoS types is respected, i.e. the value of EA_G is higher than EA_S which in turn is higher than EA_B. For example, when *TN*=600, we have EA_G=5 *nines* and EA_S=4 *nines* whilst about 60% of Bronze objects are stored in DCs such that EA_B=4 *nines* (which is not plotted in Fig. 3.4b). This is because that Gold and Silver objects have stricter requirements. Figs. 3.4a and 3.4b also show that as the *TN* value rises, the value of EA_Q increases according to P_Q. Fig. 3.5 plots *EA* against EF_Q. As it can be see, the higher *EA* value, the lower EF_Q value for three types of QoS. In addition, the value of EF_G is lower than EF_S and EF_B because the value of P_Q prioritizes Gold, Silver and Bronze objects. For example, when EA=7 nines , EF_G ≈1.6

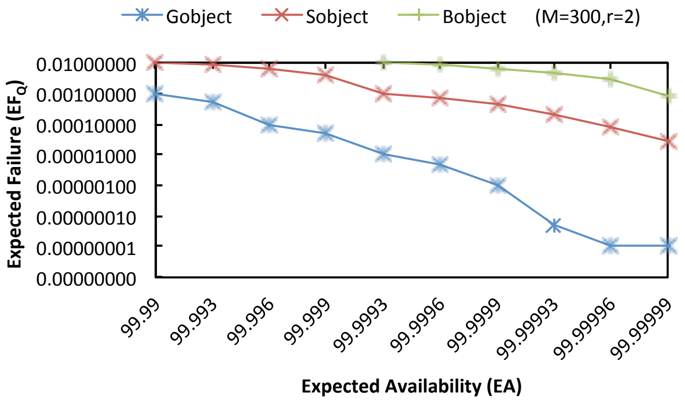


Figure 3.5: Expected failure (EF_Q) versus Expected Availability for three types of QoS

EF_S and $EF_G \approx 3\ EF_B$.

3.4.3 Algorithm 3.2: Maximum Expected Availability with a Given Budget

In this algorithm, since n and r are small constants, the value of m is small and varies between 2 and $\lfloor n/r \rfloor$. The value of m can be defined according to a trade-off between the switching cost and the availability of objects. This is because as m is increased the switching cost and the availability of objects are decreased. Finding the optimal value of m based on this trade-off is beyond the scope of this chapter. What it is important for us is to investigate how Algorithms 3.2 and 3.3 are able to find optimal placement of chunks for each arbitrary value of m. Therefore, in order to evaluate them, we fix the values of M, r, and m and vary the value of budget. Fig. 3.6 shows that the value of *EA* for objects when the budget is varied from 1000 to 3000. The following observations can be made from the results. First, with increasing the budget, the value of *EA* is increased. This should be attributed to the fact that Algorithm 3.2 explores the more expensive DCs with lower failure probability to store objects as the budget grows. Second, as the budget is increased, the rate of increment in the value of *EA* becomes smaller. Because Algorithm 3.2 chooses a subset of DCs from D as a *main* set of DCs such that the cost of replication for all objects gained the maximum expected availability is minimized. As the budget escalates, the algorithm dynamically changes some DCs in *main* set, which is termed as *auxiliary* subset, until the budget allows the algorithm to find a new *main* set of DCs. This new *main* set increases the value of *EA* with a smaller rate. This happens because marginal increment in *EA* value of using more expensive of DCs in *auxiliary* subset is

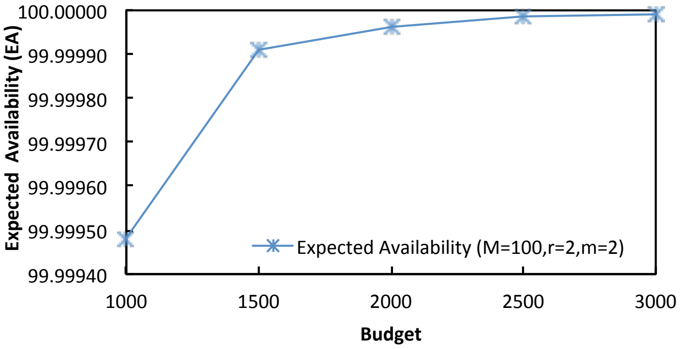


Figure 3.6: Expected availability versus Budget

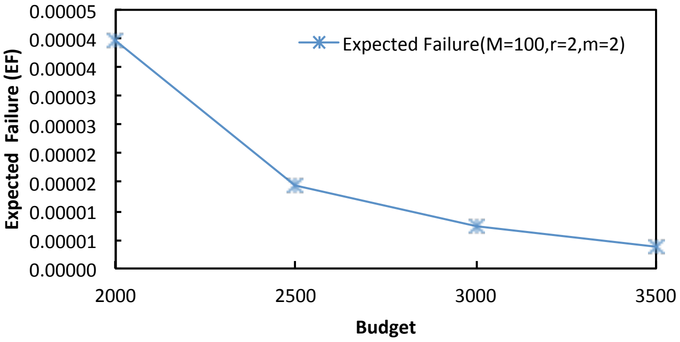


Figure 3.7: Expected failure versus Budget

decreasing. As a result, the expected availability is increased less with the same amount of additional budget. For example, when the budget increases from 1000 to 1500, the value of *EA* increases one nine that is the same as that of from 1500 to 2500. Fig. 3.7 illustrates the *EF* value for objects versus the budget. As it can be seen, the *EF* value is decreased with the increase of budget. Furthermore, the reduction in the value of *EF* becomes less when we have an increment in the budget, which confirms the second observation in Fig. 3.6.

Fig. 3.8 plots the value of EA_Q against the budget that is varied between 2000 and 6000. The higher budget results in the higher value of EA_Q for all types of QoS. As expected, the Gold objects achieve the highest expected availability and Bronze objects achieve the lowest one, because the revised Algorithm 3 divides the budget among each type of QoS according to P_Q. Fig. 3.8 also depicts that EA_B value is constant when the budget is varied from 2000 and 5000. The reason is since the revised algorithm allocates the lowest budget to Bronze objects, it is not possible to store all Bronze objects in the DCs. In this experiment, 60%, 40% and 20% Bronze objects are not placed in DCs when the budget is 2000, 3000, and 4000, respectively. This figure for Silver objects is about 40% and 20%

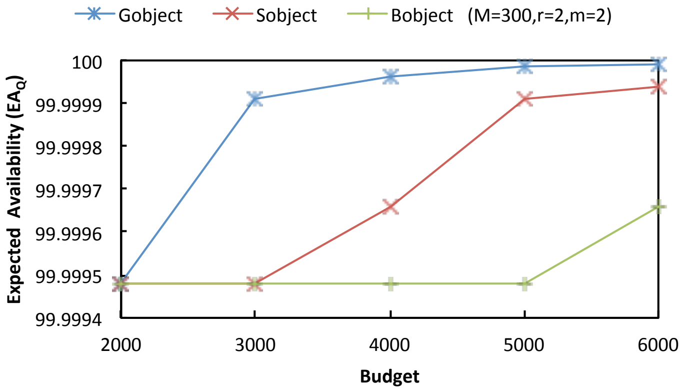


Figure 3.8: Expected availability for three types of QoS Versus Budget

when the budget is 2000 and 3000, respectively. Placing all Silver and Bronze objects happens when the amount of budget reaches 3000 and 5000, respectively.

3.5 Summary

We addressed two issues related to placing replicas of the objects in multi-cloud environment. In order to tackle these issues, we propose efficient algorithms. The first algorithm has been designed to minimize the replication cost and the expected availability of objects as the user's QoS is met. The second one is proposed to maximize the expected availability of objects under a given budget with the assumption that the objects are split to chunks. We also have conducted extensive simulation experiments to evaluate the effectiveness of our algorithms. The experiments show that the proposed algorithms are efficient to determine the optimal location of the replicas for objects with a given constraint.

Chapter 4

Cost Optimization in a Dual Cloud-based Storage Architecture

Due to diversity of pricing options and variety of storage and network resources offered by cloud providers, enterprises encounter nontrivial choice of what combination of storage options should be used in order to minimize the monetary cost of managing data in large volumes. prior to investigate this matter for Geo-replicated data stores, it is important to consider it for a dual cloud-based storage architecture (i.e., the combination of a temporal and a backup DC) as a fine-grained cloud-based architecture. For this purpose, we propose two data object placement algorithms, one optimal and another near optimal, that minimize residential (i.e., storage, data access operations), delay, and potential migration costs in this architecture. We evaluate our algorithms using real-world traces from Twitter. Results confirm the importance and effectiveness of the proposed algorithms and highlight the benefits of leveraging pricing differences and data migration across cloud storage providers (CSPs).

4.1 Introduction

DATA volume is one of the important characteristics of cloud-based application (e.g., Online Social Network) and has been changed from TB to PB with an inevitable move to ZB in current IT enterprises. From statistical perspective, 8×10^5 PB of data were stored in the world by the year of 2000 and it is expected that this number will increase to 35 ZB by 2020 [188]. Storing and retrieving such data volume demand a highly available, scalable, and cost-efficient infrastructure.

Thanks to the cloud infrastructures, management of such large volume data has been simplified and the need for capital investment has been removed from IT companies.

This chapter is derived from: **Yaser Mansouri** and Rajkumar Buyya, "To Move or Not to Move: Cost Optimization in a Dual Cloud-based Storage Architecture," *Journal of Network and Computer Applications (JNCA)*, Volume 75, Pages: 223-235, ISSN: 1084-8045, Elsevier, Amsterdam, The Netherlands, November 2016.

However, this creates a major concern for these companies regarding the cost of data management in the cloud. The cost of data storage management (simply, cost of data management) is a vital factor from companies' perspective since it is the essential driver behind the migration to the cloud. Thus, companies are in favor of optimizing data management cost in the cloud deployments. In order to optimize data management cost, choosing a suitable storage option across CSPs in the right time becomes a nontrivial task. This happens due to the two following reasons.

First, there is an array of pricing options for the variety of storage and network services across CSPs (e.g., Amazon, Google and Microsoft Azure). CSPs currently offer at least two classes of storage service: Standard Storage (SS) and Reduced Redundancy Storage (RRS). RRS enables users to reduce cost at the expenses of lower levels of redundancy (i.e., less reliability and availability) as compared to SS. These services provide users with API to *Get* (read) data from storage and to *Put* (write) data into it. In mid-2015, Amazon and Google respectively introduced Infrequent Access Storage (IAS) and Nearline storage services that aimed at hosting objects with infrequent *Gets*/*Puts*. Both services charge lower storage cost in comparison to their corresponding RRS but higher cost for *Get*s and *Put*s.

Furthermore, CSPs charge users with different out-network cost to read data from a DC to the Internet (typically in-network data transfer is free). They also offer discounts for data transfer between their DCs. For example, Amazon reduces out-network cost when data is transferred across its DCs in different regions and Google offers free of charge data exchange between its DCs in the same region. Thus, taking the advantage of diversification in price of storage and network (as well as service type) plays an important factor in *residential cost* (i.e., Storage and data access operations costs) as a major part of the data management cost. Note that data access operations are *Get* and *Put* in this chapter.

Second, there is time-varying workload on the object stored in the cloud. Presume that an object is a tweet/photo and it is posted on the user's feed (e.g., timeline in Facebook) by herself or her friends. *Get*s and *Put*s are usually high in the early lifetime of the object and we say that such object is in *hot-spot* status. As time passes, *Get*s and *Put*s are reduced and we refer that the object is in *cold-spot* status. Thus, it is cost-efficient to store the object in a DC with lower out-network cost (referred as a *temporal* DC) in its early lifetime, and then migrate it to the DC with lower cost in storage (referred as a *backup* DC). If the object migration happens between a temporal and a backup DC, the user incurs *migration cost*. This cost is another part of the data management cost, which is affected by the number of *Get*s, *Put*s, and the object size. It is important to note that the migration cost might be zero in some cases: (i) if both DCs belong to the same provider and are in the same region, then transferring objects between DCs is free, as in Google provider, and (ii) if *temporal* and *backup* DCs are the same and the object is just moved from a storage class to another (i.e., from SS to IAS) within the same DC.

Besides discussed costs, *latency* for reading from (writing into) the data store is also a vital performance criterion from the user's perspective. The latency is defined as the elapsed time between issuing a Get/Put and receiving the required object. To respect this criterion, we convert latency into monetary cost, as a *latency cost*, and integrate it in our cost model.

In summary, by wisely taking into account the discussed differences in prices across CSPs and time-varying workload, we can reduce the data management cost (i.e., residential, latency, and migration costs) as one of the main user's concern with regard to the cloud deployment. To address this issue, we make the following **contributions**:

- we propose the optimal algorithm that optimizes data management cost in the dual cloud-based architecture when the workload in terms of *Get*s and *Put*s on the objects is known;
- we also propose a near-optimal algorithm that achieves competitive cost as compared to that obtained by the optimal algorithm in the absence of future workload knowledge; and
- we demonstrate the effectiveness of the proposed algorithms by using the real-world traces from Twitter in a simulation.

The reminder of this chapter is organized as follows. Section 4.2 presents system and cost model. In Section 4.3, we describe our object placement algorithms to save cost. Section 4.4 presents our simulation experiments and evaluation of the proposed algorithms. Finally, in Section 4.5, we conclude this chapter with future work issue related to cost optimization of data management across Geo-replicated cloud-based data stores.

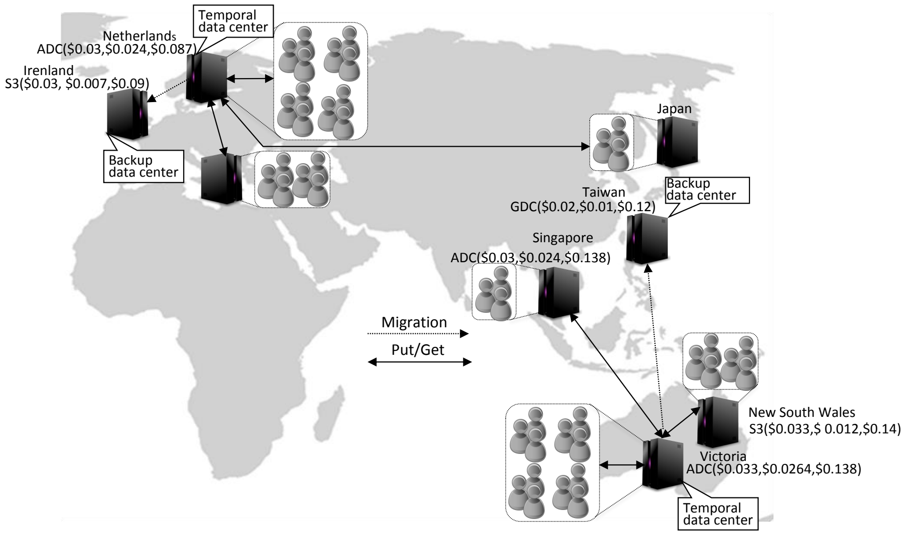


Figure 4.1: A scenario of the dual cloud-based storage architecture in the European and Asia-Pacific regions. Parenthesis close to each DC's name shows the storage price (per GB per month) for standard storage, backup storage, and network price (per GB), respectively.

4.2 System and Cost Model

We first describe the dual cloud-based architecture, which can lead to reduced monetary cost for applications. Then, we discuss the cost model and the objective function that should be minimized considering the objective and specifications of the architecture.

4.2.1 System Model

In our system model, an object is a tweet or photo posted by users on their feed. As stated earlier, the object is stored in the temporal (resp. backup) DC during its hot-spot (resp. cold-spot) status to benefit from lower network (resp. storage) cost. A transition between hot- and cold-spot probably leads to the object migration. Our architecture uses *stop and copy* migration technique [65] in which *Get*s and *Put*s are respectively served by the temporal and backup DCs during the object migration. Fig. 4.1 illustrates the scenario of the architecture in two regions. In the architecture, each user is assigned to the closest DC among the DCs which are Geo-graphically located across the world. This DC is referred as the home DC. The determined home DC for each user is then paired with the DC that is selected by the object owner or application provider[1]. The paired DCs are considered as

[1] We pair each home DC with 21 DCs in the experiment to determine which combination of temporal and backup DCs is cost-efficient.

the temporal and backup DCs. As an example, in the Asia-Pacific, a user stores the object in the Victoria Azure DC (ADC) with low network cost, while their friends in Singapore and New South Wales access the object by issuing their read/write requests from their home DCs. As time passes, the object is migrated to the Taiwan Google DC (GDC), which has lower storage cost (i.e., Nearline) in comparison to S3 and ADC in the Asia-Pacific region.

4.2.2 Cost Model

The system model is represented as a set of independent DCs D where each DC d is associated with the following cost elements to manage data:

1. **Storage cost:** $S(d)$ denotes the storage cost per unit size per time,
2. **Network cost:** $O(d)$ is the cost per byte of out-bound bandwidth transferred from DC d (in-bound bandwidth is typically free),
3. **Transaction cost:** t_g and t_p define transaction cost for a *Get* and *Put* request respectively.

Assume that the application hosts a set of objects in time slot $t \in [1...T]$. Each object is associated with $v(t)$, $r(t)$, and $w(t)$ denoting respectively size in byte, the number of read and write operations for the object in time slot t. Also suppose l denotes the latency between the DC that issues a *Put*/*Get* for the object and the DC that hosts the required object. $x_d(t)$ represents whether the object exists in DC d in time slot t ($x_d(t) = 1$) or not ($x_d(t) = 0$).

Residential cost: The residential cost of the object in time slot t is as follows. (i) The storage cost of the object is equal to its size multiplied by the storage price ($S(d)v(t)$). (ii) The read cost of the object is the cost of all Gets ($r(t)t_g(d)$) and the communication cost ($r(t)v(t)O(d)$). (iii) The write cost of the object is the cost of all *Put*s for updating of the object ($w(t)t_p(d)$). Thus, the residential cost C_R is defined as:

$$C_R(x_d, t) = \sum_{x_d} x_d(t)[v(t)(S(d) + r(t)O(d)) + r(t)t_g(d) + w(t)t_p(d)] \ . \tag{4.1}$$

Delay cost: Time is cost and user-perceived latency for reading and writing the object is a vital criterion. For example, Amazon reported that every 100 ms of latency reduces

1% of its sales. To capture this cost, the incurred latency for *Get*s and *Put*s is considered as a monetary cost [194]. We consider "Get/Put delay" as the time taken from when a user issues a Get/Put from the DC d' to when he/she gets a response from the DC d that hosts the object. In fact, the read and write requests are issued from the application hosted by the closest DC to the user. The delay cost of read and write requests, C_D, can be formally defined as:

$$C_D(x_d,t) = \sum_{x_d(t)} x_d[l(d',d)v(t)(r(t)+w(t))l_w] \ ,\forall d,d' \in D. \tag{4.2}$$

In Equ. 4.2, $l(d',d)$ denotes the latency between DC d' that issues requests and DC d that hosts the object. $l(d',d)$ in our formulation and evaluation is based on the round trip times between d and d'. This is reasonable because for the application, the size of objects is typically small (e.g., tweets, photos, small text file), and thus data transitions are dominated by the propagation delays, not by the bandwidth between the two DCs. For applications with large objects, the measured $l(d',d)$ values capture the impact of bandwidth and data size as well. In the above equation, l_w denotes the *latency cost weight*, which converts latency into a monetary cost. It is set by the application based on the importance degree of the latency from the user's perspective. The more importance the latency, the higher l_w will be.

Migration cost: As time passes, the number of *Put*s and *Get*s decreases, and it is cost-effective to migrate the object from a *temporal* DC to a *backup* DC with the lower cost in storage. We use *stop and copy* migration technique in our model for two reasons. First, the system performance is not significantly affected by this technique because (i) the transfer time of a bucket (which is at most 50 MB in size [52]) between *temporal* and *backup* DCs is about few seconds, and (ii) a significantly lower number of *Put*s and *Get*s must be served after the transition to the cold-spot status. Second, this technique imposes a lower cost as compared to the log-based technique [169]. The object migration cost is the cost of retrieving the object from the source DC ($v(t)O(d_s)$) and writing it to the destination DC ($t_p(d_d)$). Thus the migration Cost $C_M(t-1,t)$ is defined as:

$$C_M(t-1,t) = v(t)O(d_s) + t_p(d_d), \tag{4.3}$$

where d_s and d_d are the source and destination DCs respectively.

Cost optimization Problem: Considering the aforementioned cost model, we define the objective as to determinate the object placement (i.e., x_d) in each time slot t so that the overall cost (i.e., the sum of residential, delay, and potential migration costs) of the object during $[1...T]$ is minimized. Thus, the objective function can be defined as:

$$\sum_{t=1}^{T} \sum_{x_d(t)} C_R(x_d(t), t) + C_D(x_d(t), t) + C_M(t-1, t), \quad x_d(t) \in \{0, 1\}. \tag{4.4}$$

4.3 Data Management Cost Optimization

To solve the aforementioned cost optimization problem, we first propose a dynamic algorithm to minimize the overall cost while the future workload is assumed to be known a priori. Then, we present a heuristic algorithm to achieve competitive cost as compared to the cost of dynamic algorithm for unknown objects workload.

4.3.1 Optimal Object Placement (OOP) Algorithm

Let $P(d, t)$ be the minimum cost of the object in DC d in time slot t. In order to compute P, we drive a general recursive equation for P as Equ. 4.5, where $C(d, t)$ is the summation of the residential, delay, and potential migration costs of the object in time slot t (Equs. (4.1 - 4.3)).

We first enumerate all possible DCs that could store the object in time slot t and then calculate the residential and delay costs. Second, we consider all possible placements of the object in time slot $t-1$ and calculate the migration cost from each DC in time slot $t-1$ to the current DC d in time slot t.

If we store the object at DC d in time slot t, then the overall cost (i.e., $P(d, t)$) is the minimum of the overall cost in time slot $t-1$ (i.e., $P(d, t-1)$) plus the residential, delay, and potential migration costs of the object in time slot t (i.e., $C(d, t)$). This recursive equation is terminated when t is zero and its corresponding $P(d, t)$ value is zero. Therefore,

Algorithm 4.1: Optimal Object Placement (OOP) Algorithm

```
   Input  : DCs and objects specifications
   Output: $x_d^*(t)$ and the optimized overall cost during $t \in [1...T]$
1  Initialize: $\forall$ DC $d$, $P(d,0) \leftarrow 0 \wedge \forall t \in [1...T], x_d(t) \leftarrow 0$
2  for $t \leftarrow 1$ to $T$ do
3      /* DCs (i.e., d) are a temporal DC and a backup DC.*/
4      forall $x_d(t)$ do
5          forall $x_d(t-1)$ do
6              Calculate $P(d,t)$ based on Equ. 4.5.
7          end
8      end
9  end
10 Select $\min_d P(d,T)$ as the optimized overall cost (i.e., Equ. 4.4).
11 Take the optimized overall cost, i.e., $\min_d P(d,T)$, in time $T$ and set its
   corresponding $x_d(T)$ to 1 (i.e., $x_d^*(T)$). Then, the value of $x_d(T-1)$ is set to 1 if its
   corresponding cost value in time slot $T-1$ leads to the optimized overall cost in
   time slot $T$. In the same way, find the value of all $x_d(t)s$ from $T-2$ to 1.
12 All $x_d(t)$s with value of 1 are $x_d^*(t)$s.
13 Return $x_d^*(t)$ and the optimized overall cost.
```

we define the recursive equation for the OOP algorithm as:

$$P(d,t) = \begin{cases} \min_d [P(d,t-1) + C(d,t)], & t > 0 \\ 0 & t = 0 \end{cases} \tag{4.5}$$

.

Once $P(d,t)$ is calculated for all DCs d during $t \in [1...T]$, we calculate $\min_d P(d,T)$ as the minimum cost of the object. It is easy to find the optimal location of the object in time slot t (i.e., $x_d^*(t)$) by backtracking from the minimum cost in time slot T. In each time slot t, if the cost value leads to minimum cost value in time slot $t+1$, then the value of $x_d(t)$ is 1; otherwise is 0. Continuing on the backtrack step from T to 1, we find the value of $x_d^*(t)$ for all $t \in [1...T]$.

The pseudo code in Algorithm 4.1 shows the discussed OOP. Note that since the value of $x_d^*(t)$ is 1 only for one DC in each time slot t, we can safely initialize $x_d(t)$ with 0 for all DCs d in all time slots t.

The time complexity of the algorithm is dominated by three nested "for" loops in which the recursive function (Equ. 4.5) is calculated. For each paired DCs (lines 4-8), we compute the value of $P(d,t)$ for each time slot $t \in [1...T]$. Thus, this calculation takes $O(2T)$ since we have two DCs in the dual cloud-based storage architecture. To find the location of the object in each time slot, we need to backtrack the obtained results (line 12), which takes $O(T)$. Since this process (lines 2-13) is repeated for all pairs of DCs (i.e.,$\binom{n}{2}$), not shown in the algorithm), the total time complexity of the algorithm yields $O(n^2T)$, where n is the number of DCs.

4.3.2 Near-Optimal Object Placement (NOOP) Algorithm

We propose a novel heuristic algorithm that finds a competitive solution for the cost optimization problem, as compared to that of OOP. Intuitively, if the object migrations do not happen at the right time, then besides migrations cost, the residential and delay costs increase as compared to these costs in OOP. We refer to the difference between these costs in OOP and NOOP as the overhead cost. This overhead cost should be minimized by providing a strategy that leads to the object migration in time(s) t_m so that it should be near to the optimal migration time(s) obtained by OOP. To achieve this aim and minimize the overhead cost, we make a trade-off between the migration cost and the summation of residential and delay costs (for summary, denoted by C_{RD}) in the absence of the future workload knowledge. The idea behind this trade-off is that the object is migrated to a new DC when (i) the object migration leads to save monetary cost at the new DC, and (ii) the sum of the lost cost savings from the last migration time up to the current time slot gets more than or equal to the migration cost of the object between DCs. This strategy avoids migrating the object too early or too late. The "trick" is to move the object "lazily", i.e., when sum of overall cost savings that could be done by any earlier migrations from the last migration time t_m is as large as the cost of migration in the current time slot.

Now we formally define the discussed trade-off. Let $C_M(t_{m-1}, t_m)$ be the migration cost between two consecutive migration times, where t_m is the last time the object is migrated. For each time slot $v \in [t_m, t)$, we calculate the summation of the residential and delay costs of the object (i) in the DC hosting the object in the previous time slot $t-1$, and (ii) in the new DC in the current time slot as if the object is migrated to it. Now, for each time slot v, we calculate the summation of the difference between two C_{RD}s in the two previous cases from time $v = t_m$ to $v = t - 1$. This summation is equal to $\sum_{v=t_m}^{t-1}[C_{RD}(x_d(v-1),v) - C_{RD}(x_d(v),v)]$. Note that if the migration of the object happens in time slot $v = t$, we assign $t_{m-1} = t_m$ and t_m=t in Equs. 4.6 and 4.7. Based on the summation of the residential and delay costs (i.e., C_{RD}) and the migration cost (i.e., $C_M(t_{m-1}, t_m)$) in the current time slot, the algorithm decides whether the object should be migrated or not. As discussed before, to avoid the object being migrated too early or too late, the object migration happens only if both the following conditions are satisfied:

First, the object has the potential to be migrated to a new DC if

$$C_M(t_{m-1}, t_m) \leq \sum_{v=t_m}^{t-1}[C_{RD}(x_d(v-1),v) - C_{RD}(x_d(v),v)]. \tag{4.6}$$

Otherwise, the object is kept in the previous DC as determined in time slot $t-1$. This condition prevents the object being migrated too late. Second, to avoid early object migration, we enforce the following condition.

$$C_M(t_{m-1}, t_m) + C_{RD}(x_d(t),t) \leq C_{RD}(x_d(t-1),t). \tag{4.7}$$

This constraint implies that the overall cost of the object, including residential, delay, and migration costs, in the new DC should be less or equal to the summation of residential and delay costs of the object if it stays in the determined DC in time slot $t-1$. Algorithm 4.2 represents the pseudo code for NOOP.

The time complexity of the algorithm is as follows. We need to calculate the total cost (lines 4- 16) for each paired DCs for each time slot $t \in [1...T]$. This calculation takes $O(T)$(lines 6-15). Since we repeat this computation for all pairs of DCs (i.e., $\binom{n}{2}$), not shown in the algorithm), the total time complexity of the algorithm is $O(n^2T)$, where n is the number of DCs.

4.4 Performance Evaluation

In this section, we first discuss the experimental settings in terms of workload characteristics, DCs specifications, and assignment of users to DCs. Then, we study the performance of the proposed algorithms in terms of cost saving and investigate the effect of the various parameters on the cost saving.

Algorithm 4.2: Near-Optimal Object Placement (NOOP) Algorithm

```
Input  : DCs and objects specifications
Output: $\hat{x}_d(t)$ and the overall cost during $t \in [1...T]$ as denoted by $C_{ove}$.
$C_{ove} \leftarrow 0, \forall t \in [1...T], x_d(t) \leftarrow 0$
$C_{ove} \leftarrow$ Select either backup or temporal DC so that cost $C_{RD}$ is minimized in time
slot $t$, and set Corresponding $x_d(t=1)$ to 1.
$t_m \leftarrow 1$
for $t \leftarrow 2$ to $T$ do
    /* DCs (i.e., d) are a temporal DC and a backup DC.*/
    forall $x_d(t)$ do
        $C_{RD}(.) \leftarrow$ Determine $x_d(t)$ by minimizing $C_{RD}(x_d, t)$
        $C_{ove} \leftarrow C_{ove} + C_{RD}(.)$
    end
    if (Equs. 4.6, 4.7, and $x_d(t-1)! = x_d(t)$ ) then
        $t_{m-1} \leftarrow t_m, t_m \leftarrow t, C_M \leftarrow$ calculate $C_M(t_{m-1}, t_m)$
        $x_d(t) = 1, C_{ove} \leftarrow C_{ove} + C_M$
    else
        $x_d(t) \leftarrow x_d(t-1)$
    end
end
$x_d$s with value of 1 are $\hat{x}_d(t)$s.
Return $\hat{x}_d(t)$ and $C_{ove}$.
```

4.4.1 Experimental settings

We evaluated the performance of algorithms via extensive experiments using a dataset from Twitter [101]. In the dataset, each user has her own profile, tweets, and a user friendship graph over a 5-year period. We focus on tweet objects posted by the users and their friends on the timeline, and obtain the number of tweets (i.e., number of *Put*s) from the dataset. Since the dataset does not contain information of accessing the tweets (i.e., number of *Get*s), we set a Get/Put ratio of 30:1, where the pattern of *Get*s on the tweet follows Longtail distribution [16]. This pattern mimics the transition status of the object from hot- to cold-spot status. The size of each tweet varies from 1 KB to 100 KB in the dataset.

We model 22 DCs in CloudSim Toolkit [35], and among these DCs, 9 are owned by Microsoft Azure, 4 by Google, and 9 by Amazon. Each DC is referred by a name that consists of (i) provider name: Microsoft Azure (AZ), Google (GO) and Amazon (AM); (ii) location: USA (US), Europe (EU), Asia (AS), Australia (AU), Japan (JA), and Brazil (BR); and (iii) the specific part of the location: south (S), north (N), west (W), east (E),

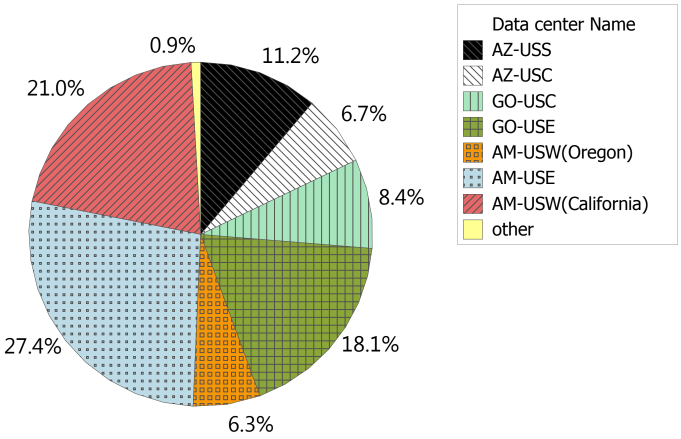


Figure 4.2: Allocated users to DCs (%).

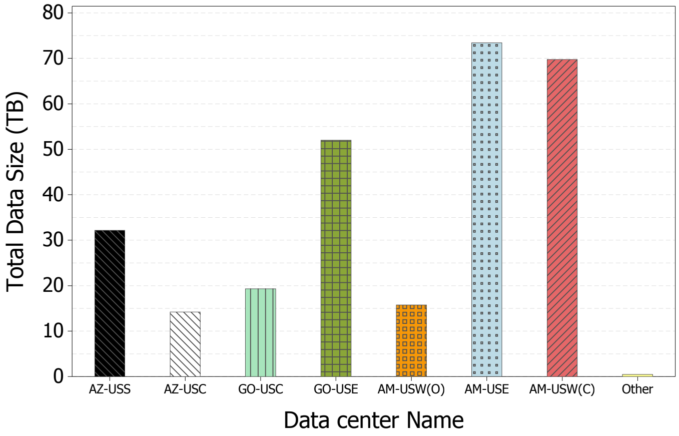


Figure 4.3: Total data size in data centers.

and center (C). Since we have two different Amazon DCs in US-West region, i.e., Oregon and California, we add letter "O" for Oregon and "C" for California. For example, based on this naming, the DC with name `AZ-USS` refers to the DC that is part of Microsoft Azure in the USA South. Every DC offers two classes of storage services: (i) SS, and (ii) RRS, that is, Locally Redundant Storage (LRS) for Azure, Nearline for Google, and IAS for Amazon. The latter storage class, i.e., RRS, is used for the object when it transits to *cold-spot* status. We set the storage and network prices of each DC as of September 2015.[2]

We measured inter-DCs latency (22*22 pairs) over several hours using instances deployed on all 22 DCs. We run `Ping` operation for this purpose, and used the medium latency values as the input for our experiments. The default value of l (as the latency cost weight) to convert delay to cost is 10. As a result, delay cost constitutes 7-10% of the total

[2] Amazon S3 storage and data transfer pricing. https://aws.amazon.com/s3/pricing/
Google storage and data transfer pricing. https://cloud.google.com/storage/pricing
Azure storage pricing. https://azure.microsoft.com/en-us/pricing/details/storage/
Azure data transfer pricing. https://azure.microsoft.com/en-us/pricing/details/data-transfers/

cost in the system.

With the help of Google Maps Geocoding API [3], we convert users' text locations to geo-coordinates (i.e., latitude and longitude) according to the users' profiles. Then, according to the coordination of users and DCs, we assigned users to the nearest DC based on their locations. In the case of two (or more) DCs with the same distance from the user, one of these DCs is randomly selected as the home DC for the user. One-month (Dec. 2010) of Tweeter dataset with more than 46K users, posting tweet on their timeline, is utilized for our experiments. As shown in Fig. 4.2, around 99.1% of users are assigned to DCs in the USA while the remaining users are designated to DCs in Europe, Asia, Australia and Brazil[4]. This is because most of the users of the dataset come from the USA region. Therefore, we focus on the cost saving for DCs in the USA including: two Azure DCs (`AZ-USS` and `AZ-USC)`, two Google DCs (`GO-USC` and `GO-USE)`, and three Amazon DCs (`AM-USW(O)`, `AM-USE` and `AM-USW(C)`). The total size of data in each DC, as depicted in Fig. 4.3, is dependent on the number of users allocated to each DC and the number of tweets posted by users.

4.4.2 Results

We compare the cost savings gained by the proposed algorithms with the following policy benchmark. It is important to mention that (1) the obtained results are valid for the current prices offered by three well-known cloud providers investigated in this chapter, and these prices may change quickly in the current competitive market, and (2) in this work, cloud provider selection is only determined based on the monetary cost, while data placement decision can be made based on other criteria such as availability, durability, scalability, and even the reputation of the cloud provider from application owner's perspective. Thus, with these results, we do not intend to advertise or harm the reputation of an individual cloud provider.

[3] The Google maps geocoding API https://developers.google.com/maps/documentation/geocoding/intro

[4] We also used the same policy to assign friends of the user to a DC. The user's friends are derived from the friendship graph of dataset.

Benchmark policy

In the benchmark policy, we permanently store the user's objects in the home DC (i.e., closest DC), and thus the object is not allowed to be migrated to another DC. This is because that application providers often deploy their data in data centers close to their user base. In all experiments, we normalize the incurred cost of the algorithms to the cost of the benchmark policy by varying the following parameters: home DCs, data size factor, latency cost weight, read to write ratio, and access pattern of read/write on the objects. Each parameter has a default value and a range of values as summarized in Table 4.1. This range is used for studying the impact of the parameter variations on the cost performance of the proposed algorithms. For clarity, Table 4.2 summarizes the specific settings in terms of parameters corresponding to each figure. In the following section, we discuss the cost saving of OOP and NOOP.

Table 4.1: Summary of Simulation Parameters

	Data size factor	Latency cost weight	Read to write ratio	Access pattern on objects
Default	1	10	30	Longtail
Range	0.2-1	1-30	1-30	Normal, Random

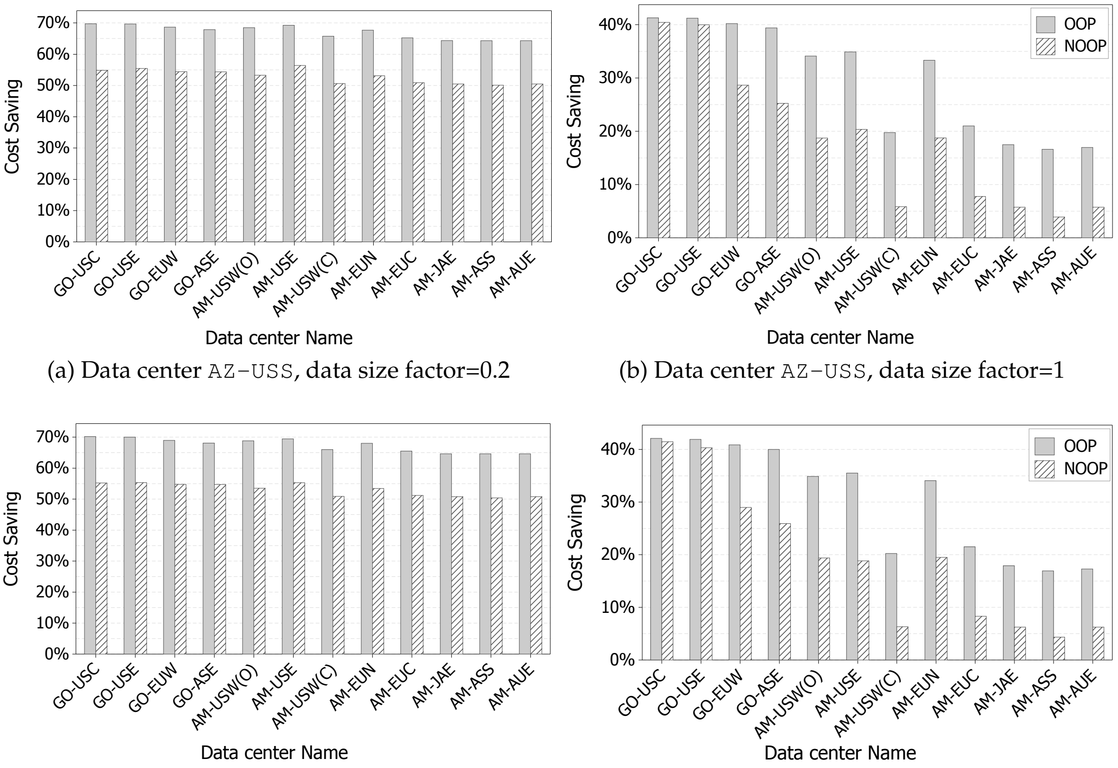
Cost Performance

In this section, we study the cost savings of the proposed algorithms for the most populated DCs (i.e., 6 home DCs) with factor size 0.2 and 1. Note that a DC with "data size factor x" means that it only stores x percent of the generated total data size, as shown in Fig. 4.3 for each home DC. A DC stores the total data size when data size factor is 1. For example, based on Fig. 4.3, `AZ-USS` with *data size factor* 0.2 stores 20% of 30 TB. It is important to note that we report result for each pairing between the home DC and each of 21 DCs in the experiments when cost can be saved.

Fig. 4.4 shows the cost savings of OOP and NOOP for `AZ-USS` and `AZ-USC` when each of these home DCs are paired with 21 DCs. As expected, the cost cannot be saved when `AZ-USS` and `AZ-USC` are paired with Azure DCs, as Azure DCs have more (or the same) cost in the network and storage services than these considered home DCs. In

Table 4.2: Evaluation Settings for Figures and Tables.

Figs./Table	Data size factor	Latency cost weight	Read to write ratio	Access pattern on objects
4.4, 4.5, 4.6	0.2,1	10	30	Longtail
4.7	0.2-1	10	30	Longtail
4.8	1	1-30	30	Longtail
4.9	1	10	1-30	Longtail
Table 4.3	1	10	30	Normal and Random



(a) Data center AZ-USS, data size factor=0.2

(b) Data center AZ-USS, data size factor=1

(c) Data center AZ-USC, data size factor=0.2

(d) Data center AZ-USC, data size factor=1

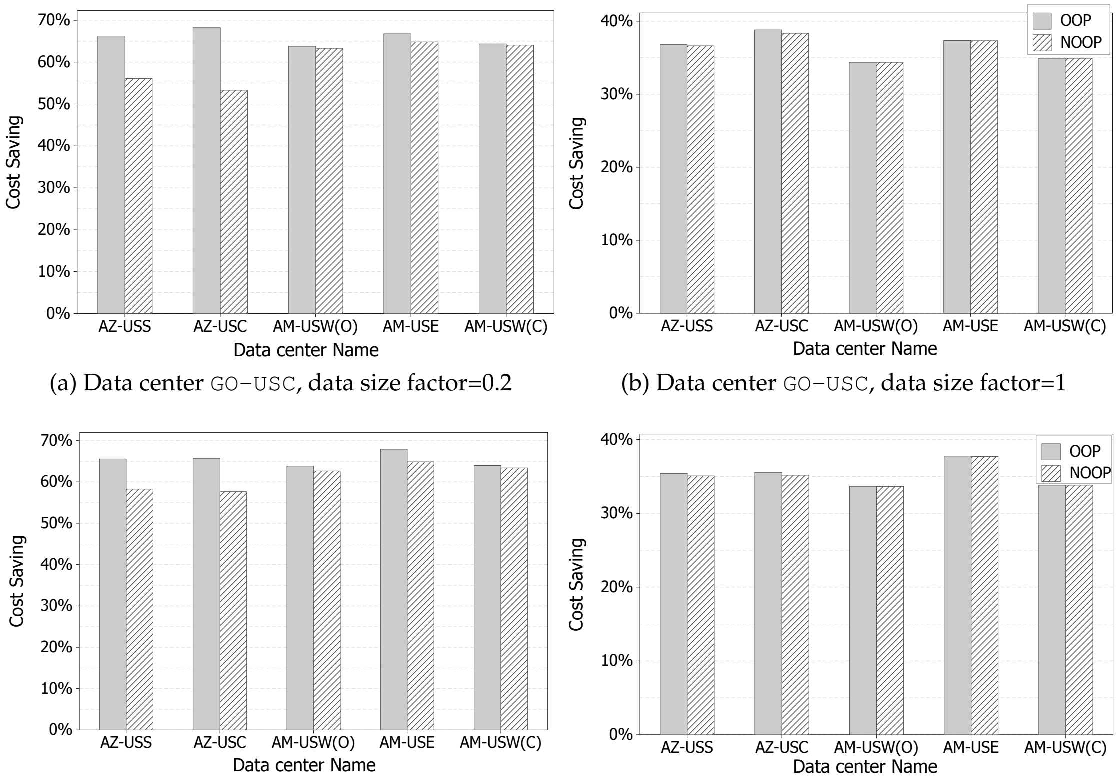
Figure 4.4: Cost saving of OOP and NOOP algorithms for two Azure DCs: AZ-USS and AZ-USC as home DCs with data size factor 0.2 and 1.

contrast, Google DCs are more suitable to pair with aforementioned home DCs compared to the DCs that belong to Amazon. The reason is that Google DCs have the cheapest cost in the storage service (i.e., Nearline) for hosting the objects in their cold-spot status. For pairing home DCs with Google DCs, OOP can save cost about 40% and 70% respectively when the data size factor is equal to 1 and 0.2. However, NOOP can reduce cost by 40% (resp. 25-28%) if the home DCs are paired with GO-USC and GO-USE (resp. GO-EUW and GO-ASE) when the data size factor is equal to 1. For data size factor=0.2, NOOP cuts the cost by around 55% (see Fig. 4.4a and 4.4c) when the considered home DCs are paired with each of Google DCs. For both algorithms, when data size factor=0.2, the different component costs (i.e., residential, delay, and migration costs) remain constant for all Google DCs; while for data size factor=1, pairing the home DCs with GO-EUW and GO-ASE incurs more delay cost compared to the case that these home DCs are paired with other Google DCs (i.e., GO-USC and GO-USE). Therefore, the latter pairing set (i.e., pairing the home DCs with GO-USC and GO-USE) gains more cost savings.

Fig. 4.4 also suggests that, when AZ-USS and AZ-USC are home DCs, Amazon DCs can be another suitable set of DCs to pair with, except AM-BRS as this DC is more expensive than home DCs in both network and storage services. OOP and NOOP gain cost savings about 32-35% and 18-20% respectively when paired with AM-USW(O), AM-USE, and AM-EUC for data size factor=1, and likewise 67%-68% and 52%-56% for data size factor=0.2. In contrast, when home DCs are paired with other Amazon DCs (i.e., AM-JAE, AM-ASA and AM-AUE), both algorithms attain lower cost savings due to two reasons: (i) these Amazon DC are more expensive in storage for backup objects and network as compared to the former subset of Amazon DCs (i.e., AM-USW(O), AM-USE, and AM-EUC), and (ii) they also impose more delay cost owing to their longer distance to home DCs.

Fig. 4.5 depicts the cost savings of OOP and NOOP when home DCs are GO-USC and GO-USE. The results show that the cost is reduced when these home DCs are paired with AZ-USS and AZ-USC that offer the same price in the network and storage. This cost reduction is 65-67% for OOP and 53-55% for NOOP when data size factor=0.2, while for data size factor=1, both algorithms approach the same cost saving (about 35-37%-see Figs. 4.5b and 4.5d). The reason behind this result for data size factor=1 is that both algorithms determine to migrate a low proportion of objects (about 25%) at roughly the same time. Moreover, the results show that OOP can save more cost by (2-3%) when GO-USC is paired with AZ-USC rather than AZ-USS (Figs. 4.5a and 4.5b) for both data size factor values. This implies that users in GO-USC and their friends (in other DCs) incur less delay cost when their read/write requests are sent to AZ-USC.

Fig. 4.5 also demonstrates that the home DCs can benefit from pairing with three Amazon DCs, but the benefit is less than pairing with the specified Azure DCs. This is because of AM-USW(O) and AM-USE are more expensive than Azure DCs in terms of



(a) Data center `GO-USC`, data size factor=0.2

(b) Data center `GO-USC`, data size factor=1

(c) Data center `GO-USE`, data size factor=0.2

(d) Data center `GO-USE`, data size factor=1

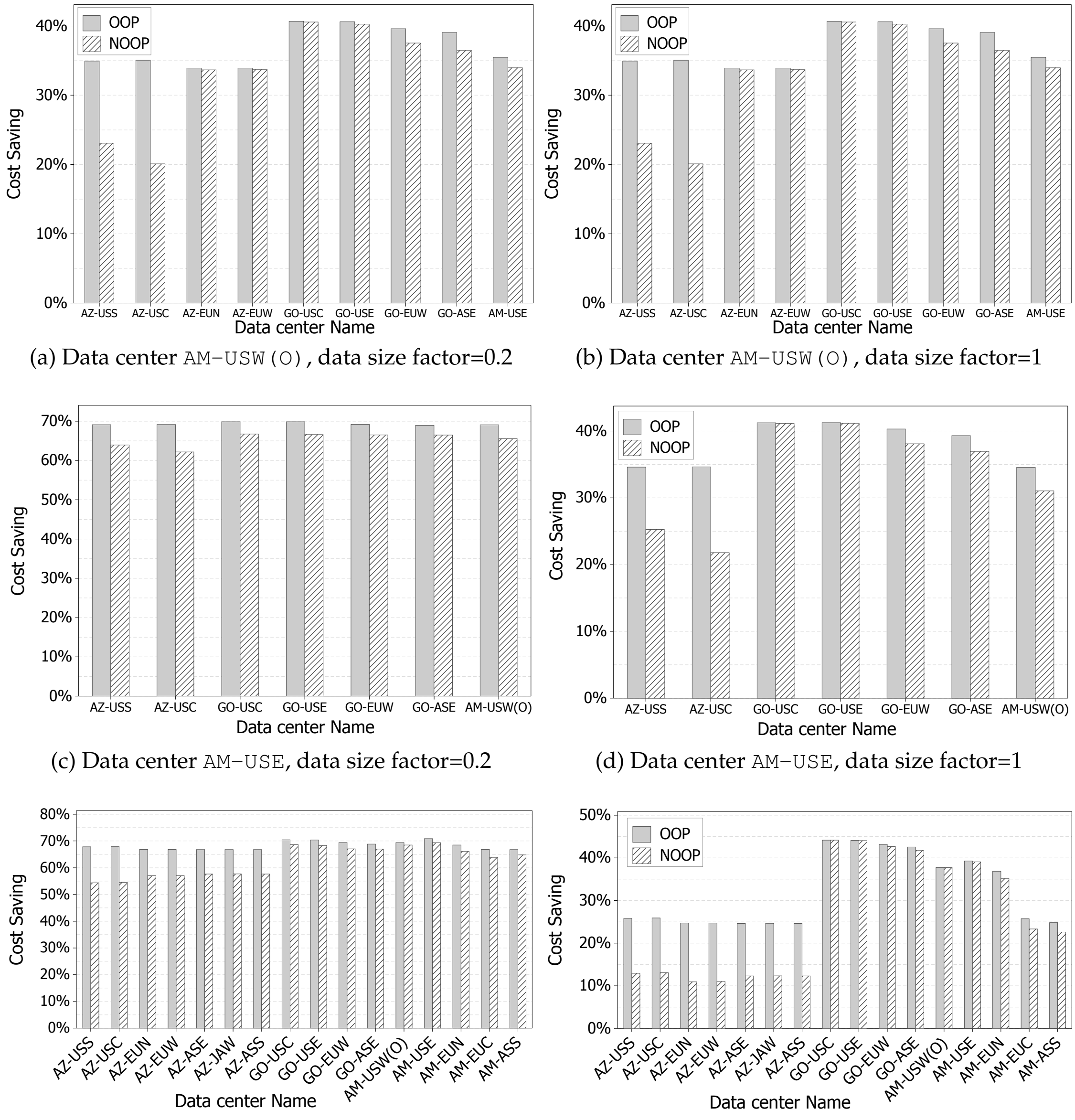
Figure 4.5: Cost saving of OOP and NOOP algorithms for two Google DCs: `GO-USC` and `GO-USE` as home DCs with data size factor 0.2 and 1.

network cost, while `AM-USW(C)` is more expensive in both storage and network costs. However, the results (Figs. 4.5c and 4.5d) depict an exception in which pairing `GO-USE` with `AM-USE` can save more cost than pairing `GO-USE` with Azure DCs. This happens because both `GO-USE` and `AM-USE` are in eastern USA, and thus read/write requests (mainly coming from this region) incur less delay cost.

Fig. 4.6 depicts the obtained cost savings from pairing each DC when home DCs are: `AM-USW(O)`, `AM-USE` and `AM-USW(C)`.

Figs. 4.6a-4.6d show that `AM-USW(O)` and `AM-USE` can benefit from pairing with at most three Azure DCs, all Google DCs, and one Amazon DC. Pairing with Google DCs can bring more cost savings than with Amazon DC [5] which in turn, save more cost than with Azure DCs (i.e., `AZ-USS` and `AZ-USS`) especially for NOOP when data size factor=1. The reason behind this is: (i) objects tend to migrate to Google DCs with the

[5] `AM-USW(O)` as a home DC is paired with `AM-USE` (Figs. 4.6a and 4.6b), and `AM-USE` as a home DC is paired with `AM-USW(O)` (Figs. 4.6c and 4.6d).



(a) Data center `AM-USW(O)`, data size factor=0.2

(b) Data center `AM-USW(O)`, data size factor=1

(c) Data center `AM-USE`, data size factor=0.2

(d) Data center `AM-USE`, data size factor=1

(e) Data center `AM-USW(C)`, data size factor=0.2

(f) Data center `AM-USW(C)`, data size factor=1

Figure 4.6: Cost saving of OOP and NOOP algorithms for three Amazon data centers: `AM-USW(O)`, `AM-USE`, and `AM-USW(C)` as home data centers with data size factor 0.2 and 1.

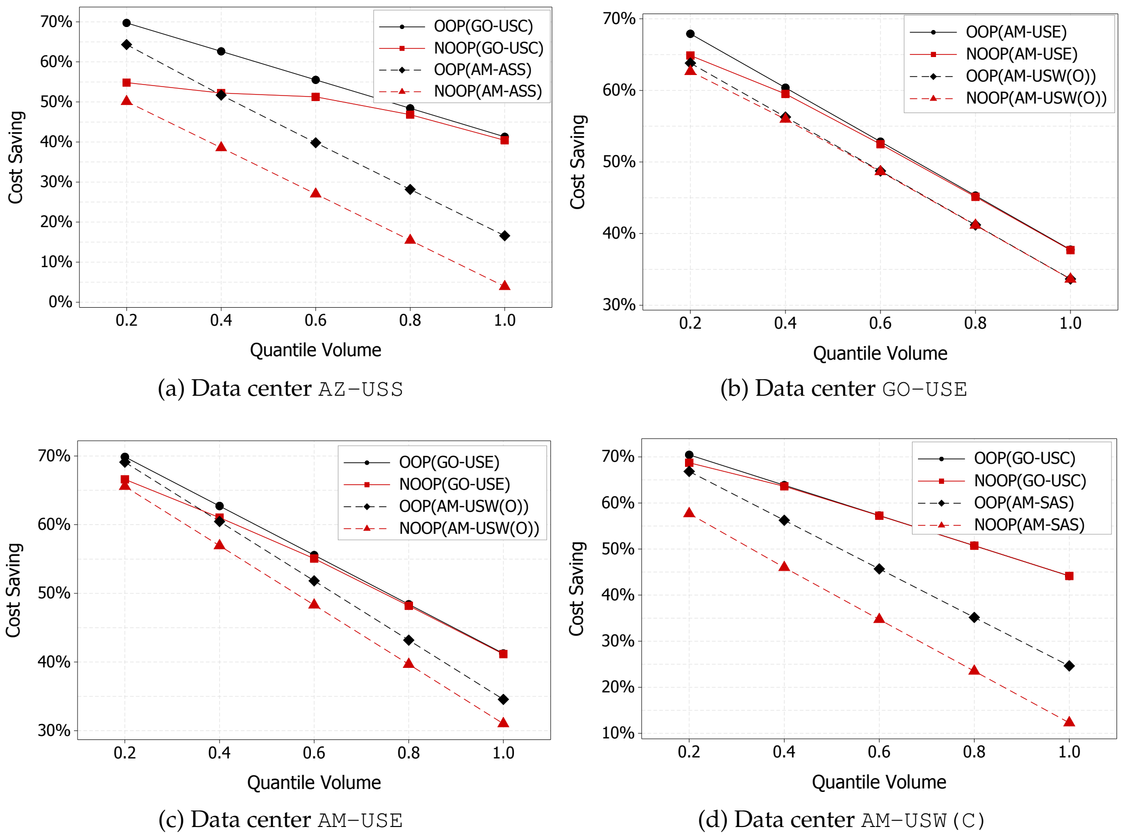
cheapest storage cost for backup objects, and (ii) Amazon DCs offer discount in network cost if the objects are migrated between two Amazon DCs. For data size factor=1, NOOP can save around 32-35% cost when `AM-USW(O)` is paired with `AM-USE` (Fig. 4.6b) or vice versa (Fig. 4.6d) by utilizing this discount, and around 20-25% when both home DCs are paired with `AZ-USS` and `AZ-USC`. The difference in cost savings of NOOP between the two pairing settings (i.e., pairing the home DCs with Amazon DC and with Azure DCs) is at most 15% (Fig. 4.6a) for the data size factor=1, and likewise at most 5% when data size factor is 0.2 (Fig. 4.6c).

Figs. 4.6e and 4.6f show that `AM-USW(C)`, as a home DC, can be paired with more DCs to save cost because its storage and network costs are substantially higher than the cost in storage and network services offered by other DCs. As it can be easily seen in the figures, the most and least profitable DCs for pairing respectively are Google and Azure DCs for both algorithms and for both values of the data size factor.

As shown in Fig. 4.6e. for OOP and NOOP, pairing the home DC `AM-USW(C)` with three Amazon DCs (i.e., `AM-USW(O)`, `AM-USE`, and `AM-EUN`) gains roughly the same cost savings than when it is paired with Google DCs. For OOP (resp. NOOP), the pairing with the remaining Amazon DCs (i.e., `AM-EUC` and `AM-ASS`) cuts the same cost (resp. more cost) as it is paired with Azure DCs. In fact, for both algorithms and for both values of the data size factor, paring with `AM-USW(O)`, `AM-USE`, and `AM-EUN` can offer more cost savings than `AM-EUC` and `AM-ASS`.

As depicted in Fig. 4.6f, for both algorithms, pairing the home DC `AM-USW(C)` with Google DCs outperforms pairing the home DC with `AM-USW(O)`, `AM-USE`, and `AM-EUN` at most by 10% in cost saving. For two pairing sets, pairing the home DC with `AM-EUC` and `AM-ASS`, and with Azure DCs, OOP gains the same cost saving in both sets, while NOOP achieves a twice cost savings in the latter pairing set in comparison to the former pairing set.

The results can be justified as the aforementioned three Amazon DCs (i.e., `AM-USW(O)`, `AM-USE`, and `AM-EUN`) benefit from the discount in network price for moving data across Amazon DCs. The amount of discount is about 3/4 of the network price for moving data out to the Internet. `AM-EUC` takes the advantage of the same amount of discount but its low profitability happens due to the small difference in the price of both storage classes (5% less for IAS and 8% more for SS) compared to these prices of the home DC `AM-USW(C)`. For `AM-ASS`, the proposed algorithms achieve a lower cost saving because the amount of discount is about 1/4 of the network price.



(a) Data center `AZ-USS`

(b) Data center `GO-USE`

(c) Data center `AM-USE`

(d) Data center `AM-USW(C)`

Figure 4.7: Cost saving of OOP and NOOP algorithms for the home DC of Azure, Google and Amazon when the data size factor is varied. The first (resp. last) two legends indicate DC with maximum (resp. minimum) cost saving when they are paired with the home DC.

The impact of data size factor value

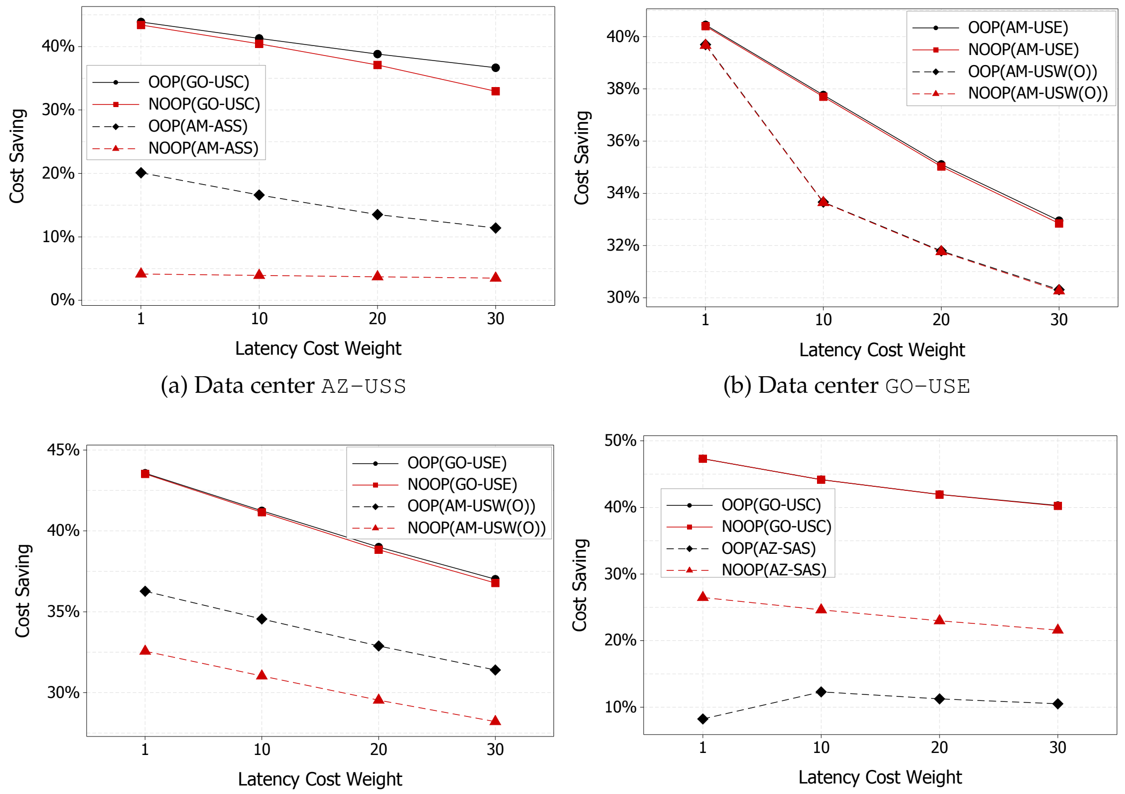
We evaluate the effect of the data size factor value by varying it from 0.2 to 1 with the step size of 0.2. The read and write requests for all data size factor values are fixed based on the default value as in Table 4.1. This setting implies that as the data size factor value is smaller, the data is more read- and write-intensive.

For the sake of brevity, from hereafter (excluding subsection 4.4.2), we report the results only for the most populated Azure DC (`AZ-USS`), Google DC (`GO-USE`), and the two most populated Amazon DCs (`AM-USE` and `AM-USW(C)`), as home DCs. We also consider the pairing of these DCs with two DCs: the ones with maximum and minimum cost savings, where the value of data size factor is 1. Note that these DCs can be easily recognized in Figs. 4.4b, 4.5d, 4.6d, and 4.6f. For example, Fig. 4.4b depicts `AZ-USS`, as a home DC, can achieve maximum (resp. minimum) cost saving when it is paired with GO-USC (resp. AM-ASS).

As shown in Fig 4.7, as the data size factor value increases, the cost saving decreases. This is because when the data size factor value is small, the network cost dominates the total cost, and the proposed algorithm exploits more difference between network prices offered by the paired DCs. On the contrary, as the value of data size factor increases, the storage cost becomes more important and thus the difference between storage prices offered by the paired DCs comes into play for optimization. We can also see that both algorithms approach the same cost saving value (referred as convergence point) in the case of maximum cost savings obtained from pairing DCs. The convergence point for each home DC is: data size factor=0.6 for GO-USE and AM-USE, and data size factor=0.8 for AM-USW(C). This implies that both algorithms decide to migrate the objects at roughly the same time, and consequently they almost achieve the same cost saving.

The impact of latency cost weight

We study the effects of *latency cost weight* by varying it from 1 to 30 on the cost performance of the algorithms on the pairing DCs as already discussed in Section 4.4.2. As shown in Fig. 4.8, when the value of *latency cost weight* increases, the cost savings gradually decrease (1-10%) for both algorithms. This is due to the fact that as the *latency cost weight* value grows, the delay cost in dual cloud-based storage increases and the impact of other parts of the total cost (i.e., residential and migration costs) diminishes. In fact, the growth in the latency cost weight reduces the potential for exploiting the difference cost between storage and network, which leads in the cost saving reduction. In summary, for both algorithms, as the *latency cost weight* value increases, the delay cost comes as a vital factor in the cost saving, while the impact of difference between storage and network costs on the cost saving decrease. As a result, a dual cloud-based storage prefers to store more objects locally.



(a) Data center `AZ-USS`

(b) Data center `GO-USE`

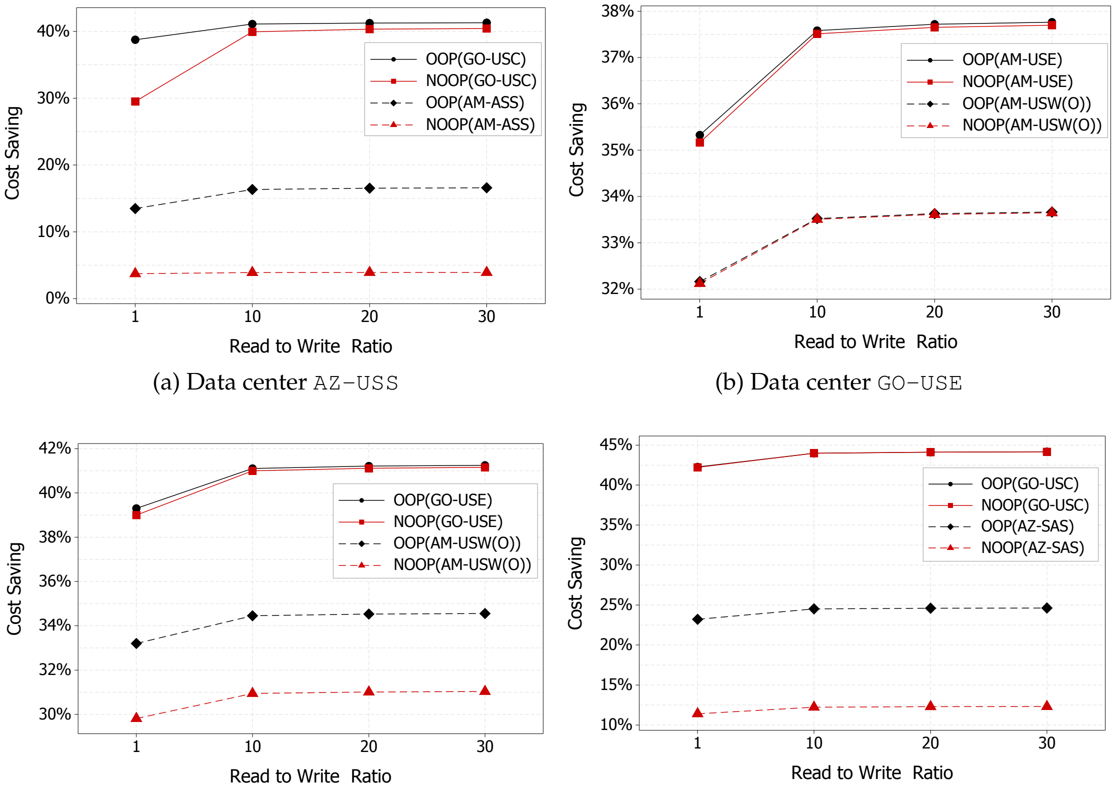
(c) Data center `AM-USE`

(d) Data center `AM-USW(C)`

Figure 4.8: Cost saving of OOP and NOOP algorithms for the home DC of Azure, Google and Amazon when the latency cost weight is varied. The first (resp. last) two legends indicate DC with the maximum (resp. minimum) cost saving when they are paired with the home DC.

The impact of read to write ratio

We explore the effects of *read to write ratio* by varying it from 1 to 30 on the cost performance of the proposed algorithms on pairing with DCs discussed in the previous section. Fig. 4.9 depicts that, as the ratio of read to write increases, the cost savings gradually increase at most 3% for OOP and 10% for NOOP. This indicates more exploitation of pricing differences in the case of network costs between the two paired DCs as the ratio grows. Also, as it can be seen, for the paired DCs with minimum cost saving, the OOP algorithm gains 3-10% more cost saving than NOOP, except for the home DC `GO-USE`. On the contrary, for the paired DCs with maximum cost saving, both algorithms approach the same cost saving. This is because the total cost is dominated by the storage cost, where in all paired DCs, Google DCs offer the same storage cost. For the data size factor values



(a) Data center `AZ-USS`

(b) Data center `GO-USE`

(c) Data center `AM-USE`

(d) Data center `AM-USW(C)`

Figure 4.9: Cost saving of OOP and NOOP algorithms for the home DC of Azure, Google and Amazon when the write to read ratio is varied. The first (resp. last) two legends indicate DC with the maximum (resp. minimum) cost saving when they are paired with the home DC.

between 0.2 and 0.6 in the case of the paired DCs with maximum cost saving (not shown in results), OOP outperforms NOOP in the cost savings. The reason is that the total cost is dominated by the network cost and home DCs are paired with different DCs (except Google DCs) in terms of network cost.

The impact of the access pattern of reads/writes on objects

We finally investigate the cost performance of the proposed algorithms when the access pattern of reads/writes on the objects follows different distributions, i.e., Normal and Random (Recall that the access pattern of reads/writes on objects follows a Longtail distribution [16], as the default). Although Normal and Random access patterns are not compatible with hot- and cold-spot status definition for objects, we investigate the im-

Table 4.3: Cost saving of OOP and NOOP (shown in bracket), and the potential DCs pairing with four home DCs when the acess patterns on the objects are Normal and Random.

Home DC	Access pattern	Azure	Google	Amazon
AZ-USS	Normal	-	2-3%[-(11-18)%]	2-4%[-(3-7)%]
	Random	-	2-3%[-(1-13)%]	2-3%[-(3-5)%]
	-	-	All DCs	USW(O,C),USE
GO-USE	Normal	4%[-(7)%]	-	4-5% [(2-4)%]
	Random	4%[-(2)%]	-	3-4%[(1-2)%]
	-	USS,USC	-	USW(O,C),USE
AM-USE	Normal	4%[0%]	4%[1-3%]	4%[2%]
	Random	4%[-5%]	3-4%[-(0-4)%]	3%[1%]
	-	USS,USC	All DCs	USW(O)
AM-USW(C)	Normal	7-8%[-3%]	4-10%[3-9%]	6-9%[6-7%]
	Random	7%[1%]	3-9%[3-6%]	6-8%[5-6%]
	-	USS,USC	All DCs	USW(O),USE

pact of these patterns separately to find out whether the algorithms can still cut cost. And if so, to what extent? We conduct the experiment for four home DCs: AZ-USS, GO-USE, AM-USE, and AM-USW(C). We use the default value of read to write ratio, latency cost weight, and data size factor, as shown in Table 4.1. Table 4.3 gives the cost savings for OOP and NOOP (shown in brackets).

As shown in Table 4.3, OOP can save cost for all home DCs under both access patterns. For AZ-USS, the algorithm saves more cost if it is paired with Amazon DCs rather than Google DCs under Normal access pattern by incurring less migration cost. For GO-USE, OOP cuts the cost by 4% if it is paired with Azure DCs and cuts slightly more costs with Amazon DCs under Normal access pattern. In contrast to the two discussed home DCs, AM-USE and AM-USW(C), as home DCs, can achieve cost reduction by pairing with DCs of all cloud providers (i.e., Microsoft Azure, Google and Amazon). The cost saving obtained from these home DCs with Amazon DCs (i.e., AM-USW(O) and AM-USE, see rows 3 and 4 under column "Amazon" in Table 4.3) approaches the one achieved through pairing with Google (i.e., all Google DCs) and Azure DCs (i.e., AZ-USS and AZ-USC). This is because home DCs exploit the discount on the network cost when data is moved across two Amazon DCs.

From the results of the OOP algorithm, we observe that the cost saving obtained from pairing potential DCs (see their name in rows 3 and 4 of Table 4.3) with home DC AM-USW(C) is approximately two times more than that achieved by pairing these DCs with home DC AM−USE. The reason is that the difference between storage and network prices offered by AM−USW(C) and its paired DCs is substantially high, while for AM−USE is comparatively low. The result for AM−USW(C) shows that the cost savings are still considerable under both Normal and Random access patterns when a home DC offers services with prices that are significantly different from those prices provided by other DCs. We can find similar DCs (i.e., a significant difference between two DCs in terms of price) in Asia-Pacific and Brazil regions as well. The companies in these regions can leverage pairing of their DCs with DCs in other regions to reduce cost not only for objects with hot- and cold-spot status, but also for objects accessed under Normal and Random access patterns.

Contrary to OOP, NOOP achieves less cost savings and in some circumstances this algorithm is not even cost-efficient. Under both access patterns, NOOP is not profitable when AZ−USS is paired with DCs of Google and Amazon. NOOP triggers more migrations that increase the total cost. Moreover, the results indicate that pairing GO−USE with Azure DCs is not cost-efficient (-7% for Normal and -2% for Random access pattern), while pairing of GO−USE with Amazon DCs can still cut cost by 4% for Normal and 2% for Random access pattern. NOOP brings more cost saving for pairing both of the Amazon home DCs with the other potential DCs, as compared to the pairing of home DCs such as AZ−USS and GO−USE with the other DCs. For instance, AM−USW(C) achieves cost savings under all circumstances except pairing with Azure DCs under Normal access pattern. As already mentioned, this is because AM−USW(C) offers more expensive storage and network as compared to its paired DC, and object migration between these paired DCs under both access patterns is cost-effective.

In summary, according to the experimental results, one can conclude that OOP is cost efficient under both access patterns. NOOP is not profitable for pairing Azure DCs with Google and Amazon DCs, while it is cost-effective in other pairing situations, like Amazon DCs with Azure, Google and Amazon DCs, as well as pairing Google DC with Amazon DC. We realized that NOOP can be profitable especially when the paired DCs can exploit the discounted price in the network cost for moving data across DCs belonging to the same provider. This discount is currently offered by Amazon, and it is likely that Google and Azure would offer their customers the same discount in the near future.

4.5 Summary

Choosing storage options across CSPs for time-varying workload is critical for optimizing data management cost. In particular, issues such as when should an object be migrated and in which storage class it should be stored need to be addressed. We consider a fine-grained architecture and propose two algorithms that determine optimal (resp. near optimal) placement of the object with (resp. without) the knowledge of the future workload. Such a fine-grained architecture provides evidence that one can achieve cost savings in Geo-replicated system where a home DC can be paired with different DCs during the lifetime of the object. This is investigated in the next two chapters.

Chapter 5

Cost Optimization across Cloud Storage Providers: Offline and Online Algorithms

Cloud Storage Providers (CSPs) offer geographically data stores providing several storage classes with different prices. An important problem facing by cloud users is how to exploit these storage classes to serve an application with time-varying workloads at minimum cost. This cost consists of residential cost (i.e., storage, Put and Get costs) and potential migration cost (i.e., network cost). This chapter addresses this problem and first proposes the optimal offline algorithm that leverages dynamic and linear programming techniques with the assumption of available knowledge of workload on objects. Due to the high time complexity of this algorithm and its requirement for a priori knowledge, it also includes two online algorithms that make a trade-off between residential and migration costs and dynamically select storage classes across CSPs. The first online algorithm is deterministic with no need of any knowledge of workload and incurs no more than $2\gamma - 1$ *times of the minimum cost obtained by the optimal offline algorithm, where* γ *is the ratio of the residential cost in the most expensive data store to the cheapest one in either network or storage cost. The second online algorithm is randomized that exploits available future workload information for* w *time slots. This algorithm incurs at most* $1 + \frac{\gamma}{w}$ *times the optimal cost. The effectiveness of the proposed algorithms is demonstrated via simulations using a workload synthesized based on the Facebook workload.*

5.1 Introduction

AMazon S3, Google Cloud Storage (GCS) and Microsoft Azure as leading CSPs offer different types of storage (i.e., blob, block, file, etc.) with different prices

This chapter is derived from: **Yaser Mansouri**, Adel Nadjaran Toosi, and Rajkumar Buyya, "Cost Optimization for Dynamic Replication and Migration of Data in Cloud Data Centers," *IEEE Transactions on Cloud Computing (TCC)*, DOI:10.1109/TCC.2017.2659728, 2017.

for at least two classes of storage services: Standard Storage (SS) and Reduced Redundancy Storage (RRS) is an Amazon S3 storage option that enables users to reduce their cost with lower levels of redundancy compared to SS. Each CSP also provides API commands to retrieve, store and delete data through network services, which imposes in- and out-network cost on an application. In leading CSPs, in-network cost is free, while out-network cost (network cost for short) is charged and may be different for providers. Moreover, data transferring across DCs of a CSP (e.g., Amazon S3) in different regions may be charged at lower rate (henceforth, it is called reduced out-network cost). Table 5.1 summarizes the prices for network and storage services of three popular CSPs in the US west region, which shows significant price differences among them. This diversification plays a central role in the cost optimization of data storage management in cloud environments. We aim at optimizing this cost that consists of *residential* cost (i.e., storage, *Put*, and *Get* costs) and potential *migration* cost (i.e., network cost).

The cost of data storage management is also affected by the expected workload of an object. There is a strong correlation between the object workload and the age of object, as observed in online social networks (OSNs) [122]. The object might be a photo, a tweet, a small video, or even an integration of these items that share similar read and write access rate pattern. The object workload is determined by how often it is read (i.e., *Get* access rate) and written (i.e., *Put* access rate). The *Get* access rate for the object uploaded to a social network is often very high in the early lifetime of the object and such object is said to be read intensive and in *hot-spot* status. In contrast, as time passes, the *Get* access rate of the object is reduced and it moves to the *cold-spot* status where it is considered as storage intensive. A similar trend happens for the *Put* workload of the object; that is, the *Put* access rate decreases as time progresses. Hence, OSNs utilize more network than storage in the early lifetime of the object, and as time passes they use the storage more than network.

Therefore, (i) with the given time-varying workloads on objects, and (ii) storage classes offered by different CSPs with different prices, acquiring the cheapest network and storage resources in the appropriate time of the object lifetime plays a vital role in the cost optimization of the data management across CSPs. To tackle this problem, cloud users are required to answer two questions: (i) which storage class from which CSP should

Table 5.1: Cloud storage pricing as of June 2015 in different DCs.

CSP	Amazon†	Amazon‡	Google	Azure
SS (GB/Month)	0.0330	0.030	0.026	0.030
RRS (GB/Month)	0.0264	0.024	0.020	0.024
Out-Network	0.09	0.09	0.12	0.087
Reduced Out-Network	0.02	0.02	0.12	0.087
Get (Per 100K requests)⋆	4.4	4	10	3.6
Put (Per 1K requests)	5.5	5	10	0.036

⋆The price of *Put* and *Get* is multiplicaton of 10^{-3}. All prices are in US dollar.

† Amazon's DC in California. ‡ Amazon's DC in Ireland.

host the object (i.e., placing), and (ii) when the object should probably be migrated from a storage class to another owned by the similar or different CSPs.

Recently, several studies take advantage of price differences of different resources in intra- and inter-cloud providers, where cost can be optimized by trading off compute vs. storage [86], storage vs. cache [23, 131], and cost optimization of data dispersion across cloud providers [177, 179]. None of these studies investigated the trade off between network and storage cost to optimize cost of replication and migration data across multiple CSPs. In addition, these approaches heavily rely on workload prediction. It is not always feasible and may lead to inaccurate results, especially in the following cases: (i) when the prediction methods are deployed to predict workloads in the future for a long term (e.g., a year), (ii) for startup companies that have limited or no history of demand data, and (iii) when workloads are highly variable and non-stationary.

Our study is motivated by these pioneer studies as none of them can simultaneously answer the aforementioned questions (i.e., *placements* and *migration times* of objects). To address these questions, we make the following key **contributions**:

- First, by exploiting dynamic programming, we formulate offline cost optimization problem in which the optimal cost of storage, *Get*, *Put*, and migration is calculated where the exact future workload is assumed to be known *a priori*.
- Second, we propose two online algorithms to find near-optimal cost as shown experimentally. The first algorithm is a *deterministic online algorithm* with the *competitive ratio* (CR) of $2\gamma - 1$, where γ is the ratio of the residential cost in the most expensive DCs to the cheapest ones either in storage or network price. The second algorithm is a *randomized online algorithm* with the CR of $1 + \frac{\gamma}{w}$, where w is the available look ahead window size for the future workload. We also analyse the

cost performance of the proposed algorithms in the forms of CR that indicates how much cost in the worst case the online algorithms incur as compared to the offline algorithm.

- In addition to the theoretical analysis, an extensive simulation-based evaluation and performance analysis of our algorithms are provided in the CloudSim simulator [35] using the synthesized workload based on Facebook workload [16].

The rest of the chapter is organized as follows. System characteristics, problem formulation and optimization cost problem are presented in Section 5.2. Section 5.3 characterizes the optimal offline algorithm for optimizing cost via dynamic programming technique. We discuss two online algorithms and provide performance guarantees in Section 5.4. Section 5.5 presents experimental evaluation results and Section 5.6 concludes the chapter.

5.2 System Model and Problem Definition

We briefly discuss challenges and objectives of the system, and then based on which we formulate a data storage management (data management for short) cost model. Afterwards, we define an optimization problem based on the cost formulation and system's constraints.

5.2.1 Challenges and Objectives

We assume that the data application includes a set of geographically distributed key-value objects. An object is an integration of items such as photos or tweets that share a similar pattern in the *Get* and *Put* access rate. In fact an object in our model is analogous to the *bucket* abstraction in Spanner [52] and is a set of contiguous keys that show a common prefix. Based on the users' needs, the objects are replicated at Geo-distributed DCs located in different regions. Each DC consists of two types of servers: computing and storage servers.

A computing server accommodates various types of VM instances for application users. A storage server provides variety of storage forms (block, key/value, database, etc.) to users charged at the granularity of megabytes to gigabytes for very short billing periods (e.g., hours). These servers are connected by high speed switches and network, and the data exchange between VMs within DC is free. However, users are charged for data transfer out from DC on a per-data size unit as well as a nominal charge per a bulk of *Get*s and *Put*s. We consider this charging method followed by most commercial CSPs in the system model.

The primary objective of the system is to optimize cost using object replication and migration across CSPs while it strives to serve the *Get* and *Put* in the latency constraint specified by the application. Providing all these objectives introduces the following challenges. (i) Inconsistency between objectives: for example, if the number of replicas decreases, then the *Get* and *Put* latency can increase while storage cost reduces, and vice versa. (ii) Variable workload of objects: when the *Get* and *Put* access rate is high in the early lifetime of an object, the object must be migrated in a DC with a lower network cost. In contrast, as *Get* and *Put* access rate decreases over time, the object must be migrated to a DC with a lower storage cost. (iii) Discrepancy in storage and network prices across CSPs: this factor complicates the primary objective, and we clarify it in the below example.

Suppose, according to Table 5.1, an application stores an object in Azure's DC when the object is in hot-spot because it has the cheapest out-network cost. Assume that after a while the object transits to its cold-spot and it must migrate to two new DCs: Amazon's DC (Ireland) and Google's DC. The object migration from Azure's DC to Amazon's DC (Ireland) is roughly 4 times (0.02 per GB vs. 0.0870 per GB) more expensive than as if the object was initially stored in Amazon's DC (California) instead of Azure's DC. The object migration from Azure's DC to Google's DC is roughly the same in the cost (0.087 per GB vs. 0.09 per GB) as if the object was initially stored in Amazon's DC (California) instead of Azure's DC. This example shows that the application can benefit from the reduced out-network price if the object migration happens between two Amazon DCs. In one hand, as long as the object is stored in Azure's DC, the application benefits from the cheapest out-network cost, while it is charged more when the object is migrated to a new DC. On the other hand, if the object is stored in Amazon's DC (California), the application saves more cost during migration but incurs more out-network and storage costs. Thus, in addition to storage and out-network costs, the reduced out-network cost plays an important factor in the cost optimization for time-varying workloads.

Table 5.2: Symbols definition

Symbol	Meaning
D	A set of DCs
K	A set of regions
$S(d)$	The storage cost of DC d per unit size per unit time
$O(d)$	Out-network price of DC d per unit size
$t_g(d)$	Transaction price for a bulk of *Get* (Read)
$t_p(d)$	Transaction price for a bulk of *Put* (Write)
T	Number of time slots
$v(t)$	The size of the object in time slot t
$r^k(t)$	Read requests number for the object from region k in time slot t
$w^k(t)$	Write requests number for the object from region k in time slot t
r	Number of replicas stored across DCs for each object
ρ	The number of DCs in destination set, excluding the intersection of the source and destination sets
γ	The ratio of the residential cost in the most expensive DCs to the cheapest ones in time slot $t \in [1...T]$
λ	The ratio of the reading volume of the objects to the objects size
$\alpha^d(t)$	A binary variable indicates whether the object is in DC d in time slot t or not
$\beta^{k,d}(t)$	A variable indicates the fraction of requests from region k directed to DC d hosting a replica of an object in time slot t
$C_R(.)$	Residential cost
$C_M(.)$	Migration cost
L	A upper bound of delay on average for Get/Put requests to receive response
T_{lp}	Time complexity of linear programming
α	The set of all r-combinations of DCs
w	The size of available look-ahead window for the future workload information

5.2.2 Preliminaries

In this section, we give some definitions, which are used throughout the chapter. The major notations are also summarized in Table 5.2.

Definition 5.1. *(DC Specification): The system model is represented as a set of independent DCs D where each DC $d \in D$ is located in region $k \in K$. Each DC d is associated with a tuple of four cost elements. (i) $S(d)$ denotes the storage cost per unit size per unit time (e.g., bytes per hour) in DC d. (ii) $O(d)$ defines out-network cost per unit size (e.g., byte) in DC d. (iii) $t_g(d)$ and $t_p(d)$ represent transaction cost for a bulk of Get and Put requests (e.g., per number of requests) in DC d, respectively.*

Definition 5.2. *(Object Specification): Assume the application contains a set of objects during each time slot $t \in [1...T]$. Let $r^k(t)$ and $w^k(t)$, respectively, be the number of Get and Put requests for the object with size $v(t)$ from region k in time slot t.*

The objective is to choose placement of object replicas, and the fraction of $r^k(t)$ (not the fraction of $w^k(t)$ since each *Put* request must be submitted to all replicas) that should be served by each replica so that the application cost including storage, *Put*, and *Get* costs for objects as well as their potential migration cost among DCs is minimized. We thus define *replication variable*, *requests distribution variable*, and *application cost* as follows.

Definition 5.3. *(Replication Variable): $\alpha^d(t) \in \{0, 1\}$ indicates whether there is a replica of the object in DC d in time slot t ($\alpha^d(t) = 1$) or not ($\alpha^d(t) = 0$). Thus, $\sum_{d \in D} \alpha^d(t) = r$. We also denote $\vec{\alpha}(t)$ as a vector of $\alpha^d(t)$s shows whether a DC d hosting a replica or not in the slot time t.*

Definition 5.4. *(Request Distribution Variable): The fraction of Get requests issued from region k to DC d hosting the object in time slot t is denoted by $\beta^{k,d} \in (0, 1)$. Thus, $\sum\limits_{k \in K} \sum\limits_{d|\alpha^d(t)=1} \beta^{k,d}(t) = 1$. We denote $\vec{\beta}(t)$ as a matrix of $|K| \times r$ represents the fraction of Get requests issued from region $k \in K$ to each replica.*

Definition 5.5. *(Storage Cost): The storage cost of an object in time slot t is equal to the storage cost of all its replicas in DCs d in time slot t. Thus, we have*

$$\sum_{d|\alpha^d(t)=1} S(d) \times v(t). \tag{5.1}$$

Definition 5.6. *(Get Cost): The Get cost of the object in time slot t is the cost of Get requests issued from all regions and the network cost for retrieving the object from DCs. Therefore,*

$$\sum_{k \in K} \sum_{d|\alpha^d(t)=1} \beta^{k,d} \times r^k(t) \times (t_g(d) + v(t) \times O(d)). \tag{5.2}$$

To keep replicas consistent, we use a simple policy that leverages the primary advantages of eventual consistency setting, which is appropriate for OSNs [177]. Thus, first, to capitalize on the network services cost, we select DC $d|\alpha^d(t) = 1$ with the minimum network cost $O(d)$ so that the upper bound of delay for *Put* requests is met[1]. Then, *Put* requests issued for the object are sent to this DC and the application incurs only *Put* transaction cost as in-network cost is free (called *initial Put cost*). Second, the other replicas are kept consistent by either DC d or another DC, hosting the replica, with the lowest network cost without considering delay constraint. This DC is responsible for data propagation and is called *propagator DC*, that is, $d_p = \min\limits_{d'|\alpha^{d'}(t)=1} (O(d'))$ (called *consistency cost*).

[1] From this point onward, whenever the migration or data transfer happens between two Amazon DCs, the reduced network cost is considered rather than the network cost.

Note that if any other DC rather than the initial selection (i.e., DC d) is selected as the propagator DC, then the application incurs one extra cost of out-network between these two DCs. Thus, in addition to the cost of *Put* transactions, the application is charged for the network cost of data from the propagator DC. For example, as illustrated in Fig. 5.1, assume that the object has been already replicated at four DCs in the European region. Let the user issue a *Put* request into Google DC (GDC) (i.e., DC d). Based on the above strategy, Azure DC (ADC) in the Netherlands is selected as the propagator DC (i.e., DC d_p) because it has the cheapest network cost among these four DCs and is responsible of updating objects in two other DCs. Based on the discussed policy, we formally define the *Put* cost as below

Definition 5.7. *(Put Cost): The Put cost of the object in time slot t is the cost of Put requests issued by all regions and the propagation cost for updating replicas of the object. Thus,*

$$\begin{aligned} & c(d, d_p) + \sum_{k \in K} [(w^k(t) \times t_p(d) + \\ & \sum_{d' | \alpha^{d'}(t)=1 \setminus \{d, d_p\}} w^k(t) \times (t_p(d') + v(t) \times O(d_p))], \end{aligned} \tag{5.3}$$

where (i) $c(d, d_p)$ is the transfer cost between d and d_p and is equal to $\sum_k w^k(t) \times (v(t) \times O(d) + t_p(d_p))$, and (ii) d' is a DC, excluding d and d_p, that hosts a replica. Note that if $d = d_p$, then $c(d, d_p) = 0$. In the above equation, $w^k(t) \times t_p(d)$ is *initial Put cost* and the second sigma is *the consistency cost*.

Definition 5.8. *(Residential Cost): The residential cost of the object in time slot t is the summation of its storage, Get, and Put costs (Equs. 5.1 - 5.3) and is denoted by $C_R(\vec{\alpha}(t), \vec{\beta}(t))$.*

The best set of DCs to replicate an object can differ in t and $t-1$. In other words, $\vec{\alpha}(t-1)$ and $\vec{\alpha}(t)$ are different. This happens because the object size, the number of requests, and the source of requests to conduct *Get*s or *Put*s would change in different time slots. Thus, if the object is in hot-spot, it is more cost-effective to replicate it at a DC with a lower network cost as long as the object is in this state. In contrast, if the object transits from hot-spot to cold-spot and grows in size, it is more profitable to migrate the object to DC(s) with a lower storage cost. Object replication based on the status of the object across DCs

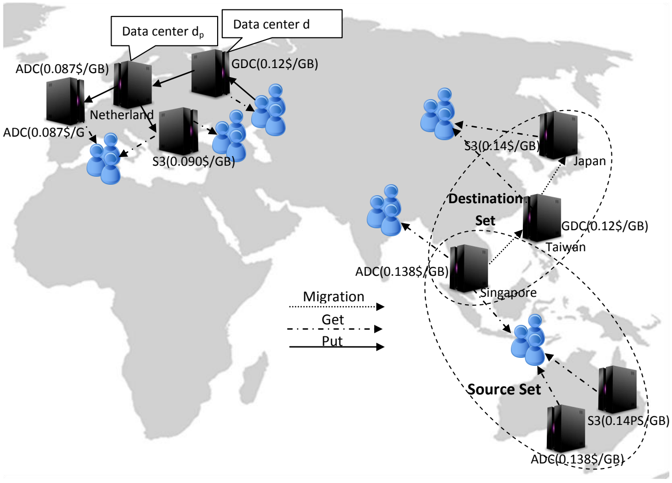


Figure 5.1: Object updating in Europe region and the object migration in Asia-pacific region.

imposes a *migration cost* on the application imposes *migration cost*. To minimize it, the object should be migrated from the DC with the lowest network cost. We thus consider two sets of DCs: one set contains DCs that the object must be migrated from (called *source set*), and the other set that the object must be migrated to (called *destination set*). The policy, first, finds the DC with the lowest network cost in each set and the first replica migration happens between these two selected DCs. For other replicas, replication is carried out from the cheapest of these two.

To clarify this simple method, we describe an example as shown in Fig. 5.1. Assume the object must be migrated from DCs in the source set to those in the destination set. Since ADC in Singapore has the cheapest network price among DCs in the source set, it is responsible to send the object to GDC in Taiwan. This is because this GDC has the lowest rate in the network price in the destination set. Then, S3 in Japan receives the object from GDC since it is cheaper than ADC in Singapore. Based on the above discussion, the migration cost is defined as below.

Definition 5.9. *(Migration Cost): If $\vec{\alpha}(t) \neq \vec{\alpha}(t-1)$, the application incurs the migration cost in time slot t that is the multiplication of the object size and the out-network cost of the DC hosting the object in time slot $t-1$. The migration cost of object denoted by $C_M(\vec{\alpha}(t-1), \vec{\alpha}(t))$ includes the migration cost from the DC $d_s = \min\limits_{d|\alpha^d(t-1)=1} O(d)$ to $d_d = \min\limits_{d|\alpha^d(t)=1} O(d)$ and the object is then replicated from the DC $d_{pm} = min(O(d_s), O(d_d))$ to all remaining DCs in the destination set if they are not in the source set. We denote by ρ as the number of DCs in destination set, excluding* *the intersection of the source and destination sets. Thus,*

$$C_M(\vec{\alpha}(t-1), \vec{\alpha}(t)) = v(t) \times (O(d_s) + O(d_{pm}) \times (\rho - 1)). \tag{5.4}$$

The discussed policy uses the *stop and copy* technique in which the application is served by the source set for Gets and destination set for Puts during migration [65]. This technique is used by the single cloud system such as HBase[2] and ElasTraS [56], and in Geo-replicated system [169]. As we desire to minimize the monetary cost of migration, we use this technique in which the amount of data moved is minimal as compared to other techniques leveraged for live migration at shared process level of abstraction[3]. We believe that this technique does not affect our system performance due to (i) the duration of migration for transferring a bucket (at most 50MB, the same as in Spanner [52]) among DCs is considerably low (i.e., about a few seconds), and (ii) most of *Gets* and *Puts* are served during the hot-spot status, and consequently the access rate to the object during the migration, which is happening in the cold-spot status, is considerably low based on the access pattern. We point out this with more details in Section 5.5 [4].

Now, we define the total cost of the object in time slot t based on Equs. 5.1- 5.4 as:

$$C(\vec{\alpha}(t), \vec{\beta}(t)) = C_R(.) + C_M(.) \tag{5.5}$$

Besides the cost optimization, satisfying the low latency response to Put/Get requests is a vital performance measure for the application. Our model respects the latency Service Level Objective (SLO) for Get/Put requests, and the latency for a Get/Put request is calculated by the delay between the time a request is issued and the time acknowledgement is received. Since the Get/Put requests time for small size objects is dominated by the network latency, similar to [177] and[175], we estimate latency by the Round Trip Time (RTT) between the source and destination DCs. Let $l(k, d')$ denote this latency, and L define the upper bound of delay for Get/Put requests on average to receive response. We generally define the latency constraint for Get and Put requests as a constraint $l(d, d') \leq L$, where d

[2] Apache HBase. https://hbase.apache.org/book.html

[3] In transactional database in the of cloud, to achieve elastic load balancing, techniques such as *stop and copy*, *iterative state replication*, and *flush and migrate* in the process level are used. The interested readers are refereed to [65] and [58].

[4] Note that our system is not designed to support database transactions, and this technique just inspired from this area.

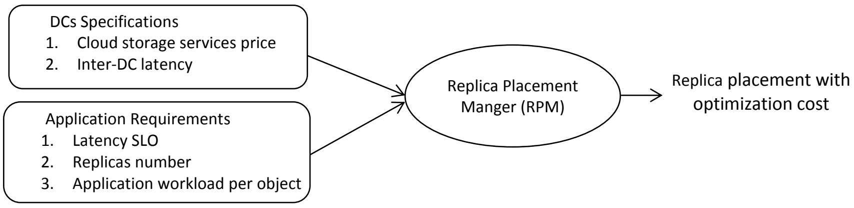

DCs Specifications
1. Cloud storage services price
2. Inter-DC latency
Application Requirements
1. Latency SLO
2. Replicas number
3. Application workload per object
Replica Placement Manger (RPM)
Replica placement with optimization cost

Figure 5.2: Overview of systems's inputs and output.

stands for the associated DC in the region k. This performance criterion will be integrated in the cost optimization problem discussed in the following section.

We also make some assumptions in the case of occurring failure and conducting *Put*/*Get* requests in the system. It is assumed that DCs are resistant to individual failures and communication links between DCs are reliable due to using redundant links [52]. In our system, the *Put*/*Get* is considered as a *complete request* once the request successfully conducted on one of the replicas. For the *Put*, this assumption suffices due to durability guarantees offered by the storage services. During migration process, if either source or destination DC fails, then the system can either postpone data migration for a limited time or re-execute the algorithms without considering the failed DC(s).

5.2.3 Optimization Problem

Given the system's input and the above cost model, we define the objective as the determination of the value of $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ in each time slot so that the overall cost for all objects during $t \in [1...T]$ is minimized. We define the overall cost optimization problem as:

$$\min_{\vec{\alpha}(t),\vec{\beta}(t)} \sum_{t} C(\vec{\alpha}(t),\vec{\beta}(t)) \tag{5.6}$$

s.t. (repeated for $\forall t \in [1...T], \forall d \in D$ and $\forall k \in K$)

(a) $\sum_{d \in D} \alpha^d(t) = r, \;\; \alpha^d(t) \in \{0,1\}$

(b) $\sum_{k \in K} \sum_{d|\alpha^d(t)=1} \beta^{k,d}(t) = 1, \;\; \beta^{k,d}(t) \in (0,1)$

(c)$\beta^{k,d}(t) \leq \alpha^d(t),$

(d) $\frac{\sum_{k \in K} \sum_{d \in D} \alpha^d(t) \times r^k \times l(k,d)}{\sum_{k \in K} r^k} \leq L,$

(e) $l(k,d') \leq L, d' = \min\limits_{d|\alpha^d(t)=1} O(d) \;\; and \;\; \forall \;\; Put$ request.

In the above optimization problem, constraint (a) indicates that only r replicas of the object exist in each time slot t. Constraint (b) ensures that all requests are served, and constraint (c) guarantees that each request for the object is only submitted to the DC hosting the object. Constraints (d) and (e) enforce the average response time of *Get* and *Put* requests in range of L respectively.

To solve the above optimization problem, in the following, we propose three algorithms as part of our Replica Placement Manager (RPM) system. As shown in Fig. 5.2, RPM uses these algorithms to optimize cost based on two inputs: DCs and application requirements.

5.3 Optimal Offline Algorithm

To solve the *Cost Optimization Problem*, we should find values $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ so that the overall cost in Equ. (5.6) is minimized [5]. So, we propose a dynamic programming algorithm to find optimal placement of replicas (i.e., $\vec{\alpha}^*(t)$) and optimal distribution of requests to replicas (i.e., $\vec{\beta}^*(t)$) for all objects during $t \in [1...T]$. Based on the above problem definition, $\vec{\beta}(t)$ in time slot t can be simply determined using a linear program once the value of $\vec{\alpha}(t)$ is fixed.

Let $\vec{\alpha} = \{\vec{\alpha}^1, \vec{\alpha}^2, .., \vec{\alpha}^i, .., \vec{\alpha}^{\binom{|D|}{r}}\}$ denote all r-combinations of distinct DCs can be chosen from D (i.e, $|\vec{\alpha}| = \binom{|D|}{r}$). Suppose that the key function of the dynamic algorithm is $P(\vec{\alpha}(t))$ that indicates the minimum cost in time slot t if the object is replicated at a set of DCs that is represented by $\vec{\alpha}(t)$.

The corresponding $P(\vec{\alpha}(t))$ to each entry of table in Fig. 5.3 should be calculated for all $t \in [1...T]$ and for all elements of $\vec{\alpha}$. In the following, we derive a general recursive equation for $P(\vec{\alpha}(t))$.

As illustrated in Fig. 5.3, to calculate $P(\vec{\alpha}(t))$ we first need to compute residential cost (i.e., $C_R(.)$) and migration cost (i.e., $C_M(.)$) between $\vec{\alpha}(t-1)$ and $\vec{\alpha}(t)$. Second, to obtain this migration cost, we enumerate over all possible $\vec{\alpha}(t-1)$ containing the object in time slot $t-1$. Thus, the cost $P(\vec{\alpha}(t))$ is the minimum of the summation of the cost $C(\vec{\alpha}(t), \vec{\beta}(t))$ in Equ. (5.5) and $P(\vec{\alpha}(t-1))$. The termination condition for the recursive equation $P(\vec{\alpha}(t))$ is $P(\vec{\alpha}(t)) = 0$ for $t = 0$, meaning there is no placement for the object. Combining all above discussions, we obtain the general recursive equation as:

[5] Note that the constraints (a-e) in Equ. (5.6) is repeated for all cost calculation equations, unless we mentioned.

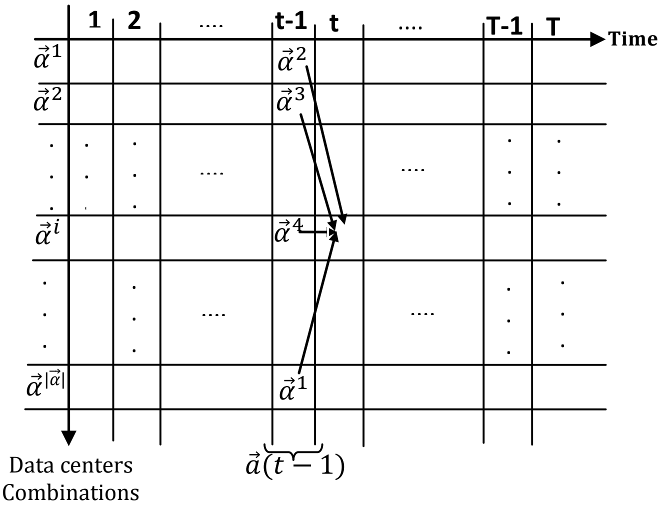


Figure 5.3: The description of $P(\vec{\alpha}(t))$ calculation in Equ. (5.7
)

$$P(\vec{\alpha}(t)) = \begin{cases} \min\limits_{\forall \vec{\alpha}(t-1) \in \vec{\alpha}} [P(\vec{\alpha}(t-1)) + C(\vec{\alpha}(t), \vec{\beta}(t))], & t > 0 \\ 0 & t = 0 \end{cases} \tag{5.7}$$

Once $P(\vec{\alpha}(t))$ is calculated for all $\vec{\alpha}(t) \in \vec{\alpha}$ during $t \in [1...T]$, the minimum cost for the object is $\min\limits_{\vec{\alpha}(t) \in \vec{\alpha}} P(\vec{\alpha}(t))$ in time slot $t = T$. The optimal placement of replicas for the object in time slot $t \in [1...T]$, $\vec{\alpha}^*(t)$, is the corresponding $\vec{\alpha}(t)$ on the path leading to the minimum value of $P(\vec{\alpha}(t))$ in time slot $t = T$. The request distribution related to $\vec{\alpha}^*(t)$ in time slot t is determined by $\vec{\beta}^*(t)$ using a linear programming.

We now analyze the time complexity of Algorithm 5.1, which comprises an $r-$combination computation and four nested loops. The computation of combinations (line 1) takes $O(|D|^2)$. The first loop repeated T times (line 3). The last two loops run for at most $|\alpha|^2$ times where $|\vec{\alpha}|$=$\binom{|D|}{r}$ is a small constant because r is at most 2 or 3 [38] and the number of DCs in the leading commercial cloud providers is 8, for example Amazon and Google. Thus, the value of $|\vec{\alpha}|$ can be $\binom{(3*8)}{2}$= 276. In the last loop, we need to solve a linear problem because we fix $\vec{\alpha}(t)$ and find variable $\vec{\beta}(t)$, which takes T_{lp}. Since $|D| \leq |\vec{\alpha}|$, the total running time of algorithm is $O(|D|^2 + |\vec{\alpha}|^2 T T_{lp}) = O(|\vec{\alpha}|^2 T T_{lp})$. This time complexity increases when the value of α (depending to $|D|$ and r) grows. Note that with a limitation on $|D|$ and an upper bound of 3 for r [38], this time complexity can be improved.

Algorithm 5.1: Optimal Offline Algorithm

```
Input  : RPM's inputs as illustrated in Fig. 5.2
Output: $\vec{\alpha}^{*}(t)$, $\vec{\beta}^{*}(t)$, and the optimized overall cost during $t \in [1...T]$
$\vec{\alpha} \leftarrow$ Calculate all $r-$combinations of distinct DCs from $D$.
Initialize: $\forall \vec{\alpha}(0) \in \vec{\alpha}$, $P(\vec{\alpha}(0)) = 0$
for $t \leftarrow 1$ to $T$ do
    forall $\vec{\alpha}(t) \in \vec{\alpha}$ do
        forall $\vec{\alpha}(t-1) \in \vec{\alpha}$ do
            Calculate $P(\vec{\alpha}(t))$ based on Equ. (5.7).
        end
    end
end
Find a sequence of $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ such that leading to $\min_{\vec{\alpha}(t) \in \vec{\alpha}} (P(\vec{\alpha}(t))$ in time slot
    $t = T$as the optimized overall cost (Equ. 5.6). This sequence of $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ are
    $\vec{\alpha}^{*}(t)$ and $\vec{\beta}^{*}(t)$.
Return $\vec{\alpha}^{*}(t)$, $\vec{\beta}^{*}(t)$, and the optimized overall cost.
```

5.4 Online Algorithms

The optimal offline algorithm as its name implies is optimal and can be solved *offline*. That is, with the given workload, we can determine the optimal placement of objects in each time slot t. However, *offline* solutions sometimes are not feasible for two main reasons: (i) we probably do not have a priori knowledge of the future workload especially for start-up firms or those applications whose workloads are highly variable and unpredictable; (ii) the proposed offline solution suffers from high time complexity and is computationally prohibitive. Thus, we present *online* algorithms to decide which placement is efficient for object replicas in each time slot t when future workloads are unknown. Before proposing online algorithms, we formally define the CR that is widely accepted to measure the performance of the online algorithms.

Definition 5.10. *(Competitive Ratio): A deterministic online algorithm DOA is c-competitive iff* $\forall I$, $C_{DOA}(I)/C_{OPT}(I) \leq c$, *where* $C_{DOA}(I)$ *is the total cost for input I (i.e., workload in our work) by DOA, and* $C_{OPT}(I)$ *is the optimal cost to serve input I by optimal offline algorithm OPT. Similarly, a randomized online algorithm ROA is c-competitive iff* $\forall I$, $E[C_{ROA}(I)]/C_{OPT}(I) \leq c$.

5.4.1 The Deterministic Online Algorithm

We propose an online algorithm based on the total cost $C(\vec{\alpha}(t), \vec{\beta}(t))$ consisting of two sub-costs: *residential* and *migration* costs. These two sub-costs of the object can potentially appear as an overhead cost for the application if the migration of the object happens at inappropriate time(s). Frequent migration of the object causes the object to be moved much more than the optimal number of migrations between DCs. As a result, the total cost of application exceeds its optimal cost. An upper bound of this cost when the optimization problem is solved in time slot t without a priori knowledge of the future workload and considering the location of the object in previous time slot $t-1$. In contrast, a lower number of migrations leads to stagnant objects that they might not be migrated even to a new DC imposing a lower cost to the application. Thus, the residential cost surpasses the optimal residential cost. The upper bound of the residential cost happens when there is no migration.

To avoid these issues, the algorithm makes a trade-off between two costs, residential and migration, in the absence of the future workload knowledge. The intuitive idea behind this algorithm is that 1) migration only happens when it causes cost saving in the current time slot and 2) the summation of the lost cost savings opportunities from the last migration (i.e., t_m) is larger or equal to the migration cost. This intuition causes to strike a balance between frequent and rare migration.

Assume that t_m denotes the last time of migration for the object. Let the migration cost between two consecutive migrations times (i.e., t_{m-1} and t_m) be defined by $C_M(\vec{\alpha}(t_{m-1}), \vec{\alpha}(t_m))$. For each time slot t, we calculate the residential cost of the object in $v \in [t_m, t)$ for two cases: (i) the residential cost of the object as if it is in the DCs that are determined in time slot $v-1$ and the requests issued to the object in time slot v are served by these DCs. This cost is defined by $C_R(\vec{\alpha}(v-1), \vec{\beta}(v))$ (see Fig. 5.4a[6]), and (ii) the residential cost of object as if the object is migrated to new DCs that are determined in time slot v and the requests for the object are served by the chosen new DCs. This cost is termed by $C_R(\vec{\alpha}(v), \vec{\beta}(v))$ (see Fig. 5.4b). Now, for each of the following time slot v, we calculate the summation of the difference between the above residential costs (i.e., (i) and (ii)) from time $v = t_m$ to $v = t-1$, which is $\sum_{v=t_m}^{t-1} [C_R(\vec{\alpha}(v-1), \vec{\beta}(v)) - C_R(\vec{\alpha}(v), \vec{\beta}(v))]$. Based on

[6] In this figure, without loss of generality, we consider only one DC that hosts an object (i.e., $r = 1$).

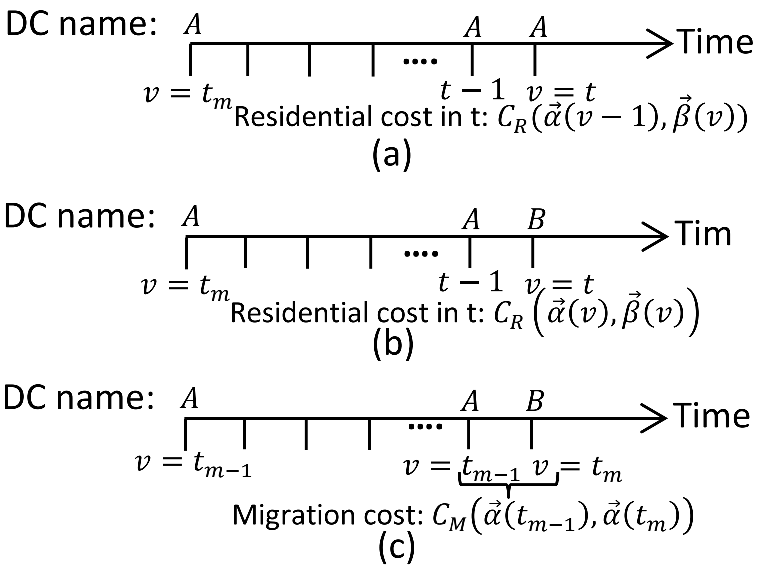


Figure 5.4: The description of Deterministic online algorithm. The residential cost of the object as if the requests on the object in slot $v = t$ are served by (a) the determined DCs in time slot $v - 1$ and (b) the determined DCs in time slot v. (c) The migration cost of the object between the determined DC in time slot $v = t_{m-1}$ and t_m.

the above calculated residential cost and migration cost (i.e., $C_M(\vec{\alpha}(t_{m-1}), \vec{\alpha}(t_m))$- see Fig. 5.4c), in the current time slot t, the algorithm makes a decision whether the object should be migrated to new DCs or not. The object is migrated to new DCs in time slot t if the two following conditions are simultaneously met.

1. The object has the potential to be migrated to a new DC if

$$C_M(\vec{\alpha}(t_{m-1}), \vec{\alpha}(t_m)) \leq \sum_{v=t_m}^{t-1} [C_R(\vec{\alpha}(v-1), \vec{\beta}(v)) - C_R(\vec{\alpha}(v), \vec{\beta}(v))] \tag{5.8}$$

   Otherwise, the object certainly stays in the previous DCs determined in time slot $t - 1$.

2. As earlier noted, to avoid migrating the object back and forth between DCs, we enforce the following condition:

$$C_M(\vec{\alpha}(t_m), \vec{\alpha}(t)) + C_R(\vec{\alpha}(t), \vec{\beta}(t)) \leq C_R(\vec{\alpha}(t-1), \vec{\beta}(t)) \tag{5.9}$$

This constraint means that the overall cost of the object in the new DCs in time slot t including the residential and migration costs should be less than or equal to the cost of the object if it stays in the chosen DCs in time slot $t - 1$.

Based on the above discussion, Algorithm 5.2 formulates the details of the deterministic online algorithm. The algorithm first finds all $r-$combinations of distinct DCs that can be chosen from D (line 2). Then, for each object in time slot $t = 1$, the algorithm determines the best placement of replicas of the object and also the proportion of requests that must be served by these replicas so that $C_R(\vec{\alpha}(t), \vec{\beta}(t))$ is minimized (line 3). After that the migration time t_m is set to 1 (line 4). For all $t \in [2...T]$ (line 5), $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ are calculated for all $\vec{\alpha}(t) \in \vec{\alpha}$ so that the residential cost $C_R(\vec{\alpha}(t), \vec{\beta}(t))$ is minimized (lines 6-9). Based on Equs. (5.8) and (5.9), if the new DC chosen in time slot t is different with that of time slot $t-1$, the object migration happens (lines 10-13). Otherwise, the object stays in the DC that is selected in time slot $t-1$, i.e., $\vec{\alpha}(t) = \vec{\alpha}(t-1)$ (line 15).

We now analyse the performance of the deterministic algorithm in terms of CR. The key insight behind the algorithm lies in Equs. (5.8) and (5.9) to make trade off between frequent and infrequent migrations of objects among DCs. According to these equations, we first calculate the upper bound for the migration cost in $[1...t]$ and then derive the CR of the algorithm.

Lemma 5.1. *The upper bound of the migration cost between two consecutive migration times* (t_{m-1}, t_m) *during* $[1, t]$ *is* γ *times of the minimum residential cost in this time period.* γ *is the ratio of the residential cost in the most expensive DCs to the cheapest ones in* $v \in [1...t]$.

Proof. Based on Equs. (5.8) and (5.9), the migration cost during $[1...t]$ consists of two sub migration costs in $[1...t-1]$ and t. Thus, we have $\sum_{t_m=1}^{t} C_M(\vec{\alpha}(t_{m-1}), \vec{\alpha}(t_m))$
$= \sum_{v=t_m}^{t} (C_R(\vec{\alpha}(v-1), \vec{\beta}(v)) - C_R(\vec{\alpha}(v), \vec{\beta}(v)))$
$\leq \sum_{v=t_m}^{t} (C_R^{max}(\vec{\alpha}(v-1), \vec{\beta}(v)) - C_R^{min}(\vec{\alpha}(v), \vec{\beta}(v)))$.

Let $\gamma = C_R^{max}(\vec{\alpha}(v-1), \vec{\beta}(v)) / C_R^{min}(\vec{\alpha}(v), \vec{\beta}(v))$ for all $v \in [1..t]$. Substituting the value of γ in the above equation, we have,

$\sum_{t_m=1}^{t} C_M(\vec{\alpha}(t_m), \vec{\alpha}(t_{m-1})) \leq$
$(1 - 1/\gamma) \sum_{v \in [1..t]} C_R^{max}(\vec{\alpha}(v), \vec{\beta}(v))$ □

Theorem 5.1. *Algorithm 5.2 is* $(2\gamma - 1)-$*competitive. Formally, for any input,* $C_{DOA}/C_{OPT} \leq 2\gamma - 1$.

Proof. The total cost incurred by DOA is the summation of migration and residential costs in $[1...T]$. Thus, $C_{DOA} = \sum_{t=1}^{T} C_R(\vec{\alpha}(t), \vec{\beta}(t)) + C_M(\vec{\alpha}(t-1), \vec{\alpha}(t))$. Since the upper bound of the residential cost for DOA is γ times the cost of the offline algorithm, and according

to the result of Lemma (5.1) we have: $C_{DOA} =$
$(1-1/\gamma)\sum_{t=1}^{T} C_R^{max}(\vec{\alpha}(t), \vec{\beta}(t)) + \sum_{t=1}^{T} C_R(\vec{\alpha}(t), \vec{\beta}(t)$
$\leq (1-1/\gamma)\gamma C_{OPT} + \gamma C_{OPT} = (\gamma-1)C_{OPT} + \gamma C_{OPT}$
$\leq (2\gamma-1)C_{OPT}$ □

The value of γ is the ratio of the residential cost between the most expensive DC to the cheapest one in the network cost in hot-spot or storage cost in cold-spot during its lifetime. Thus, if the object is read intensive (i.e., it is in hot-spot), the value of $\gamma = \max\limits_{d \neq d'} O(d)/O(d')$. Otherwise, if object is storage intensive (i.e., it is in cold-spot), then $\gamma = \max\limits_{d \neq d'} S(d)/S(d')$. Generally, if the volume of the object to be read is λ times of the object size, then $\gamma = \max\limits_{d \neq d'}(S(d) + \lambda O(d))/(S(d') + \lambda O(d'))$.

To determine the time complexity of Algorithm 5.2, we first need to compute all r-combinations of distinct DCs that runs in $O(|D|^2)$. Second, $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ should be calculated for all $\vec{\alpha}(t) \in \vec{\alpha}$ by using linear programming, which takes $O(|\vec{\alpha}|T_{lp})$. This calculation is done for T time slots. Therefore, the algorithm yields a running time of $O(|D|^2 + |\vec{\alpha}|TT_{lp}) = O(|\vec{\alpha}|TT_{lp})$. As it is observed, the time complexity of the algorithm is less than the time complexity of the optimal offline algorithm since this time complexity is not quadratic in the number of combinations of DCs.

5.4.2 The Randomized Online Algorithm

It is expected that the randomized algorithms typically improve the performance in terms of CR to their deterministic counterparts. In the following, we design a randomized online algorithm based on the subclass of Reducing Horizon Control (RHC) algorithms, which is called Fixed RHC (FRHC) [105]. RHC is a classical control policy that is used for dynamic capacity provisioning in a DC [113] [105], load balancing on a DC [106], and moving data into a DC [194].

In our algorithm, the time period T is divided into $\lceil T/w \rceil$ frames, where each frame has a size of w time slots. It is assumed that in the first time slot (i.e., t_s) of each frame, the workload in terms of *Get* and *Put* requests, and data size is known for the next $t_s + w$ time slots. Due to available future workload knowledge for the time frame $[t_s, t_s + w]$, we can calculate the optimal cost for this time frame. To do so, we re-write the cost optimization

Algorithm 5.2: Determinstic Online Algorithm (DOA)

```
Input  : RPM's inputs as illustrated in Fig. 5.2
Output: $\vec{\alpha}(t)$, $\vec{\beta}(t)$, and the overall cost denoted $C_{ove}$
$C_{ove} \leftarrow 0$
$\vec{\alpha} \leftarrow$ Calculate all $r-$combinations of distinct DCs from $D$.
$C_{ove} \leftarrow$ Determine $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ by minimizing $C_R(\vec{\alpha}(t), \vec{\beta}(t))$ for all $\vec{\alpha}(t) \in \vec{\alpha}$ in
  time slot $t = 1$.
$t_m \leftarrow 1$
for $t \leftarrow 2$ to $T$ do
    forall $\vec{\alpha}(t) \in \vec{\alpha}$ do
        $C_R(.) \leftarrow$ Determine $\vec{\alpha}(t)$ and $\vec{\beta}(t)$ by minimizing $C_R(\vec{\alpha}(t), \vec{\beta}(t))$
        $C_{ove} \leftarrow C_{ove} + C_R(.)$
    end
    if (Equs. (5.8) and (5.9), and $\vec{\alpha}(t-1)! = \vec{\alpha}(t)$ ) then
        $t_m \leftarrow t$
        $C_M(.) \leftarrow$ calculate $C_M(\vec{\alpha}(t_{m-1}), \vec{\alpha}(t_m))$
        $C_{ove} \leftarrow C_{ove} + C_M(.)$
    else
        $\vec{\alpha}(t) \leftarrow \vec{\alpha}(t-1)$
    end
end
Return $\vec{\alpha}(t)$, $\vec{\beta}(t)$ and $C_{ove}$.
```

problem based on Equ. (5.6) for the time frame $[t_s, t_s + w]$ (i.e., Equ. 5.10) and solve it by using Algorithm 5.1 to calculate the optimal cost.

$$\min_{\vec{\alpha}(t), \vec{\beta}(t)} \sum_{t=t_s}^{t_s+w} C(\vec{\alpha}(t), \vec{\beta}(t)) \tag{5.10}$$

The first time slot (i.e., t_s) of the first frame can be started from different initial time $l \in [1, w]$, which indicates different versions of the FRHC algorithm. For each specific FRHC algorithm with value l, an adversary can determine an input with a surge in *Get* and *Put* requests and produce a large size of data. These can result in increasing migration cost and degrading the cost performance of the algorithm. A randomized FRHC defeats this adversary with determining the first time slot of the first frame by a random integer $1 \leq l \leq w$.

Thus, the first slot of the first frame falls between 1 and w. The following frames are considered with the same size of w time slots sequentially. Assuming T is divisible by w, it is clear that if $l \neq 1$, then there are $\lceil T/w \rceil - 1$ *full* frames and two *partial* frames that

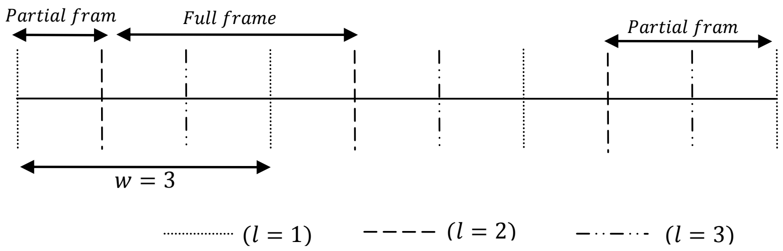


Figure 5.5: Illustration of Fixed Reduced Horizontal Control

consist of l and $w - l$ time slots. Fig. 5.5 shows partial and full frames when the algorithm randomly selects the first slot of the first frame with the value of $l = 2$, where $T = 9$ and $w = 3$. It also shows different versions of randomized FRHC for values of $1 \leq l \leq 3$.

Based on the above discussion, we design the randomized algorithm and solve the optimization problem, i.e., Equ. (5.10) according to Algorithm 5.3 for partial and full frames. In the randomized algorithm, first, we randomly choose $l \in [1, w]$ as t_s of the first frame. If $l \neq 1$, then we calculate the residential cost over two partial frames with the size of l and $w - l$ time slots (lines 2-5). For the full frames, we compute overall cost consisting residential and migration costs for each full frame and migration cost between consecutive full frames (lines 6-11). Finally the migration cost between the last full frame and its next partial frame is determined if $l \neq 1$ (lines 12-15).

We now analyse the performance of the randomized online algorithm in terms of CR as follows.

Lemma 5.2. *The upper bound cost of each frame is the offline optimal cost plus the migration cost of objects from DCs determined by randomized FRHC to those specified by the offline algorithm.*

Proof. Based on Equ. (5.10), the optimal cost of object for each frame by using randomized FRHC with value l is: $C(\vec{\alpha}_l(t), \vec{\beta}_l(t)) = \sum_{t=t_s}^{t_s+w} C_R(\vec{\alpha}_l(t), \vec{\beta}_l(t)) + \sum_{t=t_s}^{t_s+w} C_M(\vec{\alpha}_l(t-1), \vec{\alpha}_l(t))$, where $\alpha_l(t)$ indicates the location of object based on the randomized FRHC with value l. The value of $C(\vec{\alpha}_l(t), \vec{\beta}_l(t))$ is local optimal cost in the time frame $[t_s, t_s + w]$.

The cost incurred by the randomized FRHC in time frame $[t_s, t_s + w]$ should be smaller than (1) the migration cost of the object from DCs chosen by the the randomized FRHC in time slot $t = t_s - 1$ to those determined by the optimal offline algorithm in time slot t_s, i.e, $C_M(\vec{\alpha}_l(t_s - 1), \vec{\alpha}^*(t_s))$, and (2) then following the optimal offline algorithm to find optimal cost in this time slot, which is $\sum_{t=t_s+1}^{t_s+w} C_M(\vec{\alpha}^*(t-1), \vec{\alpha}^*(t)) + \sum_{t=t_s}^{t_s+w} C_R(\vec{\alpha}^*(t), \vec{\beta}^*(t))$.

We now find the upper bound of migration cost $C_M(\vec{\alpha}_l(t_s - 1), \vec{\alpha}^*(t_s))$. This cost is upper bounded by the following two sub migration costs. (i) The object is first migrated from the DCs that are chosen by the randomized FRHC in time slot $t = t_s - 1$ to those determined by the optimal offline algorithm in time slot $t_s - 1$. This migration cost is: $C_M(\vec{\alpha}_l(t_s-1), \vec{\alpha}^*(t_s-1))$. (ii) The object is then migrated from these DCs selected by the optimal offline algorithm in time slot $t_s - 1$ to DCs that are selected in time slot t_s. This cost is $C_M(\vec{\alpha}^*(t_s-1), \vec{\alpha}^*(t_s)))$. We therefore can bound the cost in the time frame as follows:

$C(\vec{\alpha}_l(t), \vec{\beta}_l(t)) \leq (i) + (ii) + \sum_{t=t_s+1}^{t_s+w} C_M(\vec{\alpha}^*(t-1), \vec{\alpha}^*(t)) + \sum_{t=t_s}^{t_s+w} C_R(\vec{\alpha}^*(t), \vec{\beta}^*(t))$
$\leq (i) + \sum_{t=t_s}^{t_s+w} C_M(\vec{\alpha}^*(t-1), \vec{\alpha}^*(t)) + \sum_{t=t_s}^{t_s+w} C_R(\vec{\alpha}^*(t), \vec{\beta}^*(t))$
$\leq (i) + \sum_{t=t_s}^{t_s+w} C(\vec{\alpha}^*(t), \vec{\beta}^*(t))$.

The right side of the above inequality gives the upper-bound of the cost for each time frame $[t_s, t_s + w]$. Hence the proof of the lemma is concluded. □

Theorem 5.2. *Algorithm 5.3 is* $(1 + \frac{\gamma}{w})$*-competitive. Formally, for any input* $C_{ROA}/C_{OPT} \leq (1 + \frac{\gamma}{w})$.

Proof. By using Lemma (2), the upper bound of the total cost incurred by the randomized FRHC is

$C_{ROA} = \sum_{t_s \in \lceil T/w \rceil} [C_M(\vec{\alpha}_l(t_s-1), \vec{\alpha}^*(t_s-1)) + \sum_{t=t_s}^{t_s+w} C(\vec{\alpha}^*(t), \vec{\beta}^*(t))]$
$= \underbrace{\sum_{t_s \in \lceil T/w \rceil} \sum_{t=t_s}^{t_s+w} C(\vec{\alpha}^*(t), \vec{\beta}^*(t))}_{C_{OPT}} + \sum_{t_s \in \lceil T/w \rceil} C_M(\vec{\alpha}_l(t_s-1), \vec{\alpha}^*(t_s-1))]$

The expected cost of Randomized is computed as:

$$E(C_{ROA}) = \frac{1}{w}[\sum_{l=1}^{w}(C_{OPT} + \sum_{t_s \in \lceil T/w \rceil} C_M(\vec{\alpha}_l(t_s-1), \vec{\alpha}^*(t)))]$$
$$= C_{OPT} + \frac{1}{w}\sum_{l=1}^{w}\sum_{t_s \in \lceil T/w \rceil} C_M(\vec{\alpha}_l(t_s-1), \vec{\alpha}^*(t_s-1)).$$

Thus, the CR of the algorithm is

Algorithm 5.3: Randomized Online Algorithm (ROA) with prediction window w

```
Input  : RPM's inputs as illustrated in Fig. 5.2
Output: $\vec{\alpha}(t)$, $\vec{\beta}(t)$, and the overall cost denoted $C_{ove}$
$l \leftarrow$ random number within $[1, w], C_{ove} \leftarrow 0$
if $l \neq 1$ then
    $C_{ove} \leftarrow$ solve Equ. (5.10) over widows $[1, l]$ and $[T - l]$
    $t_m = l + 1$
end
for $t \leftarrow l$ to $T - l + 1$ do
    $C_{ove} \leftarrow C_{ove}+$ solve Equ. (5.10) over widows $[l, l + w)$
    $C_M \leftarrow$ solve Equ. (5.4) for $(t_{m-1}, t_m)$
    $C_{ove} \leftarrow C_{ove} + C_M$, $t_m = l + w + 1$
    $t \leftarrow t + w$
end
if $l \neq 1$ then
    $C_M \leftarrow$ solve Equ. (5.4) for $(t_{m-1}, t_m)$
    $C_{ove} \leftarrow C_{ove} + C_M$
end
Return $\vec{\alpha}(t)$, $\vec{\beta}(t)$ and $C_{ove}$
```

$$E(C_{ROA})/C_{OPT} = 1 + \frac{1}{w}\left(\frac{\sum_{l=1}^{w}\sum_{t_s \in \lceil T/w \rceil} C_M(\vec{\alpha}_l(t_s - 1), \vec{\alpha}^*(t_s - 1)))}{C_{OPT}}\right)$$
$$\leq 1 + \frac{1}{w}\left(\frac{\sum_{l=1}^{w}\sum_{t_s \in \lceil T/w \rceil} C_M^{max}(\vec{\alpha}_l(t_s - 1), \vec{\alpha}^*(t_s - 1)))}{C_{OPT}}\right).$$

Based on the definition of γ in Lemma 1, we have:
$E(C_{ROA})/C_{OPT} \leq 1 + \frac{\gamma}{w}(\frac{\sum_{l=1}^{w}\sum_{t_s \in \lceil T/w \rceil} C_M^{min}(\vec{\alpha}_l(t_s-1), \vec{\alpha}^*(t_s-1)))}{C_{OPT}})$. Since the coefficient of $\frac{\gamma}{w}$ is less or equal to one, we have $E(C_{ROA})/C_{OPT} \leq 1 + \frac{\gamma}{w}$. □

Based on computed γ in Section 5.5.3, the randomized algorithm leads to a CR of $1 + \frac{1.52}{w}$, depending on the value of w, and achieves to better cost performance compared with its counterpart.

To calculate the time complexity of the algorithm, it suffices to calculate the time complexity of **for** loop. Since this algorithm produces the results in each frame, the time complexity of algorithm is $O(|D|^2 + w|\vec{\alpha}|^2 T_{lp}) = O(w|\vec{\alpha}|^2 T_{lp})$ for $w < T$; otherwise it takes the time complexity the same as the optimal offline algorithm.

5.5 Performance Evaluation

We evaluate the performance of the algorithms via simulation using the synthesized workload based on the Facebook workload [16]. Our aims are twofold: (i) we measure the cost savings achieved by the proposed algorithms relative to the benchmark algorithms, and (ii) we explore the impact of different values of parameters on the algorithms' performance. To evaluate our work, we implemented the algorithms in the CloudSim discrete event simulator [35], which has been used by both industry and academia for performance evaluation of applications in cloud environments.

5.5.1 Settings

This study uses the following setup for DC specifications, workload on objects, delay constraints, and experiment parameters setting.

DCs specifications: We span DCs across 11 regions[7] in each of which there are DCs from different CSPs. There are 23 DCs in the experiments. We set the storage and network prices of each DC as specified in June 2015. Note that we use the price of SS and RRS during hot-spot and cold-spot status of objects respectively. The object is transited from hot-spot to cold-spot when about 3/4 of its requests have been served [21]. These many requests are received within the first 1/8 of the lifetime of the object, which is considered as the hot-spot status for the object [21].

Workload on objects: It comes from the Facebook workload [16] in three terms: (i) the ratio of *Get*/*Put* requests is assigned to 30, (ii) the average size of each object retrieved from the bucket (as described in Section 5.2, the bucket is integration of hundreds of objects) is 1 KB and 100 KB on average[8][177], and (iii) the pattern for *Get* rate to retrieve items follows long-tail distribution such that 3/4 of those *Get*s happen during 1/8 of the initial lifetime of the bucket [21]. We synthetically generate the *Get* rate of each bucket based on Weibull distribution that follows the above mentioned pattern. The number of *Get* operations for each bucket is randomly assigned with the average of 1250. The low and high *Get* rate implies that the bucket contains the objects belonging to users whose profiles are accessed frequently and rarely respectively (i.e., this category of users has a

[7] California, Oregon, Virginia, Sao Paulo, Chile, Finland, Ireland, Tokyo, Singapore, Hong Kong and Sydney.

[8] Henceforth, the object with size 1 KB and 100 KB on average are called small and large object respectively.

low and high number of friends respectively).

Delay setting: The round trip time delay between each pair of DCs is measured based on the formula $RRT(ms) = 5 + 0.02 \times Distance(km)$ [133]. The latency L– a user can tolerate to receive a response of Get/Put requests– is 100 ms (i.e., tight latency) and 250 ms (i.e., loose latency). A latency higher than 250 ms deteriorates the user's experience on receiving Get/Put response [94].

Experiment parameters setting: In the experiments, we set the following parameters. The overall size of objects is 1 TB and the size of each bucket is initially 1 MB, which grows to 50 MB during the experiments. The number of replicas is set to 1 and 2 [38]. The unit of the time slot (as well w) is one day. For the prediction window, we set $w = 4$ by default, where the randomized algorithm is superior to the deterministic algorithm in the cost saving, except for large objects with two replicas under loose latency. We vary w to examine its impact on the cost saving. In all workload settings, we compute cost over a 60-day period.

5.5.2 Benchmark Algorithms

We propose two benchmark algorithms to evaluate the effectiveness of the proposed algorithms in terms of cost.

Non-migration algorithm: This is shown in Algorithm 5.4 and minimizes the residential cost $C_R(.)$ with all constraints in (5.6) such that objects are not allowed to migrate during their lifetime. This algorithm, though simple, is the most effective measure to show the impact of object migration on the cost saving (see Section 5.5.3).

Local residential algorithm: In this algorithm, an object is locally replicated at a DC located in the region that issues most Get/Put requests for the object and also in the closest DC(s) to that DC if the need for more replicas arises. All the incurred costs are normalized to the cost of *local residential algorithm*, unless otherwise mentioned.

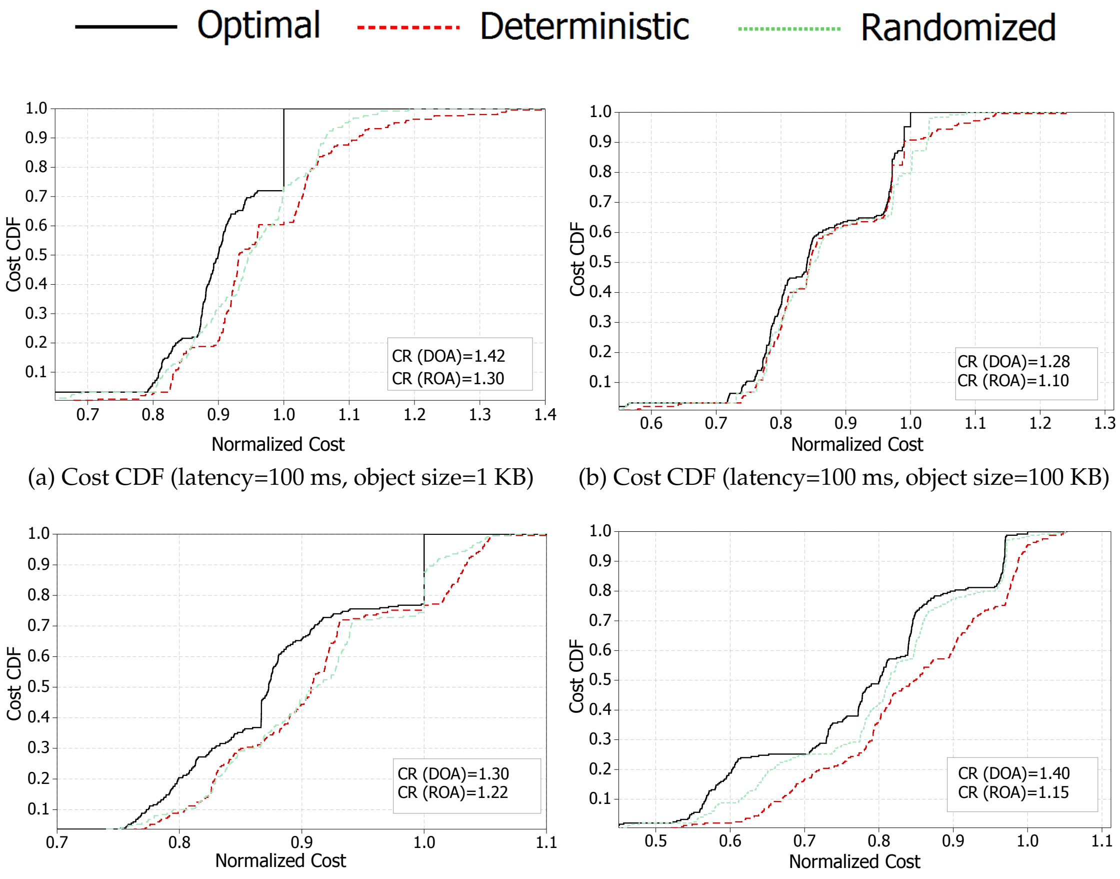
5.5.3 Results

We start by evaluating the performance of algorithms relative to the above *benchmark algorithms*.

The cost performance of all algorithms through simulations is presented in Figs. 5.6

Algorithm 5.4: The Non-migration Algorithm

```
Input  : RPM's inputs as illustrated in Fig. 5.2
Output: $\vec{\alpha}(t)$, $\vec{\beta}(t)$, and the overall cost
1 $\vec{\alpha} \leftarrow$ Calculate all $r-$combinations of distinct DCs from $D$.
2 Calculate $\sum_{t=1}^{T} C_R(\vec{\alpha}(t), \vec{\beta}(t))$ with all constraints in Equ. (5.6) for all $\vec{\alpha}(t) \in \vec{\alpha}$, and
  then select $\vec{\alpha}(t)$ as the location of the object from $t = 1$ to $T$ so that the above
  computed cost is minimized. This cost is the overall cost.
3 Return $\vec{\alpha}(t)$, $\vec{\beta}(t)$, and the overall cost.
```



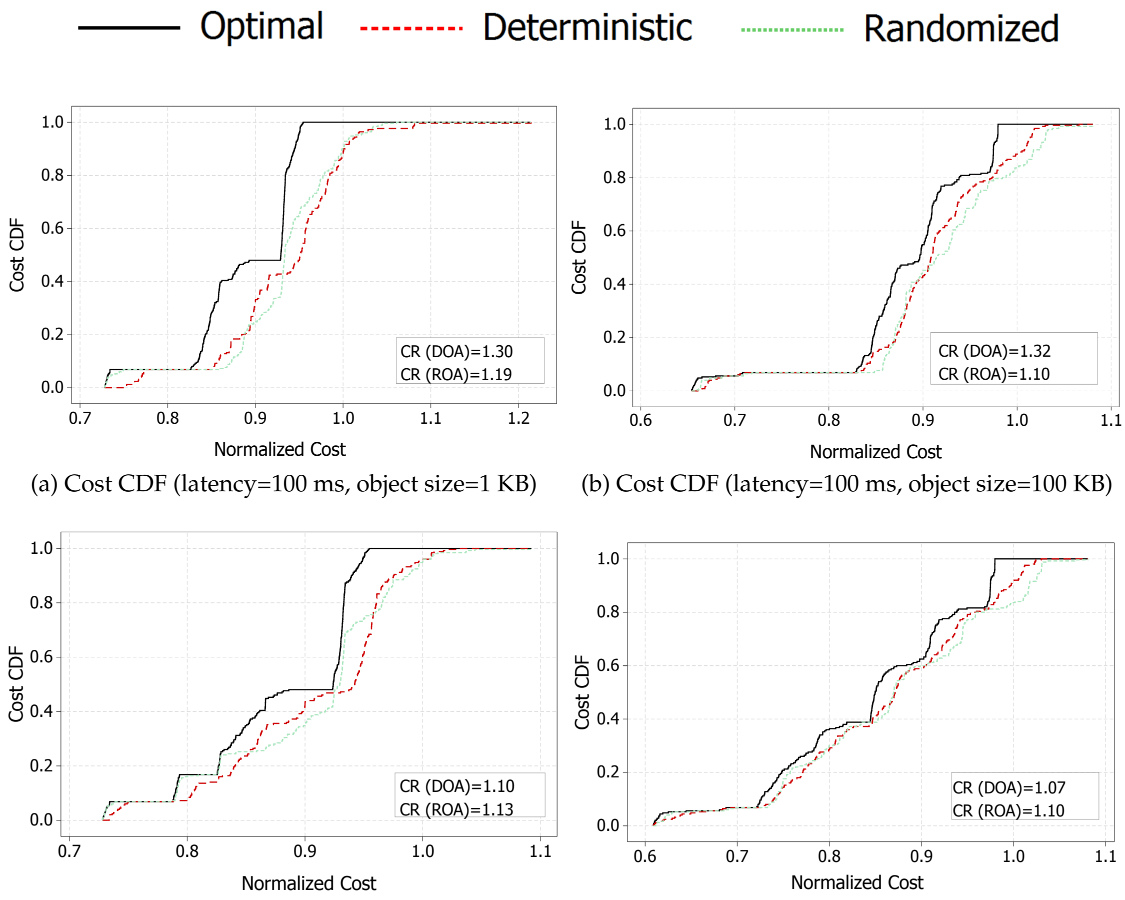
(a) Cost CDF (latency=100 ms, object size=1 KB) (b) Cost CDF (latency=100 ms, object size=100 KB)

(c) Cost CDF (latency=250 ms, object size=1 KB) (d) Cost CDF (latency=250 ms, object size=100 KB)

Figure 5.6: Cost performance of algorithms under tight and loose latency for objects with a replica. All costs are normalized to the local residential algorithm. The values in boxes show the CR of DOA and ROA in the worst case.

and 5.7, where the CDF of the normalized costs[9] are given for small and large objects with r=1,2 under tight and loose latency. The general observation is that all algorithms witness significant cost savings compared with the local residential algorithm. As expected, the

[9] Note that as normalized cost is smaller, we save more monetary cost.



(a) Cost CDF (latency=100 ms, object size=1 KB)

(b) Cost CDF (latency=100 ms, object size=100 KB)

(c) Cost CDF (latency=250 ms, object size=1 KB)

(d) Cost CDF (latency=250 ms, object size=100 KB)

Figure 5.7: Cost performance of algorithms under tight and loose latency for objects with two replicas. All costs are normalized to the local residential algorithm. The values in boxes show the CR of DOA and ROA in the worst case

results, in term of average cost saving (see Table 5.3), show that Optimal outperforms Randomized, which in turn is better than Deterministic (apart from the above mentioned exception).

Fig. 5.6a illustrates the results for small objects under tight latency. Optimal saves at most 20% of the costs for about 71% of the objects, and the online algorithms cut 10% of the costs for about 60% of the objects. In contrast, the results also show that the application incurs at most 10% more cost for about 20% of the objects by using Deterministic, and likewise at most 20% more cost for about 30% of the objects by using Randomized. Fig. 5.6b depicts the results for large objects under tight latency. We can observe that Optimal cuts costs for more than 95% of the objects, while this value reduces to about 80% of the objects in online algorithms. The cost savings for objects in Optimal, Deterministic, and

Table 5.3: Average cost performance (Normalized to the local residential algorithm)

		Latency=100 ms			Latency=250 ms		
Replicas Number	**Object Size**	**Optimal**	**DOA**†	**ROA**‡	**Optimal**	**DOA**	**ROA**
r=1	**1 KB**	0.9030	0.9694	0.9469	0.8778	0.9075	0.8974
	100 KB	0.8561	0.8734	0.8657	0.7758	0.8369	0.7997
r=2	**1 KB**	0.8879	0.9330	0.9181	0.8787	0.9045	0.8866
	100 KB	0.8831	0.9045	0.9127	0.8440	0.8625	0.8636

† Deterministic Online Algorithm
‡ Randomized Online Algorithm

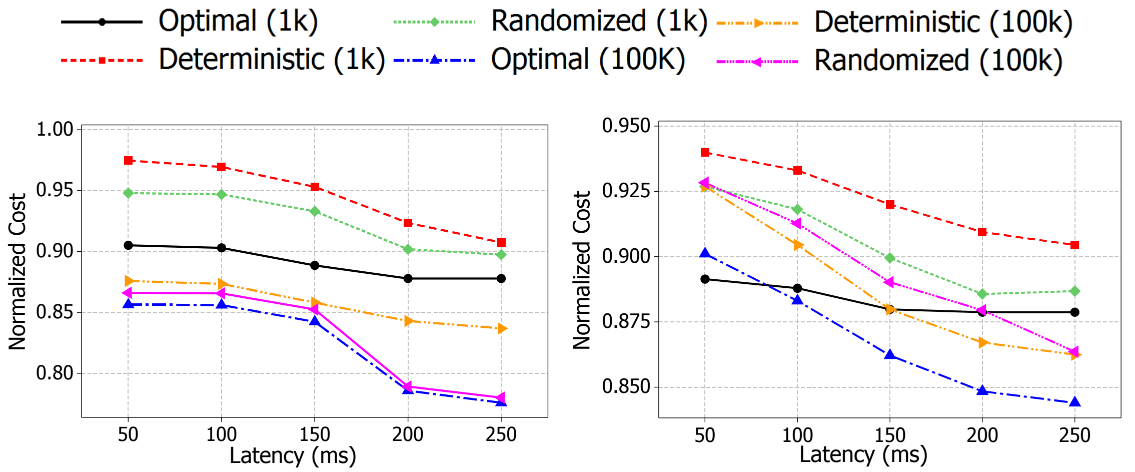
Randomized are respectively 15%, 14% and 13%. Based on comparison between results in Figs. 5.6a and 5.6b, we realize online algorithms remain highly competitive with the optimal algorithm in cost savings for large objects. This happens due to the fact that the migration of large objects in both online and offline algorithms happens roughly at the same time.

Figs. 5.6c and 5.6d show the results for small and large objects under loose latency. The algorithms cut the costs for about 78% of the small objects (Fig. 5.6c) and for about 100% large objects (Fig. 5.6d). On the average, from Table 5.3, Optimal, Deterministic, and Randomized respectively gain cost savings around 13%, 10% and 11% for small objects, and correspondingly 23%, 17% 21% for large objects. From these results in Figs. 5.6c and 5.6d to those in Figs. 5.6a and 5.6b, we observe that all algorithms are more cost effective under loose latency in comparison to tight latency. The reason is that: (i) there is a wider selection of DCs available with lower cost in storage and network resources under loose latency in comparison to tight latency, and (ii) the application can benefit from the large objects migration more than the small objects migration).

The results in Fig. 5.7 reveal that the cost performance of algorithms for objects with two replicas. By using *online algorithms*, the application witnesses the following cost savings. As illustrated in Figs. 5.7a and 5.7c, the application can reduce cost for about 90% and 95% of the small objects under loose and tight latency respectively. For these objects, Randomized and Deterministic under loose latency (resp., under tight latency) reduce the cost by 7% and 9% (resp., 10% and 12%) on average (see Table 5.3).

As shown in Figs. 5.7b, 5.7d and Table 5.3, for large objects, the cost saving of two online algorithms become very close while Deterministic is slightly better than Randomized in average cost savings. Under tight latency, the application receives 10% and 9% of cost savings by using Deterministic and Randomized, respectively, while under loose latency, the application saves the cost (around 14% on average) by using each of online algorithm. This slight superiority of Deterministic over Randomized shows that we need to choose $w > 4$ in order to allow Randomized to outperform Deterministic for this setting (i.e., r=2, for large objects under tight latency). By using the *optimal offline algorithm*, we observe the following results in Fig. 5.7. The application achieves cost savings for all objects with two replicas, while it is not the same for all objects with one replica (see Figs. 5.6a and 5.6c). On average, the application using Optimal reduces cost for small objects (resp., for large objects) by about 12% and 13% (resp., 12% and 16%) under loose and tight latency respectively.

Besides the above experimental results, we are interested to evaluate the performance of online algorithms in terms of CR values already discussed. For this purpose, we compare the value of CR obtained in theory with that of the experimental results in Figs. 5.6 and 5.7. To calculate the theoretical value of CR, we require the value of γ. Under the storage and network price used in the simulation, the gap between the network prices is more than that of between the storage prices of the same DCs in this case. The highest gap is between GCS and ACS with value 0.21 per GB and 0.138 per GB, respectively, in the Asia- Pacific region, which results in γ= 1.52. Thus, by Theorem 5.1 and the value of γ, the deterministic algorithm will lead to at most 2.04 times the optimal offline cost. And, by Theorem 5.2 and the value of γ, the randomized algorithm incurs at most 1.38 (note that the value of w is 4 in all experiments). The corresponding CR for each experimental result in Figs. 5.6 and 5.7 is shown in a box at the bottom of each figure. This value of the CR is the highest among all objects incurred by the online algorithms. All CR values obtained from experimental results are lower than those theoretical values as the object migrations conducted by the proposed algorithms does not necessarily occur between DCs with the highest and the lowest price in the network. Therefore, the online algorithms remain highly competitive in comparison to the optimal offline algorithm in the worst case in all experiments.



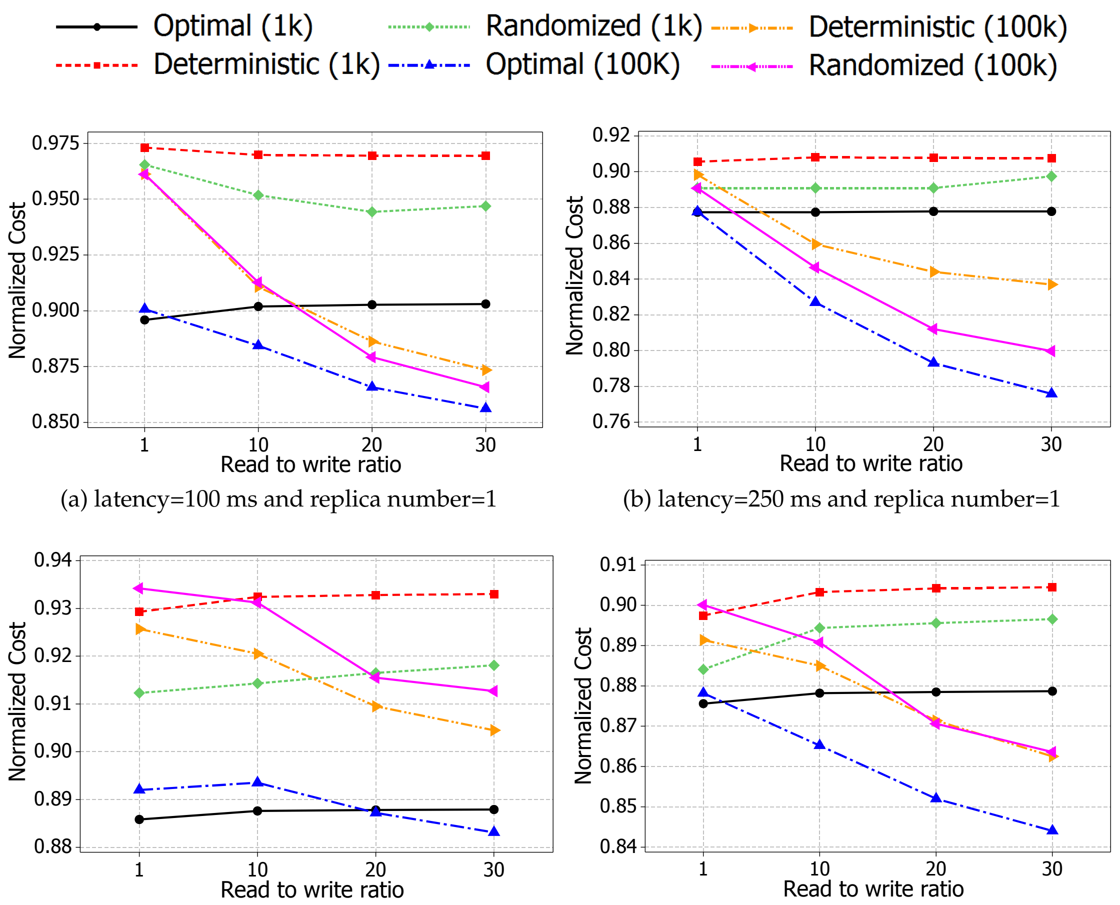
(a) Normalized cost vs. latency for one replica

(b) Normalized cost vs. latency for two replicas

Figure 5.8: Normalized cost of algorithms when the latency is varied. Legend indicates object size in KB for different algorithms. All costs are normalized to the local residential algorithm.

The Effect of Latency on Cost saving

In this experiment, we evaluate the cost performance of algorithms when the latency is varied from 50 ms to 250 ms. First, as shown in Figs. 5.8a and 5.8b, the normalized cost of all algorithms reduces when the latency increases. The reason is that when the latency is 50 ms, most objects are locally replicated at DCs; as a result normalized cost is high. As latency increases, algorithms can place objects in remote DCs which are more cost-effective, and hence the normalized cost declines. For example in Fig. 5.8a, as latency increases from 50ms to 250ms, the cost saving for Optimal, Deterministic and Randomized rises from 3-10%, 6-11% and 10-13%, respectively, for small objects, and likewise 13-17%, 14-22% and 15-23% for large objects. Second, as we expected, Optimal outperforms Randomized, which in turn is better than Deterministic in normalized cost excluding the mentioned exception (see Fig. 5.8b for large objects). This exception implies that we need to use $w > 4$ to achieve better performance of Randomized compared to that of Deterministic. Third, as shown in Fig. 5.8b, we observe that the decline in the cost savings for large objects is more steep than those of small objects when latency increases.



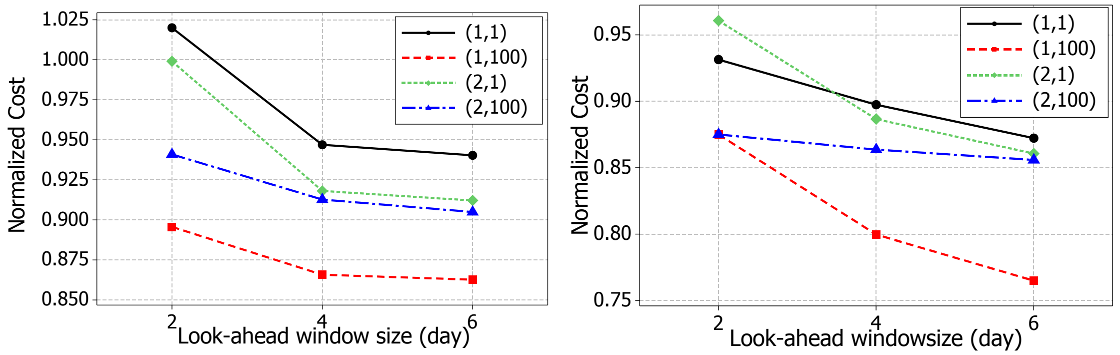
(a) latency=100 ms and replica number=1

(b) latency=250 ms and replica number=1

(c) latency=100 ms and replica number=2

(d) latency=250 ms and replica number=2

Figure 5.9: Normalized cost vs. read to write ratio under tight and loose latency for objects with one and two replicas. Legend indicates object size in KB for different algorithms. All costs are normalized to the local residential algorithm.



(a) Normalized cost vs. window size for latency 100 ms

(b) Normalized cost vs. window size for latency 250 ms

Figure 5.10: Normalized cost of the Randomized algorithm when the window size is varied. All costs are normalized to the local residential algorithm. Legend indicates replicas number and objects size in KB.

The Impact of Read to Write Ratio on Cost Saving

We plot the effect of read to write ratio, varying from 1 (read-intensive object) to 30 (write-intensive object), on the normalized cost for small and large objects under tight and loose latency in Fig. 5.9. We observe the following results.

(i) There is a hierarchy among algorithms in the normalized cost, where Optimal is better than Randomized, which in turn, outperforms Deterministic, excepts for large objects with two replicas. In this exception, Deterministic saves 1% more cost than Randomized with $w = 4$, while for $w > 4$ Randomized is better than Deterministic in this criterion (next section). (ii) For small objects with $r = 1, 2$ under both latency constraints, the normalized cost of all algorithms increases slightly as the ratio goes up, excluding the normalized cost of Randomized for small objects with one replica under tight latency (see Fig. 5.9a). The reason behind this slight increment is that when the ratio increases, less volume of data is read and written; hence the application has to leverage from less difference between storage and network services and objects are prone to stay in local DC(s). (iii) For large objects with $r = 1, 2$ under both latency constraints, the normalized cost of all algorithms reduces as the ratio raises, particularly for $r = 1$. For example, as shown in Figs. 5.9b and 5.9d, under loose latency (resp., under tight latency), the normalized cost reduces by 10%, 9% and 6% (resp., 5%, 6% and 9%) for Optimal, Randomized and Deterministic, respectively when the ratio increases from 1 to 30. The main reason for the reduction in the normalized cost for large objects is that when large objects are write-intensive, the objects migrate to the new DC(s) lately and utilize less the difference between storage and network cost. In contrast, read-intensive large objects can better leverage the difference between storage and network cost. (iv) Under both latency constraints, small objects with two replicas generate more cost savings than the same objects with one replica, while the situation is reversed for the large objects.

The Impact of Window Size on Cost Saving of the Randomized Algorithm

We investigate the impact of look-ahead window size into the available future workload on the normalized cost of the randomized algorithm. As shown in Fig. 5.10, we evaluate this effect when w varies from 2 to 6 units of time. As expected, the larger the w value, the more reduction in the normalized cost of the algorithm for small and large objects with

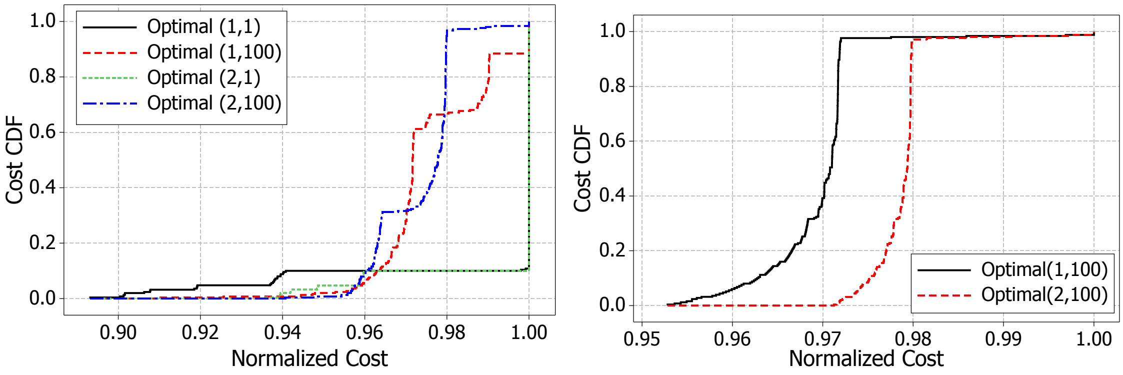
$r = 1, 2$ under tight and loose latency. As mentioned before, for the prediction window, we set $w = 4$ by default and results show that Randomized outperforms Deterministic, excluding for large objects with two replicas (see Fig. 5.7d). For this setting, Fig. 5.10b represents the normalized cost of Randomized which is lower than that of Deterministic for $w > 4$. This indicates that the more future workload information is available, more improvement in the cost saving for the algorithm happens.

The Effect of Objects Migration on Cost Saving

We now show how much cost can be saved by migrating objects in the proposed algorithms over Algorithm 4 as a benchmark. Fig. 5.11a shows that when the latency is tight, for about 11% of small objects, there is a saving of at most 10% and 6% for $r = 1$ and $r = 2$ respectively. For large objects, as expected more improvements are observed in the cost savings. In particular, the application saves 4-5% of the costs for 88% of the objects with one replica and for 98% of them with two replicas. This is because that as the object size increases, the objects are more in favor of migration due to increasing in the imposed storage cost. Fig. 5.11b shows the effect of migration on cutting the cost when the latency is loose. For small objects, cost saving is not significant (we did not plot here) because (i) in their early lifetime, they find DCs that are competitive in the cost of storage and network; and (ii) the objects do not considerably grow in size requiring to be migrated to new DCs in the end of their lifetime. Thus, the object is replicated at DCs that are cost-effective in both resources for its whole lifetime. In contrast, for 90% of the large objects, the cost saving is around 4.5% and 2.5% when r=1 and r=2 respectively.

The Run Time of Algorithms

We measure the running time of algorithms by conducting experiments on a Quad Core 2300MHz Machine (AMD Opteron 63xx series with 512 KB cache) with 16 GB RAM. Table 5.4 shows the running time for placing each object in 23 DCs per each time slot. As it can be seen, Deterministic has less running time than Randomized while Optimal is the worst case especially for $r = 2$. The algorithms are finished in less than a second when $r = 1$, while for $r = 2$ the running time increases to several seconds. However, Deterministic stays more efficient because the time complexity of this algorithm is linearly proportional



(a) Cost CDF (latency=100 ms)

(b) Cost CDF (latency=250 ms)

Figure 5.11: CDF of cost savings for objects due to their migration under tight and loose latency. All costs are normalized to the non-migration algorithm. Legend indicates replicas number and objects size in KB.

with the number of DCs.

The running time of the proposed algorithms may be further decreased through (i) using more efficient linear program solver, for example CPLEX solver, instead of LP solver used here, and (ii) reducing the number of DCs, as the main factor contributing to the time complexity, especially when the latency constraint is loose. As the latency constraint is loose (i.e., is large), the application can access more DCs with the same price or very close price. Therefore, we can reduce the number of DCs when we have similar choices. For example, in the European region, we can have only one Amazon DC in Frankfort, instead of having Amazon DCs of same prices both in Frankfort and Ireland.

Table 5.4: Running time of algorithms on 23 DCs (in Second)

Algorithms	$r = 1$	$r = 2$
Optimal	0.750	11.35
Deterministic	0.012	2.01
Randomized	0.368	8.45

5.6 Summary

To minimize the cost of data placement for applications with time-varying workloads, developers must optimally exploit the price difference between storage and network services across multiple CSPs. To achieve this goal, we designed algorithms with full and partial future workload information. We first introduced the optimal offline algorithm to minimize the cost of storage, *Put*, *Get*, and potential migration, while satisfying eventual consistency and latency. Due to the high time complexity of this algorithm coupled with possibly unavailable full knowledge of the future workload, we proposed two online algorithms with provable performance guarantees. One is deterministic with the competitive ratio of $2\gamma - 1$, where γ is the ratio of residential cost in the most expensive data center to the cheapest one either in storage or network price. The other one is randomized with the competitive ratio of $1+\frac{\gamma}{w}$, where w is the size of available lookahead windows of the future workload. Large scale simulations driven by a synthetic workload based on the Facebook workload indicate that the cost savings can be expected using the proposed algorithms under the prevailing Amazon's, Microsoft's and Google's cloud storage services prices.

Chapter 6

Cost Optimization across Cloud Storage Providers: A Lightweight Algorithm

The previous chapter has introduced the object placement algorithms that determine the location of limited and fixed number of object replicas. These algorithms fail to dynamically determine the number of object replicas that receive time-varying workloads from a wide range of DCs. They also demand high time complexity to determine the object replicas placement as the number of replicas increase. This chapter, however, addresses these issues by introducing lightweight heuristic solution inspired from an approximate algorithm for the Set Covering Problem. It jointly determines object replicas location, object replicas migration times, and redirection of Get (read) and Put (write) requests to object replicas so that the cost of data storage and access management is optimized while the user-perceived latency is satisfied. We evaluate the effectiveness of the algorithm in terms of cost savings via extensive simulations using CloudSim simulator and traces from Twitter. In addition, we have built a prototype system running over Amazon Web Service (AWS) and Microsoft Azure to evaluate the duration of objects migration within and across regions.

6.1 Introduction

WELL known Cloud Storage Providers (CSPs) such as Amazon Web Service (AWS), Microsoft Azure, and Google offer several storage classes with different prices. The price of storage classes is different across CSPs, and it is directly proportional to the performance metrics like availability, durability, etc. For example, Reduced Redundant Storage (RRS) is an AWS's storage class that enables users to reduce their cost with lower levels of redundancy as compared to Simple Storage Service (S3).

CSPs also charge their users/application providers for network resources in different prices. They charge users for outgoing data, while the cost for ingoing data is often free. They may also charge their users at a lower cost when the data are moved across Datacenters (DCs) operated by the same cloud provider (e.g., AWS provider). This diversification of the storage and network prices plays an essential role in the optimization of the monetary cost spent in using cloud-based storage resources.

This cost also affected by the expected workload of an object/data. The object might be a photo, a tweet, or even an integration of these items as a bucket [52] that shares similar Get (read) and Put(write) access rate pattern. The object is created once, read many times, and updated/written rarely. The object workload is determined by how often it is read and updated. There is a strong correlation between the object workload and the age of object, as observed in online social networks (OSNs) [122]. In other words, the object uploaded to OSNs receives dominating more Gets and Puts during its early lifetime, and such object is in hot-spot status and is said to be network-intensive. Then the object cools over time and receives fewer and fewer Gets and Puts. Such object is in cold-spot status and is said to be storage-intensive.

Therefore, with the given (i) time-varying workload of object, and (ii) storage classes offered by different CSPs with different prices, acquiring the cheapest network and storage resources in the appropriate time of the object lifetime plays a vital role in the cost optimization of the data management across CSPs. This cost consists of replica creation, storage, Get, Put, and potential migration costs. To optimize these costs, cloud users are required to answer the following questions: (i) which storage class from which CSP should host the object (i.e., placing), and (ii) which replica should serve a specific Get (i.e., Get requests redirection), and (iii) when the object replica should probably be migrated from a storage class to another one operated by the similar or different DCs.

We previously investigated some of these questions. In Chapter 4, we proposed a dual cloud-based storage architecture that optimizes the cost of object across two DCs with two storage classes. The findings in terms of cost saving obtained from this architecture motivate us to extend this architecture across multiple data stores, as discussed in Chapter 5. In Chapter 5, we proposed object placement algorithms that determine the location of limited and fixed number of object replicas with time-varying workloads. These algorithms fail to dynamically determine the number of object replicas. Moreover, they suffers from high time complexity when the object receives Gets and Puts from a wide range of DCs, and consequently demand many replicas to provision Gets and Puts within the latency constraint specified by users. To tackle these issues and answer the aforementioned questions, we propose a lightweight algorithm that demands low time complexity, thereby making them tailored for applications that host a large number of objects.

The lightweight algorithm to optimize the cost of data storage management make a three-fold decision: replicas location, replicas migration time from a storage class to another one operated by similar or different data stores, and redirection of Gets to replicas. In addition to the cost optimization, latency to read data from and write data into the data store is also a vital performance criterion from the perspective of users. We consider the latency constraint as a service level objective (SLO) and define it as the elapsed time between issuing a Get/Put from a data center (DC) and retrieving/writing the required object from/into the data store.

In summary, by wisely taking into account the pricing differences for storage and network resources across CSPs and time-varying workloads of objects, we are interested to reduce the cost of data storage management (i.e., replica creation storage, Get, Put, and migration costs) so that the user-perceived latency for Gets and Puts is met. To address this issue, we make the following key contributions:

- We introduce a cost model that includes replica creation, storage, Get, Put, and potential migration costs. This cost model integrates the user-perceived latency for reading and writing data from and into data stores.
- We optimize this cost model by exploiting linear and dynamic programming where the exact future workload is assumed to be known *a priori*. Due to the requirement of high time complexity, we propose a lightweight algorithm that makes key decisions on replica placement, Get requests redirection, and replicas migration time without any knowledge of the future workloads of objects.
- We conduct extensive experiments to show the effectiveness of the proposed solution in terms of cost savings by using real-world traces from Twitter [101] in CloudSim simulator [35].
- In addition, we have built a prototype system running over AWS and Microsoft

Azure cloud providers to evaluate the effectiveness of the proposed solution in terms of the duration of objects migration within a region and across regions.

6.2 System Model, Cost Model, and Cost Optimization Problem

We first discuss our system model and formulate the cost of data storage management, and then define an optimization cost problem according to the formulated cost model.

6.2.1 System model

Our system model employs different cloud providers that operate Geo-distributed DCs. DCs from different cloud providers may be co-located, but they offer several storage classes with different prices and performance metrics. Using each storage class can be determined by the user's objective (e.g., monetary cost optimization).

In our system, each user creating the object is assigned to his/her closest DC among the DCs, which are Geographically dispersed across the world. This DC is referred as a *home* DC. The created object can be a tweet or a photo that is posted by the user on his/her Twitter Feed or Facebook Timeline. The object is replicated in several DCs based on its workload, the number of the user's friends/followers, and the required access latency to serve Gets. These replicas are named *slave* replicas, as opposed to the *master* replica stored in the home DC. The master/slave replica of the object is in *hot-spot* status if it receives many Gets and Puts, and in *cold-spot* status if it receives a few. These statuses of the object replica probably lead to the replica migration between storage classes. To do this, our system uses the *stop and copy* migration technique in which the Gets are served by the DC that the object must be migrated from (called *source DC*) and the Puts are handled by the other DC that the object must be migrated to (called *destination DC*) [169]. The unit of data migration is the *bucket* abstraction which is the same as that in Spanner [52]. The bucket consists of the objects owned by a specific user.

In the system model, a DC is referred as a *client* DC if it issues a Get/Put for the object, and a DC is named as a *server* DC if it hosts a replica of the object. If a DC stores a replica of the object in order to serve its Puts and Gets, then this DC is client and server at the same time for this specific object.

Table 6.1: Summary of key notations

Symbol	Meaning
D	A set of DCs
D_x	If "x=c", D_c is the set of client DCs. If "x=r", D_r is the set of server DCs. If "x=p", D_p is the set of potential DCs to host a replica.
d_x	If "x=c", d_c is a client DC. If "x=r", d_r is a server DC. If "x=p", d_p is a potential DC. If "x=h", d_h is a home DC.
$S(d)$	The storage cost of DC d per unit size per unit time
$O(d)$	Out-network price of DC d per unit size
T	Number of time slots
$v(t)$	The size of the object in time t
$r_{d_c}(t)$	Number of read requests from d_c in time t
$w_{d_c}(t)$	Number of write requests issued from d_c in time t
r	Number of replicas of the object
$t_m^{d_r}$	Migration time of a replica in DC d_r
$\alpha^d(t)$	A binary variable indicates whether a replica is in DC d in time slot t or not
$d_c(t) \to d_r(t)$	A binary variable, being 1 if the DC d_c is served by DC d_r and being 0 otherwise.
$C_x(.)$	If "x=R", $C_R(.)$ is the residential cost. If "x=M", $C_M(.)$ is the migration cost. If "x=B", $C_B(.)$ is the benefit-cost that is lost for a replica in a specific DC during time t.
L	An upper bound of delay on average for Gets and Puts to receive response
$l(d_c, d_r)$	The latency between DC d_c and DC d_r

6.2.2 Cost model

We assume a time-slotted system in which each slot lasts for $t \in [1...T]$. This system is represented as a set of independent DCs, D, where each DC d is associated with a tuple of four cost elements. (i) $S(d)$ denotes the storage cost per unit size per unit time (e.g., bytes per hour) in DC d. (ii) $O(d)$ defines out-network cost per unit size (e.g., byte) in DC d. (iii) $t_g(d)$ and $t_p(d)$ represent transaction cost for a bulk of Gets and Puts in DC d, respectively

Assume that the application creates a set of objects in time slot t. Let $r_{d_c}(t)$ and $w_{d_c}(t)$, respectively, be the number of Gets and Puts for the object with size $v(t)$ from client DC $d_c(t)$ in time slot t. For Gets, let client $d_c(t)$ is served by server DC $d_r(t)$ that hosts a replica of the object in time slot t. This is denoted by $d_c(t) \to d_r(t)$, which is binary, being 1 if the DC $d_c(t)$ is served by DC $d_r(t)$ and being 0 otherwise. Note that it is no need for assignment of DCs $d_c(t)$ to $d_r(t)$ for Puts since these requests issuing form $d_c(t)s$ must be submitted to all replicas in $d_r(t)s$. The number of replicas, denoted by r, for each object is variable in each time slot, and depends on the object workload, the required

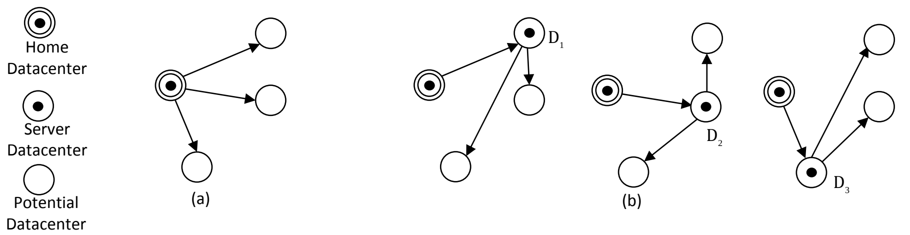


Figure 6.1: Replica creation via (a) home DC and (b)potential DCs D1, D2, and D3.

access latency, and the number of client DCs issued Puts and Gets. Table 6.1 summarizes key notations used in this chapter.

We define an objective function as to choose the placement of the object replicas (i.e., server DCs $d_r(t)$) and to determine the assignment of client DCs $d_c(t)$ to a server DC $d_r(t)$ so that the replica creation, storage, Get, Put, and potential migration costs for the object during $t \in [1...T]$ are minimized. To find the objective function, we formally define the following costs.

Replica Creation Cost: Once user creates an object in his/her home DC, the system may need to replicate this object in the DCs d. To do so, the system first requires to read the data from either the home DC d_h or other DC d_r storing the replica of the object, and then to write it into the DCs d. We refer this cost as a *replica creation cost*. To minimize it for r replicas, the system should find the minimum of the following two costs. (i) The system directly reads the object from the home DC d_h and replicates in $(r-1)$ DC[1]. This cost equals to $v \times (r-1) \times O(d_h)$ (see Fig. 6.1a). (ii) The system first reads the object from the home DC d_h to the DC d' as a server DC, and then from this DC to $(r-2)$ DCs since a replica is stored in each DC d_h and DC d' (see Fig., 6.1b). In this case, the cost is $v \times (O(d_h) + (r-2) \times O(d'))$. To minimize the cost in this case, the system requires to calculate the cost for each DC $d' \in D - \{d_h\}$ as a server DC. Therefore, the replicas creation cost is defined as

$$min_{d \backslash d_h}[\,(r-1) \times O(d_h), O(d_h) + (r-2) \times min_d\, O(d)] \times v. \tag{6.1}$$

Since in this step, the number (r) and location (d_r) of replicas have not been still specified, the system requires to calculate the *lower-bound number of replicas* as summarized in

[1] The object in the home DC is considered as a replica.

Algorithm 6.1: The lower-bound number of replicas

```
   Input  : D: a set of DCs $d$, $D_c$: a set of client DCs $d_c$, latency between each pair of
            DCs, and latency constraint $L$
   Output: $\lfloor r \rfloor$
 1 Initialize: $\lfloor r \rfloor \leftarrow 0$
 2 forall $d \in D$ do
 3     forall $d_c \in D_c$ do
 4         if $l(d, d_c) \leq L$ then
 5             Assign DC $d_c$ to DC $d$
 6         end
 7     end
 8 end
 9 Sort DCs $d$ according to their assigned number of $d_c$s in descending order.
10 while $D_c \neq \emptyset$ do
11     Select DC $d$ as $d_r$ and remove its assigned $d_c$s form $D_c$ as well as from the set of
       client DCs assigned to other DCs $d$ which are not still selected a server DCs.
12     $\lfloor r \rfloor \leftarrow \lfloor r \rfloor$+1
13 end
```

Algorithm 6.1. This algorithm assigns client DCs d_c to each DC d if the latency between DCs d_c and d is within the latency constraint (lines 2-8). Then, the algorithm sorts DCs d according to their number of assigned client DCs d_c (line 9), and finally selects the DC d one after another as a server DC d_r until all DCs d_c are served by a server DC d_r (lines 10 - 13).

Storage Cost. The storage cost of an object in time slot t is equal to the storage cost of all its replicas in DCs d_r. Thus, this cost is equal to

$$\sum_{d_r} S(d_r) \times v. \tag{6.2}$$

Get Cost. The Gets cost of an object in time slot t is the cost of Gets issued from all DCs d_c and the network cost for retrieving the object from DCs d_r. Hence, this cost is given by

$$\sum_{d_r} \sum_{d_c} (d_c \rightarrow d_r) \times r_{d_c} \times [t_g(d_r) + v \times O(d_r)]. \tag{6.3}$$

Put Cost: The Puts cost of the object in time slot t is the cost of Puts issued by all

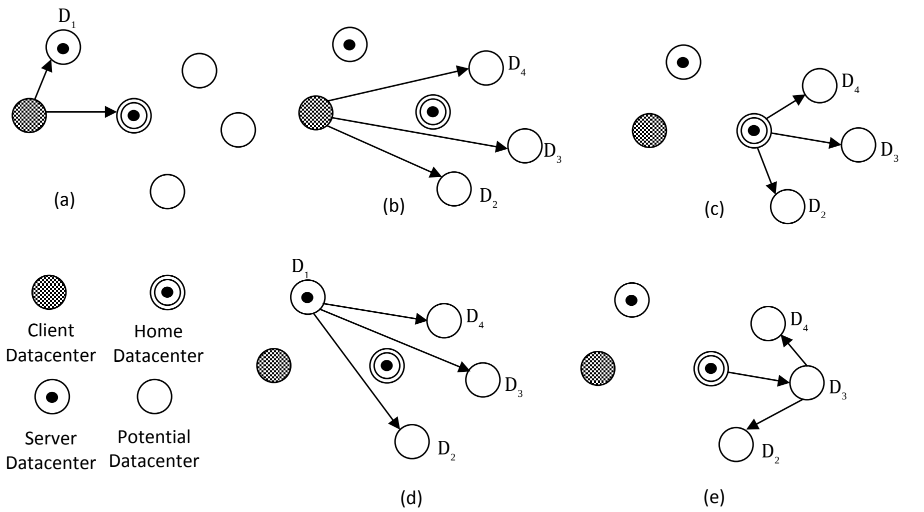


Figure 6.2: Put propagation policy. (a) Client DC first updates the server DC that serve it and the home DC. DCs hosting a replica are updated via (b) the client DC, (c) the home DC, and (d) the server DC that serve the client DC (i.e., DC D1). (e)The relayed propagation via DC D2 which is updated by the home DC.

client DCs and the propagation cost to keep slave replicas of the object consistent. For this purpose, the client DC first updates the server DC that serves it and the home DC that hosts the master replica (Fig. 6.2a). Then, one DC is selected to propagate data to other DCs. The criteria for this selection is to minimize the propagation cost. To this end, each DC should be selected to propagate data. If the selected DC already was updated, then this DC propagates data to update other DCs. For example, as shown in Figs. 6.2b-6.2d, the client DC, the server DC D1, or the home DC is selected to updates replicas in DCs D2, D3, and D4. Otherwise, if the selected DC was not already updated, it requires first to be updated via one of the updated DC and then it propagates data to the remaining DCs (i.e., $r-3$ DCs). For example, as depicted in Fig. 6.2e, if the selected DC to propagate data is DC D3, then DC D3 is first updated via the home DC, the server DC D1, or the client DC (depending to which DC has the minimum cost to transfer data from it to DC3), and then it propagates data to DCs D2 and D4. Thus, Puts cost is defined as

$$\begin{aligned}&\sum_{d_c}[(w_{d_c} \times (\left\{\begin{array}{ll} v(O(d_c)) + t_p(d_h), & if\, d_c \in D_r(t) \\ v(2O(d_c)) + t_p(d_h) + t_p(d_c \rightarrow d_r), & o.w. \end{array}\right) + \\ &\min_{d_p} \sum_{d \backslash d_h \wedge d_c \rightarrow d_r} \left\{\begin{array}{ll} v(O(d_p)) + t_p(d), & if\, d_p\, is\, d_c, d_h, or\, d_c \rightarrow d_r \\ \min\limits_{d' \backslash d_c, d_h, d_c \rightarrow d_r} (vO(d') + t_p(d_p)) + v(r-2)O(d_p + t_p(d)), & o.w. \end{array}\right.]\end{aligned} \qquad (6.4)$$

In the above Equation, the first set bracket calculates the updated cost of the home DC and the server DC that serves the client DC. The second set bracket calculates the minimum propagation cost to update $r-2$ replicas.

Migration Cost: The set of server DCs denoted by D_r for an object can be different in $t-1$ and t. That is, $D_r(t-1) \neq D_r(t)$. This happens when access pattern on the object changes, and consequently the object transits from hot-spot status to cold-spot status and vice versa. It is more cost-effective to replicate the object in a DC with the lower network cost as long as it is in the hot-spot status. Upon transition of the object to the cold spot status, it is probably more profitable to migrate it to a DC with the lower storage cost. This transition of the object across DCs incurs a *migration cost*. To optimize it, the object should be migrated from $d_r \in D_r(t-1)$ with the minimum network cost to $d_r \in D_r(t)$. Thus, the migration cost for each replica of the object in DC $d_r \in D_r(t-1)$ is as

$$C_M^{d_r}(t-1,t) = \begin{cases} 0 & if\, d_r \in D_r(t-1) \wedge D_r(t) \\ \min\limits_{d_r \in D_r(t-1)} O(d_r) \times v(t-1)) & o.w. \end{cases} \tag{6.5}$$

As seen in Equ. (6.5), if the placement of the replica in t and $t-1$ is the same, then migration cost is zero and the replica does not need to be migrated. Otherwise, users incur a migration cost when the replica is migrated from DC d_r in $t-1$ to DC in t. The benefit-cost obtained from replica migration is very important for users to make a decision on whether to migrate the replica or not. If this benefit covers the migration cost, then replica migration can probably be performed. To make a wise decision in this respect, we propose the following strategy as summarized in Algorithm 6.2.

For ease of algorithm explanation, we introduce two notations. Assume that $t_m^{d_r}$ denotes the last time of migration for the replica object in DC d_r. Also suppose $C_L^{d_r}(t_m, t)$ is the summation of benefit-cost that is lost for the replica in DC d_r during period $[t_m^{d_r}, t]$. The benefit-cost for each time slot during this period is equal to the difference between (i) the residential cost of the replica as if it is stored in the DC $d_r(t-1)$, i.e., old location,

Algorithm 6.2: Optimization of replicas migration

```
   Input  : $D_r(t-1)$, $D_r(t)$, latency between each pair of DCs, and latency constraint
            $L$
   Output: $D^*_r(t)$: the optimized location of replicas in $t$
1  Find the intersection set of $D_r(t-1)$ and $D_r(t)$ sets and remove DCs in the
   intersection set from both $D_r(t-1)$ and $D_r(t)$.
2  forall $d_r \in D_r(t)$ do
3      $C(d_r, t) \leftarrow$ Calculate residential cost according to Equs. (6.1 - 6.4)and the
       migration cost based on Equ. (6.5) .
4      forall $d'_r \in D_r(t-1)$ do
5          $C_R(d'_r(t-1), d_c(t) \ \rightarrow \ d'_r(t-1)) \leftarrow$Assume as if all $d_c$s assigned to $d_r$ are
           served by $d'_r$ subject to $l(d_c, d'_r) \leq L$ and calculate the residential cost based
           on Equs. (6.1 - 6.4) .
6          $C^{d_r}_B(t) \leftarrow [C_R(d'_r(t-1), d_c(t) \ \rightarrow \ d'_r(t-1)) - C(d_r, t)]$
7          /*Migration happens*/
8          if $C^{d_r}_L(t_m, t-1) + C^{d_r}_B(t) \geq C^{d_r}_M(t-1, t)$ then
9              $D^*_r(t) \leftarrow D^*_r(t) \cup d_r$
10             $D_r(t-1) \leftarrow D_r(t-1) - \{d'_r\}$
11             $C^{d_r}_L(t_m, t) \leftarrow 0$, $t^{d'_r}_m \leftarrow t$
12             break.
13         end
14     end
15     /*Migration does not happens*/
16     if $d_r \notin D^*_r(t)$ then
17         Find $min_{d'_r} C(d'_r, t)$
18         $D^*_r(t) \leftarrow D^*_r(t) \cup d'_r$
19         $D_r(t-1) \leftarrow D_r(t-1) - \{d'_r\}$
20         $C^{d_r}_L(t_m, t) \leftarrow C^{d_r}_L(t_m, t-1) + [C_R(d'_r(t-1), d_c(t) \ \rightarrow \ d'_r(t-1)) - C(d_r, t)]$
21     end
22 end
```

and the client DCs $d_c \in D_c(t)$ are served by DC $d_r(t-1)$, and (ii) the residential cost of replica in the DC $d_r(t)$, i.e., new location. This benefit-cost is lost when the replica stays in the old location instead of being migrated to a new one since the benefit-cost, collected from time t_m to t, cannot cover the migration cost of replica from the old location in time $t^{d_r}_m$ to the new location in t.

Algorithm 6.2 excludes the object replicas which have the same location in $t-1$ and t since these replicas do not require migration (line 1). For those replicas having potential to be migrated, the algorithm first calculates the total of *migration* cost and *residential* cost (i.e., replica creation, storage, Put, and Get costs) of the replica hosted in $d_r \in D_r(t)$, i.e.,

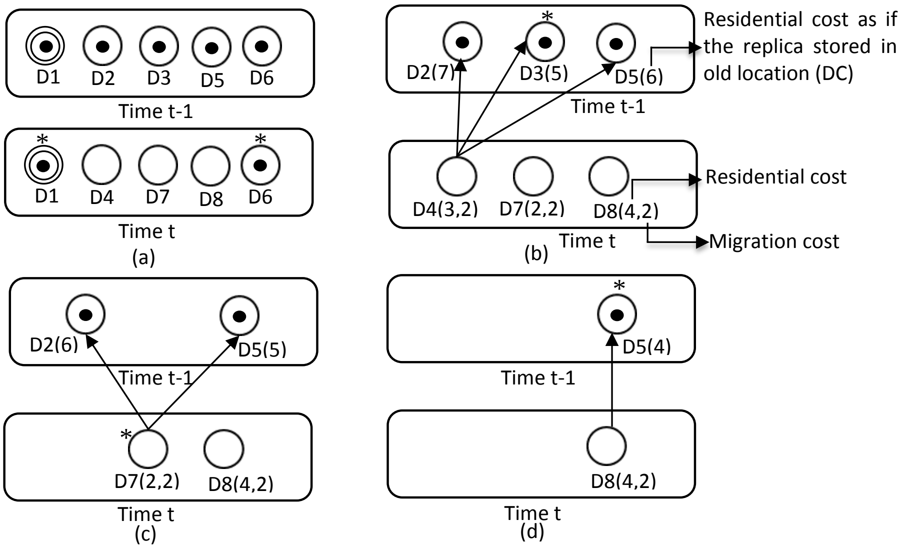


Figure 6.3: An example of illustrating the replica migration between two consecutive time slots.

$C(d_r, t)$– (line 3). Then, the algorithm computes the residential cost of the replica as if it is stored in each $d'_r \in D_r(t-1)$, i.e., $C_R(.)$– (line 5). According to these two values, $C(d_r, t)$ and $C_R(.)$, benefit-cost $C_B^{d'_r}(t)$ is calculated for time slot t (line 6). Based on the summation of benefit-cost that was lost during period $[t_m, t-1]$, the benefit-cost in time slot t, and migration cost of replica in DC d'_r in time slot t_m to that in time slot t, either of the following cases happens. **Case 1**: if the summation benefit-cost lost during $[t_m, t]$ is greater than the migration cost (i.e. line 8), then it is cost effective to migrate the replica from its location in $t-1$ (i.e., $d'_r \in D_r(t-1)$) to the location in t (i.e., $d_r \in D_r(t)$)–(line 6). In this case, the DC d_r is added to $D_r^*(t)$, and the DC d'_r is removed from $D_r(t-1)$ to avoid comparing it with the next DCs which may host a migrated replica (lines 9-10). Also, $C_L^{d_r}(t_m, t)$ is set to zero and the migration time of replica in DC d_r is updated to the current time t in order to evaluate the next potential migration time for this replica (line 11). **Case 2**: Otherwise, if the benefit-cost to bes lost during period $[t_m, t]$ cannot cover the migration cost of replica from DC d'_r in time t_m to that one in time t, then the replica migration does not happen (line 16). Thus, the algorithm finds the DC d'_r with the minimum residential cost (i.e., $min_{d'_r} C_r(d'_r, t)$), and then it adds the DC d'_r to $D_r^*(t)$ and removes it from $D_r(t-1)$ (lines 17-19). Also, it adds the benefit-cost that is lost in time slot t to the summation of those during period $[t_m, t-1]$ (line 20).

To show how this algorithm works, we will give a simple example as shown in Fig.

6.3. In this example, the DCs that host replicas in time $t-1$ are D1, D2, D3, D5, and D6. Assume the potential DCs can store replicas in time t are D1, D4, D7, D8, and D6. Since DCs D1 (as home DC) and D6 are in both sets, they are excluded from replica migration process. Thus, Algorithm 6.2 should make a decision on whether to migrate replicas to new DCs (i.e., D4, D7, and D8) in time t.

To this end, Algorithm 6.2 calculates residential and migration costs as if the replica is stored in new DCs. As shown in Fig. 6.10b, the algorithm first calculate these costs for DC D4, as represented in a pair of numbers in parenthesis. Then, it calculates the residential cost of replica as if it stays in the old DCs (i.e., D2, D3, and D5), as represented in parenthesis. Since the summation of residential and migration costs of the replica in DC4 (3+2) is less than or equal to the residential cost of replica in DCs D2 (residential cost=7), DC (5), and D5 (6). Thus, the replica stays in the old DC with the lowest residential cost, i.e., DC D3 (marked by an asterisk) and DC D4 is excluded for the next decision on the replica placement. For next decision, as shown in Fig. 6.3c, the residential cost of replica to be probably stored in DC D7 is calculated for two old DCs: D2 and D5. Since the residential and migration costs of replica in DC D7 (2+2) is less than the residential cost in DC D2 (6), DC D7 is selected to host another object replica. Thus, DCs D7 and D2 are excluded for next decision. In the same way, DC D5 is selected to host the last replica of the object.

The above discussed strategy uses the *stop and copy* technique in which the application is served by the source DCs hosting the replicas in $t-1$ for Gets and destination DCs storing the replica in t for Puts during migration [65]. This technique is used by the single cloud system such as HBase[1] and ElasTraS [56], and in Geo-replicated system [169]. As we desire to minimize the monetary cost of migration, we use this technique in which the amount of data moved is minimal as compared to other techniques leveraged for live migration at shared process level of abstraction[2]. We believe that this technique does not affect our system performance due to (i) the duration of migration for transferring a bucket (at most 50MB, the same as in Spanner [52]) among DCs is considerably low (i.e., about a few seconds), and (ii) most of *Gets* and *Puts* are served during the hot-spot status,

[1] Apache HBase. https://hbase.apache.org/book.html

[2] In cloud-based transactional database in the context cloud, to achieve elastic load balancing, techniques such as *stop and copy*, *iterative state replication*, and *flush and migrate* in the process level are used. The interested readers are referred to [65] and [58].

and consequently the access rate to the object during the migration, which is happening in the cold-spot status, is considerably low based on the access pattern[3].

Therefore, with the defined residential and migration costs based on Equs. (6.1-6.5), the total cost of the object replicas in t is defined as:

$$C_R(d_r(t), d_c(t) \ \rightarrow \ d_r(t)) + C_M^{d_r(t-1)}(t-1, t), \tag{6.6}$$

where $C_R(.)$ is the summation of the replica creation, storage, Put, and Get costs (Equs. 6.1-6.4), and is referred as the *residential cost*.

Besides the total cost optimization, our system respects the latency Service Level Objective (SLO) for Gets and Puts. The latency for a Get or Put is considered the time taken from when a user issues a Get or Put from the DC d_c to when he/she gets a response from the DC d_r that hosts the object. The latency between d_c and d_r is denoted by $l(d_c, d_r)$ and is evaluated based on the round trip times (RTT) between these two DCs since the size of objects is typically small (e.g., tweets, photos, small text file), and thus data transitions are dominated by the propagation delays, not by the bandwidth between the two DCs. For applications with large objects, the measured $l(d_c, d_r)$ values capture the impact of bandwidth and data size as well. This performance criterion is integrated in the cost optimization problem discussed in the following subsection.

6.2.3 Cost Optimization Problem

Given the defined system and cost models, we are required to determine the location of object replicas (d_r) and the client DCs (issuing Gets and Puts) redirection to a server DC ($d_c(t) \ \rightarrow \ d_r(t)$) so that the overall cost for the object replicas (Equ. 6.6) during $t \in [1...T]$ is minimized. This is defined as the following *cost optimization problem*:

[3] Note that our system is not designed to support database transactions, and this technique is just inspired by this area.

$$\min_{\substack{d_r(t)\\ d_c(t)\to d_r(t)}} \sum_t C_R(.) + C_M(.) \tag{6.7}$$

s.t. (repeated for $\forall t \in [1...T]$)

(a) $\sum_{d_c} d_c(t) \to d_r(t) = 1, \qquad \forall d_r \in D_r(t)$

(b) $\sum_{d_r} d_c(t) \to d_r(t) = |D_c(t)|, \qquad \forall d_c \in D_c(t)$

(c) $\frac{\sum_{d_r}\sum_{d_c} r_{d_c(t)} \times l(d_c(t), d_r(t))}{\sum_{d_c} r_{d_c(t)}} \leq L, \qquad \forall d_c \in D_c(t), d_r \in D_r(t)$

(d) $\sum_{d_r} d_c(t) \to d_r(t) = |D_r(t)|, \qquad \forall d_c \in D_c(t), Puts$

(e) $l(d_c(t), d_c(t) \to d_r(t)) \leq L \qquad \forall d_c \in D_c(t), d_r \in D_r(t).$

To optimally solve the above problem, we are required to replace $d_r(t)$ with $\alpha^d(t)$ that is associated to DC d. The introduced variable $\alpha^d(t)$ is binary, being 1 if the DC $d \in D$ hosts a replica of the object, otherwise 0. Thus, we apply a constraint as below:

(f) $\sum_{d \in D} \alpha^d(t) \geq 1, \qquad \forall d \in D.$

In this cost optimization problem, $C_R(.) + C_M(.)$ is calculated based on Equ. (6.6), and L is as the upper bound of delay for Gets and Puts on average to receive response. The value of L is defined by the users as their SLO. To reflect the real-world practicality, we consider the following constraints. Constraint (a) ensures that a single server DC d_r for every client DC d_c. Constraint (b) guarantees all client DCs are served. Constraint (c) enforces the average response time of Gets in range of L. Constraints (d) and (e) indicate that the Puts are received by all server DCs in the average response time L. Constraint (f) ensures at least a replica of the object is stored in DCs at any time.

6.3 Cost Optimization Solution

We first provide a brief demonstration of the optimal solution for the cost optimization problem, and then propose a heuristic solution to perform well in practice.

6.3.1 Optimal Solution

To optimally solve the cost optimization problem, we should find the values of $\alpha^d(t)$s and $d_c(t) \rightarrow d_r(t)$s, the same as that discussed in Section 5.3. The value of $d_c(t) \rightarrow d_r(t)$s can be determined via linear programming once the value of $\alpha^d(t)$s is fixed. To enumerate value of all $\alpha^d(t)$s, we need to enumerate all $r-$combinations of a given set of DCs (i.e., $\binom{|D|}{r}$). Since the value of r can be ranged between 1 and $|D|$, the number of combination of $\alpha^d(t)$s is $\sum_{r=1}^{|D|} \binom{|D|}{r} = 2^{|D|} - 1$[4].

To find the optimal solution, we use a dynamic algorithm the same as one in Section 5.3, which only differs in the number of combinations of $\alpha^d(t)$s (i.e., the combinations of DCs with r ($1 \leq r \leq |D|$) replicas. Thus, our algorithm calculates the cost for $(2^{|D|} - 1)^2$ combinations of DCs for each time slot $t \in [1...T]$. This calculation takes time complexity of O $((2^{|D|} - 1)^2 T T_{lp})$, where T_{lp} is the required time to solve the linear programming for finding the value of $d_c(t) \rightarrow d_r(t)$s. As the optimal solution is computationally intractable, we seek practical heuristic solution.

6.3.2 Heuristic Solution

We first propose Replica Placement based on Covered Load Volume (RPCLV) Algorithm. This algorithm makes the iterative decision on the assignment of client DCs to the potential DCs which are within the latency constraint L of the client DCs. Then, by using this algorithm as well as Algorithms 6.1 and 6.2, we propose an algorithm which calculates the optimized cost of the object replicas in each $t \in [1...T]$.

Replica Placement based on Covered Load Volume (RPCLV) Algorithm

The RPCLV algorithm is inspired from an approximation algorithm for the Set Covering Problem [48]. This algorithm stores a replica of the object in the potential DC which has the minimum proportion of the residential cost (Equ. 6.6) and potential migration cost (Equ. 6.5) to the volume data (in bytes) read from and written into the potential DC by the client DCs. Clearly, these client DCs are within the latency constraint L of the potential

[4] Note that neither Algorithm 6.1 nor Algorithm 6.2 are used in optimal solution since this solution enumerates all values of r as well as all combinations of object migration between $d_r \in D_r(t-1)$ and $d_r \in D_r(t)$

Algorithm 6.3: Replica Placemenet based on Covered Load Volume (RPCLV)

```
   Input  : $D_c$, latency between each pair of DCs, and latency constraint $L$
   Output: $D_r$: the location of replicas in $t$
1  Initialize: $D_r \leftarrow \emptyset$
2  /*Assignment feasible client DCs to the potential DCs. */
3  forall $d \in D$ do
4      forall $d_c \in D_c$ do
5          if $l(d_c, d) \leq L$ then
6              Consider the DC $d$ as a potential DC $d_p$ that hosts a replica of the object
7              $D_c^{d_p} \leftarrow$ assign $d_c$ to the DC $dp$
8          end
9      end
10     $D_p \leftarrow D_p \cup d_p$
11 end
12 /*Assign client DCs to the potential DCs based on the cost-volume metrics until all client
   DCs are covered. */
13 while $D_c \neq \emptyset$ do
14     forall $d_p \in D_p$ do
15         $P_{CV}(d_p) = \frac{\sum_{d_c \to d_p} C_R(d_r, d_c \to d_p) + C_M^{d_r}(t-1,t)}{\sum_{d_c \to d_p}(r_{d_c} + w_{d_c}) \times v}$
16     end
17     Find $min_{d_p} P_{CV}(d_p)$ and store a replica of the object in $d_p$ as $d_r$
18     $D_r \leftarrow D_r \cup d_r$
19     $D_c \leftarrow D_c - D_c^{d_p}$
20     Remove $D_c^{d_p}$ from $D_c^{d_{p'}}$ and recalculate $P_{CV}(d_{p'})$ for all $d_{p'}$ where $p' \neq p$.
21 end
22 /*revises the replica creation cost */
23 Adds $|(|D_r| - \lfloor r \rfloor) \times min_{d_r \in D_r} O(d_r)|$ to replicas creation cost if ($|D_r| > \lfloor r \rfloor$) and
   subtract it if ($|D_r| < \lfloor r \rfloor$).
```

DC, but they are not yet assigned. The algorithm selects this potential DC as a server DC and finds the next best potential DC to host a replica until all client DCs are assigned to server DCs. The details of this process is given in Algorithm 6.3.

The RPCLV algorithm first assigns client DCs to each potential DC $dp \in D$ if the latency between the client DC d_c and the potential DC d_p is within the latency constraint L (lines 3- 10). Then, it calculates $P_{CV}(d_p)$ as the proportion of the residential and migration costs of the replica stored in the DC d_p to the total data read and written by the set of client DCs assigned to d_p (i.e., $D_c^{d_p}$)–lines (14-16). After that, RPCLV (i) selects d_p with the minimum value of P_{CV} and stores a replica in this DC (lines 17-18), and (ii) removes all client DCs assigned to this d_p from the client DCs (i.e., D_c) as well as removes from

Algorithm 6.4: The Cost Optimization Algorithm

```
   Input  : $D_r(t-1)$, $D_r(t)$, and $T$.
   Output: $C_{ove}$
1  :The overall cost Initialize: $C_{ove} \leftarrow 0$
2  $t \leftarrow 1$
3  Call the RPCLV algorithm and determine $D_r(t)$ as well as $D_c^{d_r}$ for each $d_r \in D_r(t)$
4  $C_{ove} \leftarrow$ Calculate the residential Cost $C_R(.)$ based on Equ. (6.6)
5  for $t \leftarrow 2$ to $T$ do
6      Call the RPCLV algorithm and determine $D_r(t)$ as well as $D_c^{d_r}$ for each
       $d_r \in D_r(t)$
7      /*$C_R(.)$ and $C_M(.)$ are calculated based on Equ. (6.6)*/
8      if $D_r(t-1) \neq D_r(t)$ then
9          Call the optimization of replicas migration algorithm (Algorithm 6.2)
10         $C_{ove} \leftarrow C_{ove} + C_R(.) + C_M(.)$
11     else
12         $C_{ove} \leftarrow C_{ove} + C_R(.)$
13     end
14 end
```

those covered by the DC $d_{p'}$ for every $p' \neq p$ and recalculate $P_{CV}(d_{p'})$ (lines 19-20). Finally, RPCLV revises the replicas creation cost since in RPCLV the replica creation cost is calculated based on the lower-bound number of replicas (i.e., $\lfloor r \rfloor$) specified by Algorithm 1. Thus, RPCLV adds cost $|(|D_r| - \lfloor r \rfloor) \times min_{d_r \in D_r} O(d_r)|$ to the replicas creation cost if $|D_r| > \lfloor r \rfloor$; Otherwise if $|D_r| < \lfloor r \rfloor$, it subtracts this value from the replicas creation cost (line 23). Clearly, if $|D_r| = \lfloor r \rfloor$ there is no need to change the cost.

Cost Optimization Algorithm

Algorithm 6.4 gives the pseudo codes of the proposed heuristic solution which is composed of Algorithms 6.1, 6.2, and 6.3. In Algorithm 6.4 first RPCLV is invoked in order to determine the location of replicas (i.e., $d_r(t)$) as well as the assigned client DCs to server DCs (i.e., $d_c(t) \rightarrow d_r(t)$). According to the values of $d_r(t)$ and $d_c(t) \rightarrow d_r(t)$, first the residential cost of the replicas (i.e., $C_R(.)$) based on Equ. (6.7) is calculated for time slot $t = 1$ (lines 2-4). Then, for each time slot $t \in [2...T]$, similarly Algorithm 6.4 finds the replicas location and the assignment of client DCs to server DCs (line 6) and checks whether the location of replicas in $t-1$ and t are different or not. If they are different (i.e., $D_r(t-1) \neq D_r(t)$), Algorithm 6.2 is called to determine which object replicas of the

Table 6.2: The time complexity of Algorithms 6.1 - ch6:alg:CVRP.

Algorithm			
Algorithm 6.1	O($\|D\|^2$) Lines (2-8)	O($\|D\|log\|D\|$) Line 9	O($\|D\|$) Lines (10-14)
Algorithm 6.2	O($\|D\|$) Line 1	O($\|D\|^2$) Lines (2-18)	
Algorithm 6.3	O($\|D\|^2$) Lines (3-11)	O($\|D\|^2$) Lines (13-21)	O($\|D\|$) Line 23

object can be migrated and then the residential and migration costs are calculated based on Equ. (6.7) (lines 8-10). Otherwise (i.e., $D_r(t-1) = D_r(t)$), the migration of replicas does not happen and only the residential cost of replicas is calculated (line 12).

To calculate the time complexity of the heuristic solution, we require to compute the time complexity of Algorithm 6.4 that is composed of Algorithms 6.1, 6.2, and 6.3. Table 6.2 summarizes the time complexity of these algorithms. Algorithm 6.4 invokes Algorithm 6.3 which takes O($|D|^2$)-line 3. Then, for each time slot $t \in [2...T]$, it invokes RPCLV (Algorithm 6.3) with the time complexity of O($|D|^2$), and also runs Algorithm 6.2 with the time complexity of O($|D|^2$) if the replicas migration happens. Thus, the "for" loop takes $O(|D|^2T)$ (lines 5-14). With this analysis, the heuristic solution (Algorithm 6.4) yields the time complexity of $O(|D|^2T)$.

6.4 Performance Evaluation

We evaluate the proposed solution for replicas placement of the objects across Geo-distributed data stores with two storage classes. Our evaluations explore three key questions:

1. How significant is our solution in the cost saving?
2. How sensitive is our solution to different parameters settings which are likely to have effect on the cost saving?
3. How much time is required to migrate objects within and across regions?

We explore the first two questions via trace-driven simulations using the CloudSim discrete event simulator [35] and the Twitter workload [101]. Simulation studies enables us to evaluate our solution on a large scale (thousands of objects). We answer the last question via the implementation of our proposed solution on the Amazon and Microsoft

Azure cloud providers. This implementation, discussed in section 6.5, allows us to measure the latencies that are required for Puts, Gets, and data migration in a real test-bed.

6.4.1 Experimental setting

We use the following setup for DC specifications, objects workload, users location, and experiment parameters setting.

DCs specifications: We model 18 DCs in CloudSim Toolkit [36], and among these DCs, nine are modelled according to Amazon and nine according to Microsoft Azure in different regions: 7 in America, 4 Europe, and 7 in Asia Pacific. We use two storage classes from each cloud providers: S3 and RRS from Amazon and ZRS and LRS from Microsoft Azure. S3 and ZRS host objects with hot-spot status and the remaining storage classes store objects with cold-spot status. The price of these storage classes and network services are set for each DC based on AWS and Microsoft Azure as of November 2016[5].

Objects workload: We use Twitter traces [101] which includes users profile, a user friendship graph, and tweet objects sent by users over a 5-year period. We focus on tweet objects posted by the users and their friends on their timeline, and obtain the number of tweets (i.e., number of Puts) from the dataset. Since the dataset does not contain information of accessing the tweets (i.e., number of Gets), we set a Get/Put ratio of 30:1, where the pattern of Gets on the tweets follows Longtail distribution [21]. This pattern mimics the transition status of the object from hot- to cold-spot status. The size of each tweet object varies from 1 KB to 100 KB in the trace.

Users location: We assign each user to a DC as his/her home DC based on the following policies. (i) Closest-based policy: with the help of Google Maps Geocoding API[6], we convert the locations of users in their profiles to Geo-coordinates (i.e., latitude and longitude). Then, according to the coordination of users and DCs, we assigned users to the nearest DC based on their locations. In the case of two (or more) DCs with the same distance from the user, one of these DCs is randomly selected as the home DC for the user.

[5] Amazon S3 storage and data transfer pricing. https://aws.amazon.com/s3/pricing/
Azure storage pricing. https://azure.microsoft.com/en-us/pricing/details/storage/
Azure data transfer pricing. https://azure.microsoft.com/en-us/pricing/details/data-transfers/

[6] The Google maps geocoding API. https://developers.google.com/maps/documentation/geocoding/intro

(ii) Network-based policy: users are directed to the cheapest DC in the network cost. (iii) Storage-based policy: users are mapped to the cheapest DC in the storage cost. In the last two policies, the selected DC must be within the SLO defined by users, and in the case of two (or more) DCs with the same cost in either network or storage, one of the closest DC is selected as the same way used in the first policy. We use closest-based policy to assign friends of the user to a DC. The user's friends are derived from the friendship graph of dataset.

Experiment parameters setting: We measured inter-DCs latency (18*18 pairs) over several hours using instances deployed on all 18 DCs. We run Ping operation for this purpose, and used the medium latency values as the input for our experiments. We consider two SLOs for the values of Get and Put latencies: 100 ms and 250 ms. Recall that the Put latency is the latency between the client DC to the server DC that serves it. The stored data in the system depends on the size of tweet objects, the number of friends of the users and the rate of Get (write). To understand the effects of the total stored data size in data stores on the cost performance, we define "quantile volume" parameter. This parameter with the value of "x" means that all data stores only store "x" percent of the generated total data size. We use one-month (Dec. 2010) of Tweeter traces with more than 46K users, posting tweet on their timeline, for our experiments conducted over a 60-day period.

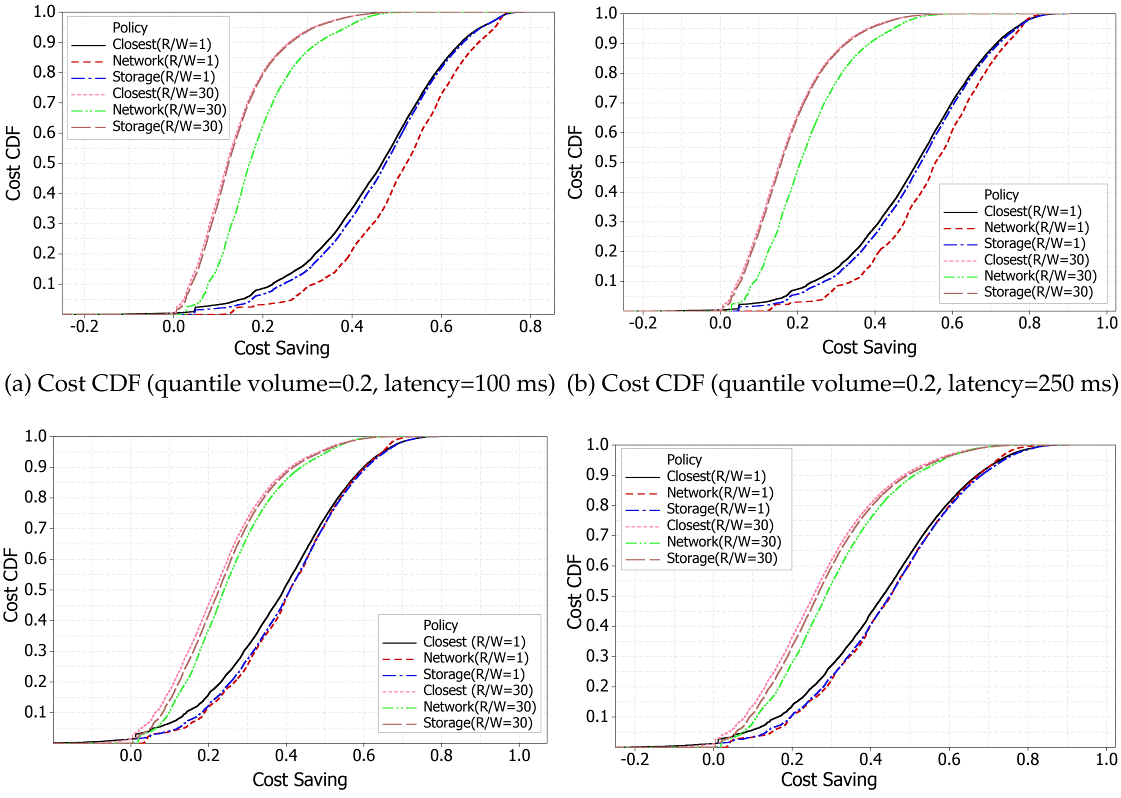
6.4.2 Results

We compare the cost savings gained by the proposed heuristic solution with the following benchmark algorithm. We also investigate the effects of parameters as their values are changed.

Table 6.3: Summary of Simulation Parameters

	Policy	Quantile Volume	Latency (ms)	Read to Write Ratio
Default	Closest-based	0.2,1	100, 250	1,30
Range	Closest-, network-, and Storage-based	0.2-1	50 -250	1-30

Benchmark Algorithm and the Range of Parameters: This algorithm permanently stores the objects in the home DC. It also replicates a replica of objects in the client DCs, upon issuing Get requests, via the home DC. The replica is stored in the client DCs until they



(a) Cost CDF (quantile volume=0.2, latency=100 ms) (b) Cost CDF (quantile volume=0.2, latency=250 ms)

(c) Cost CDF (quantile volume=1, latency=100 ms) (d) Cost CDF (quantile volume=1, latency=250 ms)

Figure 6.4: Cost saving of closest-, network-, and storage-based polices under tight (100 ms) and loose latency (250 ms).

receive Gets and Puts, and thus the replica is not allowed to migrate to another DC. This algorithm, though simple, is effective to measure the cost performance of using data stores with two storage classes offered by different CSPs. In all experiments, we normalize the incurred cost of the heuristic solution to the cost of the benchmark algorithm by varying the following parameters: policy, quantile volume, SLO latency, and read to write ratio. Each parameter has a default value and a range of values as summarized in Table 6.3. This range is used for studying the impact of the parameter variations on the cost performance of the proposed solution.

Cost performance: We present simulation results in Fig. 6.4, where the CDF of the normalized cost savings is given. These cost savings are grouped by the default value of quantile (0.2,1) and latency (100 ms and 250 ms). From the results, we observe the following facts. First, there is a hierarchy between policies in the cost savings, where the network-based policy outperforms the storage-based policy, which in turn outweighs

Table 6.4: Average cost performance normalized to the benchmark algorithm cost

		Latency=100 ms		Latency=250 ms	
Quantile volume	**Policy**	R/W=1	R/W=30	R/W=1	R/W=30
0.2	**Closest**	44.78%	13.61%	49.21%	17.12%
	Network-based	50.82%	18.78%	54.31%	22.32%
	Storage-based	46.00%	13.98%	50.48%	17.51%
1	**Closest**	38.00%	22.79%	42.05%	26.80
	Network-based	40.11%	25.43%	44.06%	29.89%
	Storage-based	39.91%	23.88%	44.08%	27.96%

the closest-policy. This is because that the network-based policy allows users to select DC which is cheaper than their closest DC. When we go deep into the cost savings obtained from each individual DC, we realized that users in California select cheaper DC within their specified SLO instead of Amazon'DC in California. Second, all policies provide cost savings for write-intensive objects (R/W=1) higher than read-intensive objects (R/W=30). For example, as shown in Figs. 6.4a and 6.4b, all policies cut costs up to 50% for all read-intensive objects, while they save similar costs for half of write-intensive objects and between 50%-80% for another half. Third, all policies cut costs for almost all objects, apart from 2-3% of the objects that incur slightly more costs than as if they were replicated in each client DCs requesting them (i.e., the strategy deployed by the benchmark algorithm).

Table 6.4 summarizes the average cost saving for each group of default values of quantile volume, R/W, and latency. We can see that the network-based policy achieves the highest cost saving, while the closest-based policy obtains the lowest, where the difference between these two policies is at most 6%. A comparison between the highest cost saving on average (obtained by the network policy for each group of default parameters) and the corresponding results in Fig. 6.4, we discover the following fact. As shown in Fig. 6.4a, all policies cut the costs above the highest average cost saving for at least 45% of write-intensive objects and for 25% of the read-intensive objects. As the quantile volume increases from 0.2 to 1, as shown in Fig. 6.4c, policies save costs above the highest average cost savings for 5o% of read- and write-intensive objects. Likewise, results remain the same, or even more improvements are experienced, for objects under loose latency (see

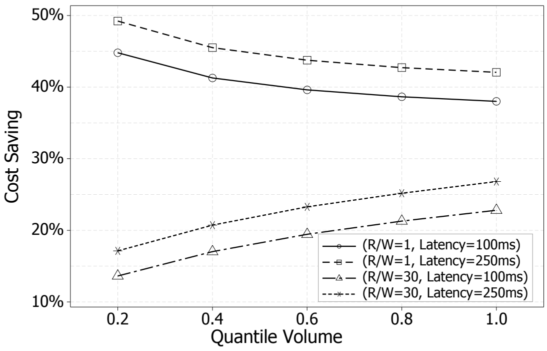


Figure 6.5: Cost saving of closest-based policy vs. quantile volume

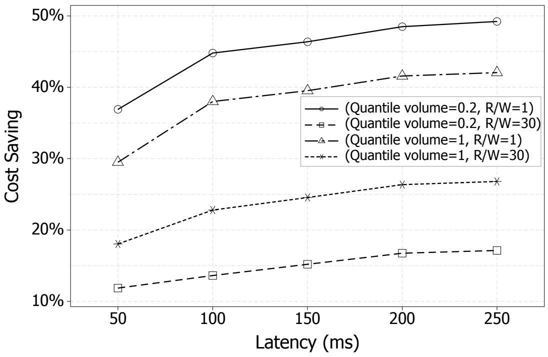


Figure 6.6: Cost saving of closest-based policy vs. latency

Figs. 6.4b and 6.4d). Thus, we can say that all policies cut costs for most of objects more than the highest cost saving on the average obtained from the network-based policy.

We now investigate the effect of different parameters on the cost saving. For the sake of brevity, from hereafter, we report the results only for default values of parameters, as shown in Table 6.3. We consider "closest" as the default value of the policy since users often are mapped to the closest DC.

Fig. 6.5 shows the effect of quantile volume by varying it from 0.2 to 1 with the step size of 0.2 on the cost saving. As the quantile volume increases from 0.2 to 1, cost savings slightly decrease by about 6% -7% for write-intensive objects under both latency constraints. The rational is that when quantile volume=0.2, the cost is dominated by write cost and our solution can optimize more cost. As the quantile volume increases to 1, the effect of this domination reduces. In contrast, cost savings increase by about 9% for read-intensive objects under both tight (100 ms) and loose (250 ms) latency constraints. This is because that as the the value of quantile volume increases, the effect of read cost reduces and storage cost becomes more dominant.

Fig. 6.6 illustrates the effect of latency on the cost savings. As the latency increase from 50 ms to 250 ms, as expected more improvement is observed in the cost savings for all default values of quantile volume and read to write ratio. This implies that there is a wider selection of DCs available with lower cost in storage and network resources for optimization under loose latency in comparison to tight latency.

Fig. 6.7 plots the effect of read to write ratio, varying from 1 (write-intensive objects) to 30 (read-intensive objects), on the cost savings. As the value of the ratio increases, the

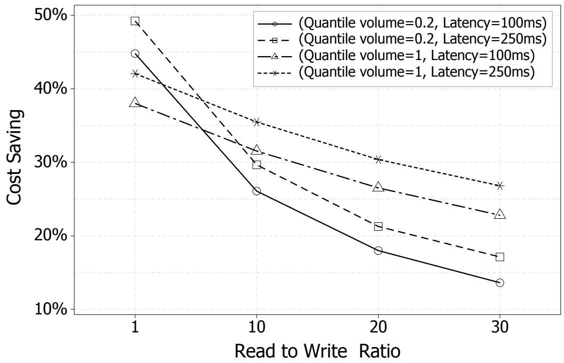


Figure 6.7: Cost saving of closest-based policy vs. read to write ratio

cost saving decreases. Under both latency constraints, the results show a 66% reduction in cost savings for quantile volume=0.2, and correspondingly at most a 42% reduction for quantile volume=1. This implies that the proposed solution is more cost-effective for write-intensive objects in comparison to the read-intensive objects due to efficient utilization of network resources for Puts.

6.5 Empirical Studies in Latency Evaluation

We implemented a prototype system to provide data access management across Amazon Web Service (AWS) and Microsoft Azure cloud providers. For this purpose, we use JAVA-based AWS S3[7] and Microsoft Azure[8] storage REST APIs. With this prototype, an individual end-user can (i) manage data across two well-known cloud providers, and (measure) the perceived latency for operations conducted on the data.

6.5.1 Data Access Management Modules

Our prototype system provides a set of modules that facilities users to store, retrieve, delete, migrate, list data across AWS and Microsoft Azure data stores. Tables 6.5 - 6.8 show the list of main web services that is used in the prototype system for data access across AWS and Microsoft Azure clouds. All these services are RESTful web services that utilize AWS S3 and Microsoft Azure storage APIs in Java. They produce response in the JavaScript Object Notations (JSON) format in successful cases and error message in

[7] Amazon S3 REST API `http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html`

[8] Azure storage REST API `https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/azure-storage-services-rest-api-reference`

the error cases. We use JSON format because it is a lightweight data-interchange format and easy to understand. In the following we discuss the provided web services in more details.

Table 6.5 shows the list of main modules that is provided by the prototype system for data management in AWS data stores. These modules are as follows:

- `amazonCreateS3Client`: This module provides users to create a client with the type of AmazonS3client for accessing the Amazon S3 web services. It also allows user to set a region for the created AmazonS3 client account.
- `createBucket`: This module creates a bucket in the AmazonS3 client specified by users. It also allows users to determine the Access Control List (ACL) in terms of private, publicRead, and PublicReadWrite.
- `amazonCreateFolder`: This module creates a folder in a bucket and allows users to determine the storage class of objects stored in a folder.
- `amazonUploadObject`: This module facilitates users to store objects in the specified directory which has a scheme like /AmazonS3 client/bucket/folder/.
- `amazonDownloadObject`: This module allows users to retrieve objects from AWS data stores in the specified directory with a scheme like /AamzonS3 client/bucket /folder/.
- `amazonDeleteFolder/Objeject`: As its name implies, it deletes folders or objects.
- `amazonListObjects`: This module lists folders in a bucket as well as the objects stored in a folder.
- `amazonChangeClassStorageFolder`: This module changes the storage class for objects stored in a folder.
- `amazonTransferFolder`: This module allows users to transfer objects (stored in a folder) from an AWS data store to another one.
- `amazonToAzureTransferFolder`: This module facilitates users to migrate objects from an AWS data store to Azure data store.

Table 6.6 summarizes the type and description of the main input parameters used in the above modules. It is worth nothing that the `storageClass` parameter can be one of the four constant values: `STANDARD`, `REDUCED_REDUNDANCY`, `STANDARD_IA` (IA for

Table 6.5: The modules used for data access management in AWS.

Module Name	Input Parameters	Output
amazonCreateS3Client	accesskey,secretkey	AmazonS3Client
createBucket	client,bucketName	Bucket
amazonCreateFolder	client,bucketName,folderName,storageClass	Folder
amazonUploadObject	client,bucketName,folderName,objectname, path,storageClass	put Object
amazonDownloadObject	client,bucketName,folderName,desPath	Get object(s)
amazonDeleteFolder/Objects	client,bucketName,folderName	Delete folder/objects
amazonListObjects	client,bucketName,folderName	Objects list
amazonChangeClassStorageFolder	client,bucketName,folderName,storageClass	Change storage class
amazonTransferFlder	srcClient,srcBucketName,srcFoldername,desClient, desBucketName	Transfer objects
amazonToAzureTransferFolder	srcClient,bucketName,srcFolderName,desClient, containerName,desFolderName	Transfer objects

Table 6.6: The input parameters used in Modules of AWS.

Input Parameter	Type	Description
accessKey	String	This key is uniquely assigned to the owner of AWS S3 account.
secretKey	String	This key is the password for the owner of AWS S3 account.
client	AmazonS3Client	This parameter allows users to invoke the service methods on AWS S3.
bucketName	String	This refers to the name of the bucket that contains folders.
folderName	String	This refers to the name of the folder containing objects.
storageClass	String	This specifies constants that include four storage classes of AWS S3.
objectName	String	This parameter specifies the name of object generated by users.
srcPath	String	This represents the path from which the data can be transferred.
desPath	String	This indicates the path to which the data can be transferred.

infrequent access), and GLACIER. In our prototype, we use the first two storage classes. For more details on these storage classes, please refer to Section 2.6.2. Some input parameters used in the above modules are the same in the type and description while they are different in the name. We excluded these parameters in Table 6.6 and give their details here. All (src/des)Client, (src/des)BucketName, and (src/des)FolderNmae parameters are respectively the same with the client, bucketName, and folderName parameters in the type and descriptions. The srcClient, srcBucketName, and srcFolderNmae parameters represent that from which client, bucket and folder data are transferred. The desClient, desBucketName, and desFolderNmae parameters indicate the location to that data are transferred.

Similarly, we provide a set of modules to manage data across Microsoft Azure data stores. As listed in Table 6.7, these modules are mostly similar to the discussed ones in the functionality. They are summarized as follow:

- azureCreateCloudBlobClient: This module creates an Azure cloud storage account in a region to access Azure cloud storage web services.
- createContainer: It creates a container in the the Azure storage account spec-

Table 6.7: The modules used for data access management in Microsoft Azure.

Module Name	Input Parameter(s)	Output
azureCreateCloudBlobClient	accessKey	CloudBlobClient
createContainer	client,containerName	Container
azureCreateFolder	client,containerName,folderName	Folder
azureUploadObject	client,containerName,folderName,objectname, path	put Object
azureDownloadObject	client,containerName,folderName,desPath	Get object(s)
azureDeleteFolder/Objects	client,containerName,folderName	Delete folder/objects
azureListObjects	client,containerName	Objects list
azureTransferFlder	srcClient,srcContainerName,srcFolderName,desClient, desContainerName, desFolderName	Transfer object(s)
azureToAmazonTransferFolder	srcClient,srcContainerName,srcFolderName,desClient, BucketName,desFolderName	Transfer object(s)

ified by users. This module can determine the type of ACL in the forms of Container, Blob, and OFF (i.e., no blob neither container). It is worth noting that container in Azure data stores and bucket in AWS data stores are the same in the concept.

- `azureTransferFolder`: It facilitates users to transfer objects (stored in a folder) from an Azure data store to another one.
- `azureToAmazonTrasnferFolder`: This allows users to transfer object from Azure data stores to Amazon data stores.
- `azureCreateFolder`,`azureUploadObject`,`azureDownloadObject`,`azure-DeleteFolder/Objeject`,`azureListObjects` modules respectively allow users to create folder, store objects, download object, and list objects in Azure data stores.

The modules of Microsoft Azure require input parameters which are similar to those of AWS data stores to the large extent. Table 6.8 gives a list of those that are only used in Microsoft Azure Modules. The `accessKey` parameter is a 512-bit storage access key which is generated when users create their storage accounts. The `client` parameter allows users to create containers in different Azure regions. The `containerName` parameter is an instance of the “CloudBlobContainer” class and its name is a string value used in Microsoft Azure modules. Note that the `desClient` parameter in the `amazonToAzureTransferFolder` module is an instance of the “cloudBlobClient” class; likewise this parameter in the `azureToAmazonTransferFolder` module is a reference variable of the “AmazonS3Client” class.

Table 6.8: The input parameters used in Modules of Microsoft Azure.

Input Parameters	Type	Description
accessKey	String	This key is used to authenticate when the storage account is accessed.
client	cloudBlobClient	This parameter allows users to invoke the service methods on blob storage.
containerName	String	This refers to the name of the container that contains the folders.

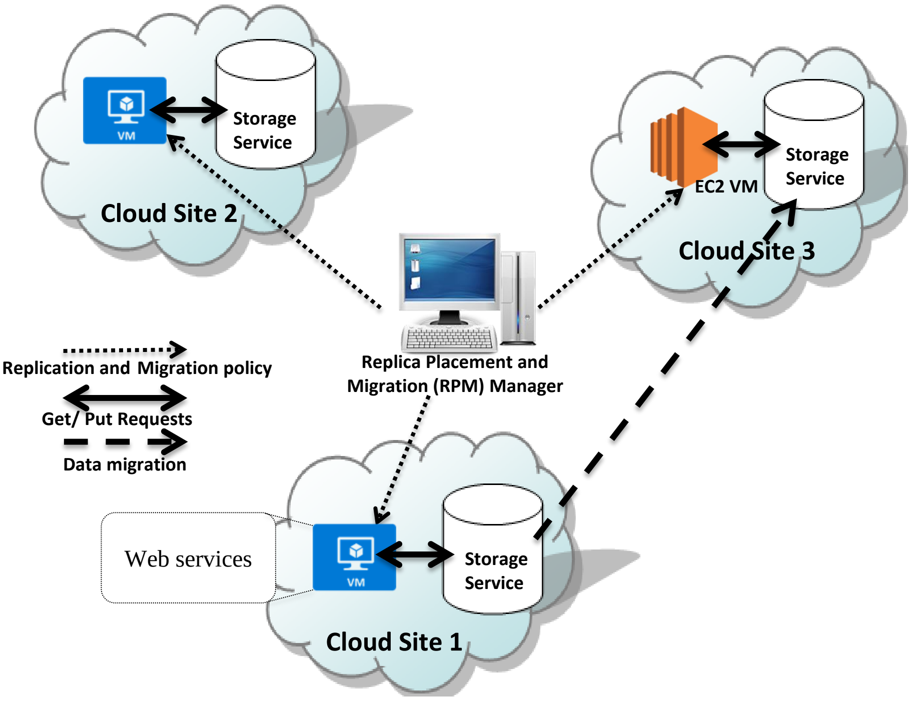


Figure 6.8: An overview of prototype

6.5.2 Measurement of Data Migration Time

We design a simple prototype as shown in Fig. 6.8. The way in which the deployed virtual machines (VMs) should serve Puts and Gets for each object is dictated by a central replica placement and migration (RPM) manager. The RPM manager makes decision on replica placement and migration across data stores based on the proposed heuristic solution. The RPM issues Http requests (REST call) to the VMs deployed in cloud sites and receives Http responses (JSON objects). The VMs process the received requests via the deployed web services that are implemented based on Spring Model-View-Controller (MVC) framework [83] as illustrated in Fig. 6.9.

To measure the time spent on data migration across DCs, we utilize the federation of cloud sites from Microsoft Azure and Amazon in our prototype. We span our prototype across 3 Microsoft Azure cloud sites in `Japan West`, `North Europe`, and `South Central US` regions and 3 Amazon cloud sites in `US East (North Virginia)`, `US`

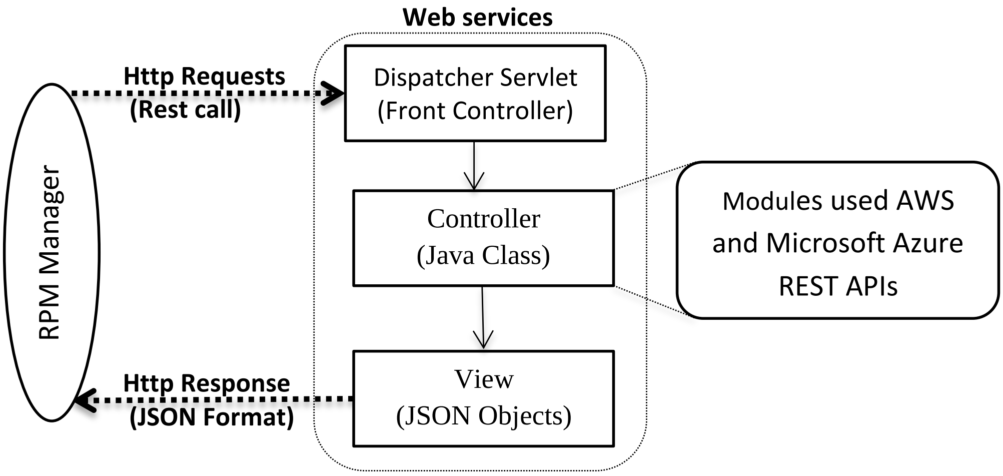
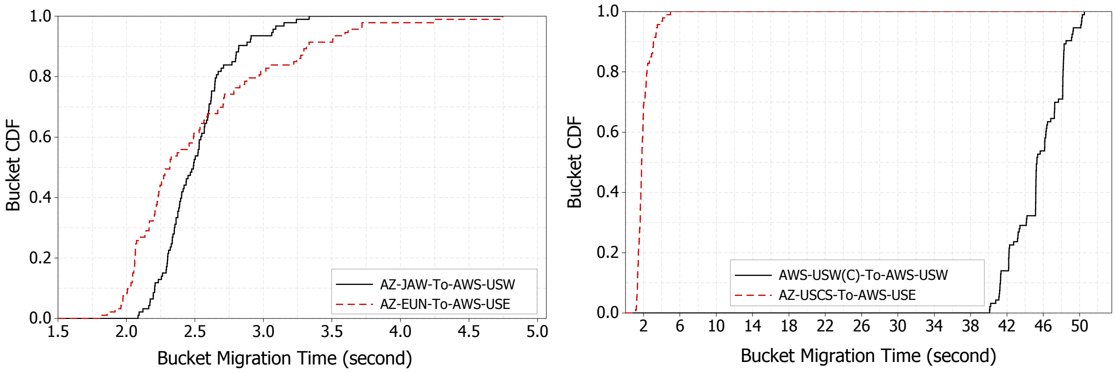


Figure 6.9: Web services components used in the prototype



(a) Bucket migration time across regions

(b) Bucket migration time within a region

Figure 6.10: CDF of data migration time (a) from Azure DC in Japan west to Amazon DC US west and from Azure DC in Europe north to Amazon US east, and (b) Amazon DC in US west (California) to Amazon DC in US west and Azure DC in US center south to Amazon US east.

West (Oregon), and US West (North California) regions. In each Azure cloud site, we create a Container and deploy a DS3_V2 Standard VM instance. In each Amazon cloud site, we create a Bucket and deploy a t2.medium VM instance. All VM instances used in the prototype run Ubuntu 16.04 LTS as operating system.

After the set-up, we run the heuristic algorithm for 100 users (in the Twitter traces) who are assigned to the aforementioned cloud sites. According to the replication and migration policy dictated by the heuristic algorithm, the data are stored in data stores and are integrated in a folder (analogous to bucket in Spanner [52]) for each user. When data migration happens we record the time of data transfer from source cloud site to the destination cloud site.

Fig. 6.10 shows the CDFs of data migration time observed for 100 buckets (each user is associated to a bucket), each of which with the size of about 47.35 MB in average. Fig. 6.10a depicts that data migration can be transmitted in several seconds across regions. About 60% of buckets are transmitted in 2.5 seconds from Azure DC in Japan west (AZ-JAW) to Amazon DC in US west (AWS-USW) as well as from Azure DC in Europe north (AZ-EUN) to Amazon DC in US east (AWS-USE). Also, all buckets are transmitted in 3.5 seconds from Asia region to US region and likewise 4.5 seconds from Europe region to US region. Fig. 6.10b illustrates the data migration time within US region. About 80% of buckets are migrated from Azure DC in US center south (AZ-USCS) to Amazon US east (AWS-USE) below 2 seconds. In contrast, bucket migration time between Amazon DC in US west (North California) (AWS-USW(C)) to another DC in US west (Oregon) (AWS-USW) is between 40-48 seconds for about 80% of buckets. From the results, we conclude that the duration of buckets migration is considerably low. In the case of a large number of buckets, we can transfer data in bulk across DCs with the help of services such as AWS Snowball[9] and Microsoft Migration Accelerator[10].

6.6 Conclusions

In this chapter, we studied the problem of optimizing monetary cost spent on the storage services when data-intensive applications with time-varying workloads are deployed across data stores with several storage classes. We formulated this optimization problem and proposed the optimal algorithm. Since high time complexity is one of the weakness of this optimal algorithm, we proposed a new heuristic solution formulated as a Set Covering problem with three polices. This solution takes advantages of pricing differences across cloud providers as well as the status of objects that changes from hot-spot to cold-spot (and vice versa) during their lifetime. We evaluated the effectiveness of the proposed solution in terms of cost saving via trace-driven simulation using CloudSim simulator and real-world traces from Twitter. We also demonstrated that the duration of objects migration across Geo-distributed data stores is negligible through implemen-

[9] AWS Snowball. https://aws.amazon.com/snowball/.

[10] Microsoft Migration Accelerator. https://azure.microsoft.com/en-au/blog/introducing-microsoft-migration-accelerator/.

tation of a prototype system running over Amazon Web Service (AWS) and Microsoft Azure cloud providers.

Chapter 7

Conclusions and Future Directions

This chapter provides a summary of the research work on monetary cost optimization of data management across cloud-based data stores from the perspective of users/application providers. It also delineates the key findings in this thesis and discusses further research directions in regard to data placement across data stores to optimize monetary cost.

7.1 Summary of Contributions

Cloud-based data stores offer the illusion of infinite storage pool to users. They bring many benefits, including ease-of-use, on-demand resource provisioning, and pay-per-use business model. However, they raise risks to some extent when users solely depend on a single data store.

First, users experience vendor lock-in because Cloud Storage Providers (CSPs) hugely charge the egress bandwidth, and thereby the cost of moving the data out of the cloud is prohibitively expensive. Therefore, users are hostage to CSP and become more vulnerable to price rise, cloud provider bankruptcy, or may lose the opportunity of moving data to a new CSP appearing in the market with a better functionality and price.

Second, unavailability of services is considered as a challenge in the context of the cloud services. Although the well-known CSPs–Amazon, Microsoft Azure, and Google–are very strict with Service Level Agreement (SLA) commitment and even compensate users when SLAs[1] are violated, they experienced the outage of their data centers (DCs) due to environmental catastrophes. This causes, in some cases, users to lose their data[2].

Third, storing data in data stores owned by a single CSP deprives users from benefits

[1] SLA for Azure storage. https://azure.microsoft.com/en-us/support/legal/sla/storage/v1_0/

[2] Data and object are used interchangeably in this chapter.

of diverse geographical locations of cloud providers. These benefits reduce the user-perceived latency by storing data close to users and the avoidance of data transfer bottlenecks by directing requests to replicas of the object hosted in different data stores across the globe. More importantly, adhering to a particular CSP also confines users to exploit pricing differences across CSPs offering various storage classes for different purposes.

With respect to the discussed shortcomings, this thesis investigates a general goal: *how to optimize the monetary cost of data placement across data stores offered at different prices for several storage classes and network resources*? This goal was split into sub optimization problems outlined in Chapter 1. To solve these optimization problems, we presented a set of algorithms to place objects based on their workload across data stores with different storage classes so that the monetary cost of data management is optimized while the required Quality of Service (QoS) is met.

To address the above research problem, we conducted an in-depth survey on data management across data stores in several aspects, mainly monetary cost of data management discussed in Chapter 2. This chapter classified the existing work on optimization monetary cost of storage and network resources and revealed the gap, open challenges, and research problems discussed in this thesis.

Chapter 3 proposed algorithms to replicate data across data stores owned by different CSPs to enhance the data availability defined by the number of nines. In this chapter, an algorithm was proposed to minimize the cost of storing object replicas across data stores under the availability constraint. Moreover, another algorithm was designed to maximize the availability of the stripped object replicas to the extent the user's budget allows. We evaluated both algorithms via extensive simulation experiments in the CloudSim simulator [35] using real prices of storage services. The results demonstrated two replicas for non-stripped objects and three replicas for stripped objects are enough to achieve 7 nines as the expected availability. This value of data availability suffices to have a 24/7 data store accessibility.

Chapter 4 proposed a dual cloud-based storage architecture (i.e., a combination of a temporal data store and a backup data store) to optimize the cost of objects placement. This cost consists of storage cost, Get (read) cost, Put (write) cost, potential migration cost of objects from the temporal data store to the backup data store or vice versa, and delay cost defined as a lost utility. This utility is the multiplication of (i) the time taken to conduct a Get/Put on the data store hosted objects, and (ii) a coefficient of delay importance from the user's perspective to convert delay to a cost monetary. To optimize the summation of these costs, two data placement algorithms were proposed: optimal and near optimal. We evaluated these algorithms via extensive simulation experiments in CloudSim simulator [35] using the real-world traces from Twitter [101]. The results demonstrated that the cost savings obtained from the optimal and near optimal algorithms are respectively 15-70% and 5-60%. These values depend to the deployed data stores, the size of the objects, as well as the workload on the objects.

Chapter 5 investigated how much monetary cost can be saved via data replication and migration across data stores offering several storage classes with different prices. This was motivated by the cost savings obtained from the proposed architecture in previous chapter. Similar to this architecture, we considered the same cost elements (i.e., storage, Get, Put, and potential migration) apart from delay cost. In this chapter, delay was defined as the latency Service Level Objective (SLO) for Get/Put requests. The latency was estimated based on the Round Trip Time (RTT) between the source and destination data stores. Given the cost elements and the requested SLA, we proposed three algorithms to place objects with a limited number of replicas so that the cost saving gained from the pricing differences among CSPs is maximized. The first algorithm is the optimal offline algorithm that yields a high time complexity and requires the exact knowledge of workload on objects. To tackle this obstacle, we proposed two online algorithms that require a limited or no knowledge of the future workload on objects. The extensive experiments using CloudSim simulator and a workload synthesized based on characteristics of Facebook workload demonstrated 10-21% cost savings gained from the optimal offline algorithm. While online algorithms achieved lower cost savings (10-30%) compared to those gained from the optimal offline algorithm.

Chapter 6 proposed a lightweight algorithm to minimize the cost of data management (i.e., storage, Get, Put, migration costs) within the specified access latency. This algorithm was designed based on the approximate algorithm for the Set Covering problem [48]. Unlike the proposed algorithms in Chapter 5, the heuristic solution dynamically determines the number of replicas for each object based on the workload on the object and the size of object. Moreover, the great merit of this solution is scalability, thereby making it appropriate for Online Social Networks hosting a large number of objects.

To evaluate the proposed algorithms, we conducted extensive simulation using real-world traces from Twitter [101] and CloudSim simulator. The results demonstrated the effectiveness of the algorithms in terms of cost saving. To understand the performance of the algorithms, we have implemented a prototype system running on Amazon S3 and Microsoft Azure data stores with the help of their REST APIs.[3][4] This prototype facilitated application providers to Put, Get, Delete, List, and Migrate data across data stores located around the world. Moreover, we measured the duration of data migration within and across regions with the help of this prototype system.

7.2 Future Directions

In spite of contributions of this thesis in regard to cost optimization of data management across CSPs, there are a number of open research challenges that require to be addressed to make further advancement in the area. Some of these challenges have been discussed in Chapter 2; others are identified here.

7.2.1 Trade-off between Availability and Monetary Cost

In Chapter 3, we proposed algorithms that help application providers to find a trade-off between availability and monetary cost spent on storage. It is important to consider the cost of Put, Get, and potential migration costs in the cost model and design algorithms to make more precise trade-off between availability and monetary cost spent on the data storage management across data stores.

As discussed in Chapter 3, we designed algorithms to maximize the expected availability for a "given budget", and to minimize the monetary cost for a "given expected availability". It would be relevant to design algorithms that cover all combinations of constraints (i.e., a given budget or expected availability) and objective functions (i.e., maximizing of the expected availability and minimizing the monetary cost) for objects.

[3] Amazon S3 REST API http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.

[4] Azure storage REST API https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/azure-storage-services-rest-api-reference.

For example, an algorithm can be proposed to minimize the replication cost with a given expected availability for the stripped objects.

7.2.2 The Selection of Home DC

In Chapters 5 and 6, we set the closest DC to the user as his/her *home* DC. This selection allowed the user to Put/Get data in/from data store within a low latency constraint. We made the home DC selection based on the assumption that the user was in a fixed location. However, it would be interesting to investigate the effect of users mobility on the cost optimization, and to determine when it is necessary to change the home DC of the user while he/she moves across the globe. We can consider simple policies to make decision for changing the home DC of the user. For example, if the user issues Get/Put requests from a specific DC in a certain period of time, then this DC can be selected as the home DC.

7.2.3 Cost Optimization of Data Management in Quorum-based Systems

In Chapter 6, we optimized the cost of data management for the system which contains a master replica to guarantee strong consistency for each object. This system provided a simple way of strong consistency while it can create data transfer bottleneck for the data store hosting the master replica. One way to tackle this issue while guaranteeing strong consistency is to use a quorum-based system. Based on the quorum configuration, the system makes a trade-off between the data consistency and the data availability.

To design such system, we first require to discover the location of replicas and then to select a set of replica to be participant in the voting. This set is called voting replicas. Second, we need to select a leader replica among voting replicas. The leader replica is responsible for sending a message to voting replicas and waiting for a quorum to respond. Finally, to capture the user-perceived latency for reading and writing the object, it would be be valuable to consider this latency as a monetary cost and add it to storage, Put, Get, and potential migration costs; that is, using the same cost model studied in Chapter 4.

7.2.4 Cost Optimization across Multiple Storage Classes

As studied in Chapters 4, 5, and 6, we considered two statuses for objects based on their workload: hot- and cold-spot statuses. This can be extended to several status, as defined in Table 2.13. With respect to this extension of objects statuses, it would be relevant to design algorithms in order to dynamically migrate data across storage classes so that the cost of data management (storage, Put, Get, potential migration costs) is optimized. This optimization problem can be mapped to the multi-shop ski-rental problem [6].

7.2.5 Fault Tolerance

The proposed algorithms in the thesis ignored handling of failures when a replica of the object is unavailable. Although Geo-replicated data stores may be available despite a DC failure, the user-perceived latency are probably negatively influenced. Therefore, it would be important to consider the scenario of DC failure in terms of both storage and virtual machine infrastructures in the cost model to support failover of DC(s) [186]. This requires wise techniques to detect soft errors resulted in the failures [185]. However, one simple solution to tolerate the failure of hardware infrastructures is replication of data in the case of storage services as discussed in this thesis and replication of hypervisor executions in the case of virtual machines [183]. This solution affects on monetary cost to provide hardware infrastructures as well as their electricity [184].

7.2.6 Cost Optimization of Using Database Instance Classes

This thesis focused on using object cloud-based storage services. The proposed algorithms can be used in conjunction with the database (DB) instance classes offered by Amazon[5] and Microsoft Azure[6]. These instance classes facilitate application providers to manage relational DBs. The instance classes support several types of instances with different size in two or three payment options. With given DB instance classes, it would be worthwhile to investigate that how to combine these instance classes so that the deployment cost is optimized.

[5] Amazon RDS product. https://aws.amazon.com/rds/details/

[6] SQL database https://azure.microsoft.com/en-us/services/sql-database/

Bibliography

[1] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, no. 2, Feb. 2012.

[2] D. J. Abadi, "Data management in the cloud: Limitations and opportunities," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 3–12, 2009.

[3] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "Racs: a case for cloud storage diversity," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. New York, NY, USA: ACM, 2010, pp. 229–240.

[4] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, no. 1, 1995.

[5] R. W. Ahmad *et al.*, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications*, vol. 52, pp. 11 – 25, 2015.

[6] L. Ai, X. Wu, L. Huang, L. Huang, P. Tang, and J. Li, "The multi-shop ski rental problem," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14. New York, NY, USA: ACM, 2014, pp. 463–475.

[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 19–19.

[8] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," in *Proceedings of third International Conference Networked Systems (NETYS)*, 2015.

[9] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 85–98.

[10] E. Alomari, A. Barnawi, and S. Sakr, "Cdport: A portability framework for nosql datastores," *Arabian Journal for Science and Engineering*, vol. 40, no. 9, pp. 2531–2553, 2015.

[11] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein, "Consistency without borders," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 23:1–23:10.

[12] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier, "Blazes: Coordination analysis for distributed programs," in *Proceedings of the 30th IEEE International Conference on Data Engineering, Chicago, ICDE, IL, USA, March 31 - April 4*, 2014, pp. 52–63.

[13] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a CALM and collected approach," in *Proceedings of the fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CIDR, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 249–260.

[14] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Proceedings of the Sixth international conference on Hot topics in system dependability*, ser. HotDep'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

[15] J. E. Armendáriz-Inigo, A. Mauch-Goya, J. de Mendívil, and F. Muñoz-Escoí, "Sipre: a partial database replication protocol with si replicas," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2008.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’12)*. New York, NY, USA: ACM, 2012, pp. 53–64.

[17] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Queue*, vol. 11, no. 3, pp. 20:20–20:32, Mar. 2013.

[18] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, “Probabilistically bounded staleness for practical partial quorums,” *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 776–787, Apr. 2012.

[19] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15. New York, NY, USA: ACM, 2015, pp. 6:1–6:16.

[20] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, “Towards predictable datacenter networks,” in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM ’11. New York, NY, USA: ACM, 2011, pp. 242–253.

[21] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, “Finding a needle in haystack: Facebook’s photo storage,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 47–60.

[22] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’95. New York, NY, USA: ACM, 1995, pp. 1–10.

[23] D. Bermbach, M. Klems, S. Tai, and M. Menzel, “Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs,” in *Proceedings of IEEE International Conference on Cloud Computing (CLOUD11)*, 2011, pp. 452–459.

[24] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, ser. MW4SOC '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:6.

[25] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," in *Proceedings of the Sixth European Conference on Computer Systems (EuroSys'11).* New York, NY, USA: ACM, 2011, pp. 31–46.

[26] C. E. Bezerra, F. Pedone, and R. V. Renesse, "Scalable state-machine replication," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 331–342.

[27] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache hadoop goes realtime at facebook," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1071–1080.

[28] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 187–198.

[29] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on s3," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 251–264.

[30] J. Broberg, R. Buyya, and Z. Tari, "Metacdn: Harnessing storage clouds for high performance content delivery," *Journal of Network and Computer Applications*, vol. 32, no. 5, pp. 1012 – 1022, 2009, next Generation Content Networks.

[31] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[32] B. Byholm, F. Jokhio, A. Ashraf, S. Lafond, J. Lilius, and I. Porres, “Cost-efficient, utility-based caching of expensive computations in the cloud,” in *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 505–513.

[33] C. Cachin, R. Haas, and M. Vukolic, “Dependable storage in the intercloud,” Research Report RZ, 3783, Tech. Rep., 2010.

[34] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, “Windows azure storage: a highly available cloud storage service with strong consistency,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11. New York, NY, USA: ACM, 2011, pp. 143–157.

[35] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, “Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms,” *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[36] D. G. Campbell, G. Kakivaya, and N. Ellis, “Extreme scale with full sql language support in microsoft sql azure,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’10. New York, NY, USA: ACM, 2010, pp. 1021–1024.

[37] R. Cattell, “Scalable sql and nosql data stores,” *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[38] C.-W. Chang, P. Liu, and J.-J. Wu, “Probability-based cloud storage providers selection algorithms with maximum availability,” *in Proceedings of the 41st International Conference on Parallel Processing*, vol. 0, pp. 199–208, 2012.

[39] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[40] F. Chen, K. Guo, J. Lin, and T. La Porta, “Intra-cloud lightning: Building cdns in the cloud,” in *Proceedings of the IEEE INFOCOM*, March 2012, pp. 433–441.

[41] H. Chen, H. Jin, and S. Wu, “Minimizing inter-server communications by exploiting self-similarity in online social networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1116–1130, April 2016.

[42] H. Chen, Z. Wang, and Y. Ban, “Access-load-aware dynamic data balancing for cloud storage service,” in *Proceedings of the 6th International Conference on Internet and Distributed Computing Systems - Volume 8223*, ser. IDCS 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 307–320.

[43] H. Chen, Y. Hu, P. Lee, and Y. Tang, “Nccloud: A network-coding-based storage system in a cloud-of-clouds,” *IEEE Transactions on Computers*, no. 1, Jan 2014.

[44] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, “Osa: An optical switching architecture for data center networks with unprecedented flexibility,” *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 498–511, Apr. 2014.

[45] R. C. Chiang, H. H. Huang, T. Wood, C. Liu, and O. Spatscheck, “Iorchestra: Supporting high-performance data-intensive applications in the cloud via collaborative virtualization,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’15. New York, NY, USA: ACM, 2015, pp. 45:1–45:12.

[46] H. E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Prez, “Consistency in the cloud: When money does matter!” in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 352–359.

[47] D. Chiu and G. Agrawal, “Evaluating caching and storage options on the amazon web services cloud,” in *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, Oct 2010, pp. 17–24.

[48] V. Chvatal, “A greedy heuristic for the set-covering problem,” *Math. Oper. Res.*, vol. 4, no. 3, pp. 233–235, Aug. 1979.

[49] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer, "Tiered replication: A cost-effective alternative to full cluster geo-replication," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 31–43.

[50] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Proceedings of the 13th USENIX Annual Technical Conference*. USENIX Association, 2013.

[51] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, no. 2, Aug. 2008.

[52] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, no. 3, Aug. 2013.

[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[54] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 29–42.

[55] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu, "Transparent and flexible network management for big data processing in the cloud," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. Berkeley, CA: USENIX, 2013.

[56] S. Das, D. Agrawal, and A. El Abbadi, "Elastras: An elastic transactional data store in the cloud," in *Proceedings of the Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009.

[57] ——, "Elastras: An elastic, scalable, and self-managing transactional database for the cloud," *ACM Trans. Database Syst.*, no. 1, Apr. 2013.

[58] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *Proc. VLDB Endow.*, vol. 4, no. 8, pp. 494–505, May 2011.

[59] DeCandia, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[60] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008, pp. 1–12.

[61] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 1–12.

[62] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, Sept 2010.

[63] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:14.

[64] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[65] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 301–312.

[66] Y. Feng, B. Li, and B. Li, "Postcard: Minimizing costs on inter-datacenter traffic with store-and-forward," in *Proceedings of the 32nd International Conference on Distributed Computing Systems Workshops*, ser. ICDCSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 43–50.

[67] I. Fetai and H. Schuldt, "Cost-based data consistency in a data-as-a-service cloud environment," in *Proceedings of the IEEE Fifth International Conference on Cloud Computing*, ser. CLOUD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 526–533.

[68] M. J. Fischer, X. Su, and Y. Yin, "Assigning tasks for efficiency in hadoop: extended abstract," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 30–39.

[69] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–7.

[70] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, 2009.

[71] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[72] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, no. 1, Mar. 2006.

[73] J. Guo, F. Liu, X. Huang, J. C. S. Lui, M. Hu, Q. Gao, and H. Jin, "On efficient bandwidth allocation for traffic variability in datacenters," in *Proceedings of the IEEE Conference on Computer Communications, INFOCOM*, April 2014, pp. 1572–1580.

[74] M. Hadji, *Scalable and Cost-Efficient Algorithms for Reliable and Distributed Cloud Storage*. Cham: Springer International Publishing, 2016, pp. 15–37.

[75] Z. Hill and M. Humphrey, "Csal: A cloud storage abstraction layer to enable portable cloud applications," in *Proceedings of the 2th IEEE International Conference on Cloud Computing Technology and Scie*, ser. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 504–511.

[76] H. Hu, Y. Wen, T. S. Chua, J. Huang, W. Zhu, and X. Li, "Joint content replication and request routing for social video distribution over cloud cdn: A community clustering method," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. PP, no. 99, pp. 1–1, 2016.

[77] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proceedings of the USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[78] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[79] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Aug. 2013.

[80] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "Eyeq: Practical network performance isolation at the edge," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 297–312.

[81] L. Jiao, J. Lit, W. Du, and X. Fu, "Multi-objective data placement for multi-cloud socially aware services," in *Proceedings of the IEEE Conference on Computer Communications, INFOCOM*, April 2014, pp. 28–36.

[82] L. Jiao, J. Li, T. Xu, W. Du, and X. Fu, "Optimizing cost for online social networks on geo-distributed clouds," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 99–112, Feb. 2016.

[83] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and D. Kopylenko, *Professional Java Development with the Spring Framework*. Birmingham, UK, UK: Wrox Press Ltd., 2005.

[84] F. Jokhio, A. Ashraf, S. Lafond, and J. Lilius, "A computation and storage trade-off strategy for cost-efficient video transcoding in the cloud," in *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications*, Sept 2013, pp. 365–372.

[85] J. Kamal, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable oltp applications," *Future Gener. Comput. Syst.*, vol. 56, no. C, pp. 421–435, Mar. 2016.

[86] A. Kathpal, M. Kulkarni, and A. Bakre, "Analyzing compute vs. storage tradeoff for video-aware storage efficiency," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 13–13.

[87] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads," in *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST, San Jose, CA, USA, February 14-17*, 2012, p. 20.

[88] A. Khanafer, M. Kodialam, and K. Puttaswamy, "The constrained ski-rental problem and its application to online cloud cost optimization," in *Proceedings of the IEEE INFOCOM*, April 2013, pp. 1492–1500.

[89] T. Kobus, M. Kokocinski, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 286–296.

[90] R. Kotla, L. Alvisi, and M. Dahlin, "Safestore: a durable and practical storage system," in *Proceedings of the USENIX Annual Technical Conference (ATC'07)*. Berkeley, CA, USA: USENIX Association, 2007, pp. 10:1–10:14.

[91] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *Proc. VLDB Endow.*, no. 1, Aug. 2009.

[92] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *PracticeProceedings of the IEEE*, vol. 103, no. 1, p. 63, 2015.

[93] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller, "Sword: workload-aware data placement and replica selection for cloud data management systems," *The VLDB Journal*, vol. 23, no. 6, pp. 845–870, 2014.

[94] R. Kuschnig, I. Kofler, and H. Hellwagner, "Improving internet video streaming performance by parallel tcp-based request-response streams," in *Proceedings of the 7th IEEE Consumer Communications and Networking Conference (CCNC'10)*, Jan 2010, pp. 1–5.

[95] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[96] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez, "Inter-datacenter bulk transfers with netstitcher," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 74–85.

[97] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, "Application-driven bandwidth guarantees in datacenters," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 467–478, Aug. 2014.

[98] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10)*. New York, NY, USA: ACM, 2010, pp. 1–14.

[99] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 265–278.

[100] M. Li, C. Qin, and P. P. C. Lee, "Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 111–124.

[101] R. Li, S. Wang, H. Deng, R. Wang, and K. C. Chang, "Towards social user profiling: unified and discriminative influence model for inferring home locations," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16,*, 2012, pp. 1023–1031.

[102] W. Li, Y. Yang, and D. Yuan, "A novel cost-effective dynamic data replication strategy for reliability in cloud data centres," in *Proceedings of the ninth IEEE International Conference on Dependable, Autonomic and Secure Computing*, ser. DASC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 496–502.

[103] G. Liang and U. C. Kozat, "Fast cloud: Pushing the envelope on delay performance of cloud storage with coding," *IEEE/ACM Trans. Netw.*, vol. 22, no. 6, pp. 2012–2025, Dec. 2014.

[104] ——, "On throughput-delay optimal access to storage clouds via load adaptive coding and chunking," *IEEE/ACM Transactions on Networking*, 2015.

[105] M. Lin, Z. Liu, A. Wierman, and L. Andrew, "Online algorithms for geographical load balancing," in *Proceedings of Green Computing Conference (IGCC'12)*, June 2012, pp. 1–10.

[106] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1378–1391, Oct. 2013.

[107] G. Liu, H. Shen, and H. Chandler, "Selective data replication for online social networks with distributed datacenters," in *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP)*, Oct 2013, pp. 1–10.

[108] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proceedings of the 18th ACM International Sym-*

posium on High Performance Distributed Computing, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 101–110.

[109] J. Liu and H. Shen, "A low-cost multi-failure resilient replication scheme for high data availability in cloud storage," in *IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016, pp. 242–251.

[110] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 401–416.

[111] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 313–328.

[112] D. Lomet, "Replicated indexes for distributed data," in *Proceedings of the fourth International Conference on Parallel and Distributed Information Systems*, 1996, pp. 108–119.

[113] T. Lu, M. Chen, and L. Andrew, "Simple and effective dynamic provisioning for power-proportional data centers," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 6, pp. 1161–1171, June 2013.

[114] Y. Ma, T. Nandagopal, K. P. N. Puttaswamy, and S. Banerjee, "An ensemble of replication and erasure codes for cloud file systems," in *Proceedings of the IEEE INFOCOM, Turin, Italy, April 14-19*, 2013, pp. 1276–1284.

[115] Y. Mansouri and R. Buyya, "To move or not to move: Cost optimization in a dual cloud-based storage architecture," *Journal of Network and Computer Applications*, vol. 75, pp. 223 – 235, 2016.

[116] Y. Mansouri, A. N. Toosi, and R. Buyya, "Brokering algorithms for optimizing the availability and cost of cloud storage services," in *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science, (CloudCom'13)*, 2013, pp. 581–589.

[117] B. Mao, S. Wu, and H. Jiang, "Exploiting workload characteristics and service diversity to improve the availability of cloud storage systems," *IEEE Transaction Parallel Distributed Systems*, vol. 27, no. 7, pp. 2010–2021, 2016.

[118] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren, "Stout: An adaptive interface to scalable cloud storage," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 4–4.

[119] V. Medina and J. M. García, "A survey of migration mechanisms of virtual machines," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 30:1–30:33, Jan. 2014.

[120] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," Gaithersburg, MD, United States, Tech. Rep., 2011.

[121] A. Mseddi, M. A. Salahuddin, M. F. Zhani, H. Elbiaze, and R. H. Glitho, "On optimizing replica migration in distributed cloud storage systems," in *Proceedings of the 4th IEEE International Conference on Cloud Networking, CloudNet, Niagara Falls, ON, Canada, October 5-7*, 2015, pp. 191–197.

[122] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm blob storage system," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 383–398.

[123] M. Naldi and L. Mastroeni, "Cloud storage pricing: A comparison of current practices," in *Proceedings of International Workshop on Hot Topics in Cloud Services*, ser. HotTopiCS '13. New York, NY, USA: ACM, 2013, pp. 27–34.

[124] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398.

[125] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," Tech. Rep., 2013.

[126] L. Pacheco, D. Sciascia, and F. Pedone, "Parallel deferred update replication," in *Proceedings of the IEEE Symposium on Network Computing and Applications (NCA)*, 2014.

[127] C. Papagianni, A. Leivadeas, and S. Papavassiliou, "A cloud-oriented content delivery network paradigm: Modeling and assessment," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 5, pp. 287–300, Sept 2013.

[128] F. Pedone, M. Wiesmann, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Proceedings of the 20th International Conference on Distributed Computing Systems, Taipei, Taiwan, April 10-13, 2000*, 2000, pp. 464–474.

[129] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing," in *Proceedings of the ACM SIGCOMM Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 351–362.

[130] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: Scaling online social networks," in *Proceedings of the ACM SIGCOMM Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 375–386.

[131] K. P. Puttaswamy, T. Nandagopal, and M. Kodialam, "Frugal storage for cloud file systems," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. New York, NY, USA: ACM, 2012, pp. 71–84.

[132] X. Qiu, H. Li, C. Wu, Z. Li, and F. C. M. Lau, "Cost-minimizing dynamic migration of content distribution services into hybrid clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3330–3345, Dec 2015.

[133] A. Qureshi, "Power-demand routing in massive geo-distributed systems," in *PhD Thesis submitted to MIT*, 2012.

[134] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, "Toward a principled framework for benchmarking consistency," in *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, ser. HotDep'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 8–8.

[135] K. Rashmi, N. Shah, and P. Kumar, "Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction," *IEEE Transactions on Information Theory*, no. 8, Aug 2011.

[136] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers," *SIGCOMM Comput. Commun. Rev.*, no. 4, Aug. 2014.

[137] R. Roth, *Introduction to Coding Theory*. New York, NY, USA: Cambridge University Press, 2006.

[138] A. Ruiz-Alvarez and M. Humphrey, "A model and decision procedure for data storage in cloud computing," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, 2012, pp. 572–579.

[139] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14, 2014.

[140] V. R. Ryan Ko, Stephen S G Lee, "Cloud computing vulnerability incidents: A statistical overview," in *Cyber Security Lab, Department of Computer Science, University of Waikato, New Zealand; Cloud Security Alliance (Asia Pacific)*, 2013.

[141] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, no. 1, Mar. 2005.

[142] S. Sakr, A. Liu, D. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Communications Surveys Tutorials*, vol. 13, no. 3, pp. 311–336, 2011.

[143] M. A. Salahuddin, H. Elbiaze, W. Ajib, and R. H. Glitho, "Social network analysis inspired content placement with qos in cloud based content delivery networks," in

Proceedings of the IEEE Global Communications Conference, GLOBECOM, San Diego, CA, USA, December 6-10, 2015, 2015, pp. 1–6.

[144] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “Xoring elephants: Novel erasure codes for big data,” *Proc. VLDB Endow.*, no. 5, Mar. 2013.

[145] D. Sciascia, F. Pedone, and F. Junqueira, “Scalable deferred update replication,” in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12.

[146] S. S. Seiden, “A guessing game and randomized online algorithms,” in *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, ser. STOC ’00. New York, NY, USA: ACM, 2000, pp. 592–601.

[147] N. B. Shah, K. Lee, and K. Ramchandran, “The MDS queue: Analysing the latency performance of erasure codes,” in *Proceedings of the IEEE International Symposium on Information Theory, Honolulu, HI, USA, June 29 - July 4*, 2014, pp. 861–865.

[148] P. N. Shankaranarayanan, A. Sivakumar, S. Rao, and M. Tawarmalani, “Performance sensitive replication in geo-distributed cloud datastores,” in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 240–251.

[149] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400.

[150] A. Sharov, A. Shraer, A. Merchant, and M. Stokely, “Take me to your leader!: Online optimization of distributed storage configurations,” *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1490–1501, Aug. 2015.

[151] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu, “Causal consistency for geo-replicated cloud storage under partial replication,” in *Proceedings of the IEEE International Par-*

allel and Distributed Processing Symposium Workshop (IPDPSW), May 2015, pp. 509–518.

[152] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 309–322.

[153] Y. Shin, D. Koo, and J. Hur, "A survey of secure data deduplication schemes for cloud storage systems," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 74:1–74:38, Jan. 2017.

[154] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 349–362.

[155] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed sql database that scales," *Proc. VLDB Endow.*, no. 11, Aug. 2013.

[156] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan, "Declarative programming over eventually consistent data stores," *SIGPLAN Not.*, vol. 50, no. 6, pp. 413–424, Jun. 2015.

[157] J. Spillner, G. Bombach, S. Matthischke, J. Muller, R. Tzschichholz, and A. Schill, "Information dispersion over redundant arrays of optimal cloud storage for desktop users," in *Proceedings of the fourth IEEE International Conference on Utility and Cloud Computing (UCC)*, 2011, pp. 1–8.

[158] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency slos," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 265–277.

[159] M. Su, L. Zhang, Y. Wu, K. Chen, and K. Li, "Systematic data placement optimization in multi-cloud storage for complex requirements," *IEEE Trans. Computers*, vol. 65, no. 6, pp. 1964–1977, 2016.

[160] C. Suh and K. Ramchandran, "Exact-repair mds code construction using interference alignment," *IEEE Transactions on Information Theory*, vol. 57, no. 3, pp. 1425–1442, March 2011.

[161] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 513–527.

[162] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., 2006.

[163] J. Tang, X. Tang, and J. Yuan, "Optimizing inter-server communication for online social networks," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2015, pp. 215–224.

[164] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 309–324.

[165] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, no. 2, Jun. 1979.

[166] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE 2010)*, March 2010, pp. 996–1005.

[167] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 7:1–7:47, May 2014.

[168] D. A. Tran, K. Nguyen, and C. Pham, "S-clone: Socially-aware data replication for social networks," *Computer Networks*, vol. 56, no. 7, pp. 2001 – 2013, 2012.

[169] N. Tran, M. K. Aguilera, and M. Balakrishnan, "Online migration for geo-distributed storage systems," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 15–15.

[170] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 115–126.

[171] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 7–7.

[172] S. Venugopal, R. Buyya, and K. Ramamohanarao, "A taxonomy of data grids for distributed data sharing, management, and processing," *ACM Comput. Surv.*, vol. 38, no. 1, Jun. 2006.

[173] T. Wang, Z. Su, Y. Xia, and M. Hamdi, "Rethinking the data center networking: Architecture, network protocols, and resource sharing," *Access, IEEE*, 2014.

[174] W. Wang, B. Li, and B. Liang, "To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds," in *Proceedings of the USENIX International Conference on Autonomic Computing (ICAC)*, 2013.

[175] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. C. M. Lau, "Scaling social media applications into geo-distributed clouds," *IEEE/ACM Trans. Netw.*, no. 3, Jun. 2015.

[176] Y. Wu, A. G. Dimakis, and K. Ramchandran, "Deterministic regenerating codes for distributed storage," *Allerton Conference on Control, Computing, and Communication*, 2007.

[177] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13).* New York, NY, USA: ACM, 2013, pp. 292–308.

[178] Z. Wu and H. V. Madhyastha, "Understanding the latency benefits of multi-cloud webservice deployments," *SIGCOMM Comput. Commun. Rev.*, no. 2, Apr. 2013.

[179] Z. Wu, C. Yu, and H. V. Madhyastha, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 543–557.

[180] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 213–226.

[181] Y. Xiang, T. Lan, V. Aggarwal, and Y. F. R. Chen, "Joint latency and cost optimization for erasurecoded data center storage," *SIGMETRICS Perform. Eval. Rev.*, no. 2, Sep. 2014.

[182] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 199–210.

[183] X. Xu and H. H. Huang, "Dualvisor: Redundant hypervisor execution for achieving hardware error resilience in datacenters," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 485–494.

[184] X. Xu, K. Teramoto, A. Morales, and H. H. Huang, "Dual: Reliability-aware power management in data centers," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 530–537.

[185] X. Xu, R. C. Chiang, and H. H. Huang, "Xentry: Hypervisor-level soft error detection," in *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, ser. BRACIS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 341–350.

[186] X. Xu and H. H. Huang, "On soft error reliability of virtualization infrastructure," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3727–3739, Dec. 2016.

[187] J. Yao, H. Zhou, J. Luo, X. Liu, and H. Guan, "Comic: Cost optimization for internet content multihoming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1851–1860, July 2015.

[188] B. Yu and J. Pan, "Location-aware associated data placement for geo-distributed data-intensive applications," in *Proceedings of the IEEE Conference on Computer Communications, INFOCOM, Kowloon, Hong Kong, April 26 - May 1*, 2015, pp. 603–611.

[189] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 239–282, Aug. 2002.

[190] D. Yuan, Y. Yang, X. Liu, and J. Chen, "On-demand minimum cost benchmarking for intermediate dataset storage in scientific cloud workflow systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 316 – 332, 2011, data Intensive Computing.

[191] D. Yuan, Y. Yang, X. Liu, W. Li, L. Cui, M. Xu, and J. Chen, "A highly practical approach toward achieving minimum data sets storage cost in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1234–1244, 2013.

[192] W. Zeng, Y. Zhao, K. Ou, and W. Song, "Research on cloud storage architecture and key technologies," in *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ser. ICIS '09. New York, NY, USA: ACM, 2009, pp. 1044–1048.

[193] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang, "Guaranteeing deadlines for inter-datacenter transfers," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 20:1–20:14.

[194] L. Zhang, C. Wu, Z. Li, C. Guo, M. Chen, and F. Lau, "Moving big data to the cloud: An online cost-minimizing approach," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 12, pp. 2710–2721, December 2013.

[195] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai, "Charm: A cost-efficient multi-cloud data hosting scheme with high availability," *IEEE Transactions on Cloud Computing*, vol. 3, no. 3, pp. 372–386, July 2015.

[196] X. Zhang, C. Liu, S. Nepal, S. Pandey, and J. Chen, "A privacy leakage upper bound constraint-based approach for cost-effective privacy preserving of intermediate data sets in cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1192–1202, June 2013.

[197] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan, "Does erasure coding have a role to play in my data center," *Microsoft research MSR-TR-2010*, vol. 52, 2010.

[198] J. Zhou, J. Fan, J. Wang, B. Cheng, and J. Jia, "Towards traffic minimization for data placement in online social networks," *Concurrency and Computation: Practice and Experience*, 2016.