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:


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 ........................................... 16
      2.2.1 Terms and Definitions ......................... 16
      2.2.2 Characteristics of data-intensive applications 17
      2.2.3 Architecture, Goals, and Challenges of Intra-cloud Storage . 19
      2.2.4 Architecture, Motivations, and Challenges of Inter-Cloud Storage . 24
   2.3 Data model ........................................... 30
      2.3.1 Data structure ................................ 30
      2.3.2 Data abstraction ............................... 31
      2.3.3 Data access model ............................. 32
   2.4 Data Dispersion ...................................... 33
      2.4.1 Data replication ................................ 33
      2.4.2 Erasure coding ................................ 40
      2.4.3 Hybrid scheme ................................ 45
   2.5 Data Consistency ..................................... 45
      2.5.1 Aspects of data consistency in distributed data stores ....... 45
      2.5.2 Eventual consistency ............................ 50
      2.5.3 Causal and Causal+ consistency ................ 54
      2.5.4 Ordering, Strong, and Adaptive-level consistency .......... 55
   2.6 Data Management Cost ............................... 56
      2.6.1 Pricing plans .................................... 56
      2.6.2 Overview of storage classes ..................... 57
      2.6.3 Cost optimization based on a single QoS metric .......... 62
      2.6.4 Cost optimization based on multi-QoS metric .......... 63
      2.6.5 Cost trade-offs ................................ 64
   2.7 Thesis Scope and Positioning ........................ 66
      2.7.1 Thesis Scope .................................... 66
2.7.2 Thesis Positioning ........................................... 67
2.8 Conclusions .................................................... 72

3 QoS-aware Brokering Algorithms for Data Replication across Data Stores .................................................. 77
  3.1 Introduction ..................................................... 77
  3.2 Minimizing cost with given expected availability ................................. 80
  3.3 Maximum Expected Availability with Given Budget ......................... 84
    3.3.1 Optimal Chunks Placement (OCP) Algorithm .................. 87
  3.4 Performance Evaluation ........................................ 89
    3.4.1 Simulation Setting ........................................ 89
    3.4.2 Algorithm 3.1: Minimum Cost Fixed Expected Availability 91
    3.4.3 Algorithm 3.2: Maximum Expected Availability with a Given Budget 93
  3.5 Summary ..................................................... 95

4 Cost Optimization in a Dual Cloud-based Storage Architecture ................................................................. 97
  4.1 Introduction ..................................................... 97
  4.2 System and Cost Model ......................................... 100
    4.2.1 System Model .............................................. 100
    4.2.2 Cost Model ................................................ 101
  4.3 Data Management Cost Optimization ................................ 103
    4.3.1 Optimal Object Placement (OOP) Algorithm ................ 103
    4.3.2 Near-Optimal Object Placement (NOOP) Algorithm ........ 105
  4.4 Performance Evaluation ........................................ 106
    4.4.1 Experimental settings ...................................... 107
    4.4.2 Results .................................................. 109
  4.5 Summary ..................................................... 122

5 Cost Optimization across Cloud Storage Providers: Offline and Online Algorithms ................................................. 123
  5.1 Introduction ..................................................... 123
  5.2 System Model and Problem Definition ................................ 126
    5.2.1 Challenges and Objectives ................................ 126
    5.2.2 Preliminaries ............................................... 128
    5.2.3 Optimization Problem .................................... 133
  5.3 Optimal Offline Algorithm ...................................... 134
  5.4 Online Algorithms .............................................. 136
    5.4.1 The Deterministic Online Algorithm ....................... 137
    5.4.2 The Randomized Online Algorithm ....................... 140
  5.5 Performance Evaluation ........................................ 145
    5.5.1 Settings .................................................. 145
    5.5.2 Benchmark Algorithms ..................................... 146
    5.5.3 Results .................................................. 146
  5.6 Summary ..................................................... 155
### 6 Cost Optimization across Cloud Storage Providers: A Lightweight Algorithm

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>157</td>
</tr>
<tr>
<td>6.2</td>
<td>System Model, Cost Model, and Cost Optimization Problem</td>
<td>160</td>
</tr>
<tr>
<td>6.2.1</td>
<td>System model</td>
<td>160</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Cost model</td>
<td>161</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Cost Optimization Problem</td>
<td>169</td>
</tr>
<tr>
<td>6.3</td>
<td>Cost Optimization Solution</td>
<td>170</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Optimal Solution</td>
<td>171</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Heuristic Solution</td>
<td>171</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance Evaluation</td>
<td>174</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Experimental setting</td>
<td>175</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Results</td>
<td>176</td>
</tr>
<tr>
<td>6.5</td>
<td>Empirical Studies in Latency Evaluation</td>
<td>180</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Data Access Management Modules</td>
<td>180</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Measurement of Data Migration Time</td>
<td>184</td>
</tr>
<tr>
<td>6.6</td>
<td>Conclusions</td>
<td>186</td>
</tr>
</tbody>
</table>

### 7 Conclusions and Future Directions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Summary of Contributions</td>
<td>189</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Directions</td>
<td>192</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Trade-off between Availability and Monetary Cost</td>
<td>192</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The Selection of Home DC</td>
<td>192</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Cost Optimization of Data Management in Quorum-based Systems</td>
<td>193</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Cost Optimization across Multiple Storage Classes</td>
<td>194</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Fault Tolerance</td>
<td>194</td>
</tr>
<tr>
<td>7.2.6</td>
<td>Cost Optimization of Using Database Instance Classes</td>
<td>194</td>
</tr>
</tbody>
</table>
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 ................. 16
2.3 NewSQL Schema ................................................................. 16
2.4 Goals of Intra-cloud storage ..................................................... 20
2.5 Challenges of Intra-cloud storage .............................................. 20
2.6 Inter-cloud storage architecture ................................................. 26
2.7 Challenges of Inter-cloud storage .............................................. 26
2.8 Motivations of Inter-cloud storage ........................................... 26
2.9 Data model taxonomy ............................................................. 30
2.10 Data replication taxonomy ....................................................... 34
2.11 Erasure coding taxonomy ....................................................... 40
2.12 Data consistency taxonomy ..................................................... 46
2.13 Mapping data stores to each pair of properties in CAP .................. 49
2.14 PACELC classification and mapping several data stores to the classification 49
2.15 Weak consistency taxonomy .................................................... 50

3.1 Cloud Storage Broker ............................................................ 79
3.2 Minimum cost of replication versus expected availability of objects ... 91
3.3 Minimum cost of replication versus expected availability for three types of QoS .................................................. 92
3.4 Expected availability versus TN for three types of QoS .................... 92
3.5 Expected failure (EFQ) versus Expected Availability for three types of QoS .................................................. 93
3.6 Expected availability versus Budget ............................................ 94
3.7 Expected failure versus Budget ................................................ 94
3.8 Expected availability for three types of QoS Versus Budget .............. 95

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. ................................ 100
4.2 Allocated users to DCs (%) ....................................................... 108
4.3 Total data size in data centers .................................................. 108
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.

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.

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.

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.

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.

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.

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

5.2 Overview of systems’s inputs and output.

5.3 The description of $P(\bar{r}(t))$ calculation in Equ. (5.7)

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 $v = t - 1$ 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 $t_m$.

5.5 Illustration of Fixed Reduced Horizontal Control.

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.

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.

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.

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.

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.
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. .......................... 155

6.1 Replica creation via (a) home DC and (b) potential DCs D1, D2, and D3. 162

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. .......................................................... 164

6.3 An example of illustrating the replica migration between two consecutive time slots. .......................................................... 167

6.4 Cost saving of closest-, network-, and storage-based policies under tight (100 ms) and loose latency (250 ms). .......................... 177

6.5 Cost saving of closest-based policy vs. quantile volume .......................... 179

6.6 Cost saving of closest-based policy vs. latency .................................. 179

6.7 Cost saving of closest-based policy vs. read to write ratio .................. 180

6.8 An overview of prototype ......................................................... 184

6.9 Web services components used in the prototype .............................. 185

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. ......................... 185

xvii
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Consistency-latency tradeoff of different replication techniques.</td>
<td>39</td>
</tr>
<tr>
<td>2.2</td>
<td>The characteristics of different storage classes for the well-known Cloud providers: Amazon Web Service (AWS), Azure(AZ), and Google(GO)</td>
<td>61</td>
</tr>
<tr>
<td>2.3</td>
<td>The Scope of Thesis.</td>
<td>66</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary of Projects with Monetary Cost Optimization</td>
<td>73</td>
</tr>
<tr>
<td>3.1</td>
<td>objective and constraint of the proposed algorithms</td>
<td>90</td>
</tr>
<tr>
<td>3.2</td>
<td>Data centers parameters</td>
<td>90</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of Simulation Parameters</td>
<td>110</td>
</tr>
<tr>
<td>4.2</td>
<td>Evaluation Settings for Figures and Tables.</td>
<td>111</td>
</tr>
<tr>
<td>4.3</td>
<td>Cost saving of OOP and NOOP (shown in bracket), and the potential DCs pairing with four home DCs when the access patterns on the objects are Normal and Random.</td>
<td>120</td>
</tr>
<tr>
<td>5.1</td>
<td>Cloud storage pricing as of June 2015 in different DCs.</td>
<td>125</td>
</tr>
<tr>
<td>5.2</td>
<td>Symbols definition</td>
<td>128</td>
</tr>
<tr>
<td>5.3</td>
<td>Average cost performance (Normalized to the local residential algorithm)</td>
<td>149</td>
</tr>
<tr>
<td>5.4</td>
<td>Running time of algorithms on 23 DCs (in Second)</td>
<td>155</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of key notations</td>
<td>161</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary of Simulation Parameters</td>
<td>176</td>
</tr>
<tr>
<td>6.4</td>
<td>Average cost performance normalized to the benchmark algorithm cost</td>
<td>178</td>
</tr>
<tr>
<td>6.5</td>
<td>The modules used for data access management in AWS.</td>
<td>182</td>
</tr>
<tr>
<td>6.6</td>
<td>The input parameters used in Modules of AWS.</td>
<td>182</td>
</tr>
<tr>
<td>6.7</td>
<td>The modules used for data access management in Microsoft Azure.</td>
<td>183</td>
</tr>
<tr>
<td>6.8</td>
<td>The input parameters used in Modules of Microsoft Azure.</td>
<td>184</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

Cloud computing has gained significant attention from 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 [37].

Cloud computing offers three types of resources delivery models to users [127]: (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
2 Introduction

2013 and 2020—from 4.4 ZB to 44 ZB.\textsuperscript{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.\textsuperscript{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,\textsuperscript{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-

\textsuperscript{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 [77]. 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) 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 [186]. (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

---

4Read and Write are respectively interchangeable with Get and Put in this 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 [129]. 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 [185] and Cosplay [91]. 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
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 design 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.
1.3 Methodology

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 [41]. We generate a synthetic workload based on the characteristics of the Facebook workload [18, 24] and use it in an optimization problem in Chapter 5. We also perform the evaluation of algorithms using a real workload from Twitter [110]. With the help of the Google Maps Geocoding APIs, 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.

---

6The 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 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 [110].
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 [18][24].

5. Cost optimization across cloud storage providers: a lightweight algorithm:
   - A lightweight algorithm based on the set cover problem [53]. 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 [110].
   - 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:
Figure 1.3: The thesis organization

- Chapter 3 proposes QoS-aware brokering algorithms for data replication across data stores. This chapter is derived from:


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


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

- 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:
  

- 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) forms a critical 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. Despite many open challenges within a cloud-based data store, application providers deploy Geo-replicated data stores in order to obtain higher availability, lower response time, and more cost efficiency. The deployment of Geo-replicated data stores is in its infancy and poses vital challenges for researchers. 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 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. The challenges posed by this growth of data...
can be overcome with aid of using cloud computing services. Cloud computing 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, data transaction support, 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 with the rejection of ACID properties as a typical feature of relational database (RDB) system. This class of data stores – such as PNUTS [56] and Dynamo [65] – 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 captures the advantages of both NoSQL and relational data stores and initially was exploited in Spanner [57].

To take the benefits of these classes of data stores, 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 solely on a single data store has risks like vendor lock-in, economic failure (e.g., a surge in price services), unavailability as outages occur, and probably data loss as an environmental catastrophe happens [31]. 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 make load balance. Third, data recovery can be possible when natural disaster and human-induced activities happen. However, the deployment of either 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-oriented data model, while OLAP applications require rich query capabilities and compliance with column-oriented data model. These requirements are faced with several challenges (Section 2.2), and they mandate that we investigate the key elements of data management in data stores, as shown in Fig. 2.1. Apart from data management cost, other key elements are mature topics in the context of distributed database systems and recently have attracted considerable attention in data stores. Thus, we focus on how these elements are applied to data stores with possible modifications and adoptions if needed.

The first element is data model that reflects how data is stored in and retrieved from data stores (Section 2.3). Data dispersion is the second element and is classified into data replication, erasure coding, and hybrid (i.e., a mix of data replication and erasure coding) schemes. Data replication improves availability and locality by moving data close to the user, but it is costly due to triplicate style (i.e., three replicas for each object) in data stores. Erasure coding 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, hybrid scheme is exploited. We clarify data dispersion schemes and identify how they influence availability, durability, and user-perceived latency (Section 2.4).

Other element of data management are data consistency that refers to coordination level between replicas within and across data stores. Based on CAP theorem [78], it is impossible to jointly attain Consistency, Availability, and Partition 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 endeavors bring transactional isolation levels in data stores.
2.2 Overview

This section discusses the terms used throughout this chapter (Section 2.2.1), the characteristics of two types of data-intensive applications deployed in data stores (Section 2.2.2), and the main goals and challenges of a single and multiple data stores leveraged to manage these applications (Section 2.2.3).

2.2.1 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 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 by users/application providers who are in the cost of extra resources and higher response time. We elaborate on the spectrum of consistency semantics in NoSQL/NewSQL data stores (Section 2.5).

We finally focus on data management cost (i.e., monetary cost) as a new concept in the context of cloud in comparison to traditional distributed systems. Cost is 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 (Section 2.6).
2.2 Overview

hosted across data centers (DCs). Metadata describe the dataset in respect to several attributes such as name, creation time, owner, replicas location, and etc.

A dataset consists of a set of objects or records which are modeled in relational databases (RDBs) or NoSQL databases. The data store that manages RDBs and NoSQL databases also are respectively called 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 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 borrows its parent’s primary key as a prefix of its primary key. As depicted in Fig. 2.3, Campaign table is a child of Customer table and inherits the primary key of Customer. This data model has a directory table in the top of hierarchical structure and each row of directory table together with all rows in the descendant table constructs a directory. In Fig. 2.3, Customer is a directory table, and each its row with rows in Campaign and AdGroup tables build a directory.

A complete or partial replica of the dataset is stored in a single data store or different data stores across the world 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.

2.2.2 Characteristics of data-intensive applications

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

Transactional applications (ref. as OLTP), by definition, must guarantee ACID properties and provide an “all-or-nothing” proposition that implies each set of operations called transactions must either complete in its entirety or have no effect whatsoever. Deploying OLTP (online transaction processing) across data stores is not straightforward for the fol-
lowing reasons [2]. (i) It is difficult to apply complex and distributed locks and commit protocols and transfer data over network bearing high latency and network congestion. This is because data is partitioned and replicated across Geo-distributed servers. (ii) It is challenging to achieve ACID properties which require a trade-off between availability and consistency. (iii) It is probably risky to transfer transactional data which consists of very secure information. Thus, OLTP is not well suited for deploying across data stores and it should be adapted to ACID properties at the expense of high latency to serve reads and writes.

Analytical applications (ref. as OLAP) usually use read-only databases and often need to handle complex queries to retrieve the desired data for business intelligence, data warehouse, and data mart. So, there is a need to create a layer above the cloud to handle such queries. The updates in OLAP are conducted on regular basis (e.g., per day or per week), and their rate is lower than that of OLTP. Thus, OLAP excludes distributed locks and commit protocols, and eventual consistency suffices.

Data management systems like BigTable [45], PNUTS [56], Dynamo [65], and Cassandra [104] are suitable for OLTP. Because they support high availability and low latency in the expense of weak consistency in which sometimes users observe stale data. Therefore, the challenge within and across data stores is how to provide stronger consistency guarantee without compromising latency, availability, and scalability. By contrary, the imperative criteria for OLAP are rich queries support and data sharing applicability, which are provided by systems like Hadoop\(^2\), Hive [173], and Pig\(^3\). But they lack rich query processing on the cloud. Thus, a feasible solution is changing data model so that the need for distributed and complex queries within and across DCs is reduced. Since OLTP applications typically access only a small portion of data (i.e., record-level accessibility), they require a data model such as the one used in Spanner and F1 [162], where the horizontal partitioning of tabular data is carefully conducted. In contrast, OLAP applications demand vertical partitioning of tabular data because they often need to manipulate large portion of data involving restricted columns of tabular data (i.e., attribute-level accessibility). Thus, OLTP and OLAP applications respectively tend to adapt row- and column-oriented layout. However, this chapter focuses on the key requirements of OLTP


\(^3\)Apache Pig. [http://hadoop.apache.org/pig/](http://hadoop.apache.org/pig/)
2.2 Overview

applications like low response time and high levels of consistency.

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

This section first describes a layered architecture of data store and then 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 [197]. 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 and manipulating 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 [40] has a primary key containing three properties: **AccountName**, **PartitionName**, and **ObjectName** that determine the owner, location, and name of the table 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 one 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

---

\(^4\)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.
Goals of Intra-Cloud Storage

Fig. 2.4 shows the main five goals of data-intensive applications deployment in a single data store. These goals are as follows:

- **Availability.** The high probability of resource failure in large-scale distributed systems challenges the cloud availability, but can be boosted from three aspects: (i) *computing availability* is feasible by using cloud federation [174], (ii) *communication availability* is achievable via redundant physical links or fault tolerant routing algorithms [179], and (iii) *data availability* is attainable through data replication and erasure coding within and across data stores [55].

- **Performance and cost saving.** Data stores offer the pay-as-you-go model in which the paid cost is proportional to the resource consumption. Thus, efficient utilization of resources has direct influence on cost saving. To achieve both 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 status of data and the required QoS. Note that hot-spot data/node receives many read and write requests, while cold-spot data/node receives a few.

- **Elasticity and load balancing.** Elasticity refers to the ability of data cloud to deal with expanding/consolidating more/fewer servers during high/low load in a live system, which directly influences the monetary cost. Elasticity is orthogonal with load balancing for which the data store should be able to dynamically move the workload from one server to another under skewed query distribution so that all
servers almost equally handle workload. A prevalent method for balancing load is to join (resp. remove) a node to (resp. from) keys group (e.g., a table in a key-value application) that is heavily (resp. lightly) loaded. This method is effective when load balancing is locally monitored, not globally [63]. Another effective method is BATON [86] in which range query processing, reparation, and redistribution of data are effectively performed. In spite of these methods, it is preferred that elasticity changes of data store are correlated with the load and requirements on-the-fly. Live data migration protocol [48] is an effective way for this purpose, but it may degrade service availability and may increase requests response time (or even cause failed requests) and network congestion across servers.

- **Fault tolerance.** This is accounted as a main goal of data store in the face of failure. Failure can be caused by two main reasons: faulty hardware (e.g., server and network) and software. The failure models are byzantine and crash-stop. In the former, the failed node presents arbitrary behavior, while in the latter a node silently stops its execution. To survive byzantine failures, $2f + 1$ replicas ($f$ is number of failures) along with non-cryptographic hashes or cryptographic primitives suffice. To tolerate crash-stop failures, the well-established approaches are data replication and erasure coding in several servers located in different racks, which in turn, are in different domains. Different types of applications demand a different degree of fault tolerance, as for the OLTP application is more important than for the OLAP one [2]. This is because that for the OLTP application, the data store should be able to recover from a failure without data loss or transaction commit. While for the read-only OLAP application, the cloud should not restart query if one of the nodes involving in the activity fails since the activity includes neither write transactions to be committed nor updates to be lost.

- **Multi-tenancy.** This allows users to run applications on shared hardware and software infrastructure so that their isolated performance is guaranteed. Multi-tenancy causes two key challenges [161]: (i) variable and unpredictable performance, and (ii) multi-tenant interference and unfairness. These issues confine migration of applications to multi-tenant cloud and happen to different multi-tenancy models. These models, from the weakest to strongest isolation, are listed as below.
Shared table [21]: the different tenants’ table with the same scheme is co-located in a shared table to provide an efficient resource sharing (e.g., salesforce.com [120]). This model needs a custom query and transaction processing framework. (ii) Shared process [60]: the process database node is shared among tenants for whom the independent table is allocated. It offers more flexibility in terms of standard query, transaction processing, and tables scheme. Shared process thus allows more tenants to be consolidated in the same process database node [60]. Pisces [161] and ElasTras [63], as examples of key value data stores, exploited this model to achieve load balancing and fairness in resource allocation. (iii) Shared machine [34]: the tenants share the physical computing server while their database processes and VM are independent. To attain elastic scaling, this model can leverage live VM migration techniques [117] though it introduces overloading leading to ineffective resource sharing.

Challenges of Intra-Cloud Storage

Fig. 2.5 shows what challenges application providers confront with the deployment of their applications in a data store. These challenges are as below:

- **Availability of services.** Although the well known data stores abide by the rigid availability, they are still at risk of inevitable and unpredictable malicious errors and natural events. Two common ways to achieve high availability are (i) data replication across storage nodes in an efficient way in data store (e.g., Copysets [55]) instead of random replica placement widely used by current data stores, and (ii) erasure coding with considering other performance metrics (Section 2.4.2).

- **Data lock-in.** Relaying on a single data store puts users in risks and prohibits them to migrate to another data store in the face of price increment by vendor, the appearance of new data store with lower price, the mobility of users, and changes in workload that demands data migration. Clearly, the same discussed strategy for the above challenge can be selected here.

- **Data transfer bottleneck.** With the deployment of data-intensive applications in the cloud, application providers are facing the problem of how to mange intra-DC traffic bottleneck. This bottleneck reflects the optimality of data placement, where data
items are accessed by distributed transactions or simple queries. Instead of using common data placement approaches, application providers deploy different techniques based on the characteristics of the application. For OLTP applications, these techniques attempt to reduce the number of distributed transactions with help of workload-aware data partitioning, and they deal with either static workloads [102] or dynamic workloads [94]. The same techniques have been also studied to reduce network traffic in respect to social networks. They jointly rely on replicating data and partitioning of social graph that represents the interaction between users. In this way, they maximize “social locality”—co-locating the data of a user and his/her friends—to reduce intra-DC traffic [47, 170].

Nevertheless, the application providers still confront network congestion. To drop such congestion, data flows are monitored in the switches and are scheduled only as data congestion occurs [9]. To avoid this coincidence, data flows can be effectively scheduled with help of data flow prediction performed by monitoring applications on servers (e.g., FlowComb [61]). Also, the optimization of data transfer of integrated flows (rather than individual data flows) is a potential way to reduce network congestion (e.g., Seawall [160]). Besides these approaches, data aggregation [59], novel network typologies [179], and optical circuit switching deployment [49] are other approaches to drop network congestion.

• **Performance unpredictability.** Shared storage services face the challenges of multi-tenant such as unpredictable performance and multi-tenant unfairness which result in degrading the requests response time. Zoolander [165], as a key-value data store, uses replication to achieve predictable response time and redundant requests for data to mask outlier response time. Pisces [161] takes the advantages of *max-min fairness* to handle a differentiated level of performance.

More importantly, the lack of bandwidth guarantee impedes predictable response time demanded by cloud applications. To achieve predictable response time under network sharing, *fairness* should be considered in approaches of allocated bandwidth for cloud applications. In this respect, fairness is achievable if *minimum bandwidth guarantee* and *proportional bandwidth share* are provided [136]. Besides fairness, these approaches try to support *work conserving* (i.e., if a link is a bottleneck for a
specific flow, then the link should be fully utilized), scalability, and practicability. The existing approaches are differed in several aspects. (i) Bandwidth guarantee policy is a static reservation that does not preserve work conserving (e.g., Oktopus [23]), or time-varying reservation that works based on the demanded bandwidth by applications (e.g., TIVC [190]). (ii) **Control model** that adjusts rate limits based on the demand bandwidth can be either centralized (e.g., Oktopus, EyeQ [89]) or distributed (e.g., ElasticSwitch [137], [80]). (iii) Bandwidth guarantee model is based on the network topology (e.g., all mentioned approaches here) or the application communication (e.g., CloudMirror [106]) that is more efficient in reserving bandwidth than its counterpart. All these approaches for network virtualization suffer from the lack of guaranteeing data delivery within deadline. They thus are suitable for batch applications, but not for OLTP applications that are stringent to predictable response time and throughput. There are new protocols to meet this need of OLTP applications and guarantee the completion of data flows within required deadline (e.g., D2TCP [177]). The interested readers are referred to survey paper by Blenk et al. [30] on this area of research.

- **No support for complex operations and transactions.** Data stores have severe limitations on a set of query types and transactions they support. **Join queries**, as a complex operation, can be provided by converting it into a set of simple primary-key queries, which is an error-prone task especially when strong consistency comes to be important [181]. Transactions support can be eased via co-located the related data in the same server (or at least in the servers in the same logic partitions) instead of using common data partitioning mechanisms [102] like random (hash-based), round robin, and key range.

### 2.2.4 Architecture, Motivations, and Challenges of Inter-Cloud Storage

To overcome the above challenges, 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.
2.2 Overview

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 [39]). Spillner et al. [164] focused more on the Inter-cloud storage architecture that allows user to select a data store based on services cost or minimal downtime. Inspired by this architecture, we pictorially clarify a layered inter-cloud architecture and discuss its layers (down to top) as shown in Fig. 2.6.

- **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 and manipulating 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 (resp. read) the object in (resp. 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 (API), which is exploited by the client application to issue operations to data stores (often the closest one).

- **Meta data 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 meta data (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, we explain how to Get (read) or Put (write) the object across data stores. An issued Get request from client library is usually directed to the closest DC, 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 (Fig. 2.6, Get in DC1). In respect to a Put request, after finding the location of replicas, the object first 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, to keep replicas consistent (i) in the case of eventual consistency, the Put operation is propagated to other replicas (Fig. 2.6, Put in DC2), and (ii) in the case of strong consistency, in a simple way, all replicas are locked and the Put request is synchronously propagated to them (Fig. 2.6, Put in DC1).
Motivations of Inter-Cloud Storage

Fig. 2.8 shows the key motivations of leveraging Geo-replicated data stores. These are listed as below:

- **Higher availability, durability, and data lock-in avoidance.** 3-way data replication across data stores is a simple way to achieve higher availability, durability, and data lock-in avoidance if the deployed data stores belong to different cloud providers. This scheme of replication provides availability of 7 nines that is adequate for most applications [124]. However, Cidon et al. [54] demonstrated that 3-way random replication is inadequate for protecting data against correlated node failures (i.e., several nodes simultaneously fail due to a specific failure like power outage), while two replicas are sufficient for guarding data against independent node failures (i.e., nodes fail due to an individual failure event like disk failure). Another way to achieve these purposes is *erasure coding*, as used by Window Azure Storage [40]. Erasure coding degrades response time of reads and writes in the case of data recovery, but it is cheaper than data replication in storage cost.

- **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. To decide which data stores are the best for cost optimization, application providers should consider the demanding QoS level such as availability, response time, consistency level, and etc. Based on these requirements, the decision on cost optimization 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 (Section 2.6).

- **Lower latency.** By deploying applications across multiple cloud services rather than a single cloud service, application providers can access more DCs to achieve lower user-perceived latency. Although this deployment intuitively reduces access latency, some applications still demand stringent latency constraints. To meet these constraints, (i) replicas should be stored close to users, (ii) the data manipulated by the same transactions should be co-located in the data stores which are close to each other, and (iii) the location and roles of replicas (either master or slave) should be carefully determined when the Geo-replicated data stores are configured in a
quorum-based scheme [158]. In spite of these efforts, users may still observe the latency variation. To reduce it, the effective ways are: caching data in memory (e.g., in Facebook [131]), issuing redundant reads/writes to replicas (e.g., CosTLO [187]), and using feedback from servers and users to prevent requests redirection to saturated servers (e.g., C3 [168]).

- Higher data confidentiality, integrity, and auditability. Cryptographic protocols with erasure coding and RAID techniques are used to create a layer on top of multiple data stores to attain higher availability, integrity, confidentiality, security, and durability even if particular data store is confronted with a malicious adversary. Geo-replicated data stores use these techniques to achieve all or some of these. For example, HAIL [33] uses a mix of RAID technique and cryptographic protocol to provide high availability and integrity for static data (i.e., not for multiple versions of data). Besides availability and integrity, DepSky [28] improves confidentiality and security of data via encryption, encoding, and replication of data across data stores. Some Geo-replicated data stores apply these techniques not only to improve the mentioned purposes but also to reduce storage and network costs [109]. There has been a spate of work on this topic and interested readers are refereed to survey paper by Fernandes et al. [73].

Challenges of Inter-Cloud Storage

The deployment of data-intensive applications across data stores is faced with the key challenges, as shown in Fig. 2.7. These challenges 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, 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 [82]) or open API (e.g., JCloud\(^6\)) to make applications portable across data clouds, and (ii)

\(^6\)JCloud project. [https://jclouds.apache.org/](https://jclouds.apache.org/)
a framework (e.g., CDPort [12]) 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 [198]) or without deadline (B4 [88]). 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 [105] increases network utilization, Postcard [72] minimizes the cost of data transferring, and Wu. et al. [183] 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) [101]. 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 [88] deployed by Google’s Inter-DC WAN and the architecture studied the transfer of Big data across DCs [183], work based on SDN to improve the utilization of Inter-DCs WAN. All 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 the data.

- **Strong consistency and transaction guarantee.** Due to considerably high communica-
2.3 Data model

Data model reflects that how data is logically organized, stored, and retrieved in the data cloud. It has a prominent effect on data consistency, transaction management, and query processing. We therefore study data model from different aspects and map data stores to the provided data model taxonomy in Fig. 2.9.

2.3.1 Data structure

Data structure affects the speed of assimilation and information retrieving. It is divided into three categories. (i) **Structured** data refers to data that defines the relation between
the fields specified with a name and value (e.g., relational databases (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 scheme known for its application and database deployments (e.g., document and extensible DBs). It supports simpler query (e.g., primitive operations) and transaction facilities as compared to structured data. (iii) Unstructured data has no identifiable structure and generally refers to data that neither have 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.

The aforementioned logical data structures can be internally organized in three schemes. In a row-oriented scheme, data is organized row-by-row within a block, where all values of rows are sequentially stored. In a column-oriented scheme, data is organized column by column and closely related to database normalization and well-suited for OLAP due to its attributes’ accessibility. In hybrid-oriented scheme [182], data is organized in both row- and column-oriented schemes with two granularities: coarse- and fine-grained. In the former, each table is stored in both row- and column-oriented, while in the latter some parts of the table are stored in row-oriented and other parts in column-oriented. With these defined schemes, structured data can be organized in all above schemes, semi-structured in row- and hybrid-oriented schemes, and Unstructured data in a row-oriented scheme.

2.3.2 Data abstraction

Data abstraction refers to different levels of storage abstraction in data stores. It does not support a strictly high to low hierarchy [83]. As the storage abstraction level increases, the storage performance reduces and its management is more difficult. These levels of data abstraction are as below:

1. Block-level is highly customizable and provides the fastest access to data for a computational unit (i.e. virtual machine (VM)). It can be classified into (i) directed-attached storage coming with the VM that provides highly sequential I/O performance with a low cost, and (ii) block storage that pairs with VM as a virtual block and can be used like a local disk.
2. **File-level** is the most common form of data abstraction due to its ease of management and has a higher abstraction level compared to block-level. File level is provided in the forms of (i) **object storage** that enables users to store binary large objects anywhere and anytime, and (ii) **online drive storage** that provides user with a folder on the Internet as storage. In both forms, data can be stored through simple APIs, but they lack customization.

3. **Database mode** offers storage in the forms of **relational data store** (RDS) 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, NoSQL 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 the ACID properties.

### 2.3.3 Data access model

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

1. **Key-value data store** stores keys and values which are indexed by keys. Key-value data store supports primitive operations and 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 data store** 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**, or both. A **shard** is horizontal partition of data in the database.

3. **Extensible record data store** is analogous to table, where rows and columns of the table are simultaneously split and distributed on nodes [43]. Table rows are split as a **tablet** based on the primary key range that represents the unit of distribution and
load balancing. Table columns are also grouped and then distributed over multiple servers. For example, as shown in Fig. 2.3, the information about Customers and their Campaign are classified in the separate groups and then all information of a customer may or may not be on the same server. Each cell of the table can contain multiple versions of the same data. These versions are indexed in decreasing timestamps order, thereby the most recent version can always read first [150]. This scheme of data model is called NewSQL which is equivalent with entity group in Megastore, shard in Spanner, and directory in F1.

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.

The above data access models should be selected based on the needs of application. Highly scalable key-value store can be suitable if the application contains a type of object that requires a simple search on one or more attributes. Document store model works well when the application requires different types of objects and tolerates eventual consistency. Extensible record store is suitable if the application requires multiple types of objects and retrieves objects based on any attribute. RDB is suitable if the application demands high scalability and ACID properties for small scale of database.

2.4 Data Dispersion

Data dispersion schemes improve QoS metrics and deploying each scheme depends on user’s objectives. These schemes, as shown in Fig. 2.1, are discussed in this section.

2.4.1 Data replication

Data replication increases availability, improves performance through serving requests by different replicas, and enhances user-perceived latency by assigning requests to the closest replica. However, it generates significant overheads like replicas coordination and storage cost. These pros and cons are affected by facets of data replication based on the taxonomy in Fig. 2.10.
Figure 2.10: Data replication taxonomy

**Data replication model**

There are two replication models for fault tolerant data stores: *state machine replication* (SMR) and *Deferred-update (transactional) replication* (DUR) [135].

In SMR (also called *active replication*), all replicas receive and process all operations in the same order. SMR implies that the servers process operations in a *deterministic way*, thereby consistency is guaranteed. This replication model has the advantages of (i) simplicity since the code in client and server is the same, and (ii) failure transparency because if a replica fails the operations still process by the other replicas. However, in SMR, every update transaction is processed by all servers and adding servers cannot increase throughput. Thus, SMR is not scalable and its throughput is confined by the processing power of a replica. Scalable-SMR (S-SMR) [29] addresses this issue in Geo-replicated data stores through (i) partitioning database and replicating each partition, and (ii) using cache techniques to reduce communication between partitions without compromising consistency. Scalable-SMR is suitable for contention-intensive and irrevocable transactions that require abort-free execution.

DUR resembles single-/multi- master replication and scales better than SMR. This is because DUR locally executes update transactions on a primary server and then propagates updates to other servers. Thus, in DUR, the update transactions do not scale with
the number of replicas, while the read-only (RO) transactions scales out with number of replicas as they are executed only on a server without communication across servers. To improve throughput of the update transaction, the Distributed-DUR (D-DUR) approach [153] first partitions the database (i.e., scaling up) and then replicates each partition (i.e., scaling out), while the Parallel-DUR (P-DUR) approach [133] conducts both of these operations in the reverse order. P-DUR achieves scalability through optimizing cross-partition transaction communication and D-DUR enhances scalability via accommodating larger database that is not fit in the main memory of a single server.

Therefore, in respect to pros and cons of both replication models, none of them is a clear winner, and the choice of model depends on factors like workload type, network congestion, and parallelism on CPU. Hybrid transaction replication [98], as a combination of SMR and DUR, borrows the parallelism in DUR and abort-free execution of transactions in SMR to improve scalability and performance in reads and writes execution.

**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.

Partial replication is more economical than full replication in storage and bandwidth costs and is also more scalable in transactions since these transactions access to a subset of DCs rather than all DCs. It is suitable for read-intensive objects that receive most requests from specific DCs (indeed requests are issued from applications deployed in DCs), and also for write-intensive objects since each request is submitted to \( r \) replicas instead of \( n \) DCs \((r \leq n)\) [159]. However, partial replication demands more complex mechanisms for consistency and transaction support which potentially degrades response time. Many partial replication protocols provide such mechanisms, but in the cost of communication with all replicas (the same as full replication) [17]. This is because that overlapping transactions cannot be predicated. Two transaction \( T_i \) and \( T_j \) are **overlapping** if 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. To circumvent such issue, the deployment of **genuine partial**
replication enforces a transaction involves only the subset of servers/DCs containing the desired replicas for coordination.

Full replication is less scalable than partial replication. The scalability of full replication is redistricted by the number of transactions that can be ordered and the capacity of each replica that certifies and processes transactions. However, full replication is more natural for read-intensive objects that receive requests from all DCs rather than a subset of DCs. Also, as the percentages of global (multi-shard) transactions increases, full replication is more suitable than partial replication since in partial replication more remote data is accessed, and consequently the latency can be raised.

With respect to pros and cons of both full and partial replication, characteristics of workload and the number of DCs are main factors in making a decision on which data replication degree should be selected. If the number of DCs is small, full replication is preferable over the partial one; otherwise, if global transactions access few DCs, partial replication is a better approach to deploy.

**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 running across data stores.

**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 [42]
and Amazon Web Service (AWS) RDS\footnote{AWS Relational Database Service (RDS). \url{https://aws.amazon.com/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.* In single master (also called *primary backup replication*), a master replica can be a replica placed at the closest distance to the user or a replica receiving the most updates. All updates are first submitted to this master replica for processing, and then the results are propagated either eagerly or lazily to all other replicas. In single master with lazy replication, replicas receive the updates in the same order accepted in the master and might miss the latest versions of updates till the next re-propagation by the master. Single master approach has the benefits of conflicts avoidance and strong consistency (e.g., PNUT and Windows Azure Storage \cite{40}). However, In this approach, the throughput of update of transactions is limited to the capacity of primary replica, thereby making a potential bottleneck and single point of failure. These issues can be mitigated somehow by chain replication \cite{178} in which all servers are linearly ordered to form a chain, and updates are directed to *head* of chain and are propagated until to reach the last node in chain (i.e., *tail*) which serves all reads. Alternatively, another solution is multi-master replication at the cost of replica coordination.

*Multi-master.* Every replica (usually the closest replica to the DC issued update operations) can accept update operations for processing, and then the master replica propagates the updated data to other replicas either eagerly or lazily. This approach increases the read performance but might receive stale data (e.g., as in Dynamo). In multi-master with lazy replication, there is no guarantee that all replicas receive the updates in the same order that results in replicas divergence and thus the need for conflict resolution.

*Quorum replication.* Under classical quorum protocol, the data store writes an object by sending it to a set of replicas (i.e, a write quorum) and reads it from a possibly different set of replicas (i.e., a read quorum). The reads and writes are submitted to a *coordinator*
replica, being single-master or multi-master, in the quorum protocol. The coordinator location, the role of different replicas (being in the write or read quorum set), and replicas location are important issues in the protocol. Sharov et al. [158] optimized operation latency by addressing these issues when the workload changes. Though the classical quorum protocol guarantees strong consistency, many Geo-replicated data stores like Cassandra, Dynamo, Riak\(^8\), and Voldemort\(^9\) deployed an adaptable version of this protocol to achieve higher availability in the cost of weaker consistency. In the adaptable protocol, a read/write is sent to all replicas, and the read/write is considered successful if the acknowledgments are received from a quorum (i.e., majority) of replicas. The adaptable protocol is configured with \(W\) and \(R\) respectively as the minimum number of replicas must participate in synchronous writes and reads. The configuration can be determined by the requested availability, durability, and consistency in two forms: strict and partial. In strict quorum (providing strong consistency), any two quorums (sets) have non-empty intersection (formally, \(W + R > N\), where \(N\) is the total number of replicas in the system), while in partial quorum (providing eventual consistency) at least two quorums should not overlap (formally, \(W + R < N\)). 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 [131] seeks to place the associated data in the same location without considering the localized data serving. (4) Associated data placement (ADP) [193] makes the

\(^8\)Riak NoSQL Database. [http://docs.basho.com/riak/kv/2.1.4/learn/why-riak-kv/](http://docs.basho.com/riak/kv/2.1.4/learn/why-riak-kv/)

\(^9\)Voldemort project. [http://www.project-voldemort.com/voldemort/](http://www.project-voldemort.com/voldemort/)
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. 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 2.1.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Synchronous propagation</th>
<th>Asynchronous propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single master</td>
<td>Case 1:</td>
<td>Case 2:</td>
</tr>
<tr>
<td></td>
<td>• Latency is high especially across DCs and is constrained by the slowest DC.</td>
<td>• Latency is lower than that in Case 1.</td>
</tr>
<tr>
<td></td>
<td>• Strong consistency is guaranteed, no matter reads are performed on which replica.</td>
<td>• If reads are submitted to the master replica, then consistency is guaranteed and latency increases. Otherwise, the trend for consistency and latency is reverse.</td>
</tr>
<tr>
<td></td>
<td>• Latency source is the latency within (is very small) and across DCs.</td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• This trades response time for consistency.</td>
<td>• Both aspects of consistency-latency trade-off are achievable.</td>
</tr>
<tr>
<td>Multi master</td>
<td>Case 3:</td>
<td>Case 4:</td>
</tr>
<tr>
<td></td>
<td>• It incurs latency the same as that in synchronous single master strategy, though to roughly lesser degrees.</td>
<td>• Latency is less than that in Case 3, and roughly in Case 2.</td>
</tr>
<tr>
<td></td>
<td>• Strong consistency is guaranteed, no matter where reads are submitted.</td>
<td>• The trade between consistency and latency is similar to that in Case 2.</td>
</tr>
<tr>
<td></td>
<td>• Latency source is the same as in Case 1 plus conflict detection and resolution time.</td>
<td>• Latency source is the same as that in Case 2.</td>
</tr>
<tr>
<td></td>
<td>• This trades consistency for latency.</td>
<td></td>
</tr>
<tr>
<td>Quorum-based</td>
<td>Case 5:</td>
<td>Case 6:</td>
</tr>
<tr>
<td></td>
<td>• If reads are submitted to synchronous updated replica precipitated in quorum, then consistency is guaranteed but latency is high.</td>
<td>• If reads are submitted to asynchronous updated replica precipitated in quorum, then consistency is not guaranteed.</td>
</tr>
<tr>
<td></td>
<td>• Latency source is similar to that in Case 2.</td>
<td>• Latency source is the same as that in Case 2.</td>
</tr>
<tr>
<td></td>
<td>• This trades consistency for latency.</td>
<td></td>
</tr>
</tbody>
</table>
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. Erasure coding increases the storage 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. Thus, erasure coding is more economical than data replication in storage cost, while both handling the same number of failed replicas. For example, a \((3,2)\)-erasure coding tolerates 2 failed replicas which is the same for a 3-way replication (i.e., a system with 3 replicas). The advantages of erasure codes vary from high availability, durability, integrity, confidentiality, and security of data to vendor lock-in avoidance, as deployed across data stores. In spite of these benefits, deciding on when to use erasure coding depends on the following aspects as shown in Fig. 2.11.

**Model**

Erasure coding has two models: **systematic** and **non-systematic**. In the former, \((original)\) \(data\) and \(parity\) chunks are separately stored in \(n - m\) and \(m\) storage nodes, while in the latter the coded data (not original data) are stored in \(n\) nodes. Systematic and non-systematic codes are respectively relevant for archival and data-intensive applications because in these applications the rate of reads is receptively low and high. Systematic codes seem more suitable for data stores because they decode data when a portion of data is unavailable. Comparatively, non-systematic codes require to decode data when it is retrieved.
due to storing the coded data not the original data. Thus these codes 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 is maximum distance separable (MDS) code. For a \((k,m)\)-MDS code, 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 these failed nodes from less than \( k \) surviving nodes. Thus, non-MDS code is more economical than MDS code in the network cost to rebuild the failed nodes. This feature makes non-MDS code more suitable for deploying across data stores, as used in Azure and Facebook. In contrast, MDS code is more efficient than non-MDS code 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 [167]. 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 [68]. 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 can determine which recovery model satisfies the application requirements. Functional repair is desired for non-secure sensitive applications because it requires the dynamics of repairing and decoding rules that results in information leakage. In contrast, the exact-repair is used for systematic codes which are appropriate for archival applications.

- **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 replication policy [202]. 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, as a MDS-code, contains m data chunks and k parity chunks, where each parity chunk is computed from k chunks. When a data chunk is unavailable, there is always a need of k chunks to rebuild the data chunk (no matter which data or parity chunks are used). Thus, the recovery cost of the (k,m)-RS code is k. Google ColossusFS uses the (6,3)-RS code [76] and Facebook HDFS [129] deploys the (10,4)-RS code in a DC (a XOR-based code–using pure XOR operation during coding computation– across DCs) respectively with 6 and 10 data chunks as recovery costs, and correspondingly the storage overheads of 1.5x and 1.4x. 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. To mitigate this issue, Hitchhiker [145] uses the adapted RS code in which a single strip RS code is divided into two correlated sub-stripes. In comparison to RS, this adapted code thus reduces network and disk traffics without additional storage overhead and degrading fault tolerance during rebuilding data block.

Similar to MDS-code, *regenerating* and non-MDS codes [184] 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 [144]. 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 [50] uses *functional*
MSR, as a non-systematic code, and maintains the same fault tolerance and storage overheads 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) \([85]\), as a non-MDS code, 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 tolerates is between \(r + 1\) and \(r + l\). Compared to the (6,3)-RS code in Google and the (10,4)-RS code in Facebook, the (12,2,2)-LRC, as used in Window Azure Storage (WAS) \([40]\), is more cost-effective in both storage and recovery costs due to requiring less chunks to construct the failed chunk. HDFS-Xorbas \([152]\) makes a reduction of 2x in the recovery cost and the growth of 14% more storage by using the LRC rather than the RS code. HACFS \([188]\) code uses LRC and Product Code (PC) \([146]\) to offer a fast code with low recovery cost for hot data and a compact code with low storage cost for cold data. Note that LRC and PC respectively have low recovery cost and storage cost/overhead.

- **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 through 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 \([96]\) codes are effective due to reading the requested data only. Also LRC and XOR-base \([145]\) 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 \([155]\). (ii) *Adaptive batching* of requests makes a trade-off between delay and throughput, as exploited by S3 (Simple Storage Service) and WAS \([125]\). (iii) *Deploying erasure codes*...
across multiple data stores improves availability and reduces latency. Fast Cloud [112] and TOFEC [113] uses 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. [189] consider these factors and investigate 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 (i) the speed of block recovery [152], and (ii) the number of failed blocks that can be handled before data loss. Failures in data stores can be correlated or independent. The correlated failures are either (i) domain failures that happen when a set of servers are down due to malicious attack, or (ii) upgrade failures that occur when a planned upgrade is conducted. The correlated failures can be avoided by placement of parity chunks in different domains to improve data availability and to deviate hot node in data store. With the same failure tolerance, LRCs are better in reliability than RS codes, which in turn, are more reliable than replication [85].

(4) Complexity refers to network topology and code size that respectively capture the number of involving nodes for recovery and the code size of the implemented erasure coding [202]. As the network topology complexity reduces, the recovery time, and thus the response time improves [179].

Data access type

There are two approaches to access chunks of the coded data: unique key and shared key [112], in which a key is respectively allocated to a chunk of coded data and the whole coded data. 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
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 a vital aspect of data management in NOSQL data stores which typically provide high availability, scalability, and low latency at the expense of the weak consistency model that probably results in the violation of application semantics. In contrast, the strong consistency model avoids this downside but increases user-perceived latency and impairs availability and scalability. To understand the pros and cons of these models, three main aspects of data consistency are discussed in this section.

2.5.1 Aspects of data consistency in distributed data stores

Fig. 2.12 shows a taxonomy based on different aspects of data consistency. These aspects are as below:

**Metric**

This is used to measure criteria that determine how much a consistency model is stronger than another and is categorized into discrete and continuous.
Discrete metrics are defined only for data-centric (i.e., from data store perspective) consistency based on time and version which are respectively measured with the maximum number of time unit (termed by t-metric) and data version (termed by v-metric). Anderson et al. [16] define three discrete metrics (Fig. 2.12): (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. Atomicity (as guaranteed in strong consistency) is stronger than regularity, which in turn, is stronger than safeness (as provided by eventual consistency) in terms of correctness and data freshness.

Continuous metrics, as shown in Fig. 2.12, are defined based on staleness and ordering from two perspectives: data- and user-centric consistency. For both perspectives, staleness metric is expressed in either t-visibility or k-staleness with the unit of probability distribution of time and version lag respectively. However, the ordering metric differs for each of these perspectives. For data-centric consistency, the ordering metric is defined in terms of violation, which is measured as the number of violations per time unit. For user-centric consistency, this metric is measured with the unit of probability distribution of violations and is defined in the forms of MR-, MW-, RYW-, WFR-violation as discussed.

Figure 2.12: Data consistency taxonomy
2.5 Data Consistency

later.

Level

Distributed data stores rely on different levels of data consistency: I/O, application, object, flow, and language [13].

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 to be made such as concurrent write-write and read-write on the same data, resulting in inefficiency. This level of consistency also should execute writes and reads in a serial order since it is unaware of the semantics of the application. Many systems are based on this level of consistency, as is studied in this chapter.

Application-level consistency exploits the semantics of the application to ensure the concreteness of invariants\(^{10}\) 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.

Deploying one of two discussed “extreme” consistency levels can degrade efficiency or reusability. To make a trade-off between these two criteria, developers resort to object-level consistency that provides the convergence of replicas to the same value without any need of synchronization across replicas. Conflicted-free Replicated Data Types (CRDTs) [157] support this level of consistency for the objects whose values 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\(^{11}\). Some components are insensitive to message order delivery

---

\(^{10}\)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 \(x \geq 0\)).

\(^{11}\)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.
as a semantic property\textsuperscript{12}, and they are confluent and produce the same set of outputs under all ordering of their inputs \cite{14}. Flow-level consistency demands manual definition for confluent components, resulting in error-prone. Moreover, flow-level consistency is more powerful than object-level consistency and it has more applicability than object- and language-level consistency. Indigo \cite{22}, as a middleware on top of Geo-replicated data stores, exploits this level of consistency and provides explicit 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 \cite{15} shows a strong connection between the need of distributed coordination across replicas and logical monotonicity. Bloom language \cite{15} 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, called monotonic program\textsuperscript{13}, provides output as it receives input elements, and thus guarantees eventual consistency under any order of inputs set. Unlike Bloom, QUELEA language \cite{163} 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) \cite{25}, and monotonic atomic view (MAV) \cite{19}.

**Model**

This is classified into two categories: user- and data-centric \cite{169} which respectively are vital to application and system developers.

User-centric consistency model refers to data consistency from the users’ perspective and can be classified into four categories, as shown in Fig. 2.12. Monotic Read (MR) guarantees that a user observing a particular value for the object will never read any previous value of that object afterwards \cite{194}. It can be forced by maintaining the highest commit timestamp record for each data item at the client cache. Monotic Write (MW) enforces

\textsuperscript{12}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 behavior.

\textsuperscript{13}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.
that updates issued by a user are performed on the data based on the arrival time of updates to the data store. It is implemented by a counter for each data item at the client, and the counter increases as a new update to the object is committed. 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. [35] designed a framework to provided these client-centric consistency models and the atomic transaction on AWS S3. Also, Bermbach et al. [26] proposed a middle-ware on eventually consistent data stores to provide MR- and RYW-consistency.

**Data-centric consistency** aims at coordinating all replicas as soon as possible from the internal state of data store perspective. It is classified into weak, strong, and adaptive consistency. Weak consistency offers low latency and high availability in the presence of network partitions, 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). On the contrary, strong consistency provides simple semantics for the developer, 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 [171]. Adaptive consistency is switching between a range of consistency models (e.g., weak and strong consistency) based on the application requirements/constraints like availability, partition tolerance, and consistency. Based on the CAP theorem [78], data stores provide only two of three desirable properties: consistency (C), availability
(A), and partition tolerance (P). Thus, 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. 2.13). 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. 2.14). In both CAP and PACELC, the consistency model has a determining effect on achieving availability, response time, and fault tolerance. Hence, we discuss the models of weak consistency (Fig. 2.15), and then define strong and adaptive-level of consistency models in the rest of section.

### 2.5.2 Eventual consistency

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

**Definition**

*Eventual* consistency is informally defined as all replicas eventually converge to the last update value. Bouajjani et al. [32] introduced a formal definition of eventual consistency with two properties possessed by distributed systems: *safety* and *liveness*. The former determines the correct effects of operations, and the latter enforces that all replicas even-
2.5 Data Consistency

Eventually converge based on the operations order. Eventual consistency purely supports liveness, while lacking safety, and consequently leads to incorrect intermediate results. The safety property is assessed in terms of inconsistency window (ICW) with two common metrics: t-visibility and k-staleness. ICW might be affected by the communication latency, system load, and number of replicas. Bailis et al. [20] studied a theory, Probabilistically Bounded Staleness (PSB), that predicts the expected bound on staleness in terms of these two metrics and measures how far data store’s behavior deviates from that of strongly consistent store. They used PSB for partial quorum (e.g., as used in Dynamo) and concluded that eventual consistency under partial quorum replication is “good enough” in data freshness while providing a considerable reduction in latency; as experimentally confirmed for S3 [27] and Cassandra [143].

Implementation components

Eventual consistency-based data stores employ optimistic/lazy replication in which (i) the operation is typically submitted to 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 [67] in the background. To do so, the following crucial components should be provided in data stores [149].

1) Scheduler. This allocates computational resources for processing both foreground update queries and background updates propagation to make a balance between delay and consistency. However, current data stores (e.g., DynamoDB) are in favour of allocating most resources for processing foreground queries. On-demand approach [4] couples the process of both update types (i.e., foreground and background), while QUTS approach [141] decouples their execution and separately allocates computational resources to each of the update types. Xu et al. [191] moved away from both extreme approaches and proposed mechanisms to schedule foreground and background updates, thus making a latency-consistency trade-off.

2) Conflict detection. Conflicts happen when operations are simultaneously submitted to the same data in different servers/DCs (i.e., in multi-master and non-blocking systems). There are four approaches to deal with conflicts. (i) Conflict detection approaches strengthen the application semantic and avoid the problems arising from ignoring con-
flicts. These approaches are classified into syntactic and semantic [149]. 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 via conflict resolution techniques (Section 2.5.2).

(3) Commitment protocol. This determines the agreement on the partial or total ordering of requests respectively in respect to weak or strong consistency. Partial ordering is typically achieved by vector clocks, while total ordering is ensured by (i) submitting all updates to a primary replica, which is not suitable in the cloud context due to non-scalability, and (ii) using a logging approach that maintains the order of submitted requests to replicas and leveraging a commitment protocol like 2 Phase Commit (2PC) [5] and Paxos [79].

Conflict resolution

This handles the detected conflicts to hinder replicas divergence (i.e., guaranteeing the safety property), and is divided into below approaches (Fig. 2.15).

(1) Last Write Win (LWW) [172] ignores conflicts, and the update with the highest timestamp is accepted (e.g., Riak (by default), SimpleDB, S3, and Azure Table). 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 procedure 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.

14 Azure Table storage. https://azure.microsoft.com/en-us/services/storage/
(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 [157].

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. [10] 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, the update first 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 [157], 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.

2.5.3 Causal and Causal+ consistency

This section first introduces a formal definition of causal and causal+ consistency models, and then it describes a description of the source and type of dependencies found in this model.

Causal consistency definition

Causal consistency maintains the merits of eventual consistency, while respecting to the causality order among requests applied to replicas. Causal consistency is stronger and more expensive than eventual consistency due to tracking and checking dependencies. It defines Lamport’s “happens-before” relation [79] between operations $o_1$ and $o_2$ as $o_1 \prec o_2$. Potential causality $o_1 \prec o_2$ maintains the following rules [6]. Execution thread: If $o_1$ and $o_2$ are two operations in the same thread of execution, then $o_1 \prec o_2$ 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 $o_1 \prec o_2$. Transitivity: if $o_1 \prec o_2$ and $o_2 \prec o_3$, then $o_1 \prec o_3$. Causal consistency does not support concurrent operations (i.e., $a \not\prec b$ and $b \not\prec a$). 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 $o_1 \prec o_2$, 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 semantic is 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 [70]. 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 [118], Eiger [119], and Orbe [69] track both dependency sources. Dependency types can be either potential or explicit and can be defined for an operation (as in Eiger and ChainReaction [11]) 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. Potential dependencies implementation 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. Explicit dependencies deployment, as used in Indigo [22], 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.

### 2.5.4 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 op-
eration 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
[178].

Strong consistency guarantees that all read and write operations receive a global time-
stamp 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 be-
cause 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 ser-
vices 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 aggressively contest on marketing
where application providers buy or rent out-house storage infrastructure. They offer
a variety of pricing plans for different storage services with a set of features such as
availability and durability. Pricing plans offered by data stores are typically divided in
two categories [130]: 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-linear 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 AWS. This pricing can been seen as a special form of the multi-part tariffs scheme, where the fixed price is zero. Note that in the standard form of multi-part tariffs, the cost is defined as a fixed cost plus multi-ranges of costs, where in each range the cost is constant. One common form of this scheme is a two-part tariffs that are utilized in a range of services like credit cards which charge an annual fee (i.e., fixed cost) plus a per-transaction fee (i.e., per-unit charge). Also, the price of some services in data stores follows the two-part tariffs which include 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 AWS 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 provider buys 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.

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. 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: (i) Simple Storage Service (S3) is a highly

---

15 Key features of storage classes. https://aws.amazon.com/s3/storage-classes/
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. RRS 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,\(^\text{17}\) 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) 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\(^\text{18}\): (i) Multi-regional storage is appropriate for


\(^{18}\)Google storage classes. https://cloud.google.com/storage/
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 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 2.2 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) \textit{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) \textit{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 2.2: The characteristics of different storage classes for the well-known Cloud providers: Amazon Web Service (AWS), Azure(AZ), and Google(GO)

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

† 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 refereed 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. [44] proposed an algorithm to replicate objects across data stores so that users obtain the specified availability subject to budget constraint. Mansouri et al. [124] 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 [108] uses the duplicate scheme to reduce replication cost while achieving the same durability as in triplicate replication. CIR [111] 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 (Section 2.5), several other studies focused on lowering cost with adaptive consistency model instead of a particular consistency model. The *consistency rationing* approach [100] 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 [51] 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 [74] 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 a crucial SLA and is defined as (i) a constant in the unit of Round Trip Time (RTT), distance, and network hops, or (ii) cost that is jointly optimized with monetary cost of other resources. Defining latency cost instead of latency as a simple constant 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. OLAP applications satisfy the latency constraint by optimization of placement of data and task, while OLTP ones achieve it via replication. Replica placement has an essential effect on optimizing latency cost or satisfying it as a constraint. This placement also can be effective in reducing the cost of resources if a joint optimization of latency and resources costs are considered rather than latency as a constraint.

2.6.4 Cost optimization based on multi-QoS metric

Application providers employ Geo-replicated data stores and require an optimization regarding the cost spent on resources under multi-QoS metrics. In addition to storage, network, and computing costs, application providers may incur data migration cost that is calculated based on the data size transferring out from data store and its corresponding network cost. Data migration happens due to the changes to the optimized data placement configuration. Several factors cause the configuration change such as data 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” [176]. In Nomad, the changes in users’ location are recognized based on simple policies that monitor the location of users when they access an object. According to these policies, the object is migrated to a given location when it is accessed more than a certain number of times from that location (e.g., 10 times), in a certain period of time (e.g., 10 days), and at a certain rate (e.g., 5 times per day). 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 [58], k-center/k-median [87], ski-rental [154], and so on. 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. FCFS [139] 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 opti-
mized 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 [138], (ii) maximizing the number of users whose locality can be preserved with a given number of replicas for each user [175], (iii) graph partitioning based on the relations between users in OSN (e.g., cosplay [91]), and (iv) selective replication of data across DCs to reduce the cost of reads and writes [116]. (4) The emergence of content cloud platform (e.g., AWS Cloud-Front\(^\text{19}\)) and Azure CDN\(^\text{20}\) offers a natural solution for building 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 [46]. These factors determine the topology of CDN on data stores against the transitional CDN with a given topology (e.g., a tree rooted at the origin server).

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 [201]. The trade-off can be also seen in video-on-demand service in which video transcoding\(^\text{21}\) is a computation-intensive operation, and storing a video with a variety of formats requires a huge amount of storage. Incoming workload on the video and the performance requirement of users determine whether the video is transcoded on-

\(^{19}\)AWS CloudFront. [https://aws.amazon.com/cloudfront/](https://aws.amazon.com/cloudfront/).


\(^{21}\)Video transcoding is the process of converting a compressed digital video format to another.
2.6 Data Management Cost

demand (i.e., trade storage for computation) or stored with different formats (i.e., trade computation for storage).

Storage-cache trade-off. Cloud providers offer different tiers of storage with different prices and performance metrics. A tier (e.g., S3) provides low storage cost but charges more for I/O, and another tier (e.g., EBS and Azure drive) provides storage at higher cost but I/O at lower cost [52]. 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/precomputed 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 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 the object in a data store with either the cheapest network or storage for its 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 [123]. 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 (e.g., NCCloud [50]).

Reserved-on demand storage trade-off. AWS 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 [180], 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 2.2. It considers these classes for key-value applications that generate time-varying workloads in the form of long-tail on the objects [24]. An instance of these applications is Online Social Network (OSN) [129] 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 2.3.

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. [44]. 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. [57] 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. [55] 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) when data automatically are moved between low- and high-cost storage media. FCFS [139] 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* [174]. 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 [107]. Being inspired by these various features, automatic selection of cloud providers based on their capabilities and user’s specified requirements are proposed to determine which cloud providers are suitable in the trade-offs such as cost vs. latency and cost vs. performance [147].

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 [81] proposed various replica placement algorithms to enhance availability and scalability for encrypted data chunks while optimizing the storage and communication cost. SPANStore [185] 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 [91] 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 [121], energy-aware dynamic server provisioning [114] and load balancing among Geo-distributed DCs
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 [139], 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 (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) [114, 199] technique to conduct dynamic migration of objects. In [199], 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 [199]. 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. [95] 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 [26] 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 [139] also made this trade-off as already discussed. The third trade-off can be bandwidth vs. cache as somehow simulated in DeepDive [132] 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 refereed to survey papers [7, 126] for detailed discussion on VM migration techniques.

Similarly, data migration is classified into two approaches. The first approach is *live data migration* [176]. 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 [64,71].

The second approach is *non-live data migration* [176]. 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 and ElasTraS [62].

There are several factors affecting data migration: the changes to cloud storage parameter (e.g., price), optimization requirements, and data access patterns. In response to these changes and requirements, a few existing studies discuss data migration from private to public cloud [140], and some study object migration across public cloud providers [128, 187]. In [187], 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 matter. Mseddi et al. [128] 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. [36] proposed MetaCDN that exploits cloud storage to enhance throughput and response time while ignoring the cost optimization. Chen et al. [46] 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. [134] 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 [151], 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 2.4 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).

## 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.
Table 2.4: Summary of Projects with Monetary Cost Optimization

<table>
<thead>
<tr>
<th>Project specification</th>
<th>Cost elements</th>
<th>QoS metrics/constraints</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mansouri et al. [124]</td>
<td>✓</td>
<td>✓</td>
<td>(i) Dynamic programming and Brute-force search. (ii) Cost-based availability optimization. (iii) Minimizing cost under a given availability for non-partitioned objects and maximizing availability under a given budget for partitioned objects.</td>
</tr>
<tr>
<td>MDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRCR [108]</td>
<td>✓</td>
<td>✓</td>
<td>(i) Proactive replica checking. (ii) Cost-based durability optimization. (iii) Reducing $\frac{3}{2}$ to $\frac{2}{3}$ storage cost as compared to triple storage cost.</td>
</tr>
<tr>
<td>SDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scientific workflows</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIR [111]</td>
<td>✓</td>
<td>✓</td>
<td>(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.</td>
</tr>
<tr>
<td>SDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scientific workflows</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistency rationing [100]</td>
<td>✓</td>
<td>✓</td>
<td>(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.</td>
</tr>
<tr>
<td>SDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OLTP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mansouri and Buyya [123]</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓</td>
<td>(i) Dynamic programming and greedy algorithms. (ii) Storage-network trade-off. (iii) Providing optimization cost for objects in OSNs.</td>
</tr>
<tr>
<td>a dual DC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3 [74]</td>
<td>✓</td>
<td>✓</td>
<td>(i) A selective solution based on defined rules. (ii) Cost-based consistency optimization. (iii) Providing serializability, snapshot isolation, and eventual consistency.</td>
</tr>
<tr>
<td>SDC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.4: Summary of Projects with Monetary Cost Optimization

<table>
<thead>
<tr>
<th>Project specification</th>
<th>Cost elements</th>
<th>QoS metrics/constraints</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Storage Read</td>
<td>Migration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write VM Latency Bandwidth Availability</td>
<td>Consistency level</td>
<td></td>
</tr>
<tr>
<td>Shankaranarayanan et al. [156]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(i) Linear programming  (ii) Cost-based latency optimization. (iii) Minimizing latency in quorum-based data stores and optimizing monetary cost under the latency constraint.</td>
</tr>
<tr>
<td>MDC OSN</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>CostLO [187] MDC Data-intensive</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>FCFS [139] SDC File system</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(i) Ski-rental problem. (ii) Storage-cache trade-off. (iii) Deploying EBS and S3 to reduce monetary cost for a file system.</td>
</tr>
<tr>
<td>Khanafer et al. [97] SDC File system</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>Jiao et al. [90] MDC OSN</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>Hu et al. [84] MDC CDN</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(i) Lyapunov optimization technique. (ii) Cost optimization based on single QoS metric. (iii) Seeking the optimal solution without the future knowledge in regard to workload.</td>
</tr>
<tr>
<td>Chen et al. [46] MDC CDN</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
</tbody>
</table>
### Table 2.4: Summary of Projects with Monetary Cost Optimization

<table>
<thead>
<tr>
<th>Project specification</th>
<th>Cost elements</th>
<th>QoS metrics/constraints</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Storage</td>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>Papagianni et al. [134]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>COMIC [192]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>Wu et al. [183]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>Qiu et al. [140]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(i) Lyapunov 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.</td>
</tr>
<tr>
<td>Ruiz-Alvarez and Humphrey [147]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
<tr>
<td>SPANStore [185]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(i) Linear programming. (ii) Cost optimization based on multi-QoS metric. (iii) VM cost optimization only for writes propagation.</td>
</tr>
<tr>
<td>Chiu and Agrawal [52]</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>(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.</td>
</tr>
</tbody>
</table>
Table 2.4: Summary of Projects with Monetary Cost Optimization

<table>
<thead>
<tr>
<th>Project specification</th>
<th>Cost elements</th>
<th>QoS metrics/constraints</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Storage</td>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>Yuan et al. [195,196]</td>
<td>✓</td>
<td>✓</td>
<td>☑</td>
</tr>
<tr>
<td>SDC Scientific workflows</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jokhio et al. [93]</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SDC Video Transcoding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byholm et al. [38]</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SDC Video Transcoding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deelman et al. [66]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SDC data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triones [166]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MDC data-intensive</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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.

(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.

(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.

(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).

(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.
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

---

\(^1\)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 ($2^{80}$ 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 [77]. 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 [99]. This is why some well-known cloud providers have experienced outages in their data centers (DCs) [77], and the number of vulnerability incidents has doubled from 2009 to 2011 [148]. 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 [44]. 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 [77]. 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.
3.1 Introduction

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 paper 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 $\delta$ 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 deter-
3.2 Minimizing cost with given expected availability

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 $\Phi_j$ indicates a subset of DCs (i.e., $\Phi_j \subseteq D$) that contains the $j$th object with $r$ replicas. Therefore, for all $\Phi_j$, $|\Phi_j| = r$ 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 $\Phi_j$. Thus, the replication cost of $\Phi$, referred to the cost of replicas for $M$ objects, is defined as:

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

(3.1)

where $C(\Phi_j) = \sum_{d_i \in \Phi_j} s(d_i)$ 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 $\Phi_j$ ($X_j = 1$) 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 $j$th object under $\Phi_j$, the value of $X_j$ is 0 only when all DCs in $\Phi_j$ fail. Otherwise, the value of $X_j$ is 1 when at least one of DCs in set $\Phi_j$ 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:

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

(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:

\[
\begin{align*}
\min C(\Phi) &= \sum_{j=1}^{M} C(\Phi_j), \\
\text{s.t. } E(\Phi) &= \sum_{j=1}^{M} (1 - \prod_{d_i \in \Phi_j} f(d_i)) \geq TN, \\
\forall Q, E(J_Q) &\propto P_Q, Q = \{ G, S, B \},
\end{align*}
\]

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:

\[
\begin{align*}
\max 1/ \sum_{j=1}^{M} C(\Phi_j),
\end{align*}
\]

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. 

\textit{Case 1: } \( j > 1 \). This solution is calculated to \( MC[j][tn] \), which means the minimum cost for
the \( j \)th object\(^2 \) \((1 < j \leq M)\) with a given number of nines \( t_n \) \((1 \leq t_n \leq T_N)\). To obtain the best placement for the \( j \)th object, we first find all combinations of \( r \) distinct DCs that can be chosen from \( D \). Assume that each combination is denoted by \( \delta \) \((|\delta| = r)\). Since the availability of \( j \)th object is \( E(\delta) \), we have the expected availability \( t_n - E(\delta) \) for the \( j - 1 \) objects. It means we should consider all possible cases with \( t_n - E(\delta) \) for the first \( j - 1 \) objects, assuming \( \delta \) is a set of DCs, which contains the \( j \)th object. Therefore, if we place the \( j \)th object in the set \( \delta \) 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 \( t_n - E(\delta) \) plus the cost of DCs in the set \( \delta \). This mathematically translated into:

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

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

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

The second step of the algorithm is the termination conditions. First, if \( t_n - E(\delta) < 0 \), then \( MC[j][t_n] = 0 \), 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 \( t_n = E(\delta) \), \( MC[1][t_n] \) also is assigned to zero. Third, the value of \( MC \) should be infinity when \( j \) and \( t_n \) 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 \([t_n \times P_Q]\) where \( 1 \leq t_n \leq T_N \). To guarantee constraint (3.5) accurately, \( E(\delta) \) is sorted decreasingly by \( \text{dif}_Q \) where \( 0 \leq \text{dif}_Q = t_n \times P_Q - E(\delta) < 1 \), and then the first \( t_n - \sum_Q E(J_Q) \) 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 \( T_N \) 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:** \(M_C[M][TN]\)

1. for \(tn \leftarrow 1\) to \(TN\) do
2. \(M_C[0][tn] \leftarrow +\infty\)
3. end

4. for \(tn \leftarrow 1\) to \(TN\) do
5. \(M_C[j][0] \leftarrow 0\)
6. for \(tn \leftarrow 1\) to \(TN\) do
7. \(M_C[j][tn] \leftarrow 0\)
8. forall combination \(\delta \in (D, r)\) do
9. if \((tn - E(\delta)) \geq 0\) then
10. if \((j = 1)\) and \((tn = E(\delta))\) then
11. \(M_C[1][tn] \leftarrow (1/C(\delta), M_C[1][tn])\)
12. end
13. if \((j > 1)\) then
14. if \((M_C[j - 1][tn - E(\delta)] = 0)\) then
15. \(M_C[j][tn] \leftarrow 0\)
16. else
17. \(C \leftarrow 1/M_C[j - 1][tn - E(\delta)] + C(\delta)\)
18. \(M_C[j][tn] \leftarrow \max(1/C, M_C[j][tn])\)
19. end
20. end
21. end
22. end
23. 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,
3.3 Maximum Expected Availability with Given Budget

Algorithm 3.2: Maximum Expected Availability with a Given Budget

Input: \( D, M, m, B, r, f(d_i), s(d_i) \)

Output: \( E[M][B] \)

1. for \( b \leftarrow 0 \) to \( B \) do
2. \( E[0][b] \leftarrow 0 \)
3. end
4. for \( j \leftarrow 1 \) to \( M \) do
5. \( E[j][0] \leftarrow -\infty \)
6. end
7. for \( j \leftarrow 1 \) to \( M \) do
8. for \( b \leftarrow 1 \) to \( B \) do
9. \( E[j][b] \leftarrow E[j][b - 1] \)
10. for all combination \( \delta \in (D, m \times r) \) do
11. if \( (b - C(\delta)) \geq 0 \) then
12. \( E(\delta) \leftarrow \text{Call OCP}(\delta, r, m) \)
13. \( e \leftarrow E[j - 1][b - C(\delta)] + E(\delta) \)
14. \( E[j][b] \leftarrow \max(E[j][b], e) \)
15. end
16. end
17. end

The total cost of \( M \) objects can be written as:

\[
C(\Phi) = \sum_{j=1}^{M} \sum_{k=1}^{m} C(\varphi_{j,k}). 
\] (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 \( k \)th 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 \( k \)th chunk with \( r \) replicas is \( \prod_{d_l \in \varphi_{j,k}} f(d_l) \). As a result, the \( k \)th 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 \( k \)th chunk with \( r \) replicas. Obviously, the \( j \)th 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 \( \Phi_j \) can be calculated as:

\[
E(X_j) = E(\Phi_j) = \prod_{k=1}^{\infty} (1 - \prod_{d_l \in \varphi_{j,k}} f(d_l)).
\]  

(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:

\[
E(\Phi) = \sum_{j=1}^{M} E(\Phi_j) = \sum_{j=1}^{M} \prod_{k=1}^{\infty} (1 - \prod_{d_l \in \varphi_{j,k}} f(d_l)).
\]  

(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:

\[
\max \ E(\Phi) \text{ s.t. } C(\Phi) \leq B \text{ and } E(J_Q) \propto P_Q.
\]  

(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 $j$th object in the set $D$ of DCs, all possible $\delta$s 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 $j$th object is $C(\delta)$, we have the budget $b - C(\delta)$ to consider for storing the $j-1$ objects. Thus, considering all possible $\delta$s ($\delta \subseteq D$) and all possible cases with budget $b - C(\delta)$ for $j-1$ objects, $E[j][b]$ is calculated as follows.

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

(3.14)

In the second step, terminal conditions are considered. Clearly, if $b - C(\delta) < 0$ then the subset $\delta$ 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, Eq. (3.11) is not maximized. In order to maximize that, we should maximize availability of each chunk, that is $(1 - \prod_{d_l \in \Phi_j} f(d_l))$, 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 $k$th chunk with $r$ replicas.

Since $n$ is a small constant [77] and the number of replicas, $r$, is 2 or 3 at most [75], 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 $k$th chunk and another one for all possible
Algorithm 3.3: Optimal Chunks Placement

<table>
<thead>
<tr>
<th>Input</th>
<th>(\delta, r, m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>(CA[k][S])</td>
</tr>
</tbody>
</table>

1. \(S = C(\delta, r)\)
2. Procedure OCP\((S, r, m)\)

3. \(\text{forall } (\varphi \in S) \text{ do}\)
   4. \(PCA[0][\varphi] \leftarrow 1\)
5. \(\text{end}\)
6. \(\text{for } k \leftarrow 1 \text{ to } m \text{ do}\)
7.     \(CA[k][S] \leftarrow 0\)
8. \(\text{forall } (\varphi \in S) \text{ do}\)
9.     \(P \leftarrow \forall \varphi' \in S | \forall d_i, d_i \in \varphi' \land d_i \notin \varphi\)
10.    \(CA[k][\varphi] \leftarrow A(\varphi) \times PCA[k-1][P]\)
11. \(\text{if } (k == 1) \text{ then}\)
12.     \(PCA[k][P] \leftarrow \max_{\varphi' \in P}(CA(\varphi'))\)
13. \(\text{end}\)
14. \(\text{if } (k > 1 \text{ and } kr < mr) \text{ then}\)
15.     \(PCA[k][P] \leftarrow \text{OCP}(P, r, k - 1)\)
16. \(\text{end}\)
17. \(CA[k][S] \leftarrow \max(CA[k][\varphi], CA[k][S])\)
18. \(\text{end}\)
19. \(\text{end}\)

Combinations of size \(r\) from \(\delta\), denoted by \(S (|S| = |\delta|)\), where \(\delta\) 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 \(\delta\) (\(|\varphi| = r\)). In more details, \(CA[k][\varphi]\) is the maximum availability of \(k\)th 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 \(\varphi'\) does not include any DC \(d_i \notin \varphi\). \(PCA[k][P]\) 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 \(k\)th chunk with \(r\) replicas, as if we were placing the \(k\)th chunk. Second, we consider all possible placement of the first \((k - 1)\) chunks with \(r\) replicas, which are placed into \(P\). Thus, if set \(\varphi\) is considered for \(k\)th 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 \(k\)th chunk, \(A(\varphi)\), when we use \(\varphi\). Thus, we have:
\[ CA[k][\varphi] = A(\varphi) \times PCA[k-1][P]. \]  

(3.15)

The computation for \( PCA[k][P] \) is a recursive approach as follows. If \((k > 1 \text{ and } kr < mr)\), 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.

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

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 \(dib(J_Q)\), where \(0 \leq dib(J_Q) = b \times P_Q - b(J_Q) < 1\), 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Replication Availability (Expected Availability, (EA))</th>
<th>Replication Cost (Budget, (B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 3.1</td>
<td>Limited by (EA)</td>
<td>Minimize</td>
</tr>
<tr>
<td>Algorithm 3.2</td>
<td>Maximize</td>
<td>Limited by (B)</td>
</tr>
</tbody>
</table>

Table 3.2: Data centers parameters

<table>
<thead>
<tr>
<th>DC#</th>
<th>FP</th>
<th>CPO</th>
<th>DC#</th>
<th>FP</th>
<th>CPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d_1)</td>
<td>0.0001</td>
<td>48</td>
<td>(d_6)</td>
<td>0.004</td>
<td>12</td>
</tr>
<tr>
<td>(d_2)</td>
<td>0.0002</td>
<td>36</td>
<td>(d_7)</td>
<td>0.01</td>
<td>6</td>
</tr>
<tr>
<td>(d_3)</td>
<td>0.0004</td>
<td>30</td>
<td>(d_8)</td>
<td>0.04</td>
<td>4</td>
</tr>
<tr>
<td>(d_4)</td>
<td>0.001</td>
<td>24</td>
<td>(d_9)</td>
<td>0.1</td>
<td>2</td>
</tr>
<tr>
<td>(d_5)</td>
<td>0.002</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 [76] 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 [44]. 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) [75].
3.4 Performance Evaluation

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, \( P_G = 50\% \), \( P_S = 30\% \) and \( P_B = 20\% \). 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,
QoS-aware Brokering Algorithms for Data Replication across Data Stores

![Graph](image)

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

![Graph](image)

(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, $EF_G \approx 1.6$.
3.4 Performance Evaluation

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 paper. 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
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%
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.

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

count 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.
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, $8 \times 10^5$ 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 [193]. 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.

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 Gets and Puts.

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 paper.

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. Gets and Puts are usually high in the early lifetime of the object and we say that such object is in hot-spot status. As time passes, Gets and Puts 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 Gets, Puts, 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 Gets and Puts 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 paper with future work issue related to cost optimization of data management across Geo-replicated cloud-based data stores.
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 [71] in which Gets and Puts 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. The paired DCs are considered as

---

1We 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 Puts for updating of the object ($w(t)t_p(d)$). Thus, the residential cost $C_R$ is defined as:

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

**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
Cost Optimization in a Dual Cloud-based Storage Architecture

1% of its sales. To capture this cost, the incurred latency for Gets and Puts is considered as a monetary cost [199]. 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 Puts and Gets 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 [57]) between temporal and backup DCs is about few seconds, and (ii) a significantly lower number of Puts and Gets 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 [176]. 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:

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

(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 $d$, $P(d, 0) \leftarrow 0$ and $\forall t \in [1...T], x_d(t) \leftarrow 0$

2. for $t \leftarrow 1$ to $T$ do
   
   /* DCs (i.e., $d$) are a temporal DC and a backup DC.*/
   
   forall $x_d(t)$ do
   
   forall $x_d(t-1)$ do
   
   Calculate $P(d, t)$ based on Equ. 4.5.
   
   end
   
   end

3. Select $\min_d P(d, T)$ as the optimized overall cost (i.e., Equ. 4.4).

4. 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.

5. All $x_d(t)$s with value of 1 are $x^*_d(t)$s.

6. Return $x^*_d(t)$ and the optimized overall cost.

we define the recursive equation for the OOP algorithm as:

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

(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., ${n \choose 2}$, 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

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

$$C_M(t_{m-1}, t_m) + C_{RD}(x_d(t), t) \leq C_{RD}(x_d(t - 1), t).$$ \hspace{1cm} (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.
4.4 Performance Evaluation

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 \( C_{ove} \).

1. \( C_{ove} \leftarrow 0, \forall t \in [1...T], x_d(t) \leftarrow 0 \)
2. \( C_{ove} \leftarrow \) 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.
3. \( t_m \leftarrow 1 \)
4. for \( t \leftarrow 2 \) to \( T \) do
   
   /* DCs (i.e., \( d \)) are a temporal DC and a backup DC.*/
   
   forall \( x_d(t) \) do
   
   7. \( C_{RD}(.) \leftarrow \) Determine \( x_d(t) \) by minimizing \( C_{RD}(x_d, t) \)
   
   8. \( C_{ove} \leftarrow C_{ove} + C_{RD}(.) \)
   
   end
   
   if (Equs. 4.6, 4.7, and \( x_d(t - 1)! = x_d(t) \)) then
   
   10. \( t_{m-1} \leftarrow t_m, t_m \leftarrow t, C_M \leftarrow \) calculate \( C_M(t_{m-1}, t_m) \)
   
   11. \( x_d(t) = 1, C_{ove} \leftarrow C_{ove} + C_M \)
   
   end
   
   else
   
   14. \( x_d(t) \leftarrow x_d(t - 1) \)
   
   end

17. \( x_d(s) \) with value of 1 are \( \hat{x}_d(t) \)s.

18. 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 [110]. 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 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 tweet follows Longtail distribution [18]. 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 [41], 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),
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-US 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.

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 cost.

---

Google storage and data transfer pricing. [https://cloud.google.com/storage/pricing](https://cloud.google.com/storage/pricing)
4.4 Performance Evaluation

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, ac-
cording 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, Aus-
tralia 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 Ama-
zon 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 pol-
cy 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 paper,
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, dura-
bility, 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 reputa-
tion 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

<table>
<thead>
<tr>
<th>Data size factor</th>
<th>Latency cost weight</th>
<th>Read to write ratio</th>
<th>Access pattern on objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>1</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>Range</td>
<td>0.2-1</td>
<td>1-30</td>
<td>1-30</td>
</tr>
</tbody>
</table>

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-US 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.

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

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.

In 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
4.4 Performance Evaluation

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 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

\textsuperscript{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).
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.
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.
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 \textsc{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
4.4 Performance Evaluation

![Graphs showing cost saving comparison between OOP and NOOP algorithms for different data centers and read to write ratios.](image)

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 [18], as the default). Although Normal and Random access patterns are not compatible with hot- and cold-spot status definition for objects, we investigate the i-
Table 4.3: Cost saving of OOP and NOOP (shown in bracket), and the potential DCs pairing with four home DCs when the access patterns on the objects are Normal and Random.

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

The impact 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 \(\gamma\) 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 \(1 + \frac{2}{w}\) 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

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) [129]. 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
5.1 Introduction

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

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

*The price of Put and Get is multiplication 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 [95], storage vs. cache [26, 139], and cost optimization of data dispersion across cloud providers [185, 187]. 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 $\gamma$ 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 [41] using the synthesized workload based on Facebook workload [18].

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 [57] 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 *Gets* and *Puts*. 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

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

5.2.2 Preliminaries

In this section, we give some definitions, which are used throughout the paper. 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 \(r^G(t)\) 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 $\mathbf{\alpha}(t)$ 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 $\beta_k^d$, $d \in (0, 1)$. Thus, $\sum_{k \in K} \sum_{d \mid \alpha^d(t) = 1} \beta_k^d(t) = 1$. We denote $\mathbf{\beta}(t)$ 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

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

**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,

$$\sum_{k \in K} \sum_{d \mid \alpha^d(t) = 1} \beta_k^d \times r^k(t) \times (t_g(d) + v(t) \times O(d)).$$

To keep replicas consistent, we use a simple policy that leverages the primary advantages of eventual consistency setting, which is appropriate for OSNs [185]. Thus, first, to capitalize on the network services cost, we select DC $d \mid \alpha^d(t) = 1$ with the minimum network cost $O(d)$ so that the upper bound of delay for Put requests is met¹. 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, $d_p = \min_{d' \mid \alpha^{d'}(t) = 1} (O(d'))$ (called consistency cost).

¹From 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,

$$c(d, d_p) + \sum_{k \in K} [(w_k(t) \times t_p(d) + \sum_{d' \mid w_k(t) = 1 \setminus \{d, d_p\}} w_k(t) \times (t_p(d') + v(t) \times O(d_p)))]$$

(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 $C_R(\tilde{\alpha}(t), \tilde{\beta}(t))$.

The best set of DCs to replicate an object can differ in $t$ and $t - 1$. In other words, $\tilde{\alpha}(t - 1)$ and $\tilde{\alpha}(t)$ are different. This happens because the object size, the number of requests, and the source of requests to conduct Gets or Puts 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
imposes a \textit{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 $\vec{a}(t) \neq \vec{a}(t-1)$, 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 $C_M(\vec{a}(t-1), \vec{a}(t))$ includes the migration cost from the DC $d_s = \min_{d | \vec{a}(t-1) = 1} O(d)$ to $d_d = \min_{d | \vec{a}(t) = 1} O(d)$ 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 $\rho$ as the number of DCs in destination set, excluding
the intersection of the source and destination sets. Thus,

$$C_M(\vec{\alpha}(t-1), \vec{\alpha}(t)) = v(t) \times (O(d_s) + O(d_{pm}) \times (\rho - 1)).$$  \hspace{1cm} (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 [71]. This technique is used by the single cloud system such as HBase\(^2\) and ElasTraS [62], and in Geo-replicated system [176]. 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 [57]) 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:

$$C(\vec{\alpha}(t), \vec{\beta}(t)) = C_R(.) + C_M(.)$$  \hspace{1cm} (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 [185] and [183], 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 \(d\)

\(^3\)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 referred to [71] and [64].
\(^4\)Note that our system is not designed to support database transactions, and this technique just inspired from this area.
5.2 System Model and Problem Definition

DCs Specifications
1. Cloud storage services price
2. Inter-DC latency

Application Requirements
1. Latency SLO
2. Replicas number
3. Application workload per object

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 [57]. 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 $\hat{\alpha}(t)$ and $\overline{\beta}(t)$ 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:

$$
\min_{\hat{\alpha}(t),\overline{\beta}(t)} \sum_{t=1}^{T} C(\hat{\alpha}(t),\overline{\beta}(t))
$$

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

(a) $\sum_{d \in D} \alpha^d(t) = r, \quad \alpha^d(t) \in \{0,1\}$
(b) $\sum_{k \in K} \sum_{d|\alpha^d(t)=1} \beta^k,d(t) = 1, \quad \beta^k,d(t) \in (0,1)$
(c) $\beta^k,d(t) \leq \alpha^d(t)$
(d) $\sum_{k \in K} \sum_{d \in D} \alpha^d(t) \times r^k \times l(k,d) \leq L$
(e) $l(k,d') \leq L, d' = \min_{d|\alpha^d(t)=1} O(d)$ 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. 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 $\vec{a} = \{\vec{a}^1, \vec{a}^2, ..., \vec{a}^i, ..., \vec{a}^{(|D|)}\}$ denote all $r$-combinations of distinct DCs can be chosen from $D$ (i.e, $|\vec{a}| = \binom{|D|}{r}$). Suppose that the key function of the dynamic algorithm is $P(\vec{a}(t))$ that indicates the minimum cost in time slot $t$ if the object is replicated at a set of DCs that is represented by $\vec{a}(t)$.

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

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

\[ P(\vec{a}(t)) = \min P(\vec{a}(t), \vec{\beta}(t)) + \sum_{\vec{a}(t-1)} P(\vec{a}(t-1)) \]
5.3 Optimal Offline Algorithm

Data centers

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>\ldots</th>
<th>t-1</th>
<th>t</th>
<th>\ldots</th>
<th>T-1</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>\vec{\alpha}_1</td>
<td>\vec{\alpha}_2</td>
<td>\ldots</td>
<td>\vec{\alpha}_{t-1}</td>
<td>\vec{\alpha}_t</td>
<td>\ldots</td>
<td>\vec{\alpha}_{T-1}</td>
<td>\vec{\alpha}_T</td>
</tr>
</tbody>
</table>

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

\[
P(\vec{\alpha}(t)) = \begin{cases} 
\min_{\vec{\alpha}(t-1) \in \vec{\alpha}} [P(\vec{\alpha}(t-1)) + C(\vec{\alpha}(t), \vec{\beta}(t))], & t > 0 \\
0 & t = 0 
\end{cases}
\] (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 \( \min_{\vec{\alpha}(t) \in \vec{\alpha}} P(\vec{\alpha}(t)) \) 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 \( |\alpha|=(\binom{|D|}{r}) \) is a small constant because \( r \) is at most 2 or 3 [44] and the number of DCs in the leading commercial cloud providers is 8, for example Amazon and Google. Thus, the value of \( |\alpha| \) 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 \( O(|D|^2 + |\vec{\alpha}|^2 TT_{lp}) = O(|\vec{\alpha}|^2 TT_{lp}) \). This time complexity increases when the value of \( \alpha \) (depending to \( |D| \) and \( r \)) grows. Note that with a limitation on \( |D| \) and an upper bound of 3 for \( r \) [44], 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 $t \in [1...T]$

1. $\vec{\alpha} \leftarrow$ Calculate all $r_-$-combinations of distinct DCs from $D$.

2. Initialize: $\forall \vec{\alpha}(0) \in \vec{\alpha}$, $P(\vec{\alpha}(0)) = 0$

3. for $t \leftarrow 1$ to $T$ do

4. for all $\vec{\alpha}(t) \in \vec{\alpha}$ do

5. for all $\vec{\alpha}(t-1) \in \vec{\alpha}$ do

6. Calculate $P(\vec{\alpha}(t))$ based on Equ. (5.7).

7. end

8. end

9. end

10. 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 $\vec{\beta}^*(t)$.

11. 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 $\forall I$, $C_{\text{DOA}}(I) / C_{\text{OPT}}(I) \leq c$, where $C_{\text{DOA}}(I)$ is the total cost for input $I$ (i.e., workload in our work) by DOA, and $C_{\text{OPT}}(I)$ is the optimal cost to serve input $I$ by optimal offline algorithm OPT. Similarly, a randomized online algorithm ROA is c-competitive iff $\forall I$, $E[C_{\text{ROA}}(I)] / C_{\text{OPT}}(I) \leq c$. 
5.4 Online Algorithms

5.4.1 The Deterministic Online Algorithm

We propose an online algorithm based on the total cost $C(\alpha(t), \beta(t))$ 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 $C_M(\alpha(t_{m-1}), \alpha(t_m))$. 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 $C_R(\alpha(v-1), \beta(v))$ (see Fig. 5.4a), 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 $C_R(\alpha(v), \beta(v))$ (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 $\sum_{v=t_m}^{t-1} [C_R(\alpha(v-1), \beta(v)) - C_R(\alpha(v), \beta(v))]$. Based on

---

6In this figure, without loss of generality, we consider only one DC that hosts an object (i.e., $r = 1$).
Cost Optimization across Cloud Storage Providers: Offline and Online Algorithms

\[ v = t_m \]

DC name: A \[ \rightarrow \text{Time} \]

Residential cost in t: \[ C_R(\tilde{a}(v-1), \tilde{\beta}(v)) \]

(a)

DC name: A \[ \rightarrow \text{Time} \]

Residential cost in t: \[ C_R(\tilde{a}(v), \tilde{\beta}(v)) \]

(b)

DC name: A \[ \rightarrow \text{Time} \]

Migration cost: \[ C_M(\tilde{a}(t_m-1), \tilde{a}(t_m)) \]

(c)

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 \( t_m \).

Based on the above discussion, Algorithm 5.2 formulates the details of the determin-

\[ C_M(\tilde{a}(t_{m-1}), \tilde{a}(t_m)) \leq \sum_{v=t_m-1}^{t_m-1} [C_R(\tilde{a}(v-1), \tilde{\beta}(v)) - C_R(\tilde{a}(v), \tilde{\beta}(v))] \]  (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:

\[ C_M(\tilde{a}(t_{m}), \tilde{a}(t)) + C_R(\tilde{a}(t), \tilde{\beta}(t)) \leq C_R(\tilde{a}(t - 1), \tilde{\beta}(t)) \]  (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 \( t - 1 \).
istic 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 \(C_R(\vec{a}(t), \vec{\beta}(t))\) 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), \(\vec{a}(t)\) and \(\vec{\beta}(t)\) are calculated for all \(\vec{a}(t) \in \vec{a}\) so that the residential cost \(C_R(\vec{a}(t), \vec{\beta}(t))\) 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., \(\vec{a}(t) = \vec{a}(t - 1)\) (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 \(\gamma\) times of the minimum residential cost in this time period. \(\gamma\) 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 \(\sum_{t_{m-1}}^{t_{m}} C_M(\vec{a}(t_{m-1}), \vec{a}(t_m))\)
\[= \sum_{v=t_{m-1}}^{t_{m}} (C_R(\vec{a}(v - 1), \vec{\beta}(v)) - C_R(\vec{a}(v), \vec{\beta}(v)))\]
\[\leq \sum_{v=t_{m-1}}^{t_{m}} (C_R^{\text{max}}(\vec{a}(v - 1), \vec{\beta}(v)) - C_R^{\text{min}}(\vec{a}(v), \vec{\beta}(v)))\).

Let \(\gamma = C_R^{\text{max}}(\vec{a}(v - 1), \vec{\beta}(v))/C_R^{\text{min}}(\vec{a}(v), \vec{\beta}(v))\) for all \(v \in [1..t]\). Substituting the value of \(\gamma\) in the above equation, we have,
\[\sum_{t_{m-1}}^{t_{m}} C_M(\vec{a}(t_{m-1}), \vec{a}(t_{m})) \leq (1 - 1/\gamma) \sum_{v \in [1..t]} C_R^{\text{max}}(\vec{a}(v), \vec{\beta}(v)) \]

**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, \(C_{DOA} = \sum_{t=1}^{T} C_R(\vec{a}(t), \vec{\beta}(t)) + C_M(\vec{a}(t - 1), \vec{a}(t))\). Since the upper bound of the residential cost for DOA is \(\gamma\) times the cost of the offline algorithm, and according
140 Cost Optimization across Cloud Storage Providers: Offline and Online Algorithms

To the result of Lemma (5.1) we have: $C_{DOA} = (1 - 1/\gamma) \sum_{t=1}^{T} C_{R}^{\text{max}}(\vec{a}(t), \vec{b}(t)) + \sum_{t=1}^{T} C_{R}(\vec{a}(t), \vec{b}(t))$

$\leq (1 - 1/\gamma) \gamma C_{OPT} + \gamma C_{OPT} = (\gamma - 1) C_{OPT} + \gamma C_{OPT}$

$\leq (2\gamma - 1) C_{OPT}$

The value of $\gamma$ 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 $\gamma = \max_{d \neq d'} O(d)/O(d')$. Otherwise, if object is storage intensive (i.e., it is in cold-spot), then $\gamma = \max_{d \neq d'} S(d)/S(d')$. Generally, if the volume of the object to be read is $\lambda$ times of the object size, then $\gamma = \max_{d \neq d'} (S(d) + \lambda O(d))/(S(d') + \lambda O(d'))$.

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, $\vec{a}(t)$ and $\vec{b}(t)$ should be calculated for all $\vec{a}(t) \in \vec{a}$ by using linear programming, which takes $O(|\vec{a}| T_{lp})$. This calculation is done for $T$ time slots. Therefore, the algorithm yields a running time of $O(|D|^2 + |\vec{a}| T T_{lp}) = O(|\vec{a}| T T_{lp})$. 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) [114]. RHC is a classical control policy that is used for dynamic capacity provisioning in a DC [121] [114], load balancing on a DC [115], and moving data into a DC [199].

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: Deterministic Online Algorithm (DOA)

Input: RPM’s inputs as illustrated in Fig. 5.2

Output: \( \tilde{\alpha}(t), \tilde{\beta}(t) \), and the overall cost denoted \( C_{ove} \)

1. \( C_{ove} \leftarrow 0 \)
2. \( \tilde{\alpha} \leftarrow \) Calculate all \( r \)-combinations of distinct DCs from \( D \).
3. \( C_{ove} \leftarrow \) Determine \( \tilde{\alpha}(t) \) and \( \tilde{\beta}(t) \) by minimizing \( C_R(\tilde{\alpha}(t), \tilde{\beta}(t)) \) for all \( \tilde{\alpha}(t) \in \tilde{\alpha} \) in time slot \( t = 1 \).

4. \( t_m \leftarrow 1 \)
5. for \( t \leftarrow 2 \) to \( T \) do
6.   forall \( \tilde{\alpha}(t) \in \tilde{\alpha} \) do
7.     \( C_R(.) \leftarrow \) Determine \( \tilde{\alpha}(t) \) and \( \tilde{\beta}(t) \) by minimizing \( C_R(\tilde{\alpha}(t), \tilde{\beta}(t)) \)
8.     \( C_{ove} \leftarrow C_{ove} + C_R(.) \)
9.   end
10. if (Equ. (5.8) and (5.9), and \( \tilde{\alpha}(t-1)! = \tilde{\alpha}(t) \)) then
11.   \( t_m \leftarrow t \)
12.   \( C_M(.) \leftarrow \) calculate \( C_M(\tilde{\alpha}(t_{m-1}), \tilde{\alpha}(t_m)) \)
13.   \( C_{ove} \leftarrow C_{ove} + C_M(.) \)
14. else
15.   \( \tilde{\alpha}(t) \leftarrow \tilde{\alpha}(t-1) \)
16. end
17. end
18. Return \( \tilde{\alpha}(t), \tilde{\beta}(t) \) and \( C_{ove} \).

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., Eq. (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 Eq. (5.10), the optimal cost of object for each frame by using randomized FRHC with value \( l \) is:

\[
C(\vec{\alpha}_l(t), \vec{\beta}_l(t)) = \sum_{t_s}^{t_s+w} C_R(\vec{\alpha}_l(t), \vec{\beta}_l(t)) + \sum_{t_s}^{t_s+w} C_M(\vec{\alpha}_l(t-1), \vec{\alpha}_l(t)),
\]

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 \( \sum_{t_s}^{t_s+w} C_M(\vec{\alpha}^*(t-1), \vec{\alpha}^*(t)) + \sum_{t_s}^{t_s+w} C_R(\vec{\alpha}^*(t), \vec{\beta}^*(t)) \).

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: 
\[
C_M(\vec{a}_I(t_s - 1), \vec{a}^*(t_s - 1)).
\]
(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 
\[
C_M(\vec{a}^*(t_s - 1), \vec{a}^*(t_s))).
\]
We therefore can bound the cost in the time frame as follows:

\[
C(\vec{a}_I(t), \vec{b}_I(t)) \leq (i) + (ii) + \sum_{t=t_1}^{t_s+1} C_M(\vec{a}^*(t-1), \vec{a}^*(t)) + \sum_{t=t_1}^{t_s+1} C_R(\vec{a}^*(t), \vec{b}^*(t))
\leq (i) + \sum_{t=t_1}^{t_s+1} C_M(\vec{a}^*(t-1), \vec{a}^*(t)) + \sum_{t=t_1}^{t_s+1} C_R(\vec{a}^*(t), \vec{b}^*(t))
\leq (i) + \sum_{t=t_1}^{t_s+1} C(\vec{a}^*(t), \vec{b}^*(t)).
\]

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. \( \square \)

**Theorem 5.2.** Algorithm 5.3 is \((1 + \frac{2}{w})\)-competitive. Formally, for any input \( C_{ROA}/C_{OPT} \leq (1 + \frac{2}{w}) \).

**Proof.** By using Lemma (2), the upper bound of the total cost incurred by the randomized FRHC is 
\[
C_{ROA} = \sum_{t_1 \in [T/w]} \left[ C_M(\vec{a}_I(t_s - 1), \vec{a}^*(t_s - 1)) + \sum_{t=t_s+1}^{t_s+w} C(\vec{a}^*(t), \vec{b}^*(t)) \right]
\leq \sum_{t_1 \in [T/w]} \left[ C_M(\vec{a}_I(t_s - 1), \vec{a}^*(t_s - 1)) + \sum_{t=t_s+1}^{t_s+w} C(\vec{a}^*(t), \vec{b}^*(t)) \right]
\leq C_{OPT} + \frac{1}{w} \sum_{t_1 = 1}^{w} \sum_{t \in [T/w]} C_M(\vec{a}_I(t_s - 1), \vec{a}^*(t)).
\]

The expected cost of Randomized is computed as:

\[
E(C_{ROA}) = \frac{1}{w} \sum_{l=1}^{w} (C_{OPT} + \sum_{t_1 \in [T/w]} C_M(\vec{a}_I(t_s - 1), \vec{a}^*(t)))
\leq C_{OPT} + \frac{1}{w} \sum_{l=1}^{w} \sum_{t \in [T/w]} C_M(\vec{a}_I(t_s - 1), \vec{a}^*(t - 1)).
\]

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**: $\bar{\alpha}(t), \bar{\beta}(t)$, and the overall cost denoted $C_{\text{ave}}$

1. $l \leftarrow$ random number within $[1, w]$; $C_{\text{ave}} \leftarrow 0$
2. if $l \neq 1$ then
   3. $C_{\text{ave}} \leftarrow$ solve Equ. (5.10) over widows $[1, l]$ and $[T - l]$
   4. $t_m = l + 1$
   5. end
6. for $t \leftarrow l$ to $T - l + 1$ do
   7. $C_{\text{ave}} \leftarrow C_{\text{ave}} +$ solve Equ. (5.10) over widows $[l, l + w]$
   8. $C_M \leftarrow$ solve Equ. (5.4) for $(t_{m-1}, t_m)$
   9. $C_{\text{ave}} \leftarrow C_{\text{ave}} + C_M, t_m = l + w + 1$
   10. $t \leftarrow t + w$
   11. end
12. if $l \neq 1$ then
   13. $C_M \leftarrow$ solve Equ. (5.4) for $(t_m - 1, t_m)$
   14. $C_{\text{ave}} \leftarrow C_{\text{ave}} + C_M$
   15. end
16. Return $\bar{\alpha}(t), \bar{\beta}(t)$ and $C_{\text{ave}}$

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

$$\leq 1 + \frac{1}{w} \left( \frac{\sum_{t_s \in \lceil T/w \rceil} C_{\text{max}} M(\bar{\alpha}_l(t_s - 1), \bar{\alpha}^*(t_s - 1))}{C_{\text{OPT}}} \right).$$

Based on the definition of $\gamma$ in Lemma 1, we have:

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

Since the coefficient of $\frac{\gamma}{w}$ is less or equal to one, we have $E(C_{\text{ROA}})/C_{\text{OPT}} \leq 1 + \frac{\gamma}{w}$. \hfill $\square$

Based on computed $\gamma$ 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 $O(|D|^2 + w|\bar{\alpha}|^2 T_{lp}) = O(w|\bar{\alpha}|^2 T_{lp})$ 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 [18]. 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 [41], 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 [24]. 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 [24].

Workload on objects: It comes from the Facebook workload [18] 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\(^8\) [185], and (iii) the pattern for Get rate to retrieve items follows long-tail distribution such that 3/4 of those Gets happen during 1/8 of the initial lifetime of the bucket [24]. 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 $RRT(ms) = 5 + 0.02 \times Distance(km)$ [142]. 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 [103].

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 [44]. 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.

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.
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)

<table>
<thead>
<tr>
<th>Replicas Number</th>
<th>Object Size</th>
<th>Optimal</th>
<th>DOA †</th>
<th>ROA ‡</th>
<th>Optimal</th>
<th>DOA</th>
<th>ROA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 KB</td>
<td>0.9030</td>
<td>0.9694</td>
<td>0.9469</td>
<td>0.8778</td>
<td>0.9075</td>
<td>0.8974</td>
</tr>
<tr>
<td></td>
<td>100 KB</td>
<td>0.8561</td>
<td>0.8734</td>
<td>0.8657</td>
<td>0.7758</td>
<td>0.8369</td>
<td>0.7997</td>
</tr>
<tr>
<td>r=2</td>
<td>1 KB</td>
<td>0.8879</td>
<td>0.9330</td>
<td>0.9181</td>
<td>0.8787</td>
<td>0.9045</td>
<td>0.8866</td>
</tr>
<tr>
<td></td>
<td>100 KB</td>
<td>0.8831</td>
<td>0.9045</td>
<td>0.9127</td>
<td>0.8440</td>
<td>0.8625</td>
<td>0.8636</td>
</tr>
</tbody>
</table>

† 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 Random-
ized 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 $\gamma$. 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 $\gamma = 1.52$. Thus, by Theorem 5.1 and the value of $\gamma$, the deterministic algorithm will lead to at most 2.04 times the optimal offline cost. And, by Theorem 5.2 and the value of $\gamma$, 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.
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.
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.

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, except 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
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)

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>$r = 1$</th>
<th>$r = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>0.750</td>
<td>11.35</td>
</tr>
<tr>
<td>Deterministic</td>
<td>0.012</td>
<td>2.01</td>
</tr>
<tr>
<td>Randomized</td>
<td>0.368</td>
<td>8.45</td>
</tr>
</tbody>
</table>

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 even-
tual 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 $\gamma$ 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 look-ahead 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 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 [57] 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) [129]. 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 suffer 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 \textit{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 [110] in CloudSim simulator [41].
- 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 Geo-graphically 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) [176]. The unit of data migration is the bucket abstraction which is the same as that in Spanner [57]. The bucket consists of the objects owned by a specific user.

In the system model, a DC is refereed 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

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

### 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) $t_d(d)$ 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 $d_c(t) \rightarrow d_r(t)$, 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
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

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

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

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

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

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

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

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

\[ (6.4) \]
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

\[
C_M^{d_r}(t - 1, t) = \begin{cases} 
0 & \text{if } d_r \in D_r(t - 1) \land D_r(t) \\
\min_{d_r \in D_r(t - 1)} O(d_r) \times v(t - 1) & \text{otherwise.}
\end{cases}
\]  

(6.5)

As seen in Eq. (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 \( t_m^d \) denotes the last time of migration for the replica object in DC \( d_r \). Also suppose \( C_M^{d_r}(t_m, t) \) is the summation of benefit-cost that is lost for the replica in DC \( d_r \) during period \([t_m^d, t] \). 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,
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 $t_m^{\ast}$ 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.,

```plaintext
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: $D_r^t(t)$: 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. $C_B^d(t) \leftarrow [C_R(d'_r(t-1), d_c(t) \rightarrow d'_r(t-1)) - C(d_r, t)]$

      7. /*Migration happens*/

      8. if $C_L^d(t_m, t-1) + C_B^d(t) \geq C_M^d(t-1, t)$ then

         9. $D_r^t(t) \leftarrow D_r^t(t) \cup d'_r$

         10. $D_r(t-1) \leftarrow D_r(t-1) - \{d'_r\}$

         11. $t_m^d \leftarrow 0, t_m^{\ast} \leftarrow t$

         12. break.

   end

end

/*Migration does not happen*/

if $d_r \notin D_r^t(t)$ then

14. Find $\min_{d'_r} C(d'_r, t)$

15. $D_r^t(t) \leftarrow D_r^t(t) \cup d'_r$

16. $D_r(t-1) \leftarrow D_r(t-1) - \{d'_r\}$

17. $C_L^d(t_m, t) \leftarrow C_L^d(t_m, t-1) + [C_R(d'_r(t-1), d_c(t) \rightarrow d'_r(t-1)) - C(d_r, t)]$

end
```


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 \( C_B^d(t) \) 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, \( C_L^d(t_m, t) \) 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 be 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 [71]. This technique is used by the single cloud system such as HBase\(^1\) and ElasTraS [62], and in Geo-replicated system [176]. 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 [57]) 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,

---


\(^2\) 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 refereed to [71] and [64].
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:

\[
C_R(d_r(t), d_c(t) \rightarrow d_r(t)) + C_M^{d_c(t),d_r(t)}(t-1,t),
\]

where \(C_R(.)\) is the summation of the replica creation, storage, Put, and Get costs (Equs. 6.1-6.4), and is refereed 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.
\[ \min_{d_c(t) \to d_r(t)} \sum_t C_R(.) + C_M(.) \quad (6.7) \]

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

(a) \( \sum_{d_c} d_c(t) \to d_r(t) = 1 \), \quad \forall d_r

(b) \( \sum_{d_c} d_c(t) \to d_r(t) = |D_c(t)| \), \quad \forall d_r

(c) \( \sum_{d_c} \frac{r_c \times l(d_c,d_r)}{\sum_{d_r} r_c} \leq L \), \quad \forall d_c,d_r

(d) \( \sum_{d_c} d_c(t) \to d_r(t) = |D_r(t)| \), \quad \forall d_c, Puts

(e) \( l(d_c(t),d_c \to d_r(t)) \leq L \) \quad \forall d_c \in D_c, 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) \( \sum_{d \in D} \alpha^{d}(t) \geq 1 \), \quad \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 \( \alpha^d(t) \) and \( d_c(t) \rightarrow d_r(t) \), the same as that discussed in Section 5.3. The value of \( d_c(t) \rightarrow d_r(t) \) can be determined via linear programming once the value of \( \alpha^d(t) \) is fixed. To enumerate value of all \( \alpha^d(t) \), we need to enumerate all \( r \)–combinations of a given set of DCs (i.e., \(|D|\)). Since the value of \( r \) can be ranged between 1 and \(|D|\), the number of combination of \( \alpha^d(t) \) is \( \sum_{r=1}^{|D|} \binom{|D|}{r} = 2^{|D|} - 1 \).

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 \( \alpha^d(t) \) (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 \( O((2^{|D|} - 1)^2 TT_{lp}) \), where \( T_{lp} \) is the required time to solve the linear programming for finding the value of \( d_c(t) \rightarrow d_r(t) \). 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 [53]. 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

\[ \text{Note that neither Algorithm 6.1 nor Algorithm 6.2 are used in optimal solution since this solution enumerates all values of } r \text{ as well as all combinations of object migration between } d_r \in D_r(t-1) \text{ and } d_r \in D_r(t) \]
The RPCLV algorithm first assigns client DCs to each potential DC $d_p \in D$ 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., $D^d_{dp}$)–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**: $D_r(t-1), D_r(t), \text{and } T.$

**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 $D_{cr}^d$ 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 $t \leftarrow 2 \text{ to } T$ do
6: Call the RPCLV algorithm and determine $D_r(t)$ as well as $D_{cr}^d$ 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 $\mid(D_r - \lfloor r \rfloor) \mid \times \min_{d_r \in D_r} O(d_r)$ 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Lines</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 6.1</td>
<td>O(</td>
<td>D</td>
<td>^2)</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>D</td>
<td>^log</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>D</td>
<td>)</td>
</tr>
<tr>
<td>Algorithm 6.2</td>
<td>O(</td>
<td>D</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>D</td>
<td>^2)</td>
</tr>
<tr>
<td>Algorithm 6.3</td>
<td>O(</td>
<td>D</td>
<td>^2)</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>D</td>
<td>^2)</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>D</td>
<td>)</td>
</tr>
</tbody>
</table>

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 [41] and the Twitter workload [110]. 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.

*Objects workload*: We use Twitter traces [110] 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 [24]. 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, 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.

---


6The Google maps geocoding API. [https://developers.google.com/maps/documentation/geocoding/intro](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>
<thead>
<tr>
<th>Policy</th>
<th>Quantile Volume</th>
<th>Latency (ms)</th>
<th>Read to Write Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Closest-based</td>
<td>0.2,1</td>
<td>100, 250</td>
</tr>
<tr>
<td>Range</td>
<td>Closest-, network-, and Storage-based</td>
<td>0.2-1</td>
<td>50 -250</td>
</tr>
</tbody>
</table>

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
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

<table>
<thead>
<tr>
<th>Quantile volume</th>
<th>Policy</th>
<th>Latency=100 ms R/W=1</th>
<th>R/W=30</th>
<th>Latency=250 ms R/W=1</th>
<th>R/W=30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>44.78%</td>
<td>13.61%</td>
<td>49.21%</td>
<td>17.12%</td>
</tr>
<tr>
<td>0.2</td>
<td>Closest</td>
<td>50.82%</td>
<td>18.78%</td>
<td>54.31%</td>
<td>22.32%</td>
</tr>
<tr>
<td></td>
<td>Network-based</td>
<td>46.00%</td>
<td>13.98%</td>
<td>50.48%</td>
<td>17.51%</td>
</tr>
<tr>
<td></td>
<td>Storage-based</td>
<td>38.00%</td>
<td>22.79%</td>
<td>42.05%</td>
<td>26.80%</td>
</tr>
<tr>
<td></td>
<td>Closest</td>
<td>40.11%</td>
<td>25.43%</td>
<td>44.06%</td>
<td>29.89%</td>
</tr>
<tr>
<td>1</td>
<td>Network-based</td>
<td>39.91%</td>
<td>23.88%</td>
<td>44.08%</td>
<td>27.96%</td>
</tr>
</tbody>
</table>

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 (W/R=1) higher than read-intensive objects (W/R=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 50% of read- and write-intensive objects. Likewise, results remain the same, or even more improvements are experienced, for objects under loose latency (see
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
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 and Microsoft Azure 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](http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html)

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/Object**: 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.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Input Parameters</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazonCreateS3Client</td>
<td>accessKey,secretKey</td>
<td>AmazonS3Client</td>
</tr>
<tr>
<td>createBucket</td>
<td>client,bucketName</td>
<td>Bucket</td>
</tr>
<tr>
<td>amazonCreateFolder</td>
<td>client,bucketName,folderName,storageClass</td>
<td>Folder</td>
</tr>
<tr>
<td>amazonUploadObject</td>
<td>client,bucketName,folderName,objectName, path,storageClass</td>
<td>Put Object</td>
</tr>
<tr>
<td>amazonDownloadObject</td>
<td>client,bucketName,folderName,desPath</td>
<td>Get object(s)</td>
</tr>
<tr>
<td>amazonDeleteFolder/Objects</td>
<td>client,bucketName,folderName</td>
<td>Delete folder/objects</td>
</tr>
<tr>
<td>amazonListObjects</td>
<td>client,bucketName,folderName</td>
<td>Objects list</td>
</tr>
<tr>
<td>amazonChangeClassStorageFolder</td>
<td>client,bucketName,folderName,storageClass</td>
<td>Change storage class</td>
</tr>
<tr>
<td>amazonTransferFolder</td>
<td>srcClient,srcBucketName,srcFoldername,desClient, desBucketName</td>
<td>Transfer objects</td>
</tr>
<tr>
<td>amazonToAzureTransferFolder</td>
<td>srcClient,bucketName,srcFolderName,desClient, containerName,desFolderName</td>
<td>Transfer objects</td>
</tr>
</tbody>
</table>

Table 6.6: The input parameters used in Modules of AWS.

<table>
<thead>
<tr>
<th>Input Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessKey</td>
<td>String</td>
<td>This key is uniquely assigned to the owner of AWS S3 account.</td>
</tr>
<tr>
<td>secretKey</td>
<td>String</td>
<td>This key is the password for the owner of AWS S3 account.</td>
</tr>
<tr>
<td>client</td>
<td>AmazonS3Client</td>
<td>This parameter allows users to invoke the service methods on AWS S3.</td>
</tr>
<tr>
<td>bucketName</td>
<td>String</td>
<td>This refers to the name of the bucket that contains folders.</td>
</tr>
<tr>
<td>folderName</td>
<td>String</td>
<td>This refers to the name of the folder containing objects.</td>
</tr>
<tr>
<td>storageClass</td>
<td>String</td>
<td>This specifies constants that include four storage classes of AWS S3.</td>
</tr>
<tr>
<td>objectName</td>
<td>String</td>
<td>This parameter specifies the name of object generated by users.</td>
</tr>
<tr>
<td>srcPath</td>
<td>String</td>
<td>This represents the path from which the data can be transferred.</td>
</tr>
<tr>
<td>desPath</td>
<td>String</td>
<td>This indicates the path to which the data can be transferred.</td>
</tr>
</tbody>
</table>

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 \((src/des)Client, (src/des)BucketName, and (src/des)FolderName\) parameters are respectively the same with the \(client, bucketName, and folderName\) parameters in the type and descriptions. The \(srcClient, srcBucketName, and srcFolderName\) parameters represent that from which client, bucket and folder data are transferred. The \(desClient, desBucketName, and desFolderName\) 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.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Input Parameter(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>azureCreateCloudBlobClient</td>
<td>accessKey, client, containerName</td>
<td>CloudBlobClient</td>
</tr>
<tr>
<td>createContainer</td>
<td>client, containerName</td>
<td>Container</td>
</tr>
<tr>
<td>azureCreateFolder</td>
<td>client, containerName, folderName</td>
<td>Folder</td>
</tr>
<tr>
<td>azureUploadObject</td>
<td>client, containerName, folderName, objectName, path</td>
<td>put Object</td>
</tr>
<tr>
<td>azureDownloadObject</td>
<td>client, containerName, folderName, desPath</td>
<td>Get object(s)</td>
</tr>
<tr>
<td>azureDeleteFolder/Objects</td>
<td>client, containerName, folderName</td>
<td>Delete folder/objects</td>
</tr>
<tr>
<td>azureListObjects</td>
<td>client, containerName</td>
<td>Objects list</td>
</tr>
<tr>
<td>azureTransferFolder</td>
<td>srcClient, srcContainerName, srcFolderName, desClient, desContainerName, desFolderName</td>
<td>Transfer object(s)</td>
</tr>
<tr>
<td>azureToAmazonTransferFolder</td>
<td>srcClient, srcContainerName, srcFolderName, desClient, bucketName, desFolderName</td>
<td>Transfer object(s)</td>
</tr>
</tbody>
</table>

The modules of Microsoft Azure require input parameters which are similar to those of AWS data stores to the large extent. 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.

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.
- azureToAmazonTransferFolder: This allows users to transfer object from Azure data stores to Amazon data stores.
- azureCreateFolder, azureUploadObject, azureDownloadObject, azureDeleteFolder/Objects, 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.

<table>
<thead>
<tr>
<th>Input Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessKey</td>
<td>String</td>
<td>This key is used to authenticate when the storage account is accessed.</td>
</tr>
<tr>
<td>client</td>
<td>cloudBlobClient</td>
<td>This parameter allows users to invoke the service methods on blob storage.</td>
</tr>
<tr>
<td>containerName</td>
<td>String</td>
<td>This refers to the name of the container that contains the folders.</td>
</tr>
</tbody>
</table>

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 [92] 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 [57]) 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-
6.6 Conclusions

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


\(^2\)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 [41] 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
7.1 Summary of Contributions

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 [41] using the real-world traces from Twitter [110]. 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 [53]. 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 [110] 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 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.

3Amazon S3 REST API http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.
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 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 2.2. 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 [8].

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 the cost model to support failover of DC(s).

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\textsuperscript{5} and Microsoft Azure\textsuperscript{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.

\textsuperscript{5}Amazon RDS product. https://aws.amazon.com/rds/details/

\textsuperscript{6}SQL database https://azure.microsoft.com/en-us/services/sql-database/
Bibliography


[16] 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 con-


