SOCA (2017) 11:411–426
https://doi.org/10.1007/s11761-017-0218-9

SPECIAL ISSUE PAPER

Cost-optimized redundant data storage in the cloud

Philipp Waibel[1] · Johannes Matt[1] · Christoph Hochreiner[1] · Olena Skarlat[1] · Ronny Hans[2] · Stefan Schulte[1]

Received: 15 March 2017 / Revised: 19 July 2017 / Accepted: 28 August 2017 / Published online: 12 September 2017

Abstract The use of cloud-based storage systems for storing data is a popular alternative to local storage systems. Beside several benefits of cloud-based storages, there are also downsides like vendor lock-in or unavailability. Moreover, the selection of the best fitting storage solution can be a tedious and cumbersome task and the storage requirements may change over time. In this paper, we formulate a system model that uses multiple cloud-based services to realize a redundant and cost-efficient storage. Within this system model, we formulate a local and a global optimization problem that considers historical data access information and predefined quality of service requirements to select a cost-efficient storage solution. Furthermore, we present a heuristic optimization approach for the global optimization. Extensive evaluations show the benefits of our work in comparison with a baseline that follows a state-of-the-art approach. We show that our solutions save up to 30% of the cumulative cost in comparison with the baseline.

✉ Philipp Waibel
p.waibel@dsg.tuwien.ac.at

Johannes Matt
johannes.matt@aon.at

Christoph Hochreiner
c.hochreiner@dsg.tuwien.ac.at

Olena Skarlat
o.skarlat@dsg.tuwien.ac.at

Ronny Hans
ronny.hans@kom.tu-darmstadt.de

Stefan Schulte
s.schulte@dsg.tuwien.ac.at

1 Argentinierstrasse 8/184-1, 1040 Vienna, Austria

2 Rundeturmstrasse 10, 64283 Darmstadt, Germany

Keywords Cloud storage · Redundant storage · Erasure coding · Vendor lock-in · Long-term storage · Global optimization · MILP

1 Introduction

The use of cloud storage services to store data is a popular alternative to traditional local storage systems, e.g., storage on a local infrastructure [20]. Companies, government organizations, and even private persons use cloud storages as an alternative to maintaining their own storage systems [4]. In comparison with local storage systems, a cloud-based storage solution can increase the availability and durability of the data while lowering the IT maintenance cost. Particularly for smaller and medium-sized enterprises, a cloud storage solution can help reducing the storage cost in comparison with maintaining a local solution [15]. This cost reduction may emerge due to a decrease in the IT maintenance cost, which has to be considered for an own storage system.

Nowadays, several cloud storage providers exist, e.g., Amazon S3,[1] Google Cloud Storage,[2] or RackSpace CloudFiles[3] [9]. Each of them offers different storage technologies, different quality of services (QoSs) and pricing models.

This huge diversity of storages makes the decision, where the data should be stored, all but trivial. A customer has to take several constraints into account to find the best fitting provider for her current use case. Among others, some considerations can be: which storage technologies should be used, which pricing model is the cheapest, which geographical location should be chosen, or which provider has to be

[1] https://aws.amazon.com/s3/.

[2] https://cloud.google.com/storage/.

[3] http://www.rackspace.com/cloud/files.

avoided due to company policies. Particularly, the different pricing models are subject to a huge variety. They not only differ from provider to provider but also between different storage technologies and different geographical locations for the same provider. Apart from "standard" storages, specialized long-term storage services like Amazon Glacier[4] exist, which require lower cost but also decreased QoS, i.e., data retrieval may take hours.

Relying on only one cloud provider may lead to risks which should be prevented: For instance, a provider could increase the price of the storage or go out of business [2,7,23]. This results in the need to migrate the data to another provider, which involves additional migration cost, as well as implementation or administrative efforts. Even more problematic is the situation when the provider is temporarily unavailable or goes out of business. In the worst case, this may lead to a total data loss. For instance, an outage in February 2017 of Amazon S3 in North Virginia [1] showed that even big cloud storage providers struggle with service failures [5]. Moreover, cloud providers' terms of usage and customer properties may evolve over time, e.g., a cloud provider modifies the pricing models, or the amount of stored data changes. One way to prevent those risks is the redundant usage of different storages. This does not only decreases the risk of vendor lock-in [25] but also increases data availability and durability.

In this work, we address the problem of cost-efficient data redundancy in the cloud. We extend our former work [26], where we formulated a local optimization problem, with a global optimization problem that optimizes the placement of all files, in the remainder of this paper called *data objects*, on several cloud storages in a redundant and cost-efficient way. Moreover, we provide a heuristic approach to solve the global optimization problem. All three optimization approaches, the local one from our former work [26], and the global and heuristic approach from this work, consider predefined storage requirements (i.e., availability, durability, and a vendor lock-in factor) of the customer. Furthermore, all optimization approaches consider access patterns of all data objects, which is a significant cost factor [28].

The remainder of this paper is organized as follows: In Sect. 2, we provide background information for our approach. Afterward, we present the local, global, and heuristic optimization approaches in Sect. 3. The evaluation setup is described in Sect. 4, and the results of the evaluation are discussed in Sect. 5. Section 6 gives an overview of the related work, and Sect. 7 concludes the paper.

[4] https://aws.amazon.com/glacier/details/.

2 Background

Before the data object placements on several storages in a redundant and cost-efficient way can be discussed, we have to discuss some preliminaries.

2.1 Quality of service

When storing data on cloud storages, several QoS aspects need to be considered. The main aspects are availability, durability, and vendor lock-in factor since those three QoS aspects define how probable or improbable a data loss is [2,11,19,27].

Availability The availability defines the probability that a service, here a storage service, is available for a specific time span [3]. This QoS parameter is declared as availability of the storage service over a given time span in percentage, e.g., 99.99% over a year.

Durability The durability defines the probability that there is no data loss, e.g., due to a hardware failure, on a storage service. This QoS parameter is declared as the durability of the stored data in a defined time span in percentage, e.g., 99.99999% over a year.

Vendor lock-in factor A vendor lock-in situation can occur if the data are stored on only one storage service which is not reachable; therefore, the data are locked on this provider and can not be accessed or migrated [28]. This unavailability of the provider can be temporarily, e.g., networking issues; or in the worst case, the provider can go out of business [5,25]. This parameter is declared as $\text{lockin} = \frac{1}{N}$, where N is the amount of used storages, and $\text{lockin} \in (0, 1]$.

2.2 Erasure coding

Erasure coding is a redundancy mechanism where a data object is split into n chunks in a way that the whole data object can be reconstructed by any subset of size m ($m < n$) of those chunks [11,21,22,24]. An erasure coding configuration is therefore defined by the tuple (m, n). By defining different erasure coding configurations, the availability of a data object can be increased or decreased.

This functionality makes erasure coding a superset of the RAID technology and normal replication systems [27]. For example, a RAID 5 can be described by the erasure coding configuration $(4, 5)$ and a normal replication by $(1, 3)$, which will generate three replications. The main advantage of using erasure coding instead of replication is the smaller additional storage needed to achieve the same level of redundancy [27].

2.3 Pricing models

Storage service pricing models vary from storage provider to storage provider. However, most pricing models have the

Table 1 Amazon S3 price for EU Frankfurt region standard storage

Usage per month	Price per GB in $
Storage price	
Next 50 TB	0.0245
Next 450 TB	0.0235
>500 TB	0.0225
Outgoing traffic price	
First 1 GB	0.000
Up to 10 TB	0.090
Next 40 TB	0.085
Next 100 TB	0.070
Next 350 TB	0.050
>524 TB	On demand

same foundation: They account for the used storage, the used outgoing traffic, and the amount of read and write operations. Most of the pricing models do not charge the incoming traffic or delete operations. Furthermore, most storage providers use a *block rate pricing model* [18], where the cost decreases the more the storage is used by a customer, i.e., the more data is stored, the cheaper it is to store additional data. In addition, large cloud storage providers often have several geographically distributed data centers, called regions, with different pricing models. Notably, those providers often offer reduced migration prices that are charged when data are migrated from one region to another.

Besides changing pricing models due to different providers or regions, the pricing models can also differ by storage technologies. For example, some providers offer long-term storage solutions for seldom used data. Those storages often have cheap storage prices but a high traffic price in comparison with conventional storages. Long-term storages often also define a minimum storage duration, i.e., a *Billing Time Unit* (BTU). If such a BTU is defined and data are stored on the storage, the whole BTU time is charged whether or not the data are deleted or moved to another storage, before the BTU is over. In addition, some storage providers also define a minimum object size, i.e., *Billing Storage Unit* (BSU). Data objects that are smaller than the BSU are charged with the BSU size, despite the fact that only a part of the storage is used.

Representative for other providers, Table 1 shows the pricing model of the Amazon S3 region EU Frankfurt "standard storage." This cloud storage defines a block rate pricing model for storage and outgoing data transfer. The billing period of an Amazon S3 storage is one month. Additionally to the prices shown in the table, Amazon charges for the request commands PUT, COPY, POST, and LIST $0.0054 per 1,000 requests and for GET and all other requests $0.0043 per 10,000 requests, except the DELETE request which is free of charge. The migration between Amazon S3 regions is charged with $0.020 per GB. Incoming data transfer is also free of charge.[5]

As has already been mentioned, beside different regions some storage providers also offer different storage technologies. For instance, Amazon offers the following storage technologies: Amazon S3 Standard, Amazon S3 Standard – Infrequent Access (IA), and Amazon Glacier. Each storage technology has different pricing schemes and properties, e.g., the Standard–IA has a cheaper storage price, in comparison with Amazon S3 Standard, but also a lower availability. Further, Standard–IA defines a BTU of 30 days and a BSU of 128KB.

While the pricing models of most cloud storages are based on a similar notion, there are some differences between them. For instance, Amazon AWS S3 applies the block rate pricing model for the traffic as well as for the storage. In comparison, Google Cloud Storage applies a block rate pricing model for the outgoing data transfer, but not for the storage cost.

3 Data object placement

After having defined some necessary preliminaries in the last section, we are now able to discuss the system model, the formal specification of the local and the global optimization models, and the heuristic optimization approach.

Each optimization approach suggests the placement of the chunks of the data objects, on several cloud storages, in a cost-efficient way without violating predefined service-level objectives (SLOs). Those SLOs are defined for each data object by the owner of the data object. The SLOs are availability, durability, and vendor lock-in as defined in Sect. 2.

3.1 System model

For the data object placement optimization, we provide a mixed-integer linear programming (MILP)-based local and a global data placement approach and a heuristic approach. In the following, we introduce the used variables, the cost model, and the used decision variables, before we discuss the optimization approaches.

3.1.1 Variables

In our system model, the set of all available storages is labeled with S where $s \in S = \{s_1, s_2, \ldots\}$ defines one storage of this set. N is the set of all data objects, and $F \in N = \{F_1, F_2, \ldots\}$ is one data object of this set. Next, $f \in F = \{f_1, f_2, \ldots\}$ defines a chunk of a data object F, with $|S| \geq |F|$. As described in Sect. 2.2, the

[5] https://aws.amazon.com/s3/pricing/, as of July 16, 2017.

amount of chunks of a data object depends on the used erasure coding configuration. The parameter τ defines the amount of historical information of a chunk (e.g., amount of read and write operations) that is used for the optimization. Under the assumption that the usage pattern of a data object does not change over a period of time [16], our optimization approach uses this information to predict future data access.

As mentioned in Sect. 2.3, several storage providers define a block rate pricing model for the storage and traffic cost. In the system model, b_s defines one block of a block rate pricing model of storage s and $b_s \in B_s^{T_{\text{out}}} = \{b_1, b_2, \ldots\}$, where $B_s^{T_{\text{out}}}$ defines the set of all outgoing traffic price blocks. Analogously, $B_s^{\text{sto}} = \{b_1, b_2, \ldots\}$ defines the pricing blocks for the storage prices. Furthermore, $b_s = (b_{L,s}, b_{U,s}, p_s)$ defines a triple that includes the lower bound of the pricing block $b_{L,s}$, the upper bound $b_{U,s}$, and the price of this block p_s.

3.1.2 Cost model

The cost model is used by the optimization to calculate the cost that accrue by storing a chunk. The cost of storing a chunk is composed of the storage cost, the traffic cost of the chunk, and the cost of the performed read and write operations.

The total cost that is charged if a chunk f is stored on storage s by considering the last τ min of the chunk's history is calculated by (1). The total cost is calculated by adding the used storage cost, the cost for read and write operations, and the cost for the used incoming and outgoing traffic. The single cost factors will be explained in the following.

$$c_{s,f}(\tau) = c_{s,f}^{S}(\tau) + c_{s,f}^{R}(\tau) + c_{s,f}^{W}(\tau) + c_{s,f}^{T_{\text{in}}}(\tau) + c_{s,f}^{T_{\text{out}}}(\tau) \tag{1}$$

The storage cost that are charged if the chunk f is stored on s based on the last τ min of the chunk's history is calculated by (2). In the equation, the term $p_{s,\gamma_{s,f}}^{S}$ calculates the current storage price. Since several cloud storage providers use the already discussed block rate pricing model, the current price can be different depending on the present usage of the storage. In $p_{s,\gamma_{s,f}}^{S}$, this is taken into account by the term $\gamma_{s,f}$ that calculates the present usage of the storage s and adds the size of f to the result if f is currently not stored on the storage. The resulting price is then multiplied with the chunk size calculated by $\sigma_f(\tau)$ considering the last τ min of the chunks history. This term also considers if a BSU is defined for the storage. If this is the case and the chunk size is smaller than the BSU, then $\sigma_f(\tau)$ uses the BSU instead of the actual chunk size.

If storage s is a long-term storage, the BTU time has to be considered as well. This is done by $\hat{\sigma}_{f,\text{BTU}} \cdot h_f$. The term $\hat{\sigma}_{f,\text{BTU}}$ calculates the storage size of the chunk f that is charged for the remaining BTU time.

$$c_{s,f}^{S}(\tau) = p_{s,\gamma_{s,f}}^{S} \cdot (\sigma_f(\tau) + \hat{\sigma}_{\text{f,BTU}} \cdot h_f) \tag{2}$$

The charged cost for the write and read operations is calculated by (3) and (4). The terms $r_f^W(\tau)$ and $r_f^R(\tau)$ return the amount of write and read operations of chunk f during the last time period τ, respectively. Further, p_s^W defines the price of a write operation and p_s^R of a read operation. Delete operations are handled analogue.

$$c_{s,f}^{W}(\tau) = r_f^W(\tau) \cdot p_s^W \tag{3}$$
$$c_{s,f}^{R}(\tau) = r_f^R(\tau) \cdot p_s^R \tag{4}$$

The outgoing and incoming traffic cost of a chunk f, in the last time period τ, is calculated by (5) and (6). Since the description of the outgoing traffic cost is also applicable to the incoming traffic cost, we will only discuss the outgoing traffic cost defined in (5). $t_f^{\text{out}}(\tau)$ defines the amount of read bytes from chunk f during the last time period τ. The term $p_{s,\beta_{s,f}}^{T_{\text{out}}}$ returns the outgoing traffic price of storage s. Analogue to storage cost calculation, defined in (2), the block rate pricing model has to be considered also for the traffic cost calculation. Analogue to $\gamma_{s,f}$ in (2), this is done by $\beta_{s,f}$ in (5). $\beta_{s,f}$ calculates the amount of read bytes from storage s, including chunk f. If storage s is a long-term storage, data retrieval cost can be charged as well. This additional price is considered by $p_{s,\beta_{s,f}}^{\text{ret}}$. The variable h_f is the same as in (2).

$$c_{s,f}^{T_{\text{out}}}(\tau) = t_f^{\text{out}}(\tau) \cdot \left(p_{s,\beta_{s,f}}^{T_{\text{out}}} + p_{s,\beta_{s,f}}^{\text{ret}} \cdot h_f\right) \tag{5}$$
$$c_{s,f}^{T_{\text{in}}}(\tau) = t_f^{\text{in}}(\tau) \cdot p_{s,\beta_{s,f}}^{T_{\text{in}}} \tag{6}$$

Optional migration cost which occurs if a chunk f has to be migrated from one storage to another is calculated by (7) and (8). Which equation is taken to calculate the migration cost depends on the migration type. If a chunk f has to be migrated from one storage to another storage of the same provider and this provider defines a special migration price, (7) is used. If a chunk f has to be migrated from one storage to another storage and the provider is different, (8) is used. In (8), $p_{s_1,\beta_{s_1,f}}^{T_{\text{out}}}$, $p_{s_2,\beta_{s_2,f}}^{T_{\text{in}}}$, and $p_{s,\beta_{s,f}}^{\text{ret}}$ are analogously defined as in (5) and (6). In (7), $p_{s_1,\beta_{s_1,f}}^{T_{\text{out,reg}}}$ and $p_{s_2,\beta_{s_2,f}}^{T_{\text{in,reg}}}$ define the same but considering region migration prices. $\hat{\sigma}_f$ specifies the size of the chunk f. $r_{s_1}^R$ and $r_{s_1}^W$ define the amount of required read and write operations. The terms $p_{s_1}^R$ and $p_{s_2}^W$ represent the same as in (3) and (4).

$$c^{M_{\mathrm{reg}}}_{s_1,s_2,f} = \left(p^{T_{\mathrm{out,reg}}}_{s_1,\beta_{s_1},f} + p^{T_{\mathrm{in,reg}}}_{s_2,\beta_{s_2},f} + p^{\mathrm{ret}}_{s,\beta_s,f} \cdot h_f\right) \cdot \hat{\sigma}_f + r^R_{s_1} \cdot p^R_{s_1} + r^W_{s_2} \cdot p^W_{s_2} \tag{7}$$

$$c^{M}_{s_1,s_2,f} = \left(p^{T_{\mathrm{out}}}_{s_1,\beta_{s_1},f} + p^{T_{\mathrm{in}}}_{s_2,\beta_{s_2},f} + p^{\mathrm{ret}}_{s,\beta_s,f} \cdot h_f\right) \cdot \hat{\sigma}_f + r^R_{s_1} \cdot p^R_{s_1} + r^W_{s_2} \cdot p^W_{s_2} \tag{8}$$

3.1.3 Decision variables

In our optimization model, $x_{(s,f)} \in \{0,1\}$ defines if a chunk f is stored on storage s ($x_{s,f} = 1$) or not ($x_{s,f} = 0$). The optimization model uses the variable $g_{\tilde{S},F} \in \{0,1\}$, where $\tilde{S} = \{s_1, s_2, \ldots, s_n\}$ is a subset of the storage set S with $|\tilde{S}| = |F|$ and $\tilde{S} \subseteq S$. $g_{\tilde{S},F} = 1$ indicates that all storages of the subset $\tilde{S}$ have one chunk of F stored and $g_{\tilde{S},F} = 0$ indicates that at least one storage of $\tilde{S}$ does not store a chunk of F.

The decision variable $h_f \in \{0,1\}$ denotes that a chunk f is currently stored on a long-term storage, indicated by $h_f = 1$, or not, indicated by $h_f = 0$. The system model further uses the decision variables z_{s_1,s_2} and y_{s_1,s_2} to indicate if two storages are the same or are different but have the same storage provider. The variable $y_{s_1,s_2} \in \{0,1\}$ defines if the storages s_1 and s_2 are not identical but have the same storage provider, indicated by $y_{s_1,s_2} = 1$; $y_{s_1,s_2} = 0$ otherwise. Analogously, $z_{s_1,s_2} \in \{0,1\}$ defines if the storages s_1 and s_2 are not the same and have different storage providers, indicated by $z_{s_1,s_2} = 1$; $z_{s_1,s_2} = 0$ otherwise.

The decision variables $u^{T_{\mathrm{out}}}_{s,b_s} \in \{0,1\}$, $v^{T_{\mathrm{out}}}_{s,b_s} \in \{0,1\}$, and $o^{T_{\mathrm{out}}}_{s,b_s} \in \{0,1\}$ are used to indicate if the overall used outgoing traffic of a storage is in a specific pricing step of a block rate pricing model. $u^{T_{\mathrm{out}}}_{s,b_s} = 1$ indicates that the outgoing traffic of storage s is bigger than the lower boundary $b_{L,s}$ defined in b_s; $u^{T_{\mathrm{out}}}_{s,b_s} = 0$ otherwise. $v^{T_{\mathrm{out}}}_{s,b_s} = 1$ indicates that the outgoing traffic of storage s is smaller than the upper boundary $b_{U,s}$ defined in b_s; $v^{T_{\mathrm{out}}}_{s,b_s} = 0$ otherwise. Furthermore, $o^{T_{\mathrm{out}}}_{s,b_s} = 1$ indicates that the traffic is between the lower and upper bound of b_s, which means that $u^{T_{\mathrm{out}}}_{s,b_s} = 1$ and $v^{T_{\mathrm{out}}}_{s,b_s} = 1$ hold. $o^{T_{\mathrm{out}}}_{s,b_s} = 0$ indicates that this is not the case. The variables $u^{\mathrm{sto}}_{s,b_s} \in \{0,1\}$, $v^{\mathrm{sto}}_{s,b_s} \in \{0,1\}$ and $o^{\mathrm{sto}}_{s,b_s} \in \{0,1\}$ are used to assess the block for the used storage of s.

3.2 Local placement problem

After defining the system model, we are now able to discuss the local optimization problem. The local optimization problem defines the problem of finding a cost-efficient placement of one data object F. Therefore, the optimization problem is provided with the data object F and all available storages S. Additionally, the problem takes the time period τ as input.

3.2.1 Objective function

(9) shows the objective function, which is set to minimize the overall cost to store F.

$$\min \sum_{f \in F} \sum_{s \in S} \left(c_{s,f}(\tau) \cdot w_{s,f} + c^M_{f_s,s,f} \cdot z_{f_s,s} + c^{M_{\mathrm{reg}}}_{f_s,s,f} \cdot y_{f_s,s} \right) \cdot x_{s,f} \tag{9}$$

$c_{s,f}(\tau) \cdot w_{s,f}$ calculates the overall cost to store the chunk f on the storage s by taking the last τ min of the chunks usage history into account. The term $c_{s,f}(\tau)$ is already discussed in Sect. 3.1.2. The term $w_{s,f} \in$[1,BTU] is a multiplier that helps to specify if the overall storage cost can be reduced by storing a chunk f on a long-term storage. If s is a long-term storage, the calculation of the resulting value of the term is initialized with the value of the BTU, i.e., $w_{s,f}$=BTU. The algorithm decreases $w_{s,f}$ each time there was no or rare usage of the chunk according to the history, where the amount of historical information is defined by the BTU. This is done until all historical information is checked or until $w_{s,f} = 1$. If s is a standard storage without any BTU, the value is always 1.

$c^M_{f_s,s,f} \cdot z_{f_s,s}$ and $c^{M_{\mathrm{reg}}}_{f_s,s,f} \cdot y_{f_s,s}$ calculate the migration cost from one storage to another, without and with special migration prices. The term f_s returns the storage on which the chunk f is currently stored. Finally, the decision variable $x_{s,f}$ decides if the chunk f is stored on the storage s or not.

3.2.2 Constraints

The first constraint (10) ensures that the selected storage solution of a data object fulfills the required vendor lock-in factor l_F. As mentioned above, the vendor lock-in factor is set for each data object as a SLO.

$$\frac{1}{\sum_{f \in F} \sum_{s \in S} x_{s,f}} \leq l_F \tag{10}$$

Constraints (11) and (12) ensure that the required durability and availability of a data object are fulfilled. In the following we will only describe (11) because the description of (12) is analogue.

$\sum_{\tilde{S}' \in r_{\tilde{S},k}} \left[\prod_{s \in \tilde{S}'} \hat{a}_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - \hat{a}_s) \right]$ calculates the availability of the storage set $\tilde{S}'$, whereas $\tilde{S}'$ holds all possible combinations of size k of the set $\tilde{S}$. Those combinations are represented by $r_{\tilde{S},k}$. $\hat{a}_s$ defines the availability of s. Conclusively, this part of the equation calculates the probability that there are k simultaneously available storages. To complete (11), we have to include the functionality that a system which uses erasure coding with a coding configuration of (m, n) and can withstand up to $n - m$ simultaneous storage failures. This is done by increasing k starting from m, i.e., the minimum amount of required chunks of F depicted by F^m, to $|\tilde{S}|$. In the equation, this is achieved by $\sum_{k=|F^m|}^{|\tilde{S}|}$. Finally, the result is compared to $a_F \cdot g_{\tilde{S},F}$ where a_F defines the required availability of the data object and $g_{\tilde{S},F}$ defines if each storage in $\tilde{S}$ has one chunk stored or not. $g_{\tilde{S},F}$ is defined by constraints (13) and (14) that define together a logic AND.

$$\sum_{k=|F^m|}^{|\tilde{S}|} \sum_{\tilde{S}' \in r_{\tilde{S},k}} \left[\prod_{s \in \tilde{S}'} \hat{a}_s \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - \hat{a}_s) \right] \geq a_F \cdot g_{\tilde{S},F} \tag{11}$$

$$\sum_{k=|F^m|}^{|\tilde{S}|} \sum_{\tilde{S}' \in r_{\tilde{S},k}} \left[\prod_{s \in \tilde{S}'} \hat{d}_s \prod_{s \in \tilde{S} \setminus \tilde{S}'} \left(1 - \hat{d}_s\right) \right] \geq d_F \cdot g_{\tilde{S},F} \tag{12}$$

$$g_{\tilde{S},F} \geq \sum_{s \in \tilde{S}} \sum_{f \in F} x_{s,f} - (|F| - 1) \tag{13}$$

$$g_{\tilde{S},F} \leq \sum_{f \in F} x_{\tilde{s},f} \qquad \forall \tilde{s} \in \tilde{S} \tag{14}$$

With (15) it is ensured that only $|F|$ assignments from the chunks $f \in F$ to the storages $s \in S$ exist. Furthermore, (16) and (17) ensure that each chunk is stored on only one storage and (18) defines the decision variable boundaries.

$$\sum_{f \in F} \sum_{s \in S} x_{s,f} = |F| \tag{15}$$

$$\sum_{s \in S} x_{s,f} \leq 1 \tag{16}$$

$$\sum_{f \in F} x_{s,f} \leq 1 \tag{17}$$

$$\begin{aligned} &g_{\tilde{S},F} \in \{0,1\}; \quad x_{s,f} \in \{0,1\} \\ &z_{s_1,s_2} \in \{0,1\}; \quad y_{s_1,s_2} \in \{0,1\} \end{aligned} \tag{18}$$

3.3 Global placement problem

In the following, we discuss the global optimization problem. In comparison with the local optimization problem, the global optimization problem defines the problem of finding the cheapest placement for all data objects on all available storages S. The global optimization problem gets as an input the sets S and N, which contain all data objects. Further, the parameter τ is set to the length of the BTU, i.e., $\tau = \text{BTU}$. This way the global optimization gets the historic information of the last BTU and can, therefore, precisely calculate if the cost can be decreased by storing the chunk on a long-term storage where the whole BTU is charged.

3.3.1 Objective function

The objective function of the global optimization problem optimizes the placement of all chunks $f \in F$ of all data objects $F \in N$ as shown in (19).

$$\begin{aligned} \min \sum_{s \in S} \Bigg[&\sum_{F \in N} \sum_{f \in F} \Big(c^R_{s,f}(\tau) + c^W_{s,f}(\tau) \\ &+ c^M_{f_s,s,f} \cdot z_{\hat{s},s} + c^{M_\text{reg}}_{f_s,s,f} \cdot y_{\hat{s},s} \Big) \cdot x_{s,f} \\ &+ c^\text{sto}_s(\tau) + c^{T_\text{out}}_s(\tau) \Bigg] \end{aligned} \tag{19}$$

While for the local optimization the consideration of the currently stored chunks of a storage was enough to identify the current price in a block rate pricing model, this is not applicable anymore for the global optimization. This is due to the fact that by adding the possibility to migrate multiple chunks at once, all chunks on a storage can change and, thus, also the pricing step. To consider this, the global optimization problem splits up the cost calculation $c_{s,f}(\tau)$. Therefore, it models those cost calculations that may include a block rate pricing model, namely the used outgoing traffic cost $c^{T_\text{out}}_s(\tau)$ and the used storage cost $c^\text{sto}_s(\tau)$, as constraints. Those calculations are done by (25) for the outgoing traffic cost and by (31) for the storage cost. The terms $c^R_{s,f}(\tau)$ and $c^W_{s,f}(\tau)$ calculate the read/write operation cost, as defined in (3) and (4). The migration cost are calculated by $c^{M_\text{reg}}_{f_s,s,f}$ and $c^M_{f_s,s,f}$ as defined in (7) and (8).

3.3.2 Constraints

The global optimization uses the same constraints as the local optimization, i.e., (10) to (18). However, due to the fact that the global optimization considers the placement of all files at once, all local optimization constraints need to be defined for all $F \in N$. Furthermore, the global optimization requires further constraints, to include the storage cost and outgoing traffic cost of a storage, which are discussed in the following.

(20) to (24) define if the outgoing traffic of storage s is in the range of a block rate pricing model step. In (20) together with (21), it is defined if the used outgoing traffic of storage s is bigger than the lower boundary $b_{L,s}$ of a pricing step $b_s \in B^{T_\text{out}}_s = \{b_1, b_2, \ldots\}$. $t^\text{out}_f(\tau)$ is defined analogue as in (5), and M is a sufficient large constant that is at least larger than the largest possible value of $\sum_{F \in N} \sum_{f \in F} t^\text{out}_f(\tau) \cdot x_{s,f}$.

$$\begin{aligned} &b_{L,s} \leq \sum_{F \in N} \sum_{f \in F} t^\text{out}_f(\tau) \cdot x_{s,f} + M \cdot \left(1 - u^{T_\text{out}}_{s,b_s}\right) \\ &\forall s \in S;\ \forall b_s \in B^{T_\text{out}}_s;\ \exists b_{L,s} \in b_s \end{aligned} \tag{20}$$

$$b_{L,s} > \sum_{F\in N}\sum_{f\in F} t_f^{\text{out}}(\tau)\cdot x_{s,f} - M\cdot u_{s,b_s}^{T_{\text{out}}} \quad \forall s\in S;\ \forall b_s\in B_s^{T_{\text{out}}};\ \exists b_{L,s}\in b_s \tag{21}$$

(22) together with (23) indicates if the used outgoing traffic of storage s is smaller than the upper boundary $b_{U,s}$ of a pricing step $b_s \in B_s^{T_{\text{out}}} = \{b_1, b_2, \ldots\}$.

$$\sum_{F\in N}\sum_{f\in F} t_f^{\text{out}}(\tau)\cdot x_{s,f} \le b_{U,s} + M\cdot\left(1 - v_{s,b_s}^{T_{\text{out}}}\right) \quad \forall s\in S;\ \forall b_s\in B_s^{T_{\text{out}}};\ \exists b_{U,s}\in b_s \tag{22}$$

$$\sum_{F\in N}\sum_{f\in F} t_f^{\text{out}}(\tau)\cdot x_{s,f} > b_{U,s} - M\cdot v_{s,b_s}^{T_{\text{out}}} \quad \forall s\in S;\ \forall b_s\in B_s^{T_{\text{out}}};\ \exists b_{U,s}\in b_s \tag{23}$$

Finally, (24) defines if the outgoing traffic is between the lower and upper boundary of a pricing block b_s. This is the case if the used outgoing traffic is bigger than the lower boundary $b_{L,s}$, indicated by $u_{s,b_s}^{T_{\text{out}}}$, and smaller than the upper boundary $b_{U,s}$, indicated by $v_{s,b_s}^{T_{\text{out}}}$.

$$0 \le u_{s,b_s}^{T_{\text{out}}} + v_{s,b_s}^{T_{\text{out}}} - 2\cdot o_{s,b_s}^{T_{\text{out}}} \le 1 \quad \forall s\in S; \forall b_s\in B_s^{T_{\text{out}}} \tag{24}$$

The information if the used traffic of a storage s is within a pricing range, defined by $o_{s,b_s}^{T_{\text{out}}}$, is then used to calculate the cost that are charged due to the used traffic of a storage, indicated by $c_s^{T_{\text{out}}}(\tau)$. This is done by (25) where p_s defines the price of the pricing range $b_s \in B_s^{T_{\text{out}}} = \{b_1, b_2, \ldots\}$.

$$\sum_{F\in N}\sum_{f\in F} t_f^{\text{out}}(\tau)\cdot p_s\cdot x_{s,f} - M\left(1 - o_{s,b_s}^{T_{\text{out}}}\right) \le c_s^{T_{\text{out}}}(\tau) \le \sum_{F\in N}\sum_{f\in F} t_f^{\text{out}}(\tau)\cdot p_s\cdot x_{s,f} + M\cdot(1 - o_{s,b_s}^{T_{\text{out}}}) \quad \forall s\in S; \forall b_s\in B_s^{T_{\text{out}}}; \exists p_s\in b_s \tag{25}$$

Analogue as (20) to (24) defines if the used outgoing traffic of a storage s is within a pricing range of a block rate pricing model step, equations (26) to (30) define if the size of the chunks stored on storage s is within a pricing range. Since the description of (20) to (24) is also applicable to (26) to (30), we will repeat it here in detail. In the following, $\sigma_f(\tau)$ is defined analogue as in (2), and M is a sufficient large constant that is at least larger than the largest possible value of $\sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot x_{s,f}$.

$$b_{L,s} \le \sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot x_{s,f} + M\cdot\left(1 - u_{s,b_s}^{\text{sto}}\right) \quad \forall s\in S;\ \forall b_s\in B_s^{\text{sto}};\ \exists b_{L,s}\in b_s \tag{26}$$

$$b_{L,s} > \sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot x_{s,f} - M\cdot u_{s,b_s}^{\text{sto}} \quad \forall s\in S;\ \forall b_s\in B_s^{\text{sto}};\ \exists b_{L,s}\in b_s \tag{27}$$

$$\sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot x_{s,f} \le b_{U,s} + M\cdot\left(1 - v_{s,b_s}^{\text{sto}}\right) \quad \forall s\in S;\ \forall b_s\in B_s^{\text{sto}};\ \exists b_{U,s}\in b_s \tag{28}$$

$$\sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot x_{s,f} > b_{U,s} - M\cdot v_{s,b_s}^{\text{sto}} \quad \forall s\in S;\ \forall b_s\in B_s^{\text{sto}};\ \exists b_{U,s}\in b_s \tag{29}$$

$$0 \le u_{s,b_s}^{\text{sto}} + v_{s,b_s}^{\text{sto}} - 2\cdot o_{s,b_s}^{\text{sto}} \le 1 \quad \forall s\in S; \forall b_s\in B_s^{\text{sto}} \tag{30}$$

Furthermore, analogue as (25) calculates the cost that occur due to the used outgoing traffic of s, (31) calculates the cost that occur due to the used storage of s.

$$\sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot p_s\cdot x_{s,f} - M\left(1 - o_{s,b_s}^{\text{sto}}\right) \le c_s^{\text{sto}}(\tau) \le \sum_{F\in N}\sum_{f\in F}\sigma_f(\tau)\cdot p_s\cdot x_{s,f} + M\cdot(1 - o_{s,b_s}^{\text{sto}}) \quad \forall s\in S;\ \forall b_s\in B_s^{\text{sto}}; \exists p_s\in b_s \tag{31}$$

Finally, (32) ensures that $c_s^{T_{\text{out}}}(\tau)$ and $c_s^{\text{sto}}(\tau)$ are positive for all $s \in S$ and that the remaining decision variables are bounded to $\{0, 1\}$.

$$\begin{aligned} &c_s^{T_{\text{out}}}(\tau) \ge 0; && c_s^{\text{sto}}(\tau) \ge 0 && \\ &u_{s,b_s}^{T_{\text{out}}} \in \{0,1\}; && o_{s,b_s}^{T_{\text{out}}} \in \{0,1\}; && v_{s,b_s}^{T_{\text{out}}} \in \{0,1\} \\ &v_{s,b_s}^{\text{sto}}x \in \{0,1\}; && u_{s,b_s}^{\text{sto}} \in \{0,1\}; && o_{s,b_s}^{\text{sto}} \in \{0,1\} \end{aligned} \tag{32}$$

3.4 Heuristic placement

As a third optimization approach, we developed a heuristic for the global optimization. This heuristic first applies a classification of all data objects by the used storage size and the used outgoing traffic. Subsequently, the best fitting storage set for each of those classes is selected by optimizing a representative data object. The result of this optimization is then applied to all data objects in a class. Therefore, a nearly optimal solution can be achieved by calculating the optimal placement for only a couple of data objects.

Based on an intensive analysis of the global and local optimization results, we identified that the main reasons of a chunk migration are the outgoing traffic of a chunk and the size of it. Therefore, the classification is based on these two aspects. For defining the upper and lower boundaries of a class, we are using a separation by quantiles for the storage size classes. However, in comparison with the storage size of the chunks, where each chunk has a size > 0, the traffic of the chunks can be 0, i.e., all not used chunks. To take care that all of them are in the same class, we define the traffic class boundaries based on the outgoing traffic instead of a separation by quantiles, which could distribute those chunks into several classes.

Algorithm 1 and 2 describe how our approach uses classification and the already discussed local optimization problem to find a cheap storage solution for all chunks of all data objects. In the following we first discuss the classification in Algorithm 1 and then the optimization, described in Algorithm 2, based on the results of the classification. Depending on the requirements, this heuristic optimization can be performed for each data object access or only in predefined intervals, e.g., each 1,000th data object access. This is due to the fact that our proposed heuristic optimizes the placement of all chunks at a time and, thus, also considers the data objects that were accessed before the optimization starts.

As an input, the classification algorithm, depicted in Algorithm 1, requires the boundaries for the traffic and storage classes. The first step of the algorithm is to sort all data objects according to their used traffic and storage size, which is done by the methods *sortByTraffic()* and *sortByStorage()* in lines 2 and 3. Subsequently, empty lists for all traffic classes, called *trafficClasses*, and for all storage classes, called *storageClasses*, are generated (lines 4 and 5) according to the classes defined by the boundaries *trafficBoundaries* and *storageBoundaries*. Those empty lists are then filled with the sorted data objects by the method *fillClasses()* (lines 6 and 7). Finally, the cartesian product of the lists *trafficClasses* and *storageBoundaries* is created and stored in the list *classes* (lines 9–14). This results in a list of all combinations of the traffic and storage classes including their corresponding chunks. This list is then used in Algorithm 2 for the optimization.

Algorithm 1 Classification

```
Require: $trafficBoundaries, storageBoundaries$
1: function PERFORMCLASSIFICATION($dataObjs$)
2:   $dataObjSortByTraffic \leftarrow sortByTraffic(dataObjs)$
3:   $dataObjSortByStorage \leftarrow sortByStorage(dataObjs)$
4:   $trafficClasses \leftarrow getTrafficClasses(trafficBoundaries)$
5:   $storageClasses \leftarrow getStoClasses(storageBoundaries)$
6:   $fillClasses(trafficClasses, dataObjSortByTraffic)$
7:   $fillClasses(storageClasses, dataObjSortByStorage)$
8:   $classes \leftarrow \emptyset$
9:   for all $trafficClass \in trafficClasses$ do
10:    for all $storageClass \in storageClasses$ do
11:      $class \leftarrow merge(trafficClass, storageClass)$
12:      $classes \leftarrow classes \cup \{class\}$
13:    end for
14:  end for
15:  return $classes$
16: end function
```

Algorithm 2 gets as input all data objects that are subject for optimization, in our case all stored data objects, and all available storages. At the beginning of Algorithm 2, the *performClassification()* method is called (line 2), which is described by Algorithm 1. The result of this call is a list of all possible traffic and storage class combinations, including the corresponding chunks. Algorithm 2 is then iterating through this list (lines 3–15) and selects at the beginning of each iteration, wherever the class is not empty (line 4), a representative data object for the current class. This is done by the method *getRepresentativeDataObj()* (line 5). The concrete data object selection depends on the implementation of this method. Possible implementations are, e.g., a random selection, the first or last data object of the class, or a data object in the middle of the class. Subsequently, method *localOpt()* (line 6) finds the best chunk placement for the representative data object by solving the local optimization problem from Sect. 3.2 and stores it in *dataObj*. Afterward, the result of the optimization is read by the method *getStorages()* (line 7) that returns the list of selected storages and stores it in *selStorages*.

This storage set is then applied for all data objects in the current class. For this, the algorithm iterates through all data objects in the class (lines 8–13). In each iteration, the first step is to map each chunk of a data object to one storage of the *selStorages* list. This is done by the method *getMap()*, and the result is stored in *chunkStgMap* (line 9). To avoid unnecessary migration steps, this method has to consider that chunks may already be stored on one of the selected storages. Therefore, *getMap()* considers if a chunk is already stored on one of the selected storages from the list *selStorages*. In this case, the chunk will be mapped to the same storage again. In the final step, each of those chunks to storage mappings is added to the resulting list *optimalPla* (lines 10–12). At the end of the execution of Algorithm 2, the list *optimalPla* holds the final optimal chunk to storage mapping for all chunks of all data objects.

Algorithm 2 Optimization

```
Require: $allDataObjects, allStorages$
1: $optimalPla \leftarrow \emptyset$
2: $classes \leftarrow performClassification(allDataObjects)$
3: for all $class \in classes$ do
4:   if $class$ not $\emptyset$ then
5:     $dataObj \leftarrow getRepresentativeDataObj(class)$
6:     $localOptResult \leftarrow localOpt(allStorages, dataObj, \tau)$
7:     $selStorages \leftarrow getStorages(localOptResult)$
8:     for all $dataObj \in class$ do
9:       $chunkStgMap \leftarrow getMap(selStorages, dataObj)$
10:      for all $chunkStorage \in chunkStgMap$ do
11:        $optimalPla \leftarrow optimalPla \cup \{chunkStorage\}$
12:      end for
13:    end for
14:  end if
15: end for
16: return $optimalPla$
```

Since there is no historical information available for the first upload of a new data object, we use a fixed storage set for the first upload. Ideally, this fixed storage set includes the cheapest storages in respect to the storage and traffic prices.

The complexity of the algorithm depends mainly on the complexity of the selected local optimization solution, i.e., line 6 in Algorithm 2, and on the sorting of the data objects, i.e., lines 2 and 3 in Algorithm 1. The complexity of Algorithm 1 is $O(\max(n \cdot log(n), m \cdot w))$, where n is the number of data objects, m is the amount of traffic classes, and w is the amount of storage classes. Algorithm 2 depends on the local optimization, which is a NP-hard problem. To reduce and limit the optimization duration a deadline-based approach can be used, e.g., take the best solution after 2 min of searching for the local optimization.

4 Evaluation setup

To evaluate all approaches, we prototypically implement them by using the middleware presented in our former work [26]. For the evaluation, we use a real-world cloud storage access trace [14]. The scope of this evaluation is the global and heuristic approaches. A detailed evaluation of the local optimization approach can be found in our previous work [26].

4.1 Prototype

We extended the middleware from our former work [26], called CORA, with the global and the heuristic optimization approaches. The middleware already provides the local optimization. We apply CPLEX[6] to solve the global optimization problem.

In case of the global optimization, the prototype executes the optimization for each data object access (i.e., read, write, update, delete). The same applies for the local optimization. However, to also include seldom used data objects, the local optimization is performed on all not or rarely accessed data objects in predefined intervals. For the heuristic approach, the prototype also executes the optimization in predefined interval steps. As a representative data object for a class, it selects the data object that is in the middle of the class. If the amount of data objects in a class is even, the implementation rounds up and takes the upper one. As a value for the constant M, the prototype uses $\sum_{F \in N} \sum_{f \in F} t_f^{\text{out}}(\tau)$ for the traffic price calculation and $\sum_{F \in N} \sum_{f \in F} \sigma_f(\tau)$ for the storage price calculation.

4.2 Storages

In the evaluation, we evaluate and analyze the behavior of all three optimization approaches with real-world cloud storage systems. Beside the public cloud storage solutions from Amazon (AWS S3) and Google (Google Cloud Storage), we use a self-hosted Swift[7] storage system. Table 2 provides an overview of the cloud storages used for the evaluation.

Table 2 Evaluation Storages

Provider	Region	Storage Class
AWS S3	US Oregon	Standard
AWS S3	US Oregon	IA
AWS S3	US North California	Standard
AWS S3	EU Frankfurt	Standard
AWS S3	EU Frankfurt	IA
AWS S3	Asia Pacific Tokyo	Standard
AWS S3	Sao Paulo	Standard
Google Cloud	–	Standard
self-hosted	–	Standard
self-hosted	–	Long-term

IA infrequent Access

The evaluation uses the AWS S3[8] and Google Cloud Storage[9] pricing models. For the self-hosted standard storage, we apply the AWS S3 Frankfurt standard storage pricing model and the AWS S3 Frankfurt IA pricing model for the self-hosted long-term storage.

4.3 Evaluation data

For the evaluation of our approaches, we use the access trace of a public available dataset presented in [14]. This dataset contains 30 days of anonymized data objects' access information on cloud storages used by more than 1,000,000 users.

For the evaluation of our optimization approaches, we extracted a set of 188 data objects, to evaluate and compare all optimization approaches, and a set of 10,400 data objects, to evaluate the scalability of the heuristic approach. To include heavily used data objects, as well as seldom used data objects, both data sets include data objects with different access frequencies. In the smaller data set (i.e., the 188 data objects set), 74% of the data objects are used less than 50 times, 16% are used more than 50 times and less than 500 times, and 10% are used more than 500 times in the full 30 days of the trace. The size of the data objects is between 20KB and 800MB. For the bigger dataset (i.e., the 10,400 data objects set), the access frequencies of the data objects are: 64% are used less than 50 times, 34% are used more than 50 times and less than 500 times, and 2% are used more than 500 times. The size of the data objects is between 3KB and 2.2GB.

[6] http://www.ibm.com/software/commerce/optimization/cplex-optimizer/.

[7] https://wiki.openstack.org/wiki/Swift.

[8] https://aws.amazon.com/s3/pricing/.

[9] https://cloud.google.com/storage/pricing.

Besides normal cloud storage usage, the used access trace includes three DDOS attacks that increase the usage of the storages drastically. We did not include those attacks in our evaluation, because we only consider the normal usage of a cloud storage.

Since the trace was taken from an already running and used cloud storage solution, the trace also includes data objects that were uploaded to the storage before the recording of the trace started but read during the recording period. To make sure that those data objects can also be used for our evaluation, we simulated an upload at the beginning of the evaluation.

Furthermore, we define that each data object has to be stored with a durability of 99.999999%, an availability of 99.99%, and a vendor lock-in factor of 0.5.

4.4 Evaluation process

For each evaluation scenario, we iterate through the access traces of each data object in the used data object set and perform the recorded operations (i.e., upload, update, read, and delete).

All evaluations use the entire 30 days of the trace and the full storage provider set of Table 2. We set the BTUs and billing periods for all long-term storages to one week, to be able to evaluate the whole behavior of the optimization approaches. Furthermore, the history time step interval is set to 12 h and the amount of used history time steps for the local and the heuristic approaches to five steps, which results in the consideration of 2.5 days of historical access information. As mentioned in Sect. 3.3, the global optimization always uses all history steps from the last BTU for the optimization. As storages for the first upload of a new data object of the heuristic approach, we use the storages AWS S3 EU Frankfurt standard storage, AWS S3 US Oregon standard storage, and self-hosted standard storage. In case of the local optimization, the optimization of all not used data objects is set to 8 days. Thus, it is guaranteed that the optimization can use the history information of a whole week.

To evaluate the behavior of the heuristic with different class boundaries, we run the 10,400 data objects set evaluation with different boundaries, as discussed in Sect. 3.4.

To also measure the required efforts of the approaches and the prototype, in terms of time and memory, we further log the duration of each optimization run and the size of the used metadata.

4.5 Baseline

As a baseline, for each evaluation scenario, we disable the optimization and define a fixed set of storages that is equally used. This baseline simulates the cost in the case when no optimization is applied. This fixed storage set contains the three cheapest standard storages: AWS S3 EU Frankfurt standard storage, AWS S3 US Oregon standard storage, and self-hosted standard storage.

5 Evaluation scenarios

In Sect. 5.1, we evaluate and compare all three optimization approaches, i.e., the local, global, and heuristic optimization, with the baseline by using the small test set. In Sect. 5.2, we then compare the heuristic with the baseline in a second evaluation scenario with the larger test set. Finally, we discuss the performance of all three approaches in Sect. 5.3.

5.1 Evaluation scenario 1

In the first evaluation scenario, we evaluate the behavior of the global optimization and compare the results with the results of the local and heuristic approaches. Since the global optimization is triggered after each data object access for all data objects at once, the runtime of this optimization increases exponentially with the amount of data objects. Therefore, the maximal amount of data objects that can be handled by the global optimization has a smaller upper bound than for the other optimization approaches. Nevertheless, our prototype can handle enough data objects to be able to compare the quality of the local optimization and heuristic approach with the global optimal solution. As an erasure coding configuration, we use for the baseline, the local approach, and the heuristic approach a (2,3) configuration since this configuration offered the best solutions in respect of the cost, the availability, and the optimization duration in our previous work [26]. For the global optimization, we evaluate the behavior with the erasure coding configurations (2,3), (2,4), and (3,4). The (2,4) configuration provides a high availability, i.e., only two of four storages have to be available to read a data object; nevertheless, this results in bigger chunks. The (3,4) configuration has smaller chunks and , however, also a smaller availability, since three storages out of four have to be available.

For the heuristic approach, we set the optimization interval to each data object access. As class boundaries for the heuristic, we use the storage quantiles (sq) = {25,50,75} and four traffic classes.

Evaluation Hypothesis At the beginning of the evaluation, all optimization approaches select the cheapest storages. Since the baseline also uses the cheapest storages, there should not be any difference. After a while, the global optimization starts to migrate chunks to long-term storages. This increases the cost of the global optimization, because of the additional BTU cost due to the chunks that are now stored on long-term storages. For the local optimization, the migration of not or rarely used chunks to long-term storages starts after

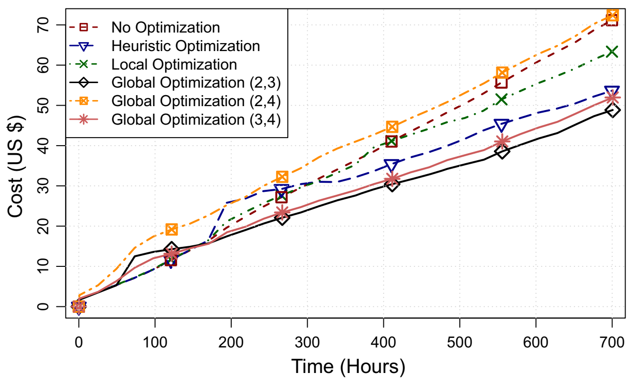

Fig. 1 Cumulative cost for the first evaluation scenario

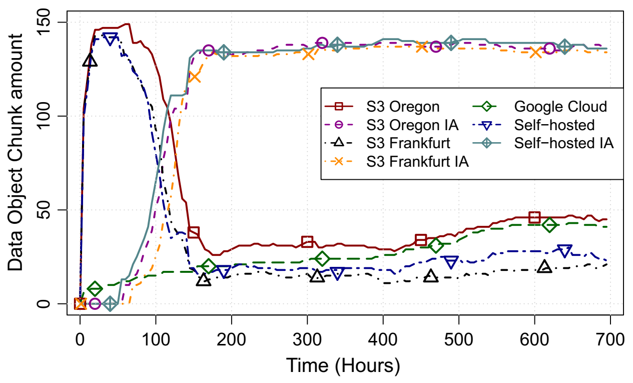

Fig. 2 Data object chunk distribution of the global optimization with a (2,3) erasure coding configuration in the first evaluation scenario

8 days. Before this point in time only chunks are optimized that are often used. Since those chunks are already on the cheapest standard storages, this optimization will not trigger any changes. Since the heuristic approach uses the local optimization as optimization strategy, the first migration of several chunks to long-term storages takes place at a similar point in time.

Evaluation Execution Figure 1 shows the cumulative cost of each optimization approach and of the baseline. Figure 2 shows the distribution of the chunks on the different storages during the global optimization evaluation with the (2,3) erasure coding configuration. For a clearer graph, only those storages that are selected are shown.

As shown in Fig. 1, at the beginning the cost for all optimization approaches (except for the global optimization with the (2,4) configuration) and the baseline are the same, due to the already cheapest standard storage selection. The global optimization with the (2,4) configuration is more expensive due to the bigger chunks; however, this configuration offers the highest availability. After 25 and 55 h, the global optimization migrates not or rarely used chunks to long-term storages, which increases the cost due to the additional BTU cost. This is also shown in Fig. 2 where chunks are transferred from standard storages, i.e., AWS S3 US Oregon, to long-term storages, i.e., AWS S3 US Oregon IA. Since the whole BTU cost is charged as soon as a chunk is stored on a long-term storage, no additional cost is added for the long-term storage as long as the BTU time is not over. After 160 h, the cost of the global optimization with the (2,3) and (3,4) configurations is again the same as for the baseline. From this point in time, the global optimization runs with those two configurations provide cheaper placement solutions than the baseline and the other optimization approaches. Since the global optimization with the (2,4) configuration was already at the beginning more expensive than the baseline, it stays more expensive also after this point in time.

After 190 h, all additional cost of the global optimization runs regarding the BTU of the long-term storages is charged and therefore the cost graph raises faster again. At approximately the same time, the heuristic and local optimization starts to migrate data from standard storages to long-term storages. Similar to the global optimization, this increases the cost due to the additional BTU cost. As shown in the cost graph (Fig. 1), the cost of the heuristic is bigger than for the local optimization. This means that the heuristic migrates more files to long-term storages than the local optimization. This happens because the heuristic performs the migration of all data objects from one class, by the use of the local optimization result for one data object, at once. Furthermore, it can be seen that the global optimization with the (3,4) configuration does again migrate some chunks to long-term storages.

After 300 h, the cost of the heuristic and local optimization is lower than the baseline. As shown in Fig. 1, from this point in time the heuristic requires less cost than the local optimization. After 380 h, it can be seen that the local optimization performs again an optimization of all not or rarely used chunks that increases the cost due to the BTU. However, after a short amount of time the cost is again lower than the baseline.

For the global optimization runs, the cost is not changing drastically after 190 h and after 300 h for the heuristic. This shows us that the selected placement does not need to be further optimized. For the global optimization with the (2,3) configuration, this is also observed in Fig. 2, where after all migrations are done no further changes are needed by the optimization.

Results Altogether at the end of the evaluation, all optimization approaches, except the global optimization with the (2,4) configuration, provide better results, with respect to the cost, than the baseline. In comparison with the baseline, we are able to save 11% with the local optimization, 24.61% with the heuristic, and 31.36% with the global optimization with the (2,3) configuration and 26.98% with the (3,4) configuration. In case of the global optimization with the (2,4) configuration, the cost increases by 1.68% in comparison with the baseline. Nevertheless, this configuration offers the highest availability of all discussed configurations, because it only requires two out of four available storages to read a data object. In comparison, the discussed local optimization needs two out of three available storages. For longer evaluation runs the pricing differences, in comparison with the baseline, would increase even more due to the linear cost increase after the optimization of the not used data objects took place.

5.2 Evaluation scenario 2

The aim of this evaluation scenario is to evaluate the heuristic with a large amount of data objects against the baseline. Since the optimization duration of the global optimization increases exponentially with the amount of data objects, and since the optimization is performed for each data object access, it is infeasible to apply it to this large dataset. Therefore, we did not include the global optimization in this evaluation scenario. The same applies for the local optimization. Since this optimization optimizes each data object after it was accessed and in predefined intervals all not used data objects, a lot of optimizations take place and the optimization duration increases with the amount of data objects. Therefore, also the local optimization is not applicable anymore for this amount of data objects.

For this evaluation scenario, we set the optimization interval of the heuristic approach to each 1,000th data object access. Furthermore, we evaluate the heuristic approach with different class boundaries. Those boundaries are (a) storage quantiles (sq) = {25, 50, 75}, amount of traffic classes (tc) = 4; (b) sq = {25, 50, 75}, tc = 2; and (c) sq = {16, 32, 47, 64, 81}, tc = 4. Furthermore, we set the erasure coding configuration for all approaches to (2,3).

Evaluation Hypothesis The heuristic approach shows a similar behavior as in the first scenario and the evaluation scenarios from our previous work [26]. Therefore, at the beginning no difference is expected between the baseline and the heuristic approach, since both approaches use the cheapest standard storages at the beginning. After some time, depending on the configuration of the heuristic, the heuristic starts to migrate chunks from standard storages to long-term storages, which will again increase the cost due to the BTU. As soon as the BTU is over, the cost will be lower than the baseline and stays lower for the rest of the evaluation.

Evaluation Execution Figure 3 presents the cumulative cost, and Fig. 4 shows the chunk distribution on the different storages with the class boundaries (b).

As observed in Fig. 3, the cost at the beginning of the evaluation is the same for the baseline and the heuristic. This is due to the already cheapest storage selection of the baseline and the fixed storage set that is used for the first upload of a data object in case of the heuristic.

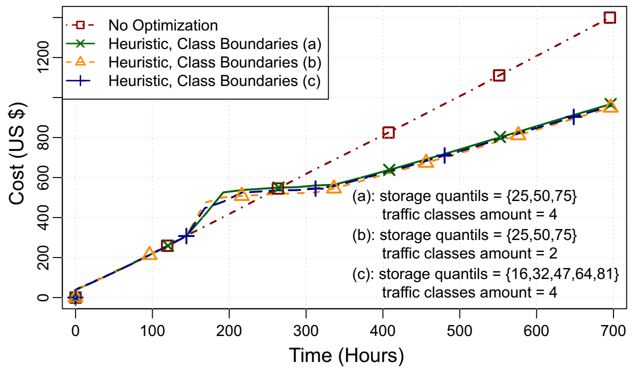

Fig. 3 Cumulative cost for the second evaluation scenario

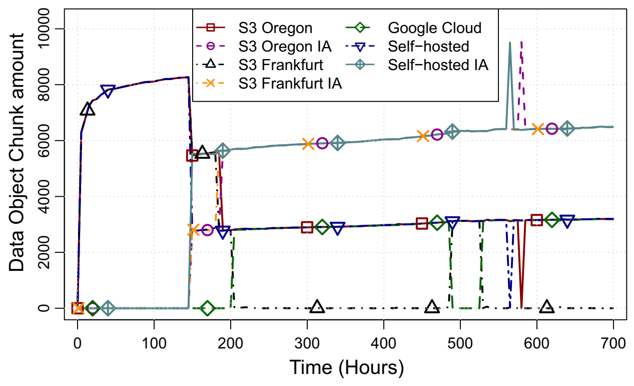

Fig. 4 Data object chunk distribution of the heuristic with class boundaries **b** in the second evaluation scenario

As can be seen after around 150 h, all three heuristic evaluations start to migrate chunks from standard storages to long-term storages. As in the first scenario, this increases the cost due to the BTU of the long-term storages. However, consequently for the remainder of the BTU no additional storage cost is charged. This migration is also observed in Fig. 4. Furthermore, in Fig. 4 it is observed that at 180 h a second migration of some chunks from standard storage to long-term storage takes place.

Figure 3 shows that after 280 h the cost of the heuristic evaluations is lower than the baseline and stays lower for the rest of the evaluation.

In Fig. 4 it is further observed that after 200 h a migration takes place that migrates chunks from Amazon AWS S3 EU Frankfurt to Google Cloud Storage. Furthermore, it can be observed that at 490 h some data chunks are migrated from Google Cloud Storage to AWS S3 EU Frankfurt and back at 520 h. Those migrations are the results of a change of the access patterns. Similar migrations take place at 560 h and 570 h. Since the heuristic optimizes the placement of all chunks in a class by optimizing a representative data object, a change in the access pattern of this data object can result in the migration of multiple chunks. However, as can be seen in the cost graph (Fig. 3) this migration does not have a big impact on the overall cost.

Table 3 Average optimization durations of the first evaluation scenario in milliseconds (standard deviation)

Period of time	Optimization approach			
	Global (2,3)	Global (2,4)	Global (3,4)	Heuristic
2014-01-12 - 2014-01-16	5234.68 (σ = 1874.43)	16545.00 (σ = 8244.95)	18441.40 (σ = 9194.95)	101.64 (σ = 111.40)
2014-01-16 - 2014-01-20	8220.39 (σ = 681.44)	30195.59 (σ = 3895.66)	31273.46 (σ = 3537.07)	66.82 (σ = 57.98)
2014-01-20 - 2014-01-24	8619.54 (σ = 735.15)	31161.40 (σ = 2385.11)	34398.01 (σ = 2018.19)	63.76 (σ = 51.82)
2014-01-24 - 2014-01-28	7848.41 (σ = 551.78)	28316.06 (σ = 1485.12)	32774.72 (σ = 2371.04)	46.02 (σ = 35.39)
2014-01-28 - 2014-02-01	7796.64 (σ = 574.05)	27802.95 (σ = 1832.27)	31537.52 (σ = 1940.05)	50.89 (σ = 46.55)
2014-02-01 - 2014-02-05	8427.38 (σ = 698.55)	32041.39 (σ = 1752.86)	35216.57 (σ = 2029.41)	129.25(σ = 82.02)
2014-02-05 - 2014-02-09	8776.35 (σ = 671.00)	33185.79 (σ = 2757.59)	36707.68 (σ = 2467.75)	83.61 (σ = 37.67)
2014-02-09 - 2014-02-13	8837.22 (σ = 666.09)	35158.29 (σ = 3728.90)	37827.12 (σ = 2075.87)	65.32 (σ = 46.79)

Table 4 Average optimization durations of the second evaluation scenario in milliseconds (standard deviation) with the class configurations defined in 5.2

Period of time	Optimization approach		
	Heuristic (a)	Heuristic (b)	Heuristic (c)
2014-01-12 - 2014-01-16	1172.64 (σ = 1197.39)	1975.32 (σ = 1538.12)	2367.85 (σ = 2254.36)
2014-01-16 - 2014-01-20	1674.16 (σ = 1235.99)	1825.43 (σ = 1587.28)	2452.91 (σ = 2356.32)
2014-01-20 - 2014-01-24	2805.21 (σ = 2043.55)	3690.85 (σ = 2509.79)	5010.23 (σ = 3270.17)
2014-01-24 - 2014-01-28	3311.27 (σ = 2042.38)	3243.43 (σ = 2401.33)	5425.47 (σ = 3161.41)
2014-01-28 - 2014-02-01	3073.95 (σ = 2430.86)	3495.16 (σ = 2668.05)	5527.75 (σ = 4072.79)
2014-02-01 - 2014-02-05	2330.78 (σ = 2137.01)	3089.97 (σ = 2841.69)	5250.59 (σ = 3957.38)
2014-02-05 - 2014-02-09	2786.73 (σ = 2208.62)	2871.69 (σ = 2861.58)	4078.78 (σ = 4018.28)
2014-02-09 - 2014-02-13	3588.81 (σ = 2340.87)	4402.05 (σ = 3440.64)	6194.82 (σ = 4666.80)

Result At the end of the evaluation, it can be seen that the heuristic approach requires less cost than the baseline for all class boundaries. In fact the biggest cost saving was achieved by the class boundaries (b) with a cost saving, in comparison with the baseline, of 32.16%. The one with the lowest cost saving was the one with the class boundaries (a) with a cost saving of 30.9% in comparison with the baseline.

5.3 Performance assessment

In the following we evaluate the performance of our implemented middleware and the global and heuristic approach by analyzing the optimization duration and amount of required metadata. A detailed performance analysis of the local optimization approach can be found in our former work [26].

Duration During the runs of the two evaluation scenarios presented in Sects. 5.1 and 5.2, we have logged the duration of each optimization of the global and heuristic approach. Table 3 shows the average optimization duration and its standard deviation in milliseconds of the first scenario and Table 4 the durations of the second scenario with the class configurations defined in 5.2. As observed in Table 3, the global optimization has a long optimization duration, with an average duration of more than 5,000 ms in case of the (2,3) configuration, since each optimization considers all chunks and all possible storage combinations of them. In case of the (2,4) and (3,4) configuration the optimization duration is bigger than 16,000 ms. This is due to the fact that each data object is split into four chunks which results in 752 chunks, in comparison with 564 chunks for the (2,3) configuration. In case of the heuristic approach optimization durations in the first scenario most of the average durations are below 80 ms.

In the second evaluation scenario (Table 4), it can be observed that the average heuristic duration also increases to durations over 1,000 ms for the class configuration (a) and to durations over 2,000 ms for the class configuration (c). This increase is, on the one hand, due to the fact that the heuristic approach uses the local optimization to calculate the best storage set for a class, and on the other hand, due to the pre-processing, i.e., sorting of the data objects, and post-processing, i.e., setting the selected storages for all chunks in a class. The complexity of both sides increases with the amount of data objects. Furthermore, it can be observed that by changing the class configuration the optimization duration increases or decreases. For instance, with the class configuration (a) the average optimization duration is under 3,600 ms; however, with the class configuration (c) most of the optimization durations are over 5,000 ms. Nevertheless, it has to be noted that this optimization technique is only executed in predefined intervals. In case of the values from Table 4, an optimization is done after 1,000 data object accesses.

Metadata Each proposed optimization approach relies on metadata of each data object, e.g., historic information of the chunks, which are stored in a database. Nevertheless, each proposed optimization approach only requires the information of the last BTU time and billing period. This allows the middleware to prune the metadata and decreases the amount of required storage for the metadata. For example, the second scenario, which uses the biggest amount of data objects and therefore has the biggest amount of metadata, stores only around 111 MB of metadata at a time.

6 Related work

In recent years, the redundant storage of data in the cloud has been a vivid field of research. Substantial efforts have been undertaken, however, with some important limitations.

Similar to our own work, *Scalia* aims at minimizing the cost for redundant data storage in the cloud [19]. To achieve this, the system focuses on performing a runtime analysis of the access patterns of the data objects and uses this information to adapt the data placement. For that, the system holds historical access information, e.g., the size of a data object chunk or input and output traffic for each data object, which are then used in a placement algorithm. Similar to CORA, Scalia applies erasure coding. For the placement optimization, Scalia relies on a heuristic to find a cost-efficient data placement solution. The heuristic resembles the well-known multi-dimensional knapsack problem. In comparison with our work, the proposed optimization approaches of Scalia do not provide a local and global optimization solution based on MILP. In addition, Scalia does not include long-term storage solutions and therefore does not recognize BTUs. Further, Scalia also does not include the block rate pricing models of some providers. Instead, it uses a simplified pricing model. As a result, the pricing model applied by Scalia is not completely realistic. Nevertheless, Scalia comes closest to our own work.

Similar to Scalia, *RACS* uses erasure coding to split data objects into several data object chunks and to store them on several cloud providers [2]. In contrast to Scalia and our work, RACS does not monitor the usage of the data objects. Hence, RACS is not able to take this information into account for finding a cost-efficient data placement.

Another cost-efficient multi-cloud storage system is *CHARM* [28]. Similar to our work, CHARM offers the functionality to find the cheapest storage solution from a set of available cloud storage providers to realize high availability and to avoid vendor lock-in. However, in comparison with our work, the system uses two separate redundancy mechanisms, *replication* and *erasure coding*. The system uses the access history of a data object to determine whether the storage cost is lower for one of these two mechanisms. CHARM uses a similar pricing model as Scalia, leading to the same limitations.

MetaStorage uses full replication to store data objects on several cloud storage providers aiming at a high data availability [6]. To distribute data objects among the available providers, the system uses a distributed hash table, which makes MetaStorage highly scalable. In contrast to our work, MetaStorage does not include any optimization of the placement to find the cheapest provider set. Furthermore, all data objects are fully replicated among the different storage providers. This redundancy mechanism raises the amount of needed traffic and storage and therefore increases the cost.

Chang et al. present in their work [10] a mathematical solution to choose the placement of data objects on different cloud storage systems to maximize the availability for a given budget. However, they only take the storage cost into account and do not include the traffic or migration cost. Furthermore, their equations do not include long-term storage solutions. The same also applies for the work of Mansouri et al. [17] where they also present a mathematical formulation of the same problem as in [10]. Nevertheless, they do not include the additional cost for traffic, data migration and long-term storage, as well. By not taking the traffic cost into account, the system cannot react to changes in the usage pattern if, for example, a data object changes from not used to often used.

While the cost of storing data (without transfer cost) is regarded in further approaches [10,17], other approaches do not take into account any cost [8]. Beside the usage of a redundant storage functionality in the cloud, there are also several peer-to-peer (P2P) systems that are offering similar functionalities, like [12,13]. The usage of different redundancy mechanisms, i.e., replication and erasure coding, was also analyzed in P2P settings [22,27].

Apart from Scalia and CHARM, none of the abovementioned approaches provide a cloud-based redundant storage system that monitors the usage of the data objects and dynamically optimizes the placement of the data objects in a cost-efficient way while taking into account SLOs. To the best of our knowledge, none of the discussed approaches include the usage of long-term storages to store not or rarely accessed data objects. Therefore, state-of-the-art solutions, as discussed above, do not recognize BTUs and BSUs. Last but not least, none of the discussed works models the problem using MILP.

7 Conclusion

To use all benefits of cloud storages without the risk of a vendor lock-in, the use of several cloud storages is an obvious choice. In the work at hand, we extended our previous work in this area by formulating the global optimization problem of storing data objects on a set of storages in a redundant and cost-efficient way. To overcome the data object amount limit of the global optimization approach, we further formulated a heuristic approach for the global optimization based on a classification solution. In the end, we evaluated the two new approaches and compared the results with placement solutions without an optimization but with a fixed provider set and the local optimization approach from our former work [26].

In our evaluation, we showed that our solutions provide less cost in comparison with the baseline. Further, we presented that our heuristic approach can also handle larger amount of data despite the complexity of the optimization problem.

In our future work, we plan to improve our approaches even more. This will include the analysis of different approaches to predict the usage of the data more precisely so cheaper storage solutions can be found earlier. To get a more dynamic solution, we plan to extend our approach in a way that it can dynamically adapt its configuration parameters, e.g., erasure coding configuration or optimization intervals, to elastically adapt to different situations. In regard to the architecture of the middleware CORA, we plan to extend it to a decentralized architecture that is capable to serve several requests in parallel and to increase the resilience of the middleware. Furthermore, we plan to evaluate our approaches in different real-world scenarios, e.g., big data processing.

Besides the cost efficiency, the privacy and latency aspects of cloud storages play important roles. We plan to analyze how the privacy aspects can be considered by using hybrid cloud storage solutions and encryption. With respect to the latency, we plan to extend the optimization approaches in a way that they also consider latency aspects during the data object placement optimization. This extension then minimizes, besides the cost, also the traffic latencies, i.e., download and upload latency.

Acknowledgements Open access funding provided by TU Wien (TUW). This work is partially supported by the Commission of the European Union within the CREMA H2020-RIA project (Grant agreement no. 637066) and by TU Wien research funds.

References

1. Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/41926/. Accessed: March 2017
2. Abu-Libdeh H, Princehouse L, Weatherspoon H (2010) RACS: A case for cloud storage diversity. In: 1st ACM Symposium on Cloud Computing, pp. 229–240
3. Alhamad M, Dillon T, Chang E (2010) Conceptual SLA framework for cloud computing. In: 4th IEEE International Conference on Digital Ecosystems and Technologies, pp. 606–610
4. Allen E, Morris C.M (2009) Library of congress and duracloud launch pilot program using cloud technologies to test perpetual access to digital content. In: Library of Congress, News Release. Accessed: October 2015
5. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) Above the clouds: a berkeley view of cloud computing. Comm ACM 53(4):50–58
6. Bermbach D, Klems M, Tai S, Menzel M (2011) MetaStorage: A Federated Cloud Storage System to Manage Consistency-Latency Tradeoffs. In: IEEE International Conference on Cloud Computing, pp. 452–459
7. Bermbach D, Kurze T, Tai S (2013) Cloud federation: Effects of federated compute resources on quality of service and cost. In: 2013 IEEE International Conference on Cloud Engineering, pp. 31–37
8. Bowers K.D, Juels A, Oprea A (2009) HAIL: A high-availability and integrity layer for cloud storage. In: 16th ACM Conference on Computer and Communications Security, pp. 187–198
9. Butler B Gartner (2015) Top 10 cloud storage providers. Accessed: http://www.networkworld.com/article/2162466/cloud-computing/cloud-computing-gartner-top-10-cloud-storage-providers.html
10. Chang C.W, Liu P, Wu J.J (2012) Probability-based cloud storage providers selection algorithms with maximum availability. In: 41st International Conference on Parallel Processing, pp. 199–208
11. Chen YFR (2015) The growing pains of cloud storage. IEEE Int Comput 19(1):4–7
12. Chun B.G, Dabek F, Haeberlen A, Sit E, Weatherspoon H, Kaashoek M.F, Kubiatowicz J, Morris R (2006) Efficient replica maintenance for distributed storage systems. In: 3rd Conference on Networked Systems Design and Implementation, pp. 45–58
13. Dabek F, Kaashoek M.F, Karger D, Morris R, Stoica I (2001) Wide-area cooperative storage with cfs. In: 18th ACM Symposium on Operating Systems Principles, pp. 202–215
14. Gracia-Tinedo R, Tian Y, Sampé J, Harkous H, Lenton J, García-López P, Sánchez-Artigas M, Vukolic M (2015) Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end. In: 2015 ACM Conference on Internet Measurement Conference, pp. 155–168
15. Gupta P, Seetharaman A, Raj JR (2013) The usage and adoption of cloud computing by small and medium businesses. Int J Inf Manag 33(5):861–874
16. Mahanti A, Eager D, Williamson C (2000) Temporal locality and its impact on web proxy cache performance. Perf Evalu 42(2):187–203
17. Mansouri Y, Toosi A.N, Buyya R (2013) Brokering algorithms for optimizing the availability and cost of cloud storage services. In: 5th International Conference on Cloud Computing Technology and Science, pp. 581–589
18. Naldi M, Mastroeni L (2013) Cloud storage pricing: a comparison of current practices. In: 2013 International Workshop on Hot Topics in Cloud Services, pp. 27–34
19. Papaioannou T.G, Bonvin N, Aberer K Scalia (2012) An adaptive scheme for efficient multi-cloud storage. In: International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 20:1–20:10
20. Pettey C.: Gartner says that consumers will store more than a third of their digital content in the cloud by 2016. www.gartner.com/newsroom/id/2060215. Accessed: March 2017

21. Plank J.S (2013) Erasure codes for storage systems: A brief primer. Login: The USENIX Magzine pp. 44–50
22. Rodrigues R, Liskov B (2005) High availability in DHTs: Erasure coding vs. replication. In: 4th International Conference on P2P Systems, pp. 226–239
23. Satzger B, Hummer W, Inzinger C, Leitner P, Dustdar S (2013) Winds of change: from vendor lock-in to the meta cloud. IEEE Int Comput 1:69–73
24. Schnjakin M, Metzke T, Meinel C (2013) Applying erasure codes for fault tolerance in cloud-raid. In: 2013 IEEE 16th International Conference on Computational Science and Engineering, pp. 66–75
25. Vernik G, Shulman-Peleg A, Dippl S, Formisano C, Jaeger M.C, Kolodner E.K, Villari M (2013) Data on-boarding in federated storage clouds. In: 6th International Conference on Cloud Computing, pp. 244–251
26. Waibel P, Hochreiner C, Schulte S (2016) Cost-efficient data redundancy in the cloud. In: 9th International Conference on Service-Oriented Computing and Applications, pp. 1–9
27. Weatherspoon H, Kubiatowicz J (2002) Erasure coding vs. replication: A quantitative comparison. In: Revised Papers from the First International Workshop on P2P Systems, pp. 328–338
28. Zhang Q, Li S, Li Z, Xing Y, Yang Z, Dai Y (2015) CHARM: A cost-efficient multi-cloud data hosting scheme with high availability. IEEE Trans Cloud Comput 3(3):372–386