SkyPIE: A Fast & Accurate Oracle for Object Placement

TIEMO BANG, University of California, Berkeley, USA
CHRIS DOUGLAS, University of California, Berkeley, USA
NATACHA CROOKS, University of California, Berkeley, USA
JOSEPH M. HELLERSTEIN, University of California, Berkeley, USA

Cloud object stores offer vastly different price points for object storage as a function of workload and geography. Poor object placement can thus lead to significant cost overheads. Prior cost-saving techniques attempt to optimize placement policies on the fly, deciding object placements for each object individually. In practice, these techniques do not scale to the size of the modern cloud. In this work, we leverage the static nature and pay-per-use pricing model of cloud environments to explore a different approach. Rather than computing object placements on the fly, we *precompute* a SkyPIE oracle—a lookup structure representing all possible placement policies and the workloads for which they are optimal. Internally, SkyPIE represents placement policies as a matrix of cost-hyperplanes, which we effectively precompute through pruning and convex optimization. By leveraging a fast geometric algorithm, online queries then are *1 to 8 orders of magnitude faster* but as accurate as Integer-Linear-Programming. This makes exact optimization tractable for real workloads and we show >10x cost savings compared to state-of-the-art heuristic approaches.

CCS Concepts: • **Information systems** → **Cloud based storage**; *Data replication tools*; Remote replication; Data analytics; • **Mathematics of computing** → *Solvers*; • **Computer systems organization** → *Cloud computing*.

Additional Key Words and Phrases: object placement, data placement, offline, exact, cloud oracle

ACM Reference Format:
Tiemo Bang, Chris Douglas, Natacha Crooks, and Joseph M. Hellerstein. 2024. SkyPIE: A Fast & Accurate Oracle for Object Placement. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 55 (February 2024), 27 pages. https://doi.org/10.1145/3639310

1 INTRODUCTION

Modern cloud services rely heavily on object stores like AWS S3, Azure Blob Storage or Google Cloud Storage, due to their high convenience [6, 12, 27, 32, 44, 45, 66]. Besides the simple API, these storage systems offer unbounded elastic capacity, only charge for actual usage (pay-per-use), and are globally available. Cloud services hence have ample opportunity to optimize their service quality and storage costs through object placement and replication across these object stores. For example, objects can be placed close to the clients—across many cloud regions and possibly across multiple clouds [46, 55, 79].

Unfortunately, there is no easy way especially for large-scale users to match their usage patterns to optimal placement policies and many end up overpaying by large margins; costs vary across providers, regions and service levels, and depend on workload features like the network costs between the clients making requests and the location of object stores. However, the pay-per-use

Authors' addresses: Tiemo Bang, University of California, Berkeley, USA, tbang@berkeley.edu; Chris Douglas, University of California, Berkeley, USA, chris_douglas@berkeley.edu; Natacha Crooks, University of California, Berkeley, USA, ncrooks@berkeley.edu; Joseph M. Hellerstein, University of California, Berkeley, USA, hellerstein@berkeley.edu.

ACM 2836-6573/2024/2-ART55
https://doi.org/10.1145/3639310

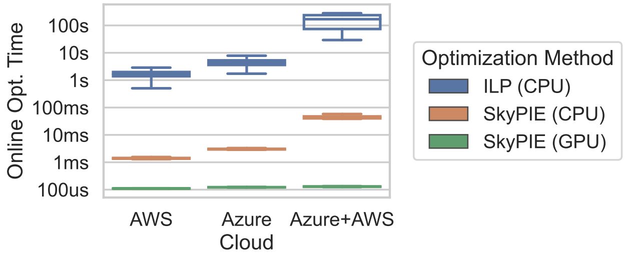


Fig. 1. Object placement via Integer-Linear-Program (ILP) versus SkyPIE: A comparison of the online computation time for the placement of a single object on the object stores of the indicated cloud vendors.

costs models are linear functions over the basic workload features—one can easily compute the storage costs of each object. In theory, it should be possible to choose the perfect mix of object stores and replicas to minimize dollar cost.

To date, the NP-hard problem of optimal object placement has defied practical solutions that are both acceptably fast and accurate. The best way to minimize network costs and achieve latency SLOs is often to *replicate* objects to multiple stores, but this introduces exponential blow-up in the number of available object stores—reaching $\sim 10^{465}$ choices across AWS and Azure today[1].

The state of the art in this problem is still nascent. Current solutions optimizing object placement include pricing tools [17, 69] and full-fledged object replication systems [2, 3, 7, 8, 47, 49, 78], but none of them can accurately optimize cloud-scale workloads. ILP-based *optimizers* [5, 53, 56, 71, 78] compute exactly the optimal solutions, but with exponential time complexity [48]. As shown in Figure 1, their optimization time grows drastically with the number of available object stores, exceeding 100 seconds to compute the placement for a single object. Accordingly, faster heuristics and approximates have been proposed, such as SpanStore's workload aggregation, partitioning the ILP or reinforcement learning [1, 20, 25, 39, 40, 50, 76, 78]. These, however, offer only loose guarantees on placement quality, thus can lead to storage bills *many times* of what an optimal placement could produce on real-world workloads, as we will see in §6.

Despite the inherent complexity, we show that this trade-off is not fundamental in practice: we can achieve optimal object placement at the scale of the modern cloud with reasonable compute time. Our insight begins with observing two key properties of cloud object storage. The first is that storage and serving costs in these systems change very slowly (on the order of months), so we can assume *a priori* knowledge of those costs. The second is that for the consumer, the elasticity of cloud storage is effectively unbounded, so we can ignore transient issues of system load in our optimization. These observations lead us to a counter-intuitive, eager approach. Rather than optimizing placements online, we eagerly construct an *oracle* for any placement question—an exhaustive and accurate lookup structure.

We embrace precomputation, "flipping the script" on the problem. A traditional optimizer computes: $p = \arg\min_{p_i \in P} cost_{p_i}(\vec{w})$ on demand, a function $Opt\colon \mathbb{W} \rightarrow P$ from *object signatures* $\vec{w}$ (read/write frequency per client, object size, etc.) to the optimal *placements* p (choice of object store, region, replication factor). Instead, we eagerly compute *all optimal* pairs $(\vec{w}, p) \in \mathbb{W} \times P$. Because we are being exhaustive, we can build our oracle by working "backwards", computing $Opt^{-1}\colon P \rightarrow 2^{\mathbb{W}}$ for all of P, mapping each placement to the set of object signatures for which it is optimal. We then retain only those placements in P that are optimal for some $\vec{w} \in \mathbb{W}$. In the forward direction, our representation of the oracle can be queried for any given $\vec{w}$ and accurately returns the optimal placement like previous ILP-based optimizers. However, as shown in Figure 1, query times are orders of magnitude faster for placement in a single- and multi-cloud scenarios.

[1]

A natural concern with this approach is the cost of precomputing the oracle. In this paper, we propose the SkyPIE algorithm that scales computation of oracles to hundreds of object stores and replication factor 5. This oracle computation requires only few hours of parallel computation, thanks to careful optimization and embarrassing parallelism. Also, the resulting oracles fit well into memory—even of today's GPUs (<32GB). When the cloud setting does eventually change, a new SkyPIE can be computed in a few hours and redistributed.

The technical underpinnings of our work rest on the linearity of the unbounded, pay-per-use cost functions. In a nutshell, each object signature $\vec{w} \in \mathbb{W}$ has an associated cost under any placement p—a point in $\mathbb{W} \times cost$ space. We illustrate a simple version of this space in Figure 2. Since the cost function of any placement p is linear in its inputs, the cost of a placement p for all object signatures $\vec{w}$ forms a *hyperplane*. Different placements will have different (linear) cost formulae, resulting in many hyperplanes in the space. Our oracle-building approach computes the lowest-cost placements of all possible workloads (§4).

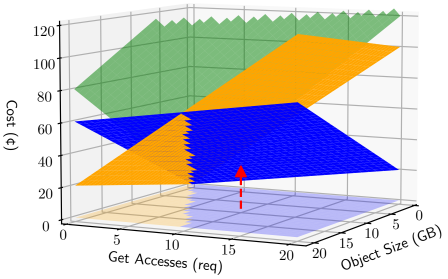


Fig. 2. A simplistic workload/cost space with two workload dimensions $\mathbb{W} = (\texttt{object size} \times \texttt{get accesses})$ and the cost of three placements (green, blue, and orange planes). The lowest-cost placement for each point $\vec{w} \in \mathbb{W}$ is projected down to the bottom (cost=0) to show what ray-shooting would choose for each point in the workload.

To place an object with vector $\vec{w}$ optimally, we want to choose the lowest-cost placement for $\vec{w}$. To query the oracle for a $\vec{w}$, we use geometric techniques to "shoot a ray" from the floor of the space (i.e. $[\vec{w}, 0]$) upward along the cost axis; the hyperplane that we hit first identifies the lowest point $[\vec{w}, z]$ and hence the cost-optimal placement policy. Figure 2 colors the contiguous regions on the floor of the space according to the plane that each ray would hit first.

In summary, we make three contributions:

(1) An inverse perspective on the object placement problem enabling the construction of an oracle.
(2) Effective precomputation of a compact oracle, SkyPIE, including all optimal placements.
(3) Fast and accurate querying of specific optima in SkyPIE by geometric computation.

Artifacts are available on GitHub: https://github.com/hydro-project/cloud_oracle [18].

2 TWO PERSPECTIVES ON OBJECT PLACEMENT

We next highlight the benefits of precomputing the SkyPIE oracle. The *traditional approach* is to take a workload with the access patterns and sizes of all objects (object signatures) and iteratively compute the optimal placements: $\forall \vec{w} \in \mathbb{W} : opt(\vec{w}) = p_{\vec{w}}$. Our *inverse approach* instead starts from the placements and maps them back to the object signatures for which they are optimal: $\forall p \in P : opt^{-1}(p) = \mathbb{W}_p$, where $\bigcup \mathbb{W}_p = \mathbb{W}$. This inverse strategy avoids repeated optimization and embraces eager computation: we precompute a universal lookup structure (an "oracle") that represents all optimal placements for all possible object signatures.

We contrast these two approaches for the simple object placement problem of storage tiering between *S3 Standard (Std.)* and *S3 Infrequent Access (IA)*, similar to S3's intelligent tiering [64]. Assume a cloud service like Slack that stores and reads (does not write) images on S3 [6], as listed on the left of Table 1. The pricing of S3 Std. targets small frequently accessed objects and the pricing of the S3 IA targets large infrequently accessed objects, as our simplified prices on the top of Table 1 resemble.

Accordingly, a placement optimizer must consider the number of gets and the size of an object signature to find the cheapest storage tier. Object placement generally considers replication of an object to several object stores, so we consider the replicated placement on both S3 Std. and IA for exposition. Note that, this simplified example disregards the network transfer costs and the request source (cloud region). In the general, cost-optimal routing of get requests to object replicas is an important aspect of the placement problem, as we detail in §3.

Object signatures	**Placements with pay-per-use costs**		
	p_1={*S3 Std.*} [3¢/GB, 1¢/get]	p_2={*S3 IA*} [1¢/GB, 5¢/get]	p_3={*S3 Std., S3 IA*} [4¢/GB, 6¢/get]
$\vec{w}_1$: [3GB, 8gets]	$\underline{17¢}$	43¢	60¢
$\vec{w}_2$: [6GB, 2gets]	20¢	$\underline{16¢}$	36¢
$\vec{w}_3$: [9GB, 10gets]	$\underline{37¢}$	59¢	96¢

Table 1. Example of storage tiering as placement optimization for 3 images with object signatures $\vec{w}_1$–$\vec{w}_3$ and object stores *S3 Standard (Std.)* and *Infrequent Access (IA)* under simplified pay-per-use pricing for storage and get requests.

2.1 Traditional: From Workload to Placement

For every object signature in the workload, current optimizers compute the serving costs of each object store. Then they feed an Integer-Linear Program (ILP) into a solver to compute the combination of object stores that minimizes the total serving cost for a particular object signature—its optimal placement.

ILP solvers, in essence, search subsets of the available object stores for the choice of object stores that has the least serving costs. Despite the fact that ILPs search the same object stores for every object signature $\vec{w}_i$, solvers have to start the search from scratch, because each $\vec{w}_i$ can have distinct serving costs, e.g., as shown in Table 1.

ILP computation is a complex search and remain specific to a specific object signature. The online optimization overhead is exponential in the number of object stores [48]. To make matters worse, reuse is limited and heuristic-based [10, 29, 36, 41]. Such an approach is clearly infeasible in today's cloud landscape, which consists of hundreds of object stores, serving workloads with millions of objects.

Heuristics such as relaxed ILPs, reinforcement learning or optimizing over aggregate data like SpanStore can reduce computation overhead [1, 25, 39, 40, 50, 76, 78] but come with significant risks: their optimization quality is volatile—it depends on the workload with at best loose guarantees. For instance, SpanStore would aggregate $\vec{w}_1$ and $\vec{w}_2$ via the element-wise sum, yielding the aggregate object size and accesses like $\vec{w}_3$. However, the cheapest placement p_2 of $\vec{w}_3$ costs 37¢ rather than 17¢+16¢=33¢ of $\vec{w}_1$ and $\vec{w}_2$ individually. In reality, this error further increases when replicating to multiple object stores, e.g., reaching >6x for our real-world trace in §6.3.

2.2 SkyPIE's Inverse Approach

From Placement to Workload. In this work, we go backwards. We start from the placements and map these to cost functions that are independent of a specific workload. Based on these cost functions, we compute which placements have an optimal object signature, i.e., achieve lowest cost for some workload. All these optimal placements together make up the SkyPIE oracle, which we can query online for the lowest-cost placement of any given object signature.

Figure 3 illustrates our inverse approach. Given the object stores o_1 and o_2 and their pay-per-use pricing, we can construct the placements p_1–p_3 with corresponding cost functions:

$$cost_{p_1}(\vec{w}) = 3¢ * \vec{w}^{size} + 1¢ * \vec{w}^{gets} \tag{1}$$

$$cost_{p_2}(\vec{w}) = 1¢ * \vec{w}^{size} + 5¢ * \vec{w}^{gets} \tag{2}$$

$$cost_{p_3}(\vec{w}) = 4¢ * \vec{w}^{size} + 6¢ * \vec{w}^{gets} \tag{3}$$

These linear functions can be modeled as *planes* in workload-cost space, with the z-axis for cost and axes for each object signature dimension in $\vec{w}$, e.g., object size and number of gets. We plot these planes in Figure 3 and project on the "floor" (z=0) the color of the lowest plane for each (x,y) point. It is then straightforward to determine which placement is cheapest for a given object signature: given the corresponding workload coordinates, the colored projection reveals the best solution. We can see that along the purple line at coordinates of object size 3GB and 8 `gets` ($\vec{w_1}$) the blue plane of p_1 is the lowest. Right next to it in workload space at 6GB and 2 `gets` (aquamarine line), one can quickly see that $\vec{w_2}$ also has p_1 as its lowest plane—both have p_1 as their optimal placement. We can see that among all the planes, only p_1 (blue) and p_2 (yellow) achieve lowest cost for some object signature. Hence, **the key idea of our inverse approach is a subset of placements and their planes suffice to optimize every object signature**.

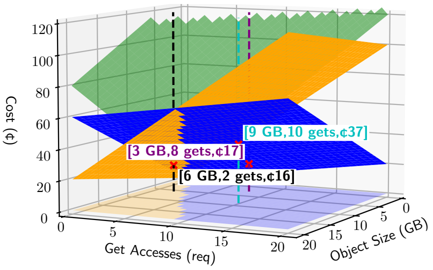


Fig. 3. SkyPIE's inverse, from placement to workload: Cost planes of placements specify the cost for all workload and the lowest plane at the coordinates of an object signature identifies the optimal placement.

Precomputation of the SkyPIE Oracle. At first blush, the precomputation of the SkyPIE oracle seems prohibitively expensive. Three key findings allow us to leverage efficient pruning and convex optimization that make precomputation tractable:

(1) ***Offline Information.*** We observe that precomputation is actually possible and worthwhile. The available object stores, their pricing, SLOs, etc. are known *a priori* and change relatively infrequently. Moreover, the elasticity of object stores makes us independent of the load at runtime. It is thus possible to enumerate placements, derive cost planes and precompute offline.

(2) ***Few Optimal Placements.*** Despite a massive space of placements, relatively few placements are ever optimal. In our simple example, we can disregard the aquamarine p_3 plane and still find the lowest cost point of every object signature. We empirically find that there are many high-cost placements which precomputation can sift out.

(3) ***Leveraging Convexity.*** Naïvely comparing the costs of all placements to find the lowest-cost ones would be intractable. Luckily, since their cost planes are linear and continuous, the space underneath all the cost planes necessarily forms a convex polyhedron. We can thus rely on *linear* convex optimization, which is still exact but significantly cheaper than the discrete optimization necessary to solve ILPs.

Building on these observations, we design the tractable SkyPIE algorithm to precompute oracles in §4. For coarsely discretized ("T-shirt sized") SLOs and replication factors, a vendor can even precompute default oracles that can be shared and reused, as we discuss in §8 "Oracle Lifetime".

Querying the SkyPIE Oracle. After building the SkyPIE oracle, the next step is to query that oracle fast and accurately—in terms of Figure 3, to find the color of a point on the cost floor. This must remain time- and space-efficient in many dimensions. To this effect, we recast the problem as a computational geometry question: the optimal placement policy in Figure 3 corresponds to the plane that intersects each signature's vertical line at the lowest point. More generally, we can construct geometric queries to operate on a compact matrix, reducing them to matrix-vector computations that are cheap and embarrassingly parallel (see §5).

One might be tempted to employ traditional spatial search structures such as BSP-trees, kd-trees, R+-trees, etc. [21, 22, 34, 63], but their space overhead, construction time or query time grow exponentially with the number of dimensions in the worst case.

3 THE OBJECT PLACEMENT PROBLEM

We now formalize this object placement problem— how can we place an object on one or more object stores given its object signature, so that serving costs are minimal and SLOs are fulfilled. The problem definition summarized in Table 2 reflects the definition used by related traditional optimizers [11, 56, 71, 74, 78]. Table 2d further details our cost model.

Table 2. Definition of the object placement problem.

(a) Input parameters of the object placement problem.

Parameter	Definition
Object stores	$\mathbf{O}$, object stores available for object placement
Pay-per-use pricing	$\vec{\mathbf{c}}_\mathbf{o}$, cost feature per metered workload feature
App cloud regions	$\mathbf{A}$, cloud regions of apps accessing objects
Observed workload	$\vec{\mathbf{w}} \in \mathbb{W}$, object signature vectors (see Table 2d)
Replication factor	$\mathbf{f}$, the least number of object stores to choose
Max. replication factor	$\mathbf{f_{max}}$, the most number of object stores to choose
Latency SLOs	$\mathbf{SLO_{per}}$, SLOs on access latency by percentile
Latency profile	$\mathbf{l_{r,o,per}}$, latency between each application region r and object o by percentile

(b) Definition of placements.

Parameter	Definition
Placement (policy)	$p{:=}(W, R)$, pair of write choice W and read choice R.
Write choice	$W \subseteq O$, choice of object stores each storing (replicas of) objects
Read choice	$R{:=}\{(r, o) \mid \forall r \in A : \exists o \in O\}$, assignments of region r to read objects placed on object store o
Pay-per-use pricing	$\vec{\mathbf{c}}_\mathbf{p}$, cost feature vector composed of object stores' pay-per-use pricing (see Table 2d)
Cost	$\vec{\mathbf{c}}_\mathbf{p} \cdot \vec{\mathbf{w}}$, cost of p for object signature $\vec{w}$

(c) Object placement optimization model.

Optimization Model	
Objective	$\arg\min_{p \in P} \quad \vec{c_p} \cdot \vec{w}, \forall \vec{w} \in \mathbb{W}$
subject to:	
All regions assigned	$\forall r \in A : \exists o \in W : (r, o) \in R$
Replication fulfilled	$\lvert W \rvert \geq f$
Latency SLOs fulfilled	$l_{r,o,per} \leq SLO_{per}, \forall (r, o) \in R$
~~Capacity~~	

(d) Object signature and cost features for placement.

	Object signature (Workload) $\vec{w}$	Cost $\vec{c_p}$
Object size	w^{size}	$c^{size} := \sum_{o \in W} c_o^{size}$
put op.	$w^{put} := \sum_{r \in R} w_r^{put}$	$c^{put} := \sum_{o \in W} c_o^{put}$
get op.	w_r^{get}	$c_r^{get} := c_o^{get} : (r, o) \in R$
Net. ingress	$w_r^{in} := w_r^{put} * w^{size}$	$c_r^{in} := \sum_{o \in W} c_{o,r}^{in}$
Net. egress	$w_r^{out} := w_r^{get} * w^{size}$	$c_r^{out} := c_{o,r}^{out} : (r, o) \in R$

$$\vec{w} := (\quad w^{size}, \quad w^{put}, \quad w_r^{get}, \dots, \quad w_r^{in}, \dots, \quad w_r^{out}, \dots)$$

$$\vec{c_p} := (\quad c^{size}, \quad c^{put}, \quad c_r^{get}, \dots, \quad c_r^{in}, \dots, \quad c_r^{out}, \dots)$$

3.1 Problem Definition

Input Parameters. As listed in Table 2a, the input is the set of available object stores O (spanning storage tiers, cloud regions, cloud vendors), their associated pay-per-use pricing $\vec{c}_o$, as well as the list of cloud regions A from which applications will access these object stores. Typically, users will determine A based on the regions where their applications are deployed and will determine O based on their overall intended storage deployment, e.g., single-cloud versus multi-cloud. We associate each object in the system with an object signature capturing all billable workload features (like put or get operations), as we describe in our cost model in §3.2. We refer to the aggregate set of all object signatures as observed workload $\mathbb{W}$. The remaining inputs allow the user to parameterize placements to be considered as valid solutions, as we will detail.

Placement Policies. Given these inputs, *placements* $p \in P$ are defined via a *write choice* W and *read choice* R. W is a set of object stores that defines the chosen "locations" of an object. R is a mapping of regions to object stores that defines, for each region $r \in R$, from which object store an application should read. Here, we assume that a full copy is stored in each chosen object store $o \in W$ and that every write updates all copies but only reads from one.[2]

[2] We focus on a single-reader multiple-writer replication model for brevity. Other replication models like erasure-coded storage or reading a quorum would change the definition of placements and the cost model but would not contradict our approach, since these placements still can be enumerated and still have a linear cost function over the workload.

The pay-per-use pricing $\vec{c_p}$ of a placement reflects the costs for storing, writing, and reading an object, which depend upon both the write choice and read choice as well as the pay-per-use pricing of the chosen object stores. The cost of a placement p for a particular object signature $\vec{w}$ is the dot product $\vec{c_p} \cdot \vec{w}$—a linear function. We detail the cost model in §3.2.

Optimization Model. Our goal is to identify, for each object signature, the placement with lowest cost. More formally, $\forall \vec{w} \in \mathbb{W}$, we need to find $\arg\min_{p \in P} \ \vec{c_p} \cdot \vec{w}$ where the write choice and read choice of p have minimal costs and are valid: (1) the read choice has to include an assignment for all application regions, (2) the write choice has to meet the minimum replication factor f, and (3) the chosen object stores must be reached within the specified SLOs by all regions. Further constraints on performance metrics are conceivable, as we will discuss in §8.

Complexity: NP-Hard. This optimization problem is NP-hard as it is equivalent to the *Uncapacited Facility Location* (UFL) problem with general cost functions—and reduces to the set cover problem [42]. Write choices are equivalent to choosing which facilities to open and read choices are equivalent to choosing which open facility serves which client. The storage and `put` costs correspond the facility opening costs and the `get`/ingress/egress costs to the service costs—costs by multiple variables of general cost functions.

3.2 Cost Model

The example of §2 was intentionally simple. In reality, modern cloud object stores charge for used storage capacity, operations (`put`, `get`, etc.), and network ingress/egress. The cost model in Table 2d reflects all these features in the workload feature vector $\vec{w}$ of the object signature and cost feature vector $\vec{c_p}$ of the placement. In fact, the prices for network ingress/egress even differ for application regions, so that the cost model further distinguishes *global* and *region-specific* features.

- ***Object Size—Global.*** The cost of storing an object can be directly inferred from the object size and the storage costs of each chosen object store in W.
- **`put`** ***Operations—Global.*** `put` operations have a single feature in both $\vec{w}$ and $\vec{c_p}$ as every `put` operation writes to all chosen object stores independent of its origin. The cost thus is the sum of the cost of each individual write.
- **`get`** ***Operations—Region.*** In contrast, `get` operations have a feature per region, as the application regions read from a single object store according to the read choice R. Their cost is thus directly the cost of reading from that object store.
- ***Network Ingress/Egress—Region.*** Network ingress corresponds to the data volume emanating from a specific application region into all object stores, as a result of `put` operations. Conversely, network egress refers to the data volume reaching a given application region following a `get` operation. Although these data volumes are based on the object size and number of operations, we consider these as distinct features to obtain a linear model.[3] To capture all dimensions of transfer cost across regions, availability zones, and data centers—including intra-region transfers, when the cost coefficient is zero—each of these is a feature per region.

Note that, the region-specific features yield high-dimensional object signatures. Vectors $\vec{c_p}$ and $\vec{w_i}$ grow proportional to the number of application regions in A ($|\vec{c_p}| = |\vec{w_i}| = 2 + 3 * |A|$)—reaching tens to hundreds of dimensions. Finally, it is noteworthy that this cost model is straightforward to extend with linear cost components.

[3] The network data volumes $w_r^{put} * w^{size}$ and $w_r^{get} * w_{size}$ are products—quadratic terms. We can substitute for new workload features w_r^{in} and w_r^{out} without loss of generality: $c_r^{in} * w_r^{put} * w^{size} = c_r^{in} * w_r^{in}$ and $c_r^{out} * w_r^{get} * w^{size} = c_r^{out} * w_r^{out}$.

4 PRECOMPUTING THE SkyPIE ORACLE

The SkyPIE oracle, once precomputed, comprises of a set of object placements that can be quickly queried to find the exact cost-optimal placement of any given object signature. To ensure the accuracy (exactness) of these queries, all "winning" placements must be included in the oracle—winning placements are those which are optimal for at least one object signature. Conversely, to ensure the efficiency (speed) of these queries, it is preferable to include as few non-winning policies as possible. While non-winning policies do not affect the query's accuracy thanks to the convex nature of the problem, their presence enlarges the oracle, thus slowing down queries.

4.1 Precomputation by Inverse Enumeration

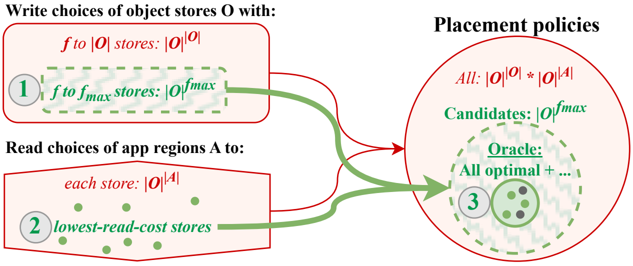

Write choices of object stores O with:
f to $|O|$ stores: $|O|^{|O|}$
1
f to f_{max} stores: $|O|^{f_{max}}$
Read choices of app regions A to:
each store: $|O|^{|A|}$
2
lowest-read-cost stores
Placement policies
All: $|O|^{|O|} * |O|^{|A|}$
Candidates: $|O|^{f_{max}}$
Oracle:
All optimal + ...
3

Fig. 4. Precomputation by Inverse Enumeration.

```
1  def precompute_skypie(O: Object stores, A: App regions, f:rep. factor, f_max:max
      rep. factor) -> Oracle:
2    C = list()
3    # Write choices of f to f_max object stores
4    for W in combinations(O, f..f_max):                                    ▷ O(|O|^f_max)
5      # Candidate placements of write choice W with optimal read choices
6      C += enumerate_candidates(W, A)                                     ▷ O(|A| * f_max^2)
7    Oracle = reduce_oracle(C) # Tighter set of placements            ▷ O(|C| * b * LP(b, |A|))
8    return Oracle
```

Listing 1. SkyPIE precomputation, $O(|O|^{f_{max}})$.

Rooted in our inverse approach, our precomputation algorithm efficiently enumerates candidate placements and then computes a compact oracle. This requires some care, as the space of options to consider during precomputation can be large, as depicted in Figure 4. Specifically, the possible write choices (the red rectangle) are all combinations of at least f and at most $|O|$ object stores: $O(|O|^{|O|})$. The possible read choices (the red hexagon) are all the independent choices of any object store for each app region: $O(|O|^{|A|})$. The cross-product of these two is the massive space of all placements (the red circle). With hundreds of available object stores and cloud regions, precomputation needs to discern over 10^{100} possibilities. We reduce the cost of our algorithm in Listing 1 via the following three high-level steps.

First, we compute the set of possible write choices, assuming that replication is bounded to a small number of at most f_{max} object stores—the green rectangle labeled (1) in Figure 4. This significantly reduces the possible write choices and the complexity of the outer loop in Line 4 from $O(|O|^{|O|})$ to $O(|O|^{f_{max}})$.

Second, we avoid exhaustively enumerating read choices by computing the lowest-cost read choices directly—the green points in the hexagon labeled (2) in Figure 4. As we will discuss in §4.2, this reduces the number of read choices dramatically, constraining the aforementioned cross-product to a tractable, but sufficient set of candidate placements. With $O(|A| * f_{max}^2)$ time complexity this computation in Line 6 is far more efficient than exhaustive enumeration.

Third, after generating candidates, we reduce the size of the oracle using a textbook *redundancy elimination* technique in Line 7—resulting in the small, solid-edged circle labeled (3) in Figure 4. As we discuss in §4.3, we conservatively speed up this technique by applying it to partitions of the candidates. This results in quasi-linear time complexity by solving linearly many LPs of small, constant size b: $O(|C| * b * LP(b, |A|))$.

As a result, our precomputation algorithm achieves $O(|O|^{f_{max}})$ time complexity, while current ILP-based optimizers have exponential complexity in both the number of object stores and cloud regions. Our approach makes precomputation of a single exact oracle tractable, even under hundreds of object stores and cloud regions—i.e., today's entire cloud. We proceed to elaborate on the second and third of these steps in more detail.

4.2 Candidate Enumeration

```
1 def enumerate_candidates(W: Write choice,A: App regions) -> Placements:
2   R = dict()
3   # Enumerate for app region r all optimal read choices with object stores in W
4   parallel for r in A:                                                      ▷ $O(|A|)$
5     R[r] = opt_assignments(W, r)                                            ▷ $O(f_{max}^2)$
6     if R[r].empty(): # No SLO-compliant assignments found
7       return [] # Abort, no SLO-compliant placements exists
8   # Merge to placements of W and the optimal read choices
9   return merge_placements(W, R)                                             ▷ $O(|R| * log(|R|))$
```

Listing 2. Candidate enumeration algorithm, computing per write choice the lowest-cost read choices and combining these to the lowest-cost placements, in $O(|A| * f_{max}^2)$.

Listing 2 presents the high-level algorithm for enumerating candidate placements, before we discuss its subroutines. Recall that a placement is a combination of a write choice (from the red rectangle) and a read assignment (from the red hexagon). Given a bounded write choice C (from the green rectangle), we compute optimal read choices for every possible object signature $w \in \mathbb{W}$ (green points in the red hexagon) in the subroutine `opt_assignments`. Subroutine `merge_placements` then constructs candidate placements p_C (the dashed green circle). Both routines have low complexity, while the former is also embarrassingly parallel, making enumeration tractable for large inputs.

Note that `opt_assignments` ensures mandatory compliance with latency SLOs. When it does not find any SLO-compliant read choice for a region it is not possible to construct any SLO-compliant placement. In this case, candidate enumeration aborts with an empty result for the given write choice (Line 6). We now discuss the two subroutines in turn.

Optimal Assignment Computation. The `opt_assignments` algorithm in Listing 3 computes all the cost-optimal read choices (green points in the red hexagon) associated with the given write choice W and application region r. Note that `opt_assignments` optimizes each region's read choice independently of other regions. Optimizing the read choice per region is much simpler than the overall optimization of placements and permits parallel computation. Assigning a region to an object store of the given write choice establishes `get` and egress costs for that region but does not affect the costs of the read choices of other regions. Since the fixed write choice predetermines the object stores and their costs, there is no trade-off between the costs (`get` and egress) of application regions. Hence, `opt_assignments` can independently compute the cost-optimal read choices for individual application regions—with low complexity and in parallel.

```
1    def opt_assignments(W, r) -> List[Tuple[Range, Store]]:
2      # SLO-compliant object stores in write choice
3      compliant = list(o for o in W if r.is_slo_compliant(o))
4      # Read choices of region r to object store o with optimal size range
5      R = dict(o : Range(0,inf) for o in compliant)
6      parallel for $o_1$,$o_2$ in combinations(compliant,2):              ▷ $O(f_{max}^2)$
7        # Boundary of $o_1$'s and $o_2$'s read choice costs
8        size,o_low,o_high = boundary(r, $o_1$, $o_2$)                        ▷ $O(1)$
9        # Update upper/lower bound of optimal object size range for read choices
10       # to the object stores on the low/high side of the intersection
11       R[o_low].up = min(R[o_low].up, size)
12       R[o_high].low = max(R[o_high].low, size)
13     # Filter object stores with non-empty optimal size range
14     # that make an optimal assignment for r
15     return [(ra,o) for o,ra in R.items() if not ra.empty()]
```

Listing 3. Object stores for optimal read-assignment of region, using cost-optimal object size as proxy for cost-optimal number of get requests and network egress volume.

In Line 3, the algorithm starts by identifying the object stores (`compliant`) whose access latency to region r fulfills the SLOs. This is the point in the overall precomputation algorithm where we enforce SLOs. Any change in latency SLOs or the latency profiles necessitates the precomputation of a new oracle. Subject to those SLOs, the remainder of this routine computes the optimal read choices for all possible object signatures of the given region.

To compute cost-optimal read choices, in principle we need to consider costs in two dimensions for our objects: the number of gets and the network egress volume. Specifically, the cost function for the assignment of region r to object store o_i (Table 2) is:

$$cost^{read}_{o_i,r}(\vec{w}) = c^{get}_{o_i} w^{get}_r + c^{out}_{o_i,r} w^{out}_r \tag{4}$$

We can view these cost functions as planes in space, as illustrated in Figure 5 (a). However, we can reduce the complexity of our algorithm by mapping these three-dimensional planes to a single dimension—object size—as in Figure 5 (b). To understand this intuition better, consider how each plane in Figure 5 (a) shows the cost of read choices of region r_1 to object stores o_1, o_2, o_3: (r_1, o_1), (r_1, o_2), (r_1, o_3). The shading on the "cost floor" indicates which read choice is optimal for each [`get`, *egress*] point: points in the blue-shaded area are optimal for the read choice (r_1, o_1), and those in the yellow-shaded areas are optimal for the read choice (r_1, o_2). There is no optimal point for the read choice (r_1, o_3). Note that the boundaries between the optimal read choices arise from the intersections of the planes above. Our reduction simplifies the computation of the boundaries between optimal read choices as follows. Consider the read choices (r_1, o_1) and (r_1, o_2). The boundary on the cost floor is where their costs are equal:

$$c^{get}_{o_1} w^{get}_r + c^{out}_{o_1,r} \mathbf{w}^{\mathbf{out}}_{\mathbf{r}} = c^{get}_{o_2} w^{get}_r + c^{out}_{o_2,r} \mathbf{w}^{\mathbf{out}}_{\mathbf{r}} \tag{5}$$

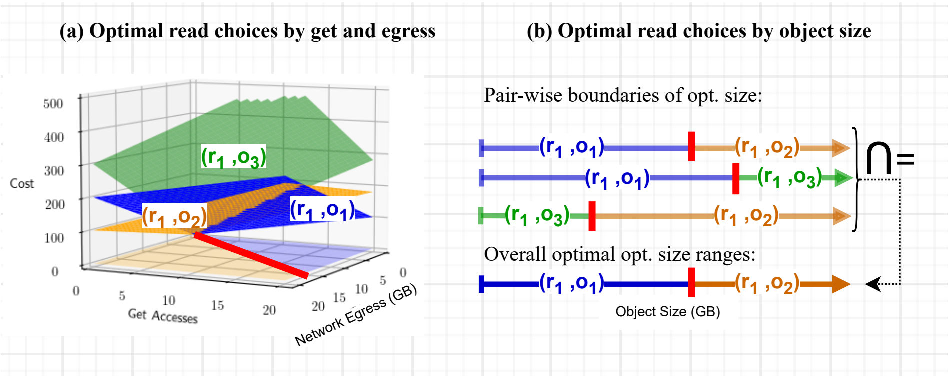


Fig. 5. Optimal workload for read choices for region r_1 to object stores o_1, o_2, o_3: (a) Cost of read choices in 3-dimensional get-egress-cost space with optimal get-egress workload shaded on the floor (z=0). (b) Optimal object size of read choices, equivalent to (a) due to reformulation.

Note that the total egress volume w^{out}_r is equivalent to $w^{get}_r w^{size}$, so we can simplify:

$$c^{get}_{o_1} w^{get}_{r_k} + c^{out}_{o_1,r}\mathbf{w}^{\mathbf{get}}_{\mathbf{r}}\mathbf{w}^{\mathbf{size}} = c^{get}_{o_2} w^{get}_{r} + c^{out}_{o_2,r}\mathbf{w}^{\mathbf{get}}_{\mathbf{r}}\mathbf{w}^{\mathbf{size}} \tag{6}$$

And now we can solve for a single variable w^{size}:

$$\mathbf{w}^{\mathbf{size}} = (c^{get}_{o_1} - c^{get}_{o_2})/(c^{out}_{o_2,r} - c^{out}_{o_1,r}) \tag{7}$$

Effectively, the boundary between two read choices is a single point in one dimension: object size. Consider for example the boundary of the blue and yellow read choices in the first row of Figure 5 (b). Blue dominates yellow in a *half-line* of object sizes: all object sizes lower than the red boundary. Inversely yellow dominates blue in the upper half-line of the boundary.

Now, we can find all object sizes for which blue is optimal by collecting the boundaries of blue with every other read choice—yellow and green — and forming the intersection of the half-lines that result. We repeat this procedure for every read choice (yellow and green). The result at bottom of Figure 5 (b) represents the optimal read choice for every object size. Note that intersection of the green half-lines is empty, as it is never optimal.

Returning to the opt_assignment algorithm of Listing 3, Line 5 initializes a dictionary of optimal ranges for each SLO-compliant read choice. The loop of Line 6 and onwards, computes the boundaries between each pair of read choices and maintains running intersections of the resulting half-lines. Notably, Line 8 computes the boundary and which read choice is lower and which is upper. Finally, the algorithm returns the list of optimal read choices with non-empty size ranges.

Merging Read choices to Candidate Placements. We are now ready to construct candidate placements. Recall from Listing 1, Line 4 that our overall algorithm begins by enumerating all write choices. At this point, we are considering a specific write choice W. Given our computation of the optimal read choices per region R, we are now ready to enumerate the placements of W that jointly optimize read choices across *all* regions.

Given a fixed W and read choices R, merge_placements in Listing 4 now constructs a single placement for every object size. For every object size, it selects the optimal read choice of each region and constructs a placement with the write choice W. Of course, we do not enumerate all object sizes $[0..\infty]$, but instead compute placements for distinct ranges along the object size axis. We do this by segmenting the object size axis into ranges at the finest granularity: all the boundaries in R across all regions.

Specifically, the `merge_placements` algorithm first initializes the joint read choice `cur_R` with the optimal read choice per region for size 0 (Line 5). In Line 7 it loads all read choices `(r,o)` into a list and sorts these by the upper bounds of the previously computed ranges. Walking this sorted list in Lines 10–14, it constructs a placement for every *distinct* range on the size axis (Line 12). Also, it gradually updates the optimal read choice for every boundary stored in `cur_R` (Line 14)—note that two read choices in `cur_R` may share a boundary. The returned candidates `P` are all the placements combining the given write choice W with its optimal read choices across regions.

When `merge_placements` completes for a particular W, control returns to the top-level routine—Line 6 of Listing 1. At that point, we have assembled the set `P` containing the candidate placements with optimized read choices of each write choice. This guarantees that we now have all exact solutions in our oracle: the set `P` includes all optimal placements, taking into account read choices and write choices. The set `P` likely includes suboptimal placements as well; we handle this next in `reduce_oracle`.

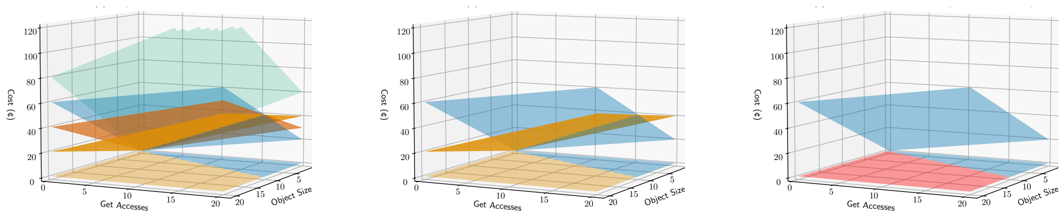
```
1   def merge_placements(W: Write choice, R: Dict[Region, List[Tuple[Range, Store]]])
        -> Placements:
2     P = [] # Candidate placements
3     cur_s = -1 # Current size boundary
4     # Initialize with read choice for 0 object size
5     cur_R = {r: l.pop(argmin(l))[1] for r, l in R.items()}
6     # List of all read choices sorted by upper boundary of optimal size range
7     L = sorted([(up,(r,o)) \                                      ▷ O(|R| ∗ log(|R|))
8       for r,l in R.items() for (_,up),o in l])
9     # Create a placement for all regions at each size boundary
10    for s, (r, o) in L:                                                    ▷ O(|R|)
11      if s > cur_s:
12        P.append(Placement(W, cur_R)) # Placement of cur_s
13        cur_s = s # Move on to next boundary
14      cur_R[r] = o # Read choice of r for cur_s
15    P.append(Placement(W, cur_R)) # Final placement
16    return P
```

Listing 4. Algorithm for merging optimal read choices per region to joint candidate placements.

4.3 Oracle Size Reduction

The efficiency of querying the SkyPIE oracle can be significantly improved by discarding candidate placements that are not in fact optimal. That is, candidates enumerated by `enumerate_candidates` for one write choice which are superseded by candidates of another write choice. As we will explain, dominated placements are equivalent to redundant constraints in Linear-Programs (LPs). The convex optimization literature applies redundancy elimination to speed up LPs without affecting their accuracy [35]. We utilize this technique for our oracle, specifically Clarkson's algorithm [26]. But we apply a relaxation to curb the overhead for large candidate sets.

Utilizing Redundancy Elimination. Figures 6a–c illustrate how redundancy elimination applies to our placements. In the convex optimization literature, non-redundant constraints *tightly bound the convex region* of feasible solutions in LPs. *Redundant* constraints do not further bound the solutions but still incur computation costs, so that redundancy elimination removes these without changing the solutions. Now, recall that the lowest plane at the coordinates of an object signature indicates the lowest-cost placement and consider the space underneath the planes in Figures 6a–b.



(a) Large oracle of candidate placements.

(b) Tight oracle of optimal placements.

(c) Sub-optimal oracle missing placements.

Fig. 6. Framing computation of optimal placements of all workloads as partitioned redundancy elimination.

We realize that the lowest planes (blue and yellow) are *tight bounds* to the space underneath, while the green and orange planes are redundant bounds to that space. If we were to remove the yellow plane as in Figure 6c, the height of the bounded space would increase in the red-shaded area. So, the realization is that the space underneath the planes is a *convex region* like the feasible solutions of LPs, where redundancy elimination can identify the tight bounds that belong to the optimal placements.[4]

Technically, redundancy elimination uses Linear-Programs (LPs) to test if a plane is tight *and linearly independent* compared to all other planes. It utilizes the linear convex optimization algorithms of LPs to efficiently test if a plane has a point that is truly lowest (<) anywhere in the high-dimensional space. Notably, it eliminates the tangential orange plane, as it is only lowest where also the blue and yellow plane are lowest—the orange plane is a tight but linearly *dependent* bound, hence redundant. Only convex optimization algorithms can efficiently test linear dependency in high dimensions. Simpler algorithms outside of convex optimization are ineffective for our purpose, as these are either slow or do not identify placements with linearly dependent planes as non-optimal.

To apply redundancy elimination, we describe the convex region underneath the planes of the candidates. We start by reformulating the cost functions:

$$\textit{Function:} \qquad cost_p(\vec{w}) = \vec{c_p} \cdot \vec{w} \tag{8}$$

$$\textit{Plane:} \qquad (v = \vec{c_p} \cdot \vec{w}) \;\Leftrightarrow\; (0 = \vec{c_p} \cdot \vec{w} - v) \tag{9}$$

$$\textit{Half-plane:} \qquad 0 \geq \vec{c_p} \cdot \vec{w} - v \tag{10}$$

As shown in Eq. 8, a cost function has the policy's cost feature vector as coefficients and the object signature vector as variables. In Eq. 9, we reformulate this cost function to the plane equation in standard form, introducing the variable v in addition to $\vec{w}$. In Eq. 10, we then express the space underneath the plane (a half-plane), by changing the equality to the inequality $\geq$.

Our `reduce_oracle` algorithm in Listing 5 executes this conversion in Line 4 by expanding the cost feature vector by the additional -1 coefficient of v. In Line 6, we further add inequalities to bound the region to positive object signatures and cost, i.e., the positive *orthant*. Otherwise, some expensive placements would appear as tight bounds for "negative workload" and would not be detected as non-optimal. This convex region describes the same space for all sets of candidates that contain the same cost-optimal placements.[5]

[4] The space underneath the planes is a convex polyhedron—an open convex region bound by linear planes. Like every convex object, it has many representations—a minimal one with the tight bounds of the lowest planes and many further ones including additional redundant bounds of farther out planes.

[5] The convex polyhedron is the intersection of all the lower half-planes (Eq.10): $\bigcap_{p \in C} \{(v, \vec{w}) | 0 \geq \vec{c_p} \cdot \vec{w} - v \wedge 0 \leq v \wedge 0 \leq \vec{w}\}$. This intersection is identical $\forall C \subseteq P$ that contain the same optimal placements $P^* \subseteq C$.

Relaxing Redundancy Elimination. Redundancy elimination can be prohibitively expensive for large inputs. To address this, we employ a conservative performance heuristic: we partition the candidate policies into subsets. In Lines 8–12, our algorithm applies redundancy elimination to subsets of b candidates, along with the additional inequalities of Line 6 to bound each partition to positive orthant. It stores the non-redundant placements of each subset and finally returns all of these as oracle with reduced size.

Importantly, our resulting oracle still gives exact results. Any truly non-redundant plane—one that is lowest for some workload in the resulting oracle—is always lowest in its partition as well, and will not be pruned.[6]

This relaxation proves to be surprisingly effective at speeding up redundancy elimination. As we measure in §6.2, even the smallest partition size we considered significantly reduces the number of policies in the oracle.[7] The exponential complexity of LPs makes the aggregate compute time of many small LPs much smaller than that of one large LP [48]; moreover, the multiple small LPs can be handled independently in an embarrassingly parallel fashion.

```
1  def reduce_oracle(C: Candidate placements, b_size = len(P)) -> Placements:
2    Compact = list() # Compact set of placements
3    # Convert cost functions to inequalities of half-planes
4    H = [ p.c + [-1] for p in C]
5    # Additional half-planes of positive orthant
6    PO = pos_halfplane_per_dimension(dims=len(H[0]))
7    # Apply redundancy elimination to partitions
8    parallel for start, end in partitions(H, b_size):                ▷ O(|C|/b)
9      # Compute indexes of non-redundant half-planes
10     non_red = redundancy_elimination(H[start:end] + PO)       ▷ O(b ∗ LP(b, |A|))
11     # Save policies of non-redundant half-planes
12     Compact.extend(C[start+i] for i in non_red)
13   return Compact # Placements for the oracle
```

Listing 5. Algorithm removing non-optimal candidate placements from the oracle to speed up queries without affecting accuracy.

5 QUERYING THE SkyPIE ORACLE

The SkyPIE oracle computed by the techniques of §4 is comprised of (1) a list of the precomputed placements and (2) a matrix of their cost planes, where the rows are policies and the columns are cost coefficients (Listing 6).[8] To achieve fast and accurate querying, we apply computational geometry directly on the precomputed cost planes. The size of the cost plane matrix is deterministic and compact. Similarly, the matrix-vector computations of our geometric queries have fixed quadratic cost, yet are cheap and are easy to accelerate. As a result, querying the SkyPIE oracle is practically orders of magnitude faster than current ILP-based optimizers and has low resource demands.

[6] Exactness is within the limits of the employed LP algorithms. Their numerical precision is a research topic of its own. Optimizing these algorithms or exploring their trade-offs in depth is outside our scope. We refer interested readers to the literature [36, 41, 61].

[7] The partition size should be greater than the number of workload dimensions for redundancy elimination to be effective. Otherwise, the convex polyhedron of the policies in a smaller partition may not be full-dimensional, so that redundant planes become unlikely.

[8] We use the plane equality of Eq. 9, see §4.3.

```
def load_oracale(P: Placements) -> Oracle:
 # Convert placements' costs to matrix of plane inequalities
 return Oracle(P, Planes=Tensor([p.c+[-1] for p in P]))
```

Listing 6. Loading the oracle from precomputed policies.

```
1 def query_opt_policy(Oracle, size, gets, puts) -> Tuple[Placement,Cost]:
2  # Construct workload vector, materializing network ingress and egress
3  W=Tensor([size]+puts+gets+[sum(put*size)]+gets*size)
4  # Vertical ray-shooting for workload vectors W
5  idx, min_cost = Oracle.Planes.matmul(W).min()                    ▷ O(|P| * |R|)
6  return (Oracle.P[idx],min_cost) # Return optimal placement and costs
```

Listing 7. Algorithm for batched querying of the oracle for the optimal placements of object signatures.

Notably, we do not adopt spatial search structures. The optimal workload of optimal policies is high-dimensional and unbounded, so that spatial search structures have poor worst-case performance [43].[9] Instead, we are able to utilize straightforward matrix-vector multiplication, as we describe next.

Optimal Placement Queries. The `query_opt_policy` algorithm in Listing 7 queries the oracle for the optimal placement of an object signature via vertical ray-shooting—the intersection search initially mentioned in §2.2. Given the object size and `put`/`get` requests of application regions, it first constructs the complete object signature, computing the ingress and egress based on the object size and `put`/`get` requests of the individual regions (Line 3). In Line 5, it then computes the lowest intersecting plane along a vertical ray (parallel to the cost axis) that originates from the coordinates of the signature on the "cost floor". The special case of a vertical ray simplifies ray-shooting to a single matrix-vector multiplication and subsequent search of the minimal element, identifying the index of the lowest plane and associated costs. These operations, while quadratic in complexity, can be efficiently vectorized on CPUs or GPUs. They also allow for batched queries of multiple object signatures (omitted from the Listing for brevity). This vertical ray-shooting is as exact as current ILP-based optimizers but orders of magnitude faster, particularly on a GPU.

6 EXPERIMENTAL EVALUATION

In this evaluation, we answer three questions:

§6.1 How does online optimization compare with the current exact approach?
§6.2 How does precomputation scale?
§6.3 How does the end-to-end performance of the SkyPIE compare against the state-of-the-art?

Placement Scenarios. We evaluate object placement under 25 placement scenarios—replication factor 1–5 and 5 cloud deployments of increasing size: (**C1**) AWS regions in the EU (8), standard tier object stores (8); (**C2**) AWS regions in the EU (8), all object stores (24); (**C3**) All AWS regions (25), all object stores (75); (**C4**) All Azure regions (22), all object stores (88); (**C5**) All AWS and Azure regions (47), all object stores (163) [16, 68].

[9] Object signature incorporates the `get`/ingress/egress of each application region, hence is high-dimensional even for few regions. Additionally, the pay-per-use pricing of object stores "scales to zero" and generally is monotonically increasing. Therefore, the intersections of placements' cost planes all go through the origin, so that the optimal workload of optimal policies starts at the origin and extends to infinity.

All specified cloud regions are considered as application regions and each object store of the specified storage tiers in these regions is considered for placement.[10] We consider the cloud regions, object stores, and pricing information as of January 9th, 2023 [16, 68]. Notably, in order to perform a controlled experiment of optimizer performance, we do not impose latency SLOs. Latency SLOs would reduce the number of considered object stores in an application-specific way; instead, we directly specify the object stores in our placement scenarios. SkyPIE generally supports latency SLOs, cf. §3.1/§4.2.

Setup. We execute all online optimization (SkyPIE and baselines) on the 40 threads of one processor in our Nvidia DGX-1 hardware [30] but execute the SkyPIE's embarrassingly parallel precomputation on both available processors. That is, the ILP solver does not benefit from multiple processors due to limited parallelism in the algorithm and NUMA effects.[11] We hence execute the online optimization of all approaches on a single processor. We employ the commercial solver Mosek [9] for both the ILP and the redundancy elimination of our precomputation. Queries to the SkyPIE oracle are executed via Pytorch 2.0—GPU-based queries on a NVIDIA V100 [31].

6.1 Online Optimization Performance

We first benchmark the online optimization of the SkyPIE oracle against an exact ILP—the gold standard for truly optimal placements. We use SpanStore's ILP formulation [78]. We compare online optimization time and accuracy. Thereby, we seek to verify the premise of the SkyPIE oracle—that precomputation yields a compact but complete lookup structure (cf. §4) which makes geometric online querying fast and accurate (cf. §5).

In this experiment, we compute the placement of a given object signature with our oracle and the ILP, comparing their performance for each of the placement scenarios. Here, we sample 5000 object signatures drawn from a uniform random distribution over common object sizes (0-1000 GB) and access frequencies (0-1000 `put`/`get` requests per application region). These 5000 samples are practical in the sense that our measurements are robust and the ILP becomes prohibitively expensive to compute for more samples. Due this prohibitive overhead of the ILP, we also limit the number of samples to 100 for the large placement scenarios (AWS+Azure at replication factor >2).

Online Optimization Time. Figure 7(a)-(c) compares the online optimization time for computing the placement of a given object signature, i.e., the *wall clock time* for solving one ILP with 40 threads versus one SkyPIE query with 40 threads and on the GPU. For completeness, we measure the online optimization time including the loading time of the object signature vector[12].

Figure 7(a) details the effect of the cloud deployment size (object stores and cloud regions) on the optimization time under replication factor 3. The optimization time of SkyPIE on the GPU remains close to 100us throughout. On the CPU, SkyPIE's optimization time noticeably increases with the problem size, but query times remain under 100ms. In contrast, the time to solve a single ILP starts at 100ms increases to 100s.[13] Figure 7(b) shows that these observations generalize to the average optimization time for all our placement scenarios. SkyPIE is 10^1–10^6x faster on the CPU and GPU. Also, SkyPIE retains sub second response time while the ILP reaches ~100s.

[10] Excluding archive tiers [13, 70], AWS object stores: *Standard, Non-Critical*, and *Infrequent Access* storage tiers of S3 in all AWS cloud regions. Azure object stores: *Premium, Hot*, and *Cool* storage tiers of *Blob Storage/Block Block* in all Azure cloud regions. The workload dimensions according to the number of application regions are 28, 28, 77, 68, 143, respectively.

[11] ILP solvers rely on sequential refinement of solutions that belie parallelism. Commercial solvers thus operate only on CPUs and do not support GPU acceleration [37]

[12] For SkyPIE loading time involves initializing a tensor on the CPU or GPU, while for the ILP it is initialization of the model parameters. These loading times do not affect the general trends.

[13] The variance in the optimization time relates to sensitivity of the ILP's convergence to the specific model parameter values. For example, it is known that (I)LPs can achieve polynomial time for specific placement scenarios, but incur exponential time for other placement scenarios (their worst-case complexity) [48].

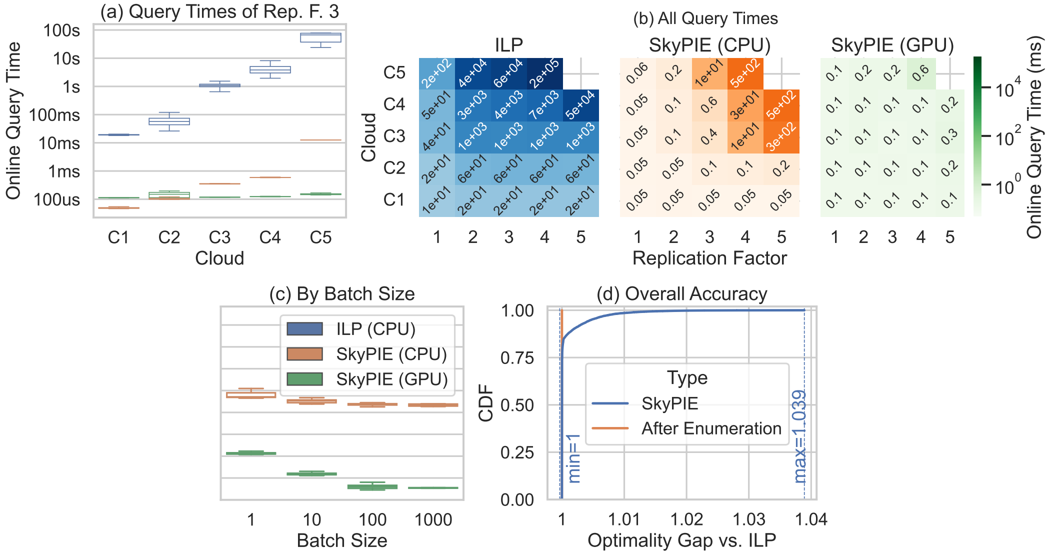


Fig. 7. (a)–(c) detail the time to compute placements for given object signatures including preparation—(a) the time per query for all cloud deployments and replication factor 3, (b) the average query times for all placement scenarios, and (c) the time per query when batch computing for C5 and replication factor 5. (d) details the gap between the cost of the placements computed by SkyPIE versus the exact ILP. Floating point arithmetic in the oracle reduction introduces minor inaccuracy.

Figure 7(c) illustrates the benefits of optimizing batches of several object signatures, under replication factor 3 and the largest deployment (C5). Batching improves optimization efficiency, with the same resources as before the median speedup of batched optimization is 1.5–2.3x on the CPU and 8–37x on the GPU. Optimization efficiency improves with increasing batch size, but with diminishing returns beyond batch size 100. It improves memory utilization on the CPU and additionally utilizes the hardware parallelism on the GPU, but eventually saturates these resources.

Online Optimization Accuracy. Figure 7(d) reports the accuracy of SkyPIE as a cumulative distribution over all queries. To quantify accuracy we measure, for each query, the optimality gap between SkyPIE and the ILP—which we define as the ratio of the serving costs of SkyPIE's computed placement versus the true optimum of the ILP. The y-axis indicates the proportion of queries with an optimality gap according to the x-axis. The blue line shows that SkyPIE's overall accuracy is a steep short-tailed distribution close to 1. There is no query with gap <1, so SkyPIE returns no incorrect results that claim impossibly inexpensive placements. About 80% of queries return placements with truly minimal costs with gap 1. The remaining 20% of queries have minor inaccuracy with costs less than 4% more than the optimum (gap <1.04). This inaccuracy related to the Linear-Program of the redundancy elimination as expected (§4.3), i.e., the orange line shows that an oracle after enumeration without redundancy elimination is perfectly accurate.

Insight: SkyPIE achieves sub second object placement—10^1–10^8x faster than Integer-Linear-Programs (ILPs). SkyPIE is almost as accurate as the ILP with minor inaccuracy (1–1.04x) due to numerical imprecision in the reduction step of the precomputation.

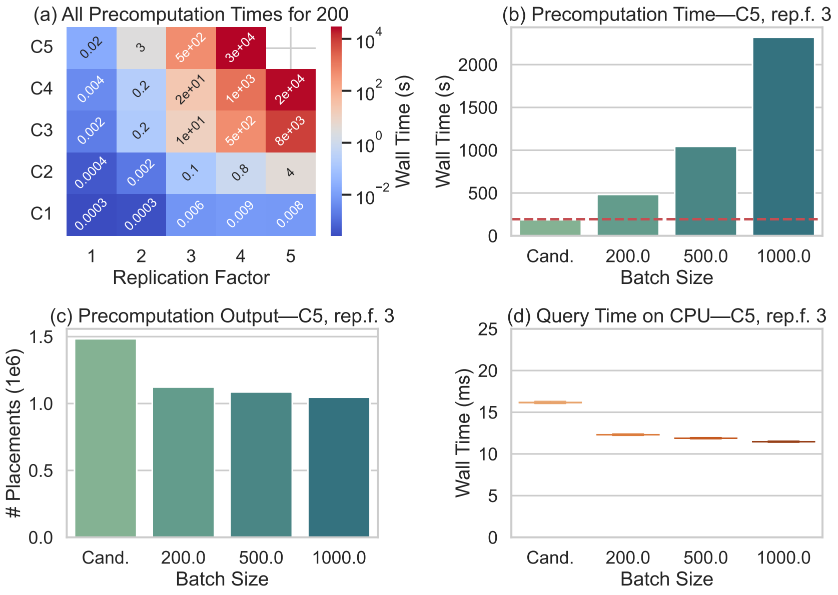


Fig. 8. (a) Precomputation times for all placement scenarios under batch size 200. (b)–(d) Precomputation time, number of placements in the resulting oracle, and the execution time for queries on the CPU of candidate enumeration (*Cand.*) and subsequent relaxed oracle reduction with indicated partition size under C5 (AWS+Azure) with replication factor 3.

6.2 Precomputation Performance

Tractable precomputation of a compact oracle is key to reap the previously presented optimization speedup. We now analyze the precomputation time of oracles across placement scenarios varying in replication factor (1–5) and cloud deployment size. Specifically, we seek understanding for the efficiency of the three steps that yield our $O(|O|^{f_{max}})$-algorithm; the bounded replication factor, enumeration of all candidate placements of a given cloud deployment, and batched reduction of the oracle, cf. §4.

Figure 8(a) provides an overview over the precomputation times across all considered placement scenarios, for our recommended batch size of 200. It shows that precomputation with 80 threads takes 3ms to 50min wall clock time. The precomputation overhead grows significantly with the replication factor and cloud deployment size, due to the exponential growth of the search space. This currently caps our precomputation at a maximum replication factor of 5 for the largest deployment (47 cloud regions and 163 object stores of AWS and Azure)—precomputation did not terminate after 4 days due to enumeration of too many candidates. Nevertheless, we demonstrate in the subsequent experiment that this limitation introduces marginal inaccuracy in object placement for real-world workloads.

Figures 8(b)–(d) detail the benefits of the relaxed oracle size reduction during precomputation. We separate on the x-axis results for precomputation with only the candidate enumeration (denoted as *Cand.*) and the subsequent relaxed reduction under a range of partition sizes. For this detailed analysis, we consider the largest cloud deployment (C5) under replication factor 3.

In summary, we can see that the small partition size 200 is highly effective.[14] In Figure 8(b), this partition size reduces the precomputation time by >4x down to 500s from 2500s—with about 250s of that time being candidate enumeration. Still, compared to the larger partition sizes, the partition size of 200 achieves almost the same oracle size and subsequent query time, as Figures 8(c)–(d) show. However, one has to take these empirical observations with a grain of salt. The efficacy of this trade-off between precomputation and query time depends on the specific distribution of (non)-optimal placements across the redundancy elimination partitions—and in turn on price distribution of the object stores. The effective partition size can change in future and users may have their own preferences for this trade-off.

Insight: Precomputation is tractable up to replication factor 5. Relaxed reduction with small partitions proves highly effective, yielding substantial speedup for both precomputation and querying.

6.3 End-to-end Performance for Real-World Workload

We finally benchmark object placement end-to-end—for the real-world workload of the file hosting service Ubuntu One [38] and three state-of-the-art heuristic optimization techniques. Today, these heuristics trade off optimization time for accuracy. We now expose the real-world benefit of accurate optimization with the SkyPIE oracle and contrast its optimization time including the offline precomputation.

We set up the following realistic scenario. Ubuntu One operates front-end applications in various cloud regions that access files in AWS S3 object stores. We simulate application cloud regions with a randomly chosen set of 10 AWS regions[15], as the trace was recorded does not contain locations. We randomly select one as the *home* region accessing the object according to the trace. The remaining *remote* regions access the object at a configurable probability. This probability defines the likelihood that any of the remote regions besides the home region accesses an object. We extract object signatures (object sizes and access frequencies) for the first week of the processed trace, as published here [18].

In this realistic scenario, we compare SkyPIE's accuracy and optimization time against the heuristic approaches: (1) SpanStore [78], (2) *Kmeans* clustering [72, 80], and (3) *profit-based* ranking [4]. SkyPIE, *kmeans*, and *profit-based* optimize placements per object (~1.1M trace records), where *kmeans* clusters only on the access costs and *profit-based* computes a ranking based on access and storage costs. SpanStore instead aggregates the object signature of objects with shared access and then optimizes the placements of the resulting 150–1500 aggregates with an ILP [78]. We set the maximum replication factor required by SkyPIE and *kmeans* to 5. We exclude the ILP, as its overhead is intractable for this real-world use case.

Real-World Accuracy. Figure 9(a) shows on the y-axis the accuracy as the cost of the computed placements relative to SkyPIE and on the x-axis the percent of accesses to objects from remote cloud regions other than the home region. It shows that SkyPIE computes placements with costs orders of magnitude lower than the heuristics. Also, we can see the workload sensitivity of the heuristics, especially sudden degradation of SpanStore and the *profit-based* heuristics. Note that, even under accesses from many regions, SkyPIE maintains high accuracy, despite the constrained replication factor of 5. Consider for example 1000% remote accesses, where all 10 regions jointly access the same objects. Intuitively, the cost of transferring objects from a nearby replica is only marginally higher than the costs of writing and storing an object across all 10 regions. Despite the limited replication factor, SkyPIE thus computes significantly cheaper placements compared to the heuristics.

[14] 200 placements is the minimum batch size in this case, since the partition size must be greater than the number of workload dimensions, cf. §4.3.

[15] AP-South-1, AP-Southeast-2, EU-Central-2, EU-North-1, EU-South-1, EU-West-1, EU-West-2, ME-Central-1, US-East-2, US-West-2

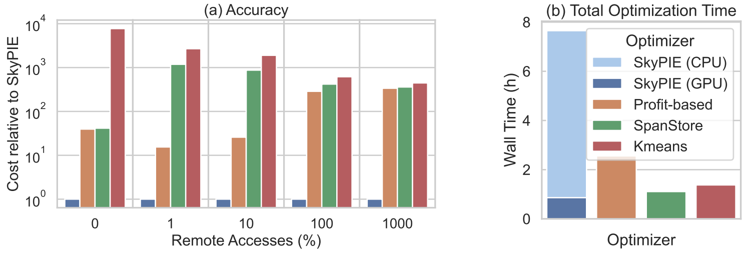


Fig. 9. Object placement accuracy and optimization time for real-world workload derived from Ubuntu One.

Real-World Optimization Time. While Figure 9 (a) shows the accuracy of the optimization approaches as the percentage of remote accesses increases, Figure 9(b) summarizes the total optimization time of all runs and including offline precomputation for SkyPIE. To compute the placements for this first week of Ubuntu One's trace, the heuristics take ~1–2.5 hours while SkyPIE's end-to-end optimization time takes 8 hours and <1 hour, respectively. As such, SkyPIE end-to-end offline and online optimization takes less time than the heuristics when using the GPU for querying.

Insight: SkyPIE significantly outperforms the current object placement heuristics. It achieves >10x lower cost while taking comparable time for the end-to-end offline and online optimization. SkyPIE thus promises significant cost savings through accurate placement and low optimization overhead even for large scale, geo-distributed cloud services.

7 RELATED WORK

We now discuss how related optimization approaches address the inherent complexity of data placement problems. Data placement problems appear in various shapes of the *Facility Location Problem*—with various objective and constraint formulations. The Facility Location Problem even in the uncapacitated case of elastic capacity is NP-hard. Optimizers thus must navigate this inherent complexity for tractable overhead at the large scale of the cloud.

SpanStore [78] reduces the online optimization overhead assuming uniform workload as input. Rather than solving the placement one object signature at a time, SpanStore heuristically aggregates groups of objects that are accessed from the same cloud regions. They solve the placement problem in a general form using Integer-Linear-Programming, but only solve for few workload aggregates. As we show in our experiments, SpanStore yields accurate placement if the workload is uniform, but it is sensitive to access skew.

Other ILP solutions find similar ways to limit the input and invocations of the the solver. TripS [56] coarsens object signatures to the average object size in a data center (DC). Other ILP solutions [71] similarly assume aggregate workloads or a small number of DCs for tractable optimization.

Baruah et al. [20] propose partitioning to avoid costly solving of large placement problems. They propose placement optimization based on Linear-Programming, but on individual partitions of the overall problem. They show that this partitioned approach significantly speedup optimization and remains close to the optimum if the number of partitions is relatively small compared to number of available placement locations (e.g., servers).

Sharov et al. [72] propose a weighted k-means clustering approach for latency-centric placement at Google. They reduce the problem complexity, addressing the realistic scenario of latency minimization and assume a low replication factor (similar to SkyPIE). In their weighted k-means algorithm, they further avoid high-dimensions by devising a 1-dimensional weight reflecting overall latency-sensitivity, since the latency-sensitivity of operations is independent of the placement. Notably, their evaluation on Google's production workload shows diminishing returns for increasing replica numbers—underlining that a low replication factor is a realistic problem setting.

Tuba [11, 74] assumes restrictive constraints for tractable online enumeration of placements. They propose to enumerate placements via constraint satisfaction and subsequently rank the candidates. Restrictive constraints allow effective enumeration without assumptions on the problem. Restrictive constrains also yield a small candidate set, so that the subsequent ranking is cheap and likely close to the optimum.

Liu et al. [51] propose to decouple optimization into offline optimization and online adaptation. For tractable offline optimization, they address placement under elastic capacity and devise a distributed optimization algorithm based on community detection. This offline optimization as well as their online adaptation are an approximate optimization, for which they provide a parameterized worst-case bound.

Akkio [8] apply a ranking-based approach for very low overhead placement of petabytes of small data shards. They decide the placement of a data shard based on its access pattern, the resource availability of data centers, and a customizable ranking policy. This ranking has very low overhead so that is a heuristic placement approach scales well to the large problem size at Meta.

Wang et al. [76] propose reinforcement-based ML agents to decide placement at each data center individually. The formulate the problem as a placement policy per data center and as an unconstrained trade-off between latency and costs. Offline training of the agents allows low overhead online optimization.

SkyPIE leverages a realistic problem formulation and offline optimization to circumvent high online overhead. It addresses a less complex problem, considering a low replication factor. It further relaxes the online optimization, shifting the heavy optimization to offline precomputation.

Note that, the specific problem formulations of the cited approaches differ. SkyPIE's enumeration (§4.2) can be extended with alternative constraints and its reduction step (§4.3) is compatible with any linear cost functions, so that SkyPIE can be extended to these formulations.

8 LIMITATIONS & DIRECTIONS

8.1 SkyPIE In Broader Context

Use Cases. SkyPIE can be applied to a range of use cases. We have focused on object placement use case and the cost minimization task that cloud users immediately face. Here, SkyPIE oracles can serve as optimizers for the cloud vendors' replication mechanism [14, 59, 65] or as drop-in replacement for the prior optimizers [8, 11, 78, 80]—minimizing object placement cost or latency at fine-granularity and with low overhead. In addition, SkyPIE oracles neatly extend to further optimization tasks, serving these use cases with fast and accurate stochastic simulation for "what-if" scenario planning, and minimization under workload drift for migration planning [19].

Placement optimization of dynamic workloads is another more involved use case. Current systems assume fairly stationary workloads that afford periodic refinement of placements [8, 78]. However, workloads with strong unpredictable variance, e.g., in access frequencies, require continuous refinement of placements. Dynamic *online* optimization algorithms (similar to [24, 54]) can employ SkyPIE as an inner-loop optimizer to improve accuracy while maintaining low overhead. By contrast, SkyPIE's utility for *prediction-based* optimization algorithms (like [52, 62, 77]) is currently limited. These algorithms need to solve the more complex placement problem that also includes the cost of object migration. The resulting search space is too large for SkyPIE, but an oracle for the problem without migration costs can seed starting points for search techniques. Dynamic pricing is not well supported either—all price changes require computing new oracles. Predictable price changes may warrant stochastic oracle computation over price distributions. Exploring dynamic optimization based on SkyPIE is an interesting avenue.

Oracle Lifetime. SkyPIE benefits the above use cases with fast and accurate placement optimization based on offline precomputation. A natural concern is whether the expected lifetime of an oracle is short. Overall, an oracle is valid as long as the precomputation inputs remain unchanged, e.g., prices, latency SLOs, available object stores, etc. Accordingly, if a limited number of scenarios is expected, one can precompute an oracle for each. For example, cloud vendors may offer default oracles for common sets of SLO classes (e.g., interactive, background) and global vs. continental deployments. Applications may use several oracles simultaneously and may compute custom oracles for custom SLOs or pricing.

Changes of the object stores, e.g., new pricing, as well as any changes of the precomputation inputs require a new oracle. Crucially, such changes also invalidate placements. A new SkyPIE oracle can be computed from scratch and shipped just a few hours after such pricing changes, to begin the process of moving data to optimal locations. It is attractive to consider *incremental* techniques to modify an existing SkyPIE oracle and data placements in the face of changes to costs, but this is challenging due to the supermodular structure of the underlying set cover problem (§3.1).

Further Performance Optimizations. Applications may require higher performance than offered out of the box from individual object stores, e.g., a higher request rate or tighter request latency. Starting points for improvements are optimizations for object store performance, e.g., overlapping concurrent requests, issuing redundant requests or partitioning/load balancing objects [23, 33, 57, 67, 73]. Optimizations that generally improve performance are complementary to the placement optimization, while optimizations that incur trade-offs may need integration with the objective function of the placement optimization.

Additionally, the placement optimization of SkyPIE can be extended with further performance constraints. For example, one can add constraints for applications that require high aggregate bandwidth across multiple object stores. Performance constraints require extension of SkyPIE's candidate enumeration (§4.2).

8.2 Complex Pricing Models

Object stores can have pricing models that are more complex than assumed throughput the paper. In addition to discounts based on monthly usage, Google Cloud Storage and Cloudflare R2 offer limited free storage and accesses per month [28, 58], while Azure offers discounts for reserved storage capacity [15]. Also, there are alternate storage services with distinct pricing models. SkyPIE requires extensions to optimally handle these cases, as we discuss next.

Free & Reserved Capacity. SkyPIE can support efficient optimal placement under discounts from free and reserved capacity. The required extension is to consider the active discounts based on the current usage of the object stores. It can be realized by introducing "discounted instances" of object stores (for each combination of discounts) and introducing conditional placements. Fortunately, placement optimization of individual objects remains sufficient, since plainly the consumption of all available discounted capacity minimizes costs regardless of the objects that occupy that capacity. Hence, offline precomputation of all cost-optimal placements across all possible discounts allows SkyPIE to offer fast and optimal online optimization.

Precomputation must handle mutually exclusive discounts, i.e., enumeration of candidates (§4.2) must avoid choosing the same object store with different discounts in the same placement and the oracle size reduction (§4.3) must only eliminate placements that are active for the same usage. The overhead of this approach will depend on the number of discounts for different workload features, due to combinatorics. Given the current number of free usage and reserved capacity discounts, we expect this straightforward extension to be sufficient. Alternatively, ad hoc recomputation oracles could be an avenue.

Online querying (§5) must resolve the active placements for the current usage. Placements need to be indexed by the discounts of their object stores and the matrix of the active placements has to be assembled. We expect the effective overhead to be marginal, since the active placements change infrequently given the typically large amounts of usage covered by discounts. One challenge is the accurate placement of objects when their object signature exceeds the discounted capacity.

Usage Discounts. Object placement under usage discounts requires further extensions beyond those described above, and cost-optimal placement appears difficult to achieve. Usage discounts reduce prices based on the account-wide monthly usage[17, 60, 69]—the higher the usage of a particular object store the higher the discounts. This incentivizes cloud services to consolidate their *monthly workload*. However, it implies that minimizing costs requires the joint placement optimization of all objects together and requires knowledge of the monthly workload in advance. Cost-optimal placement hence requires massive scale optimization across all objects and prediction of the monthly workload. Considering the currently narrow gap between discounted and non-discounted prices, it seems that heuristics could be adequate here, but this merits further study.

Since perfect workload prediction is unlikely, we propose considering placement under usage discounts as online problem where objects are gradually revealed. In this setting, a sophisticated approach could simulate object migration to identify reachable usage discounts. A simpler approach could greedily consolidate workload among similarly priced object stores. SkyPIE would accelerate either approach, given the above extension for discount support.

Further Pricing Models. Cost-optimal placement under the non-linear pricing models of alternate storage services require new approaches to cost modeling and precomputation. For example, the stepwise linear pricing of DynamoDB likely requires approximation as linear function. Pricing by provisioned bandwidth (e.g., provisioned IOPS) will require workload prediction to bridge to the pay-per-use pricing of object stores. Exploring non-linear optimization and approximation techniques is an interesting avenue to support further cost models.

CONCLUSION

SkyPIE demonstrates that is not only possible but highly beneficial to optimize cloud object placement by precomputing an oracle. The predictability of cloud object storage costs allows us to attack the problem exhaustively. This predictability leads us to an inverse perspective on the problem, approaching it from the placements. Subsequent querying via geometric intersection search proves to be as accurate as the best prior approaches, but orders of magnitude faster.

From a pragmatic perspective, SkyPIE works at the scale of the modern cloud, requiring only a few hours to precompute an oracle. Cloud vendors change their prices and offerings far less frequently, so the overhead of oracle precomputation is practically acceptable. As a result, SkyPIE can replace the optimizers in current pricing tools and object replication systems, offering nano- to millisecond response time for queries.

ACKNOWLEDGMENTS

We thank Observe Inc. and Tomas Karnagel for their support in processing workload traces on their Observability Cloud. This work was supported by gifts from AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

REFERENCES

[1] Karen Aardal, Fabián A. Chudak, and David B. Shmoys. 1999. A 3-approximation algorithm for the k-level uncapacitated facility location problem. *Inform. Process. Lett.* 72, 5 (1999), 161–167. https://doi.org/10.1016/S0020-0190(99)00144-1

[2] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*. Association for Computing Machinery, New York, NY, USA, 229–240.

[3] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Habinder Bhogan. 2010. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*. USENIX Association, USA, 2.

[4] Muhannad Alghamdi, Bin Tang, and Yutian Chen. 2017. Profit-based file replication in data intensive cloud data centers. In *2017 IEEE International Conference on Communications (ICC)*. IEEE, Paris, France, 1–7. https://doi.org/10.1109/ICC.2017.7996728

[5] Guillermo A Alvarez, Elizabeth Borowsky, Susie Go, Theodore H Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. 2001. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)* 19, 4 (2001), 483–518.

[6] Amazon Web Services. 2023. Slack Case Study. https://aws.amazon.com/solutions/case-studies/slack/

[7] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. 2002. Hippodrome: running circles around storage administration. In *Conference on File and Storage Technologies (FAST 02)*. USENIX Association, USA, 13.

[8] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 445–460. https://www.usenix.org/conference/osdi18/presentation/annamalai

[9] Mosek ApS. 2019. Mosek optimization toolbox for matlab. *User's Guide and Reference Manual, Version* 4 (2019), 1.

[10] Mosek ApS. 2023. *Advanced hot-start.* https://docs.mosek.com/10.0/toolbox/advanced-hotstart.html

[11] Masoud Saeida Ardekani and Douglas B Terry. 2014. A Self-Configurable Geo-Replicated Cloud Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, USA, 367–381.

[12] Microsoft Azure. 2023. *Azure Blob Storage.* https://azure.microsoft.com/en-us/products/storage/blobs/

[13] Microsoft Azure. 2023. *Estimate the cost of archiving data.* https://learn.microsoft.com/en-us/azure/storage/blobs/archive-cost-estimation#the-cost-to-rehydrate

[14] Microsoft Azure. 2023. *Object replication for block blobs.* https://learn.microsoft.com/en-us/azure/storage/blobs/object-replication-overview

[15] Microsoft Azure. 2023. *Optimize costs for Blob storage with reserved capacity.* https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-reserved-capacity

[16] Microsoft Azure. 2023. *Pricing API.* https://learn.microsoft.com/en-us/rest/api/cost-management/retail-prices/azure-retail-prices

[17] Microsoft Azure. 2023. *Pricing Calculator.* https://azure.microsoft.com/en-us/pricing/calculator/

[18] Tiemo Bang, Chris Dougals, Natacha Crooks, and Joeseph M. Hellerstein. 2023. *Cloud Oracle/SkyPIE Github Repo.* https://github.com/hydro-project/cloud_oracle

[19] Tiemo Bang, Conor Power, Siavash Ameli, Natacha Crooks, and Joseph M. Hellerstein. 2024. Optimizing the cloud? Don't train models. Build oracles!. In *14th Annual Conference on Innovative Data Systems Research, CIDR 2024 Chaminade, USA, January 14-17, 2024.* www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p47-bang.pdf

[20] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. 2022. Parallelism-Optimizing Data Placement for Faster Data-Parallel Computations. *Proceedings of the VLDB Endowment* 16, 4 (Dec. 2022), 760–771. https://doi.org/10.14778/3574245.3574260

[21] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[22] Jon Louis Bentley. 1979. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering* 5, 4 (1979), 333–340. https://doi.org/10.1109/TSE.1979.234200

[23] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An efficient column store for cloud data lakes. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, USA, 3078–3090. https://doi.org/10.1109/ICDE53745.2022.00276

[24] Sebastien Bubeck. 2011. Introduction to Online Optimization. http://sbubeck.com/BubeckLectureNotes.pdf

[25] Amina Chikhaoui, Laurent Lemarchand, Kamel Boukhalfa, and Jalil Boukhobza. 2021. StorNIR, a multi-objective replica placement strategy for cloud federations. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/3412841.3441886

[26] K.L. Clarkson. 1994. More output-sensitive geometric algorithms. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, USA, 695–702. https://doi.org/10.1109/SFCS.1994.365723

[27] Google Cloud. 2023. *Cloud Storage.* https://cloud.google.com/storage/

[28] CloudFlare. 2023. *Pricing CloudFlare R2 Docs.* https://developers.cloudflare.com/r2/pricing/

[29] IBM Corporation. 2022. *Starting from a solution: MIP starts.* https://www.ibm.com/docs/en/icos/22.1.0?topic=mip-starting-from-solution-starts

[30] NVIDIA Corporation. 2023. *NVIDIA DGX-1.* https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf

[31] NVIDIA Corporation. 2023. *NVIDIA Tesla V100 GPU Architecture.* https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[32] Oracle Corporation. 2023. *Object Storage.* https://www.oracle.com/cloud/storage/object-storage/

[33] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2769–2782.

[34] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. 1980. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques.* Association for Computing Machinery, New York, NY, USA, 124–133. https://doi.org/10.1145/800250.807481

[35] Komei Fukuda. 2015. Lecture - Polyhedral Computation, Spring 2015. https://people.inf.ethz.ch/fukudak/lect/pclect/notes2015/PolyComp2015.pdf

[36] Gerald Gamrath, Benjamin Hiller, and Jakob Witzig. 2015. Reoptimization techniques for MIP solvers. In *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29–July 1, 2015, Proceedings 14.* Springer, Springer-Verlag, Berlin, Heidelberg, 181–192. https://doi.org/10.1007/978-3-319-20086-6_14

[37] Greg Glockner. 2023. *Does Gurobi support GPUs?* https://support.gurobi.com/hc/en-us/articles/360012237852-Does-Gurobi-support-GPUs

[38] Raúl Gracia-Tinedo, Yongchao Tian, Josep Sampé, Hamza Harkous, John Lenton, Pedro García-López, Marc Sánchez-Artigas, and Marko Vukolic. 2015. Dissecting UbuntuOne: Autopsy of a Global-Scale Personal Cloud Back-End. In *Proceedings of the 2015 Internet Measurement Conference (IMC '15).* Association for Computing Machinery, New York, NY, USA, 155–168. https://doi.org/10.1145/2815675.2815677 event-place: Tokyo, Japan.

[39] Sudipto Guha and Samir Khuller. 1999. Greedy Strikes Back: Improved Facility Location Algorithms. *Journal of Algorithms* 31, 1 (April 1999), 228–248. https://doi.org/10.1006/jagm.1998.0993

[40] Xiangyu Guo, Janardhan Kulkarni, Shi Li, and Jiayi Xian. 2020. On the Facility Location Problem in Online and Dynamic Models. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 176),* Jaros\law Byrka and Raghu Meka (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 42:1–42:23. https://doi.org/10.4230/LIPIcs.APPROX/RANDOM.2020.42 ISSN: 1868-8969.

[41] Menal Guzelsoy. 2009. *Dual methods in mixed integer linear programming.* Ph. D. Dissertation. Lehigh University PhD.

[42] M. T. Hajiaghayi, M. Mahdian, and V. S. Mirrokni. 2003. The Facility Location Problem with General Cost Functions. *Networks* 42, 1 (2003), 42–47. https://doi.org/10.1002/net.10080 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.10080.

[43] Joseph M. Hellerstein, Elias Koutsoupias, and Christos H. Papadimitriou. 1997. On the Analysis of Indexing Schemes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA.* Association for Computing Machinery, New York, NY, USA, 249–256. https://doi.org/10.1145/263661.263688

[44] Backblaze Inc. 2023. *B2 Cloud Storage.* https://www.backblaze.com/b2/cloud-storage.html

[45] Cloudflare Inc. 2023. *Cloudflare R2.* https://www.cloudflare.com/products/r2/

[46] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. 2023. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. USENIX Association, USA, 1375–1389. https://www.usenix.org/conference/nsdi23/presentation/jain

[47] Sudarshan Kadambi, Jianjun Chen, Brian F Cooper, David Lomax, Raghu Ramakrishnan, Adam Silberstein, Erwin Tam, and Hector Garcia-Molina. 2011. Where in the world is my data? *Proceedings of the VLDB Endowment* 4, 11 (2011), 1040–1050.

[48] Richard M Karp, RE Miller, and JW Thatcher. 1972. Reducibility among combinatorial problems, Complexity of computer computations. *Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, NY, 1972* 378476, 51 (1972), 14644.

[49] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. 2007. SafeStore: A durable and practical storage system. In *USENIX Annual Technical Conference.* USENIX Association, USA, 129–142.

[50] Shi Li. 2013. A 1.488 approximation algorithm for the uncapacitated facility location problem. *Information and Computation* 222 (Jan. 2013), 45–58. https://doi.org/10.1016/j.ic.2012.01.007

[51] Kaiyang Liu, Jun Peng, Jingrong Wang, Weirong Liu, Zhiwu Huang, and Jianping Pan. 2020. Scalable and Adaptive Data Replica Placement for Geo-Distributed Cloud Storages. *IEEE Transactions on Parallel and Distributed Systems* 31, 7 (2020), 1575–1587. https://doi.org/10.1109/TPDS.2020.2968321

[52] Qingsong Liu, Zhuoran Li, and Zhixuan Fang. 2022. Online Convex Optimization with Switching Costs: Algorithms and Performance. In *2022 20th International Symposium on Modeling and Optimization in Mobile, Ad hoc, and Wireless Networks (WiOpt)*. IEEE Computer Society, 1–8. https://doi.org/10.23919/WiOpt56218.2022.9930570

[53] Harsha V Madhyastha, John McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C Snoeren, and Amin Vahdat. 2012. Scc: Cluster storage provisioning informed by application characteristics and SLAs.. In *FAST*. USENIX Association, USA, 23.

[54] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. 2019. Cost Optimization for Dynamic Replication and Migration of Data in Cloud Data Centers. *IEEE Transactions on Cloud Computing* 7, 3 (2019), 705–718. https://doi.org/10.1109/TCC.2017.2659728

[55] Paul Miller, Pascal Matzke, Will McKeon-White, Christopher Voce, and Ian McPherson. 2018. *A Clear Multicloud Strategy Delivers Business Value*. https://www.forrester.com/report/a-clear-multicloud-strategy-delivers-business-value/RES128781

[56] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2017. TripS: Automated Multi-Tiered Data Placement in a Geo-Distributed Cloud Environment. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3078468.3078485 event-place: Haifa, Israel.

[57] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[58] Google Cloud Platform. 2023. *Cloud Storage Always Free usage limits*. https://cloud.google.com/storage/pricing#cloud-storage-always-free

[59] Google Cloud Platform. 2023. *Data Availability and Durability*. https://cloud.google.com/storage/docs/availability-durability#cross-region-redundancy

[60] Google Cloud Platform. 2023. *Pricing Calculator*. https://cloud.google.com/products/calculator

[61] Ted Ralphs. 2006. Duality and Warm Starting in Integer Programming. https://coral.ise.lehigh.edu/~ted/files/papers/DMII06.pdf

[62] James Blake Rawlings, David Q. Mayne, and Moritz Diehl. 2017. *Model predictive control: theory, computation, and design* (2nd edition ed.). Nob Hill Publishing, Madison, Wisconsin.

[63] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Technical Report. University of Maryland.

[64] Amazon Web Services. 2023. *Amazon S3 Intelligent-Tiering storage class*. https://aws.amazon.com/s3/storage-classes/intelligent-tiering/

[65] Amazon Web Services. 2023. *Amazon S3 Replication*. https://aws.amazon.com/s3/features/replication/

[66] Amazon Web Services. 2023. *Cloud Object Storage*. https://aws.amazon.com/s3/

[67] Amazon Web Services. 2023. *Performance Design Patterns for Amazon S3*. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-design-patterns.html

[68] Amazon Web Services. 2023. *Pricing API*. https://pricing.us-east-1.amazonaws.com/offers/v1.0/aws/index.json

[69] Amazon Web Services. 2023. *Pricing Calculator*. https://calculator.aws/

[70] Amazon Web Services. 2023. *Restoring an archived object*. https://docs.aws.amazon.com/AmazonS3/latest/userguide/restoring-objects.html

[71] P.N. Shankaranarayanan, Ashiwan Sivakumar, Sanjay Rao, and Mohit Tawarmalani. 2014. Performance Sensitive Replication in Geo-distributed Cloud Datastores. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society, USA, 240–251. https://doi.org/10.1109/DSN.2014.34

[72] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader! online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1490–1501. https://doi.org/10.14778/2824032.2824047

[73] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a cloud DBMS: architectures and tradeoffs. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2170–2182. https://doi.org/10.14778/3352063.3352133

[74] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731 event-place: Farminton, Pennsylvania.

[75] Johann Wolfgang Von Goethe. 1843. *Faust: A Tragedy, in Two Parts*. Chapman and Hall.

[76] Haoyu Wang, Haiying Shen, Zijian Li, and Shuhao Tian. 2021. GeoCol: A geo-distributed cloud storage system with low cost and latency using reinforcement learning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 149–159. https://doi.org/10.1109/ICDCS51616.2021.00023

[77] Tyler Westenbroek, Max Simchowitz, Michael I. Jordan, and S. Shankar Sastry. 2021. On the Stability of Nonlinear Receding Horizon Control: A Geometric Perspective. In *2021 60th IEEE Conference on Decision and Control (CDC)*. IEEE, Austin, TX, USA, 742–749. https://doi.org/10.1109/CDC45484.2021.9682955

[78] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 292–308. https://doi.org/10.1145/2517349.2522730 event-place: Farminton, Pennsylvania.

[79] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. 2023. SkyPilot: An Intercloud Broker for Sky Computing. USENIX Association, USA, 437–455. https://www.usenix.org/conference/nsdi23/presentation/yang-zongheng

[80] Hamidreza Zare, Viveck Ramesh Cadambe, Bhuvan Urgaonkar, Nader Alfares, Praneet Soni, Chetan Sharma, and Arif A Merchant. 2022. LEGOStore: a linearizable geo-distributed store combining replication and erasure coding. *Proceedings of the VLDB Endowment* 15, 10 (Sept. 2022), 2201–2215. https://doi.org/10.14778/3547305.3547323

Received July 2023; revised October 2023; accepted November 2023