Adapting to Access Locality via Live Data Migration in Globally Distributed Datastores

Aleksey Charapko
University at Buffalo, SUNY
acharapk@buffalo.edu

Ailidani Ailijiang
Microsoft
aiailiji@microsoft.com

Murat Demirbas
University at Buffalo, SUNY
demirbas@buffalo.edu

***Abstract*—Storing data close to where it is used improves the performance of cloud applications. However, data access patterns change dynamically over time. Many datastores statically shard data making locality-adaptation difficult, and some provide limited capability for controlling the data-placement or migration. This leads to increased latency, reduced throughput, and expensive operations. To address this problem, we investigate the requirements for live data-migration and design four data-migration polices. Our policies use heuristics to determine the optimal data placement based on the access locality in the workload and load-balancing constraints. We show that even simple heuristics can be effective, and the topology-aware policies demonstrate overall better results with up to 70% latency improvement in medium locality workloads and nearly 95% improvement in workloads exhibiting very strong single-region access locality.**

I. Introduction

The latency of reaching out to the data across the globe becomes prohibitively large and detrimental to the user experience [6], [8] for social, e-commerce, and IoT applications. Globally distributed cloud databases [10], [11], [9], [14] have been adopted to address this problem. The de facto method of keeping the data close to the consumers in distributed databases is full replication [10], [14], [4], however, this often comes at the cost of weaker consistency provided to the clients. Moreover, fully replicated systems often have a dedicated region to perform write operations, making updates incur much higher latency than reads.

Other designs [9], [11] provide strong consistency at the global scale, but must restrict the replication of data to a subset of the available regions. Such systems often partition the data into small chunks and replicate across a handful of regions, with partitions possibly replicated to different and not always overlapping subsets of all available regions. If users happen to request the data from a partition located in the nearby regions, then they observe a relatively small latency. On the other hand, if users request the data located far away, they pay a heavy penalty of up to a few hundred milliseconds just to reach over to the regions hosting the data.

Many of today's cloud applications need both strong consistency and low latency. This combination of requirements is hard to achieve and depends upon a careful placement of data to reduce the access latency. In a recent study [5], Facebook found that placing data according to access locality can shed as much as 50% of the request latency, all while reducing storage costs by 40% and WAN-traffic by 50%.

Despite the obvious benefits of keeping the data where it is needed the most, many databases [10], [15], [12] offer only static data partitioning and placement. Some solutions [11], [9] provide limited support for data movement and do not fully take advantage of the access locality.

Contributions. To address these problems and fill this gap, we define the criteria that live data migration systems should follow to benefit from access locality. We posit that effective *data-migration policies* should:

- Minimize access latency,
- Preserve load balancing with regards to data storage and processing capacity,
- Preserve collocation of related data, and
- Minimize the number of data migrations.

We design and evaluate four data-migration policies following these requirements. Our policies work in two phases: first they find an optimal location for some data object given its access history and then adjust this location to adhere to the balancing requirements. Our policies work at an arbitrary data-granularity, such as individual data items or shards/partitions. They preserve collocation even at the most granular level for related objects having similarities in their access patterns.

Our simplest policy, the **n-consecutive accesses policy**, uses a threshold of consecutive accesses to the object to make the placement decision. Although simple, this policy works well for workloads with strong locality in a single region. In workloads that exhibit no locality or have more than one region with high rate of access, n-consecutive policy often fails to find the optimal placement, and sometimes it may even cause unnecessary data movement.

The **majority access policy** decides on the placement by examining the request history of the object and migrating it to a zone with majority access. This policy has similar drawbacks to n-consecutive one, although these drawbacks tend to manifest themselves to a lesser extent.

The **exponential moving average (EMA)** policy computes the average region for all requests to the object, therefore this policy can potentially find better placement for objects that have more than one high-access region. EMA policy, however, requires the regions to have numerical IDs arranged in the order of region's proximity to each other. This policy falters for deployments with complicated geography and may require multiple migrations to move data to the best location.

Finally, our **center-of-gravity (CoG)** policy calculates the optimal placement for data by taking into account the requests distribution between the regions and the topology of the regions. With the topology information, such as inter-region distances measured in terms of communication delays, CoG policy calculates the region closest to the central location for any access locality workload.

Results. We evaluate our migration policies with comprehensive simulations and show how they perform under different static and dynamic locality conditions. Our simulations consider a geo-distributed datastore deployed over 15 regions following the AWS region topology, with the inter-region latency taken from [1].

Our migration policies show a significant improvement in access latency and reduction in WAN traffic. In some workloads exhibiting high access locality, the improvement can reach as high as 70% compared to static non-migrating system. We also show the ability of policies to adjust to changes in the workload's access locality, whereas non-migrating statically partitioned solution performance varies greatly depending on how well the access locality aligns with static data-placement.

Our experimentation revealed that different policies may be better suitable for different workloads. For example, n-consecutive accesses policy can adjust to changes very quickly, and maybe a good choice for workloads with strong single region locality that changes between regions quickly over time. On the contrary, workloads with bad locality cause n-consecutive access policy to behave erratically and unnecessarily migrate the data back-and-forth. In general, the policies aware of the region topology fare better under a broad spectrum of conditions. For instance, CoG policy outperforms all other policies in no-locality, high locality and split locality workloads.

II. Related Work

Volley [2] is an offline data-migration recommendation system. It relies on processing all access logs for an object offline before making a migration recommendation that must be carried out by some other system or engineers. Volley's offline processing is cumbersome and time-consuming, taking as much as 14 hours to sift through 1 month worth of logs. Additionally, Volley focuses on data-collocation and load balancing more than on the optimal placement due to access-locality. Similarly, associated data placement (ADP) [22] and Clay [18] focus on collocating related data for transactional workloads.

Akkio [5] is Facebook's data placement system that adapts to the changing access locality patters. It manages data placement by grouping the related data that exhibit similar access locality into μ-shards. Each μ-shard then can be moved between datacenters based on up to a few days of recent access history. Akkio mainly focuses on the granularity of data migration, the storage requirements for all the access history and integration into the existing databases at Facebook, leaving the questions of determining the optimal data-placement largely unexplored.

GPlacer [23] solves the problem of optimal data placement in the context of the transnational workloads. The protocol is topology aware and considers both distances between datacenters and between a client and prospective replicas. GPlacer computes optimal locations for database deployment given some workload, whereas our policies assume running database capable of migrating data between datacenters. Our policies are more suitable for live migration at arbitrary data granularity in response to changes in access locality, since such migration decisions often need to be made quickly, rely on limited data and consider only migration of affected data objects. Additionally, some of our migration policies use simpler topology-unaware heuristics.

Distributed storage overlays [20] explore a problem of efficiently migrating the data to prevent the downtime, data unavailability and data-loss during the migration. Overlays control what data and from which location is visible to the client while the migration is in progress.

Some databases also provide means to control the location of data. Selective replication extension to PNUTS [13] analyzes data access patterns and selectively replicates the data to regions that will benefit from having a local copy. This reduces the network bandwidth and storage requirements, but offers little advantage in terms of access latency compared to regular full replication in PNUTS. Spanner [11] uses *movedir* command to migrate the data between Paxos-groups. CockroachDB[9] can change raft-group's leaseholder and move it closer to the region requesting data the most. Tuba [7] adopts to changes in the access locality and workload intensity by dynamically reconfiguring itself at the shard/partition granularity. DPaxos [16] data-management/replication protocol based on the WPaxos [3] algorithm targets high access locality workloads to bring highly granular data to the consumers on the edge.

Some systems address a bigger problem of live application migration from one datacenter to another [19], [21]. For instance, Supercloud [19] is a live virtual machine migration service placed on top of existing public cloud infrastructure. It moves VMs in response to changes in access pattern. Supercloud can move a small VM with 1GB of RAM in under one second: quick enough to not even cause TCP connection to be dropped. Despite its speed, VM migration requires a lot of bandwidth and must transfer data, such as storage and memory prior to making a final switch.

III. Live Migration of Data

Many geo-distributed cloud applications, such as social networks [5], IoT [17] and e-commerce are sensitive to the physical location of data. If the data is not available close to the consumer, the penalty may be too large to tolerate. On the global scale, misplacement of data can add as much few hundreds of milliseconds to the access latency in some extreme cases. For that reason, many applications rely on replicating the data and/or caches across the globe to facilitate the local clients to quickly read the required information. This approach works well when strong consistency is not necessary [5]. However, full global replication in strongly-consistent systems is expensive, since it requires at least one round trip to synchronize all the datacenters, making all update operations always complete with a large WAN-scale latency.

Proper placement of data dramatically improves the access latency in sharded, strongly-consistent systems by alleviating a need for a full global replication and reducing the number of far, out-of-region requests. Another benefit of locality-aware data placement is smaller operating costs achieved through a reduction in storage requirements and WAN traffic [5].

Unfortunately, data access locality is not static and changes due to a variety of reasons, such as diurnal patterns [19], people's travel and migration, business needs, etc. Many datastores[10], [15], [12], however, only provide static data placement that cannot accommodate the access locality changes. Some databases [11], [9] have limited capability to move data, such as Spanners *movedir* command. In this paper we formulate the criteria for optimal live data migration and present a four migration policies that can be applied to a variety of databases to improve the access locality.

A. Problem Definition

Practical live data-migration solution needs to satisfy a number of often contradicting criteria. Most importantly, the system needs to optimize for access latency and move the data closer to where it is needed. This, however, may come in contradiction with load balancing constraints, since the migration policy also needs to respect the datacenter capacity and refrain from moving data to overloaded regions. Live system may also find itself in a situation with limited data available to make migration decisions. This contrasts greatly with offline systems [2] that may use a week's worth of request logs to compute the best object placement strategy. To keep the policy on the slim side, it needs conserves resources of the system and eliminate unnecessary movement of data across regions. Finally, data-collocation is another constraint to consider while making migration decisions.

For the purpose describing the object migration problem, we assume a distributed datastore handling many different objects. Each object represents some inseparable data that always gets replicated, stored and migrated together. The granularity of the objects is up to the database design, and it may be different for different systems. For example, an object can be a partition or shard that has to be stored or migrated as one piece. However, such partition may also contain smaller units of data available to the users, such as key-value pairs or documents. Alternatively, a migration object may also be the smallest granularity of data in the systems, such as individual key-value pair or a document.

The database is spread out across Z different regions or datacenters. For the simplicity of the model, each migration object must belong to a single region. For databases that perform replication across a handfull of regions, such as Spanner [11] or CockroachDB [9], a datacenter with a leader is said to own the migration object. For instance, a CockroachDB raft-group leader or leaseholder provides access to the partition's data for both reads and writes, therefore a region hosting the leader owns the partition, while the followers in other datacenters ensure fault-tolerance. Some designs may replicate data across multiple regions and not have a dedicated node for serving reads [14], [10], in this case the owner is a region responsible for writes.

At each region z, clients c_z interact with the system and produce some load w_z on a replication object n, such that the total load W is the sum of the loads generated in each region:

$$W = \sum_{z=1}^{Z} w_z = 1$$

In other words, w_z represents the proportion of a total load for some replication group n generated in a region z and describes the locality of the workload.

Minimizing access latency. Data migration policy must strive to minimize the average request latency by adjusting to the access locality. Intuitively speaking, we want the data to be located close to where it is needed. Let L_z be the average latency for operations originating in a region z for some replication object n, and d_z be a distance, expressed as communication latency, between a region z and the region owning an object n. We can express region latency as $L_z = d_z + \epsilon$, where ϵ is some overhead for processing and replicating data in a database.

We can compute the global average latency L_{avg} for an object by taking into account the workload characteristics:

$$L_{avg} = \sum_{z=1}^{Z} w_z L_z$$

The object migration policy needs to minimize the L_{avg} by moving the object to an optimal location for a given workload w and therefore adjusting distances d. In the best case, the object is placed to a zone closest to the workload's center of gravity or location in the middle of object's access distribution.

Minimizing load disbalance. Migration policy purely driven by access latency optimization may jeopardize the stability of the system in some situations. In case of highly skewed workloads, the latency optimization may cause all object to migrate to a handful of regions, causing disbalance in the system. As a consequence, migration policy needs to halt migration to datacenters that are already at their maximum capacity, and resort to moving the objects to the next best place for their locality.

Assuming the storage capacity $\mathcal{C}_z$ in a region z, migration policy needs to ensure that storage requirement $\sum Storage(o)$ for all objects in a zone is at or below the capacity:

$$\forall z \in Regions : \mathcal{C}_z \geq \sum_{o \in O_z} Storage(o)$$

However, a region may be limited not only in storage, but in the processing capacity $\mathcal{P}(t)$ over some time-interval t. Similarly to storage, a migration policy should not assign more objects to a region than it can process:

$$\forall z \in Regions : \mathcal{P}(\Delta t)_z \geq \sum_{o \in O_z} \epsilon |Request(\Delta t, o)|$$

where O_z is a set of all objects in region z, ϵ is the cost of processing a single request, and $Request(\Delta t, o)$ is a set of requests for object o over the time-interval Δt.

Minimizing network utilization. A migration policy needs to provide some stability to the object placement and eliminate unnecessary migrations. If the policy is too sensitive it can cause erratic object movement in response to minuscule variation in the access patterns. These unnecessary migrations not only cost network resources, but may also increase the access latency. On the other side of the spectrum, a policy that is too slow to adjust to the new workload locality, will miss significant opportunity to reduce the latency and operating costs by reacting too late.

Maximizing data collocation. Many application exhibit tendency for some objects to frequently get accessed together. For performance reasons, it is important to keep such related data close together. If the policy disregards the collocation, the benefits from migrating one object may be entirely offset by out-of-region access to other related objects.

IV. Policies for Live Data Migration

We are solving the problem of live data migration by separating it in two phases. First, we find the optimal placement for the data object without considering other constraints, such as load balancing and object collocation. Once we find the best possible region/node for the object, we check if that location meets the balancing criteria. In cases when optimal placement is not possible due to the balancing reasons, we find a different node to be as close as possible to the previously computed optimal location. Algorithm 1 shows the high-level overview of our generic two-phase migration policy.

Algorithm 1 Two-phase Object Migration Policy

```
1: Initialize:
       $req_o$ := request for object $o$
       $r_{current}$ := current location of $o$
2: adjust workload parameters $w_o$ to reflect $req_o$
3: Compute optimal location $r_{new}$ for workload $w_o$        ▷ Phase-1
4: if $r_{current} \neq r_{new}$ then
5:     if Overloaded($l_{new}$) then
6:         $r_{new}$ := next closest non-overloaded node        ▷ Phase-2
   return $r_{new}$
```

Finding the best place to move an object requires a global knowledge on how such object is being used in the systems. Such global usage information is often available at the node/region currently hosting the object, since that region is responsible for answering all requests for the object. Some database designs do no send all requests through the owner region, for example Cosmos DB can perform read operations without notifying the leader. These designs, however, can either piggyback the local usage statistics to some message that requires the owner or utilize a gossiping protocol. Consolidating all the usage information about the owner allows us to make design a placement/migration policy that requires no communication with other nodes when computing the migration decision.

We have designed four different data migration/placement policies that differ mainly in the algorithms for identifying the best node/region for a data object: n-consecutive accesses, majority access, exponential-moving average (EMA) policy and center-of-gravity (CoG) policy.

A. n-Consecutive Accesses Policy

The most trivial policy is n-consecutive access. This policy uses the number of consecutive requests originating in the same region as a heuristic value for optimal placement. As a result, the region that accesses the object n times in a sequence receives the ownership. We provide the pseudocode for the policy in Algorithm 2.

This policy works best in workloads that exhibit good locality. In more sporadic workloads with no regions being able to make n-consecutive accesses to the object policy will not performing any migration and the object will remain stationary at its current leader. Sporadic workloads, however, can often create another anomaly as well: different regions may be able to access the object n times, causing it to move erratically between regions.

Algorithm 2 n-consecutive accesses

```
1: Initialize:
       $req_o$ := request for object $o$
       $r_{current}$ := current location of $o$
       $r_{migrate}$ := current migration candidate region
2: if $r_{migrate}$ = Region($req_o$) then
3:     $consecutive := consecutive + 1$
4: else
5:     $r_{migrate}$ := Region($req_o$)
6:     $consecutive := 1$
7: if $consecutive >= n$ then
8:     $r_{current}$ = $r_{migrate}$
   return $r_{current}$
```

B. Majority Access Policy

Majority access policy uses a different heuristic for finding the best region for an object. It relies on counting all requests coming from each region over some period of time and migrating the object over to the region with the most requests. Naïve implementation of this policy is sensitive to the window size over which the request statistics is accumulating. If the window is too small, there may be too few requests coming in to make a good placement decision. For example, if only a single request came in during the window, that request alone will constitute the majority of accesses and cause the policy to relocate the object. To reduce the severity of this problem we establish a minimal number of requests required to make a decision in order to accumulate more reliable sage statistics.

C. Exponential Moving Average Policy

Exponential Moving Average policy (EMA) relies on computing a region of average access for an object. This differs from the above two policies in a way that we no longer tailor the migration to one region exhibiting the most need for the object. Instead, EMA tries to compute the "central" region for a particular workload.

EMA works by requiring the users to assign integer IDs in the range [1, n] for each of n regions in the system. The

Algorithm 3 Majority access

```
1: Initialize:
       $req_o$ := request for object $o$
       $r_{current}$ := current location of $o$
       $counters$ := list of access counts per region
       $t_w$ := start time for request statistics window
2: $counters[r_{current}] := counters[r_{current}] + 1$
3: if $t_w + \Delta t < Time(req_o)$ and $\sum counters \geq n$ then
4:     $r_{current}$ := region with greatest # of accesses
5:     $counters := Zeroes()$                    ▷ reset the counts
6:     $t_w = Time(req_o)$                       ▷ Reset the window start time
   return $r_{current}$
```

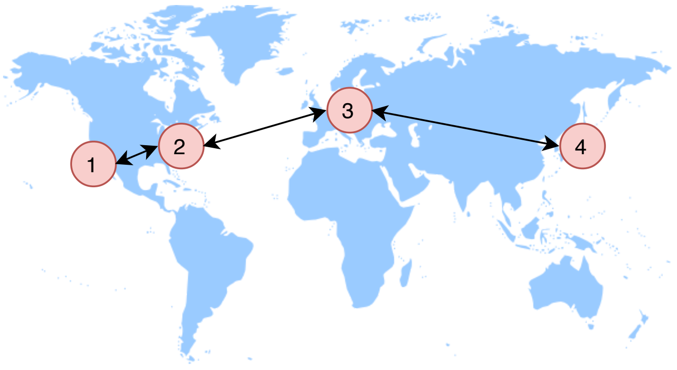

Fig. 1: Exponential moving average topology; regions have left and right neighbors

ID assignment is not arbitrary and represents the proximity of regions to each other. We say that a region i has a left neighbor $i-1$ and a right neighbor $i+1$, as shown in Figure 1. Regions 1 and n have only one neighbor.

Region IDs are then used in calculating the region of average access with the following EMA formula:

$$e_s = \alpha r + (1-\alpha)e_s$$

where e_s is the EMA, and α is the parameter to control how much weight new requests have on the average. Higher α places higher weight to the new object requests.

We also use parameter ϵ to prevent the migration of the object until its average is within the ϵ of the new region ID. This helps prevent unnecessary object migrations or jitter. We present EMA policy in Algorithm 4.

Algorithm 4 EMA

```
1: Initialize:
       $req_o$ := request for object $o$
       $r_{current}$ := current location of $o$
2: $e_s := \alpha * RegionId(req_o) + (1-\alpha) * e_s$     ▷ $e_s$ is EMA value
3: $r_{next} := Round(e_s)$
4: if $Abs(e_s - r_{next}) \leq \epsilon$ then
5:     $r_{current} := r_{next}$
   return $r_{current}$
```

D. Center of Gravity Policy

Center-of-gravity (CoG) policy fixes the shortcoming of the EMA policy and can be applied efficiently to any region topology. Instead of relying on preconfigured region IDs to represent region proximity, the CoG policy uses distances between datacenters to construct a fully-connected weighted graph, with vertices representing the regions and edge weights standing for the distances d between the regions.

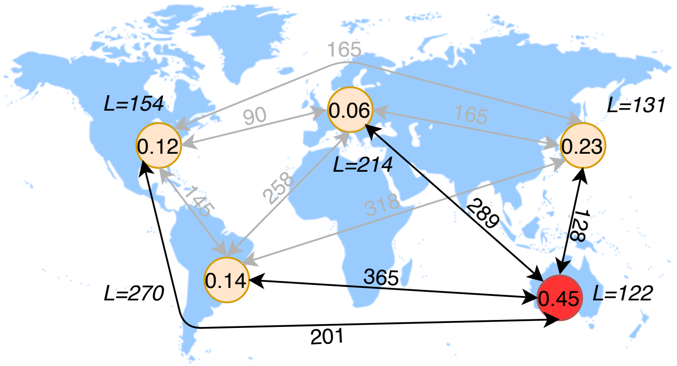

Fig. 2: Computing CoG Weights. Australia is the owner, since it is closest to the workload center of gravity and yields the best latency

We use the topology graph to compute the center of gravity for a workload. A policy assigns each vertex a score $w_z o$ representing the proportion of the workload generated at region z for object o. Knowing all the distances and the workload distribution across regions, policy can compute the region that minimizes the average latency. To that order, a migration policy calculates the latency score L_r for every region r: $L_r = \sum_{z=1}^{Z} w_{zo} d_{rz}$, where d_{rz} is the distance from region r to region some region z. After computing the L-scores, a policy picks the region with lowest score. We describe this process in Algorithm 5 and visually illustrate it in Figure 2.

Algorithm 5 CoG policy

```
1: Initialize:
       $req_o$ := request for object $o$
       $r_{current}$ := current location of $o$
       Let $w_{zo}$ be the workload distribution for object $o$
       Let $R$ be a set of all regions: $R := 1..Z$
2: for each $r \in R$ do
3:     $L_r = \sum_{z=1}^{Z} w_{zo} d_{rz}$
4: $r_{current} := r_1$ such that $\forall r_1, r_2 \in R : L_{r_1} \leq L_{r_2}$
5: return $r_{current}$
```

Since the CoG policy depends on knowing the workload distribution w_{zo} across all regions for object o, it is imperative we compute it as accurately as possible. The simplest way to compute this workload distribution is similar to computing the accesses in the majority-access policy. We can establish a time interval over which to count requests from different regions and use that data to get the proportions of access at each region. As with the majority-access policy, we need to ensure some minimum number of requests are available to make an accurate computation.

Another approach is to continuously estimate the workload ratio for each region. We can adjust each regions' workload weights after every request. The region from which a request is received improves its weight score, while all other regions' cumulative score degrades by the same amount.

V. Evaluation

We evaluate our migration policies with comprehensive simulations and show how they perform under different static and dynamic locality conditions. First we study individual object placement policies and how their parameters affect the performance. Then, we conduct a comparison study to compare different migration strategies with each other. In the first two sets of experiments we do not use balancing component to see which policy can yield the best object placement without any other constraints. Finally, in the last set of experiments we test the load-balancing component and illustrate how it impact the latency in certain workloads.

For our simulations we consider a geo-distributed datastore deployed over 15 regions following the AWS region topology, with the inter-region latency taken from [1]. We simulate clients across regions that make requests to a single object at a time. We estimate the latency of the request as the round-trip-time (RTT) between the client's region and the region owning the object. Additionally, we assume that migration between regions to take one inter-region RTT, and if some requests comes it before the migration has completed, this request is penalized by the time required to finish the migration. For the purpose of simulation we ignore the costs of replication, as these are database dependent and beyond the control of the migration policy.

We used a few distinct workloads to simulate the different access locality patterns: locality, changing locality, no locality and split ownership. We used a pool of N migration objects with each object having a unique id in a range $[0, N)$.

In locality workload, the probability of selecting an object at every region is governed by a Normal distribution $\mathcal{N}(\frac{N}{Z}z, \sigma)$ over a ring of integers modulo N called $\mathbb{Z}_N$, where Z is number of regions and z is a region id in a range $[0, Z)$. For our typical setup with 15 regions and 3000 objects, this translates to $\mathcal{N}(200z, \sigma)$. Using normal distribution to drive object access allows every region to use all objects, but puts a higher probability of accessing objects with ids close to the region's mean. We further subdivide the locality workload into two: medium locality and high locality. The workloads differ only in their standard deviation σ, with high locality workload having $\sigma = 50$ and medium locality workload $\sigma = 100$. Larger σ means that every region is more likely to access objects further from its mean and reach out to the objects in access locality of neighboring zones.

Our changing locality workload is similar to the locality one, except the mean of each normal distribution slowly changes over time to mimic diurnal workload variation. This makes a region more likely to access different objects as time progresses and requires the policies to adapt in real time. No locality workload makes each object to have an equal probability of being selected by a client from any region. Finally, the split ownership workload assigns every object to up to k distinct regions, with an object having an equal chance of being requested at any of its assigned regions.

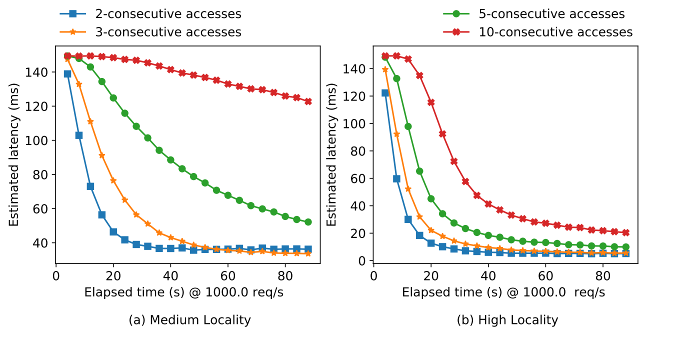

Fig. 3: Different values of n in n-consecutive policy under the locality workloads

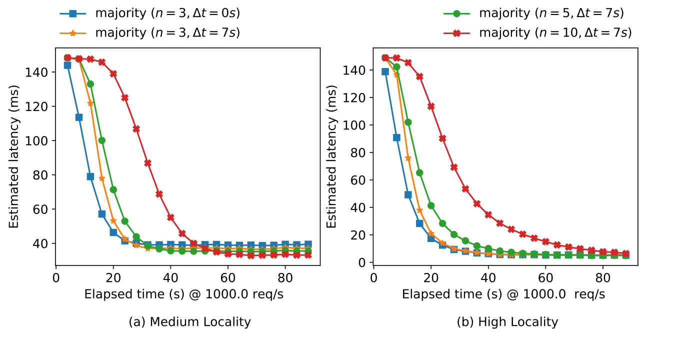

Fig. 4: Majority access policy with different parameters under the locality workloads

A. Tuning Migration Policy Parameters

First we conduct a series of simulations on each of the individual policies to evaluate how various policy parameters impact the performance. We conduct these experiments with our locality workloads on 3000 objects initially placed in random regions.

n-consecutive accesses policy uses consecutive requests coming from a region as the heuristic for optimal object placement. This policy is quick to react for low value of n, but this also makes it possible for objects to continuously move between regions if access is bursty at each region. Figure 3 shows the policy with different values of n in two locality workloads: medium locality and high locality. Larger values of n perform poorly in medium locality setting due to the inability to reach a threshold of n consecutive accesses on many objects whose access is shared between adjacent regions. High locality workload improves all tested values of n.

Majority accesses policy keeps tab on every region and decides on the migration based on which region has requested an object the most over some time period Δt. The policy also enforces the minimal history of n requests before it can make a placement decision. In Figure 4 we show majority access policy in locality workload.

The policies with higher statistics collection interval Δt and minimum history n shows slower adaptation to the workload locality, however more request statistics also allows such

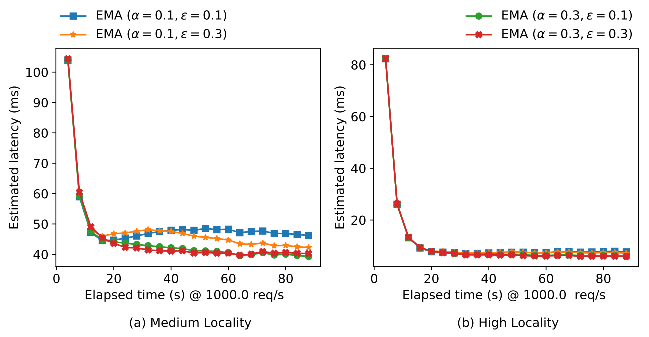

Fig. 5: EMA policy with different α and β under the locality workloads

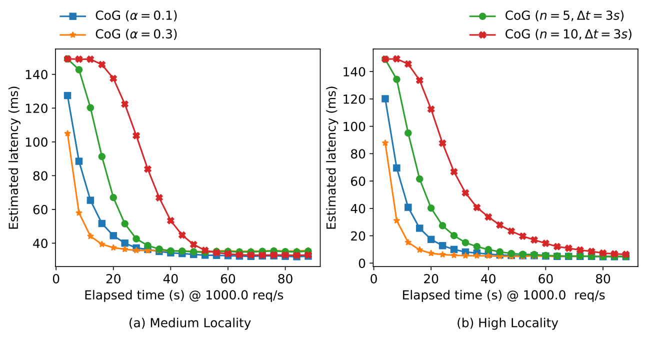

Fig. 6: CoG policy with different parameters under the locality workloads

policies to eventually make better placements, especially in the medium locality workloads.

EMA policy computes the average region of the object access. Unlike the previous two policies, EMA may require multiple steps to migrate the object from one region to another. This is because in most cases the policy cannot move objects to an arbitrary region and can only migrate data to either its left or right neighbor as the average gradually changes in response to the workload. In EMA policy, the parameter α controls how much weight is put to the newer requests, with higher α causing the policy to put more weight to the recent requests. Parameter ϵ controls how close to average must get to the region ID before the migration happens. This parameter reduces jitter, as it prevents small short-term workload irregularities from causing the migrations of objects with access locality split almost evenly between regions.

Figure 5 illustrates the performance of EMA policy in medium and high locality settings. Lower values of α make EMA perform slightly worse than higher α. This is likely due to the fact that higher α makes the policy more eager to migrate objects at the later stages of execution when an average score has been well established over many requests. Parameter ϵ also affects the performance by making policy more likely to migrate borderline objects whose locality is nearly evenly split between adjacent regions. Even though medium locality workload shows some variation due to the difference in parameters, high locality situation nearly erases all the differences.

CoG policy uses the topology information to compute central location for each object given its access patterns. Our CoG policy comes in two flavors: window based and continuous. In window based, the policy collects usage statistics similar to the majority access policy, and uses the per-region access counts to compute w_z for each region z in a given time window Δt. Continuous CoG policy does not compute actual workload W from the request history, and instead assigns a workload score to each region. The score is updated with every request and favors more recent requests to the older ones.

We illustrate the performance of CoG policy in Figure 6. The continuous version of the policy appears eager to migrate objects at first, since it reevaluates the placement after every request, thus making the decisions quickly. It is worth noting, that in some cases, this quick initial decision making may not be correct, since at start-up the policy assumes equal access locality at each region. The window based CoG is slower to react initially, but it eventually get to similar levels of performance as the continuous option.

B. Policy Comparison

In this set of experiments we compare our four different policies together under a variety of workloads simulating various access locality patterns. We also add a *never-migrate* policy to demonstrate the performance of static, non-migrating solution. An *always migrate* policy moves an object to a different region every time it is accessed from another region, illustrating the most aggressive object migration possible. We also measure the number of object migrations each policy performs to estimate the impact on the cross-region network utilization. The lower migration number indicates that the policy is making less unnecessary movements and conserves the network resources.

First we start with no locality workload that accesses every object in uniform manner across all regions. The lack of apparent locality may cause unnecessary migration in the policies, both degrading the performance and flooding the network with many messages. Figure 7 shows the results. As expected, majority accesses policy performs the worst in this experiment, since it erratically moves objects around and incurs the migration penalty. n-consecutive accesses policy matches the performance of never migrate baseline, since n-consecutive with $n = 3$ cannot per from any migrations in this workload. CoG and EMA policies were able to improve the performance by moving the objects to a more central location in the topology, allowing on average faster access from any region. This, however, is possible only because we do not take load balancing into consideration in this experiment, allowing all objects to migrate to a single region.

The medium locality workload, shown in Figure 8 allows all policies to improve the access latency. Unlike no-locality workload, EMA shows the worst performance in this experiment, barely improving over the always migrate baseline. CoG policy showed the best performance in this experiment,

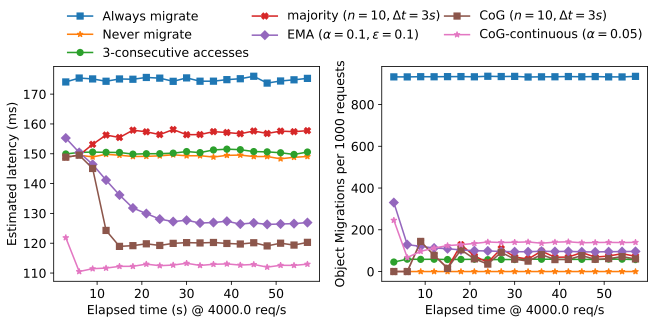
Fig. 7: No locality workload

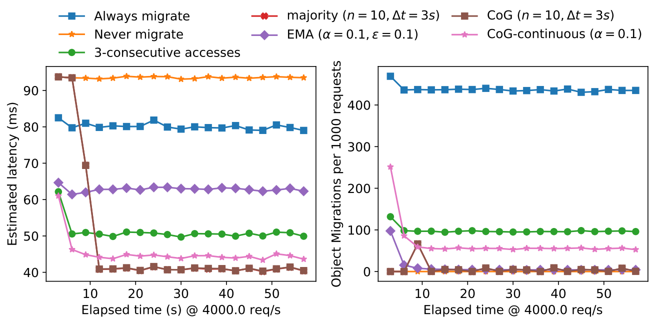
Fig. 9: Medium locality workload in 5 regions

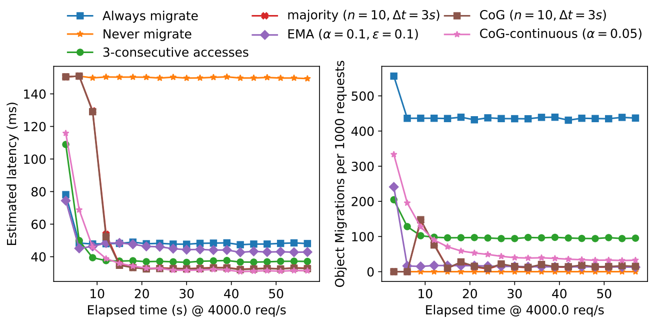
Fig. 8: Medium locality workload in 15 regions

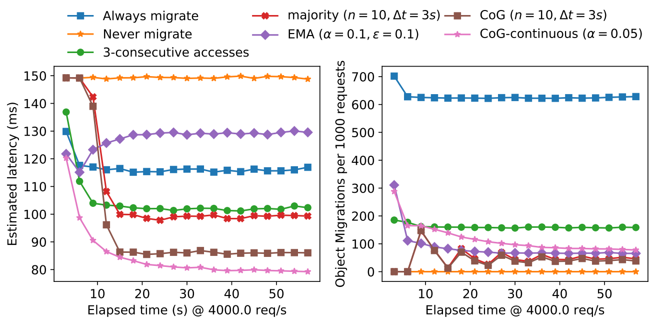
Fig. 10: Up to 3 regions share object

although it was tied with the majority accesses policy. This is because the workload we used favors a single region for object with other accessing being in the neighboring regions, making a simple majority region heuristic very effective. n-consecutive accesses policy trails the CoG and majority accesses since it exhibits jittering behavior and moves the objects shared between neighboring regions often.

The relative performance of the policies, however, may change for different clusters. For example, Figure 9 shows the same medium locality, but with 5 regions instead of 15. In particular, we used inter-region latency corresponding to California, Ireland, Japan, Australia and Brazil AWS regions. We observed that in this topology EMA performs relatively better compared to the always migrate approach. This is likely because the topology and the region ID assignment is more favorable to EMA.

In some instances, the access locality for an object may be shared between multiple regions. When this happens with 3 or more regions, it is often possible to find another region in the middle to host the object and optimize the latency for all regions haring it. Figure 10 demonstrates this scenario by having every object equally accessed between utmost 3 random regions. Non-topology aware policies, such as n-consecutive and majority accesses do not provide as much benefit in this scenario as CoG policy. This is because this policies use single-region heuristic, while the access locality is shared across many regions. Majority and n-consecutive accesses policies still improve the latency, since they are likely to migrate the object to at least one of the region accessing it, therefore reducing latency of all requests from that region. CoG, on the other hand can optimize the placement for all 3 regions sharing the object, and such placement may not always be in one of these 3 regions.

Surprisingly, EMA policy performed the worst in this workload. Even though EMA tries to compute the average region for a workload, it is not aware of the region topology and relies on imperfect user assignment of region IDs to estimate the proximity between regions. We followed the order of regions as it appears in the table from [1] to assign IDs, but this leads to situations when regions' neighbors are not necessarily geographically close. In this topology, as EMA gradually moves objects to the average region, it actually overshoots the best region for many objects, causing the latency to degrade after the initial improvement. It is still worth noting that EMA performed better than purely static object placement.

Workload locality often changes overtime in response to various environmental factors. In Figure 11 we present the performance of our migration policies under such changing locality workload. In this experiment, we changed the access locality relatively fast, causing the object's locality completely change the region in around 20 seconds of elapsed simulated workload. Similar to static workload with single region dominating object's use, majority accesses and CoG performed similarly and showed lowest latency. Under these highly dynamic conditions, n-consecutive access policy performed on par with CoG and majority accesses, due to its ability to quickly react to the changes. This policy, however, causes many unnecessary migrations over time. EMA policy

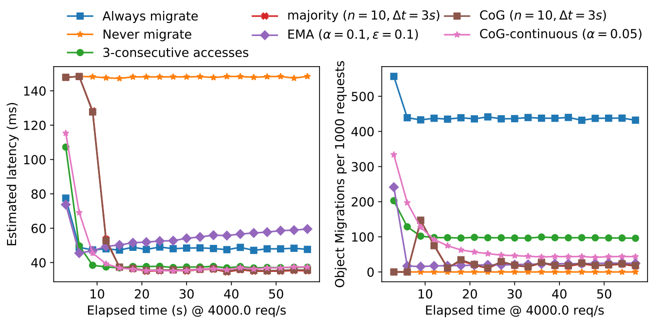

Fig. 11: Drifting locality in 15 regions

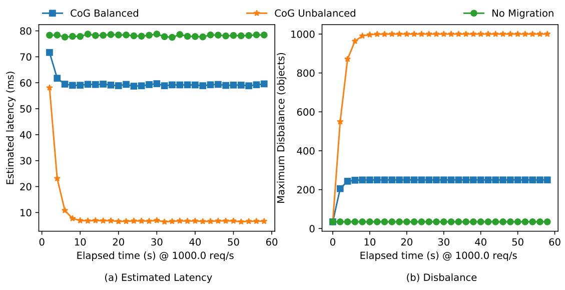

Fig. 12: Load balancing with skewed access pattern

continues to suffer from complex geography, slow migration and overshooting the optimal regions due to bad region ID assignments. In a separate longer running experiment we observed this configuration of EMA to settle at the latency of 59 milliseconds, still significantly outperforming random partitioning with no migration.

C. Load Balancing

In this experiment we test the load balancing component of our migration policy shown in Algorithm 1. We use the CoG policy to determine the data placement before adjusting the migration decision to adhere to the balancing constraints. We used a 1000 object, 5 region setup corresponding to AWS California, Ireland, Japan, Australia and Brazil regions. We applied a highly skewed workload with 5 times as many requests coming from a single region (California) than all other regions combined.

Under these conditions, unbalanced solution is likely to migrate most objects to a single region, optimizing for latency. While this has a potential to provide better access latency, in real system this it is likely to overwhelm the capacity of the datacenter. Balanced policy, on the other hand, minimizes the latency while staying within the allowed balancing envelope.

To illustrate this trade-off, we measure the load difference between the most and least loaded region along with the best possible latency. For simulation simplicity we do not implement request balancing and only show data balancing. The capacity of each region is limited to $\frac{1}{4}$ of all objects.

The results of our skewed workload experiment, shown in Figure 12, suggest that while unbalanced policy has a potential to deliver substantially better performance due to the access locality, it may significantly overload the system by skewing all the work to a single region. In unbalanced CoG, the difference between the region owning most objects and the one with least number of objects grows fast as the policy starts adjusting to locality. When CoG policy takes into account the balancing constraints, it can only place so much data into a single region before being forced to find the next closest region with free capacity. Balanced CoG shows a relatively modest improvement of roughly 25% compared to static object placement, while unbalanced CoG may potentially reduce the latency multiple-fold.

VI. Concluding Remarks

Geo-distributed cloud databases solve the problem of placing data close to the user. Strongly consistent global databases, however, have to rely on partial replication to keep costs low and performance high. This requires datastores to abandon static sharding/placement of data and invent ways to strategically place the data where it is needed the most.

We formulated the criteria for data-migration protocols/policies to optimize for access latency, load balancing, data-collocation, and network usage. We introduced four migration policies for latency optimization. Our data-migration policies help reduce the latency by as much as 70% compared to non-migration approach in workloads exhibiting some access locality. Tuning the policy parameters with values best fit for anticipated workloads also improves the performance. However, some policies are hyper-sensitive to such tuning, while others are more forgiving leading to easier maintenance. We also demonstrate that the best policies are aware of the region topology and are conservative to migrate the data until sufficient usage statistics has become available. On the other hand, simple heuristics such as n-consecutive accesses from the same region prove to be surprisingly effective for determining optimal data placement for workloads that show good access locality.

More complex policies may take advantage of larger pools of request data to learn the workload patterns and be proactive instead of reactive. For instance, an object may migrate from one region to another at the same time workload access locality moves with the diurnal pattern. Our current policies are reactive and take some time to sense the change in the workload to perform a migration.

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under XPS-1533870, XPS-1533802

References

[1] M. Adorjan. AWS inter-region latency. https://www.cloudping.co/, 2018.
[2] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, pages 17–32. USENIX Association, 2010.

[3] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. WPaxos: Ruling the Archipelago with Fast Consensus. *ArXiv e-prints*, March 2017.

[4] Amazon. Amazon dyanmoDB - nonrelational database for applications that need performance at any scale. https://aws.amazon.com/dynamodb/, 2018.

[5] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the shards: Managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460. USENIX Association, 2018.

[6] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 103–112. ACM, 2014.

[7] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381. USENIX Association, 2014.

[8] J. D. Brutlag, H. Hutchinson, and M. Stone. User preference and search engine latency. *JSM Proceedings, Qualtiy and Productivity Research Section*, 2008.

[9] Cockroach Labs. Cockroachdb architecture overview. https://www.cockroachlabs.com/docs/stable/architecture/overview.html, 2018.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[12] D. Corporation. Introduction to apache cassandra. Technical report, July 2013.

[13] S. Kadambi, J. Chen, B. F. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. Garcia-Molina. Where in the world is my data. In *Proceedings International Conference on Very Large Data Bases*. VLDB Endowment, 2011.

[14] Microsoft. Cosmos DB - globally distributed, multi-model database service. https://azure.microsoft.com/en-us/services/cosmos-db/, 2018.

[15] MongoDB. MongoDB for GIANT Ideas. https://www.mongodb.com/, 2018.

[16] F. Nawab, D. Agrawal, and A. El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1221–1236. ACM, 2018.

[17] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel, et al. Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine*, 55(2):70–78, 2017.

[18] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[19] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon. Follow the sun through the clouds: Application migration for geographically shifting workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 141–154. ACM, 2016.

[20] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. In *USENIX Annual Technical Conference*, 2011.

[21] K. Tsakalozos, V. Verroios, M. Roussopoulos, and A. Delis. Live vm migration under time-constraints in share-nothing iaas-clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2285–2298, 2017.

[22] B. Yu and J. Pan. Location-aware associated data placement for geo-distributed data-intensive applications. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 603–611. IEEE, 2015.

[23] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi. Global-scale placement of transactional data stores. In *EDBT*, pages 385–396, 2018.