View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Columbia University Academic Commons

GRANDET: A Unified, Economical Object Store for Web Applications

Yang Tang, Gang Hu, Xinhao Yuan, Lingmei Weng, Junfeng Yang
Columbia University
{ty,ganghu,xinhaoyuan,lingmei,junfeng}@cs.columbia.edu

Abstract

Web applications are getting ubiquitous every day because they offer many useful services to consumers and businesses. Many of these web applications are quite storage-intensive. Cloud computing offers attractive and economical choices for meeting their storage needs. Unfortunately, it remains challenging for developers to best leverage them to minimize cost. This paper presents GRANDET, a storage system that greatly reduces storage cost for web applications deployed in the cloud. GRANDET provides both a key-value interface and a file system interface, supporting a broad spectrum of web applications. Under the hood, it supports multiple heterogeneous stores, and unifies them by placing each data object at the store deemed most economical. We implemented GRANDET on Amazon Web Services and evaluated GRANDET on a diverse set of four popular open-source web applications. Our results show that GRANDET reduces their cost by an average of 42.4%, and it is fast, scalable, and easy to use. The source code of GRANDET is at http://columbia.github.io/grandet.

1 Introduction

Web applications are getting more ubiquitous every day because they offer many useful services to consumers and businesses. Examples include Instagram and Flickr for hosting, processing, and sharing images; YouTube and Vimeo for videos; Pandora and Spotify for music; and Dropbox and Google Drive for files.

Many of these web applications can become quite storage-intensive. At the initial deployment of these applications, a single server might be enough to host the data objects from their limited number of users. However, as they become more successful, hosting images, videos, files, and other data objects from millions of users, their storage needs increase dramatically. For instance, Facebook has over 500 million users with 260 billion images, totaling 20 PB [8]. Dropbox has more than 50 million users, who save 500 million files daily [16].

Cloud computing provides an attractive, economical choice for meeting the storage (and computational) needs of web applications. Besides the usual benefits of elastic scaling and no hardware (over-)provisioning, each cloud platform typically supports a range of storage options with different performance, durability, and price characteristics. For instance, Amazon Web Services (AWS) supports non-persistent virtual disks (*instance store*), persistent virtual disks (*elastic block store*, or EBS), and key-value object store (*simple storage service*, or S3). Each of these options typically has more sub-options, such as EBS on SSD or magnetic disks, and S3 with reduced redundancy or infrequent access. This rich set of options gives developers the flexibility to pick the best options that meet their applications' needs. Unsurprisingly, most web startups today choose to deploy their apps in the cloud, so that they can focus their scarce manpower and funding on features of their applications [33].

Unfortunately, despite all these storage options, it remains quite challenging for developers to best leverage them to minimize cost. For simplicity in programming, it is common practice for a developer to pick a store she thinks is the best and places all objects of the same data collection (*e.g.*, all images) within the store. However, at its core, minimizing cost requires developers to make fine-grained decisions on *which* store is the best for *which* object. The reason is that the pricing models of different stores are quite complex and subtle, depending on such factors as the size of the object, the number and types of the access requests, and the amount and destinations of the network transfers. Two objects in the same data collection may differ hugely regarding these factors, and therefore should be placed at different stores. Consider two AWS stores, EBS on SSD which charges a high price for storage and nothing for requests, and S3 which charges a moderate price for both storage and requests. A large but cold (*i.e.*, few read and write requests) object should be stored in S3, whereas a small but hot object should be stored in EBS. It is both non-intuitive and im-

practical to require developers, especially those at startups with scarce manpower and funding, to make such fine-grained placement decisions on a per-object basis.

In addition, many of the factors affecting price are highly dynamic, frequently requiring objects to be migrated from one store to another to minimize cost. For instance, the hotness of an object varies over time, so that the best store for the object now may be the worst fit in the future. Even the pricing models change over time due to technology improvements [5] and competitions [6]. It is impractical to require developers to predict these changes accurately or migrate objects manually.

Lastly, different stores provide heterogeneous interfaces, and a web application written against one storage interface (*e.g.*, the file system interface) may not be able to use another more economical storage option easily or at all. Many popular web applications, such as MediaWiki (the most popular wiki app) and WordPress (the most popular blogging app), still store data objects such as images in file systems. To run these applications in the cloud without significant modifications, developers have to store the data objects, however large they are, in a file system on top of EBS, an option potentially much more expensive than storing the objects in S3. While newer web applications tend to adopt S3, they may still manipulate the data objects using existing utilities that require the file system interface. Examples include a photo gallery using `ImageMagick` to process images or generate thumbnails, a video sharing application using `ffmpeg` to convert video formats, and a file sharing application using `bzip2` to compress files. Thus, developers have to explicitly move the objects between S3 and the file system. These movements, if frequent, are both complex to program and expensive to execute, because S3 charges for both requests and network transfers.

Because of these reasons, it is difficult for developers to place objects optimally for minimizing cost. The cost of misplacement can be quite high. At a micro level, each `PUT` request on S3 costs as much money as storing 5 MB of data for a day; so it is extremely costly to store frequently accessed data objects on S3. The storage cost on EBS is up to 8× as much as on S3; so putting an infrequently accessed large object in a file system on EBS is expensive, too. At a macro level, our experiments show that misplacement costs up to 572% more.

We present GRANDET, a storage system that greatly reduces storage cost for web applications deployed in the cloud. GRANDET provides both a file system interface and an S3-like key-value interface, supporting a broad spectrum of web applications. Under the hood, GRANDET supports multiple heterogeneous stores, and unifies them by placing each data object at the store deemed most economical. Specifically, for each supported store, GRANDET maintains a profile capturing the store's pricing model, availability, durability, and consistency guarantees, and performance such as latency. It updates the performance part of this profile by periodically running its *profiler*, and the other parts based on crawling or user-supplied configurations. Given a data object, GRANDET runs its *predictor* to predict the future workload on the object, and its *decider* to determine on a fine-grained, per-object basis the most economical store that meets the default or developer-specified quality of service (QoS) requirements—even the default is better than the typical web practice. It preserves the availability, durability, and consistency that the cloud stores provide. When the workloads or pricing models change, GRANDET migrates objects automatically as needed to reduce cost. We explicitly designed GRANDET to be extensible so that developers can add new stores easily.

We implemented GRANDET in AWS and evaluated GRANDET on a diverse set of four popular open-source web applications, namely CumulusClips, Piwigo, Elgg, and FileSender. Our results show that:

1. GRANDET greatly reduces the cost spent on storage for web applications. On average, GRANDET can reduce the storage cost by 42.4%.
2. GRANDET has small overhead. It can be deployed with little impact on application performance.
3. GRANDET scales well when the workload increases.
4. Web applications can use GRANDET to save cost with no modification at all, and several lines of changes would reduce the cost even further.

The remainder of this paper is organized as follows. The next section introduces the background of cloud storage services. §3 extends our motivation with a study and an example. §4 describes GRANDET's architecture. §5 shows the data placement strategy. §6 presents the file system interface. §7 describes the implementation. §8 shows evaluation results. §9 discusses some design implications, §10 presents related work, and §11 concludes.

2 Background: cloud storage services

The variety of cloud storage options can be mainly divided into two categories: file storage and blob storage. File storage generally provides a disk or file system interface. Applications can mount it and manipulate data using file system operations such as `open()`, `read()`, and `write()`. Examples of file storage are Amazon elastic block store (EBS), Microsoft Azure file storage, and Google compute engine persistent disks. On the other hand, blob storage generally provides a minimal key-value interface, such as `PUT`, `GET`, and `DELETE`. A blob is normally treated as a whole, and operations such as partially updating a blob are not supported. Examples of blob storage are Amazon simple storage service (S3), Microsoft Azure blob storage, and Google cloud storage.

Even more options are available for each category

Storage service	Type	Durability	Availability	Latency
Instance store	file	ephemeral	99.95%	lowest
EBS (SSD)	file	99.8-98.9%	99.999%	lowest
EBS (magnetic)	file	99.8-98.9%	99.999%	low
S3 (standard)	blob	$1 - 10^{-11}$	99.99%	medium
S3 (reduced)	blob	99.99%	99.99%	medium
S3 (infrequent)	blob	$1 - 10^{-11}$	99.9%	medium
Glacier	blob	$1 - 10^{-11}$	n/a	high

Table 1: *Overview of AWS storage services (January 2016).*

Storage service	Storage (/GB)	Request (/million) PUT	Request (/million) GET	Transfer (/GB) In	Transfer (/GB) Out
EBS (SSD)	0.1	0	0	0	0.09
EBS (magnetic)	0.05	0.05	0.05	0	0.09
S3 (standard)	0.03	5	0.4	0	0.09
S3 (reduced)	0.024	5	0.4	0	0.09
S3 (infrequent)	0.0125†	10	1	0	0.09
Glacier	0.007	50	50	0	0.09

Table 2: *Approximate monthly price for AWS storage services (January 2016).* Prices are shown in dollars. †S3 infrequent access charges a minimum of 128KB storage for smaller objects.

of cloud storage. For instance, Amazon Web Services (AWS) supports four types of stores (see Table 1). *Instance store* provides free, non-persistent virtual disks to an AWS elastic compute cloud (EC2) instance. These virtual disks are non-persistent because they are stored in the physical disks of the host machine that happens to run the EC2 instance. *Elastic block store (EBS)* provides persistent virtual disks, based on either SSD or magnetic. *Simple storage service (S3)* is a key-value store for objects, with standard, reduced-redundancy, or infrequent-access options. *Glacier* is a backup store with an extremely low cost and long read latency (3–5 hours).

Not only do these storage options have different service levels, they also have complex and diverse pricing models. A typical pricing model depends on (1) the total storage size, (2) the number of each type of request, and (3) the amount and destination of network data transfer. Table 2 shows a snippet of the pricing scheme for AWS storage services. Although they have the same data transfer cost, the discrepancies in storage pricing are up to an order of magnitude, and those in request pricing can be as large as three orders of magnitude. No option is cheaper across all dimensions. For example, EBS on SSD does not charge for I/O requests, but its storage price is more than three times as high as S3. By contrast, S3, despite charging less for storage, has high per-request cost.

To further illustrate the pricing discrepancies, let us study how much money it costs to put one data object on each of these storage services. Figure 1 shows the cost with (a) variable object size, and (b) variable number of requests. We exclude data transfer cost in the figures for better clarity, because it is the same for all these services. In each figure, the optimal choice is the minimum of all lines (shaded), and the threshold points are marked. We

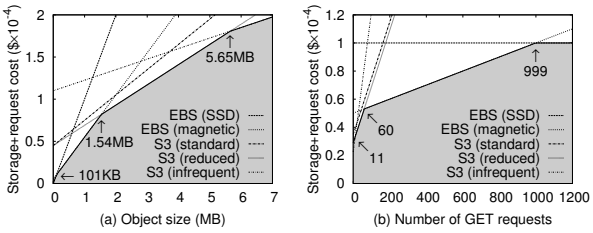


Figure 1: *Monthly cost with (a) variable object size, and (b) variable number of requests.* Each line corresponds to a storage service. Assuming (a) has fixed 100 `GET` requests, and (b) has fixed 1MB object size. Each request counts as one EBS I/O.

Web application	Category	Web application	Category
FileSender	file sharing	selfoss	RSS reader
Piwigo	photo sharing	Tiny Tiny RSS	RSS reader
OpenPhoto	photo sharing	Elgg	social network
CumulusClips	video sharing	MediaWiki	wiki
OpenCart	shopping	LionWiki	wiki
PrestaShop	shopping	Wikka	wiki
Zen Cart	shopping	Drupal	CMS
Wordpress	blog	October	CMS
NibbleBlog	blog	Anchor	CMS
Chyrp	blog		

Table 3: *List of studied web applications.*

can see that the optimal choice depends on both object size and the number of requests, let alone each choice also has different durability, availability, and latency.

Thus, the heterogeneity of service levels and pricing schemes lead to extremely difficult decisions that web applications should make when using cloud storage services. Misplacing data at non-optimal storage locations may not only cause service degradation but also cost a lot of money, negating the benefits that the cloud brings.

3 Extended motivation and example

We motivated the design of GRANDET by studying 19 popular open-source web applications of various kinds, including file sharing, photo and video sharing, shopping, blogging, news-reading, social networking, wiki, and content management systems (see Table 3 for the list). We observed two insights from our study.

Our first insight is that data files have diverse sizes and access patterns. For example, the original photo or video files are large, while the thumbnails are small. In addition, some files are frequently read, such as a celebrity's photo, while other files stay cold after they are stored, and the access pattern of files may change over time. For example, Figure 2 shows the distribution of file size for the Piwigo photo sharing application from a scaled-down workload based on real-world statistics (see §8 for workload details). About 30% of all files are original images (7–12MB), 13% are large thumbnails (600–800KB), 30% are small thumbnails (90–160KB), and the

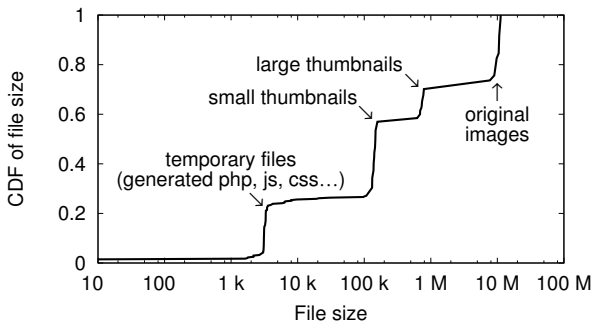


Figure 2: *CDF of file size for Piwigo.*

rest are temporary files. The reason that there are fewer large thumbnails is that Piwigo generates them lazily, and many photos are not accessed yet. This diversity gives us perfect opportunity for optimization, because file size and access pattern are the two most important factors affecting storage cost, as we have shown in §2.

Our second insight is that, despite their complexity, all the 19 applications manipulate data files only in simple ways. Each file corresponds to a logical data object, such as a photo or a video. These files are written sequentially and free of sub-file updates. Therefore, both file storage and blob storage are capable of storing these data objects.

Because of these two insights, we design GRANDET as a transparent gateway for a variety of heterogeneous storage services. Data objects are always stored at the optimal service based on the characteristics of the data and workload as well as the pricing and network condition. They are also automatically migrated among the storage services when the workload, pricing, or network condition changes. Next, we present a motivating example about how the CumulusClips video sharing application [14] stores and uses data, to illustrate how GRANDET can help it reduce storage cost.

When a user uploads a video file, CumulusClips stores it to the file system. It then calls an external program, `ffmpeg`, to convert the uploaded file into multiple formats, such as a high-definition version for broadband connections and a low-definition version for mobile devices. It also generates a static thumbnail of the video. All the derived files are stored in the file system, too. Later, viewers of the website see a list of thumbnails. When the viewer clicks into a thumbnail, depending on her platform, one of the converted video is played.

GRANDET helps CumulusClips by transparently handling the storage for all files. Although GRANDET internally stores data as key-value objects, it is mounted to CumulusClips's `uploads` directory as a file system, and no modification to CumulusClips's source code is required. Whenever CumulusClips wants to write a file to the directory, GRANDET puts the file to its optimal storage service based on its prediction of the file's workload.

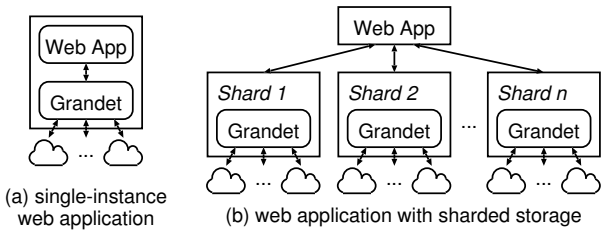


Figure 3: GRANDET *deployment scenarios.*

For example, it would put a small thumbnail file on EBS if it predicts that the file would be frequently read, but put a large high-definition video file on S3. GRANDET also migrates data over time to reflect latest conditions. For example, if an unknown video on S3 suddenly becomes a sensation (the "slashdot effect"), then GRANDET would move it to EBS for cheaper request cost.

4 Architecture

We now give an overview of GRANDET's deployment scenarios and present the architecture of GRANDET.

4.1 Overview

GRANDET unifies multiple heterogeneous cloud storages into a single service. Its primary goal is to reduce storage cost for web applications. Thus, instead of running standalone GRANDET servers that would incur additional cost, GRANDET leverages piggyback deployment.

Figure 3 shows two typical deployment scenarios. For single-instance web applications, the GRANDET service simply co-locates on the same machine with the application (Figure 3(a)). Large-scale web applications (*e.g.*, MediaWiki [43]) typically shard their files into multiple storage servers, each storing a disjoint subset of the files, and mount them via a distributed file system (*e.g.*, NFS). In this case, each shard independently runs a GRANDET service on it (Figure 3(b)). Because the files stored on each shard are disjoint, GRANDET does not need to worry about consistency among shards.

GRANDET does not introduce new availability, durability, or consistency concerns due to two reasons. First, each object is stored on exactly one cloud storage; so the availability, durability, and consistency of GRANDET's storage is as good as the underlying cloud storage. The application developer can specify the minimum availability, durability, and consistency requirement on a per-object basis (see §4.3). Second, since the GRANDET service itself resides on the same server as the web application or the storage shard, they share the same availability.

4.2 GRANDET components

Figure 4 shows the components inside the GRANDET service. The GRANDET frontend exports a key-value SDK for various languages as well as a general file system interface to the web application or storage shards. The

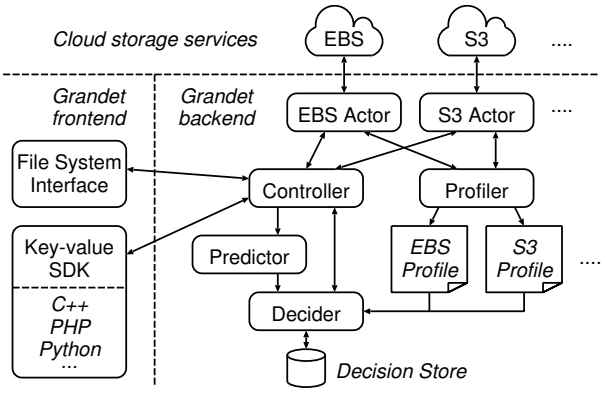


Figure 4: GRANDET *components.*

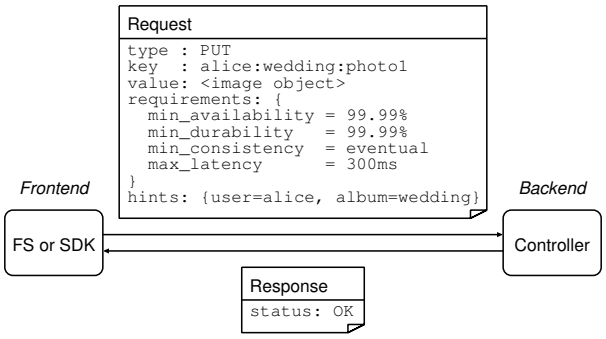


Figure 5: *An example of* PUT *request and response.*

frontend and the backend communicate through Unix domain socket IPC.

The GRANDET backend stores data as key-value objects. It consists of five components. The `Controller` handles all requests from the frontend, and coordinates all the other backend components. A set of `Actors` executes storage actions on a variety of storage backends. The `Profiler` periodically probes the current pricing model and network conditions for each storage backend, and stores them as *profiles*. The `Predictor` keeps track of the frequency of all `PUT`, `GET`, and `DELETE` requests, and predicts future request patterns. The `Decider` decides upon the best storage option based on the application's requirements, the predicted request pattern, and the storage profiles. Decisions are kept on the *decision store* in Redis [30], further persisted on EBS or S3. Note that the durability of the decision store is not critical, because it can be fully rebuilt by scanning objects on all storage.

4.3 GRANDET workflow

All communications start with the application[1] sending a request to the `Contoller` by using either the key-value SDK or the file system interface, where the latter internally represents files as key-value objects (see §6). Regardless of frontend, the request is one of the following:

PUT. The application requests to store a data object to GRANDET's storage (Figure 5). The application should assign a unique *key* to the data object based on its own needs. For example, a photo sharing application may assign the image file that Alice uploads to her Wedding album the key `alice:wedding:photo1`. The *value* of the data object can be an arbitrary length of binary content.

Along with the `PUT` request the application can specify its *requirements* on the storage service for this particular data object. The requirements include the minimum availability, durability, and consistency required, as well as the maximum latency allowed. Only the services that meet these requirements are considered as candidates for storing this data object (see Table 1 for an overview of storage services). Requirements are optional. If the application does not specify requirements, then GRANDET assumes all non-ephemeral (*i.e.*, not the EC2 instance store) and moderate-latency (*i.e.*, not Glacier) services can be chosen. It is worth mentioning that even this default assumption provides better guarantees than a typical application's setup, since both EBS and S3 are at least 20× more reliable than typical commodity disks [3].

The application can also give *hints* to GRANDET for a better placement decision. Hints are also optional, and we have implemented two default hints. §5 discusses the placement strategy and default hints in detail.

Upon receiving the request, the `Controller` first asks the `Decider` for the placement decision, which in turn looks at the current profile for each storage service and asks the `Predictor` for the predicted future request pattern. Based on this, the `Decider` finds the most cost-effective storage choice that satisfies all the application's requirements, memorizes the choice at the decision store, and returns the choice to the `Controller` Then the `Controller` tells the corresponding `Actor` to store the data object to the actual storage and notifies the `Predictor` to bookkeep this action. Finally it tells the application that the `PUT` has completed.

GET. The application requests to retrieve a data object from GRANDET's storage. The `Controller` asks the `Decider` to recall the previous placement decision from the decision store, and then asks the corresponding `Actor` to retrieve the data object from the actual storage. The `Controller` also asks the decider to check if the optimal placement decision would change because the current workload, pricing scheme, and network conditions may have changed. If not, it notifies the `Predictor` to bookkeep this action, and returns the data object to the application. Otherwise, it also migrates the data object to the new storage service and deletes the old copy.

DELETE. The application requests to delete a data object from GRANDET's storage. The `Controller` asks

[1] If deployed with sharded storage, it is actually the shard that sends the request. However, there is no difference from GRANDET's view.

```
// PHP SDK interface
function put($key, $value, $requirements=[], $hints=[])
function get($key)
function del($key)

// Example: PUT an image with requirements and hints.
require_once 'grandet.phar';
grandet\put('alice:wedding:photo1', $uploaded_image,
            ['min_availability_required' => 99.99,
             'min_durability_required'   => 99.99,
             'min_consistency_required'  => 'eventual',
             'max_latency_required'      => 300],
            ['user' => 'alice', 'album' => 'wedding']);

// Example: PUT with no requirements and default hints.
grandet\put('alice:wedding:photo2', $another_image);

// Example: GET an image.
$image = grandet\get('alice:wedding:photo1');
```

Figure 6: GRANDET*'s PHP SDK and usage examples.*

the `Decider` to recall the previous placement decision from the decision store, and then asks the corresponding `Actor` to delete the data object from the actual storage. It also notifies the `Predictor` to bookkeep this action.

4.4 Frontend interface

GRANDET has two types of frontend interface. The key-value SDK provides bindings for these requests for various programming languages such as C++, PHP, and Python. For instance, Figure 6 shows GRANDET's PHP interface and examples of putting and getting an image. The interface is similar to current cloud blob storage services such as S3. Therefore, web applications that are already aware of S3-like blob storages can just switch to GRANDET's SDK and seamlessly get all the cost-savings that GRANDET brings.

For applications that only work with file systems, GRANDET also provides a file system interface using FUSE, which applications can mount to their data directory directly. §6 describes it in detail.

5 Deciding data object placement

The cornerstone of GRANDET is the decision engine for placing each data object onto the optimal storage service. It makes a decision each time the application `PUT`s or `GET`s a data object. The decision engine closely follows the pricing model of all storage options. As mentioned in §2, a typical pricing model consists of three factors: storage (data size and lifetime), number of requests, and data transfer. The data size is known, and transfer prices are usually the same for all services within the same cloud region. Therefore, the key to making placement decision is predicting the future access pattern of the data object.

5.1 Prediction of access pattern

For each request of a certain object, the predictor uses the request's metadata to classify the object into the class of objects similar to this object. The metadata include the object size, the object name, the requirements of the request, and other hints (§5.3) provided by the developer.

For each class, the predictor keeps track of the number of `GET` and `PUT` requests issued on the objects in this class recently, and it also records the number of recently accessed objects and the average lifetime of the objects in this class. Each record is kept for r seconds.

Suppose that for the class the current object belongs to, there are g `GET` requests and p `PUT` requests recently, and there are n objects accessed in this class, then the predictor would predict that in the following t seconds, there would be $\frac{gt}{nr}$ `GET` requests and $\frac{pt}{nr}$ `PUT` requests for this object. It would also predict the object's lifetime to be the average object lifetime in its class.

5.2 Decision making

GRANDET's decider works with the predictor to decide where to place the object. It uses the object size and the predicted access pattern to make the decision. For each backend, GRANDET's decider uses its pricing model to calculate the storage cost of the object in its predicted lifetime, and chooses the backend with the lowest cost.

The optimal placement decision for an object may change over time because of changed workload, pricing scheme, or network condition. A migration happens on a `PUT` or `GET` request when the extra cost for migration is less than the cost savings at the new storage service.

The extra cost for a `GET`-triggered migration is the total cost of an additional `PUT` request, a `DELELTE` request, and data transfer cost, while a `PUT`-triggered migration does not need the extra `PUT` request. For migration within the same Amazon cloud region, such as from S3 to EBS, data transfer is free, and `DELETE` requests are also free.

5.3 Hints

The application can give additional hints to GRANDET for better prediction. A hint is an arbitrary set of key-value pairs. For example, a photo sharing application can provide the hint {`user=alice, album=wedding`} when storing an image file. The predictor will predict the workload of this file by considering files with similar hints, such as images uploaded by the same user in the same album. Hints are optional, and we have implemented two types of default hints if the application does not provide any hint. For the file system interface (see §6), the default hint is the directory hierarchy. For example, if the photo sharing application stores a file at `alice/wedding/photo1.jpg`, the default hints would be {`hint1=alice, hint2=wedding`}. For the SDK interface (see §4.4), the default hints are the object's key split by colons. We evaluate the effect of hints in §8.4.

6 File system interface

Providing POSIX-like file system semantics is arguably the best way to support the widest range of legacy web

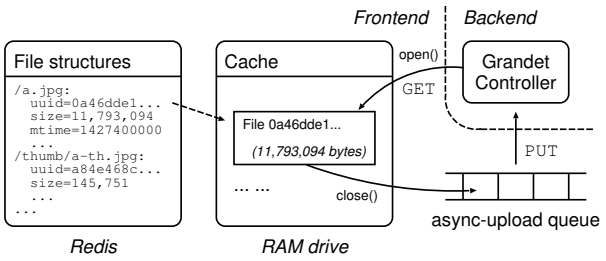


Figure 7: *Implementation of* GRANDET*'s file system interface.* Solid arrows show the data flow. Dashed arrow shows the logical relationship between the file structure and its content.

applications seamlessly, because it does not require modifications of their source code. Hence, GRANDET also implements a file system interface using FUSE. It can be directly mounted to the web application's data directory.

The design of GRANDET's file system interface follows our insight that most files are accessed sequentially and wholly by web applications, such as photo and video files. So, it is best to store each file as one object, as opposed to dividing files into blocks. Besides, web applications often need to rename files, such as moving a temporary file to its final directory. So, it is essential to support fast rename operations, although S3 does not support renaming objects other than a copy followed by a delete. Last but not least, some web applications generate many intermediate files when doing backend processing, and remove them when it finishes. So, it is desirable to skip putting these intermediate files to the backend storage.

Figure 7 shows the implementation of GRANDET's file system interface. At the backend, it stores each file as a UUID-keyed object and puts the actual file name in its metadata. At the frontend, it maintains a *cache* of file contents on a RAM drive, and keeps the *file structure* hierarchy and metadata (*e.g.*, UUID, file size) in Redis. Therefore, renaming a file only touches its metadata.

We next describe the file operations. On `creat()`, we create a file in the cache and pass the file descriptor to the application. On `open()`, we `GET` the file data from the backend storage if it does not exist in the cache, then open the cached file and return the file descriptor. For file manipulations such as read, write, and truncate, we pass them through to the corresponding file system operations of the cache. We also update our file structure for the new file size and modification time. On `close()`, if the file content has been modified, we append it into an *async-upload queue* so that the file will be `PUT` to the backend storage, and we block on `fsync()` until the `PUT` finishes.

Our implementation `PUT`s file contents to the backend storage asynchronously. It has two benefits. First, it skips short-lived intermediate files if they are deleted before the actual `PUT` happens. Second, it allows an application to specify hints as extended attributes ("xattr") efficiently after a file has been closed. This is useful when the creation of the file is beyond the application's control, such as files generated externally. For example, the CumulusClips video sharing application executes `ffmpeg` to convert a video file to another format. It can set extended attributes to the converted file thereafter.

Component	LOC	Component	LOC	Component	LOC
S3 Actor	120	EBS Actor	236	Decider	230
Predictor	356	Controller	1401	Profiler	235
C++ SDK	159	PHP SDK	168	Python SDK	69
FS interface	1979	Console	349	Misc	1191
Total : 6493					

Table 4: *Lines of code of* GRANDET*'s components.*

Since GRANDET's backend makes decision on the optimal storage location based on each file's predicted usage pattern, the replacement algorithm on the cache is not critical. A simple LRU algorithm works well in practice.

7 Implementation and system extensibility

We designed GRANDET as a extensible framework where each component, such as the storage services, the prediction algorithm, or the frontend SDK, can be easily replaced or extended. We implemented the GRANDET backend in C++14, the file system interface with FUSE [20], and key-value SDK in various languages. We modified LIBAWS [7] to communicate with Amazon Web Services. Table 4 shows the numbers of lines of GRANDET's components. Metadata such as placement decisions are stored in Redis [30]. All components can be easily extended by plugging in a new subclass, or customized by changing a configuration file. This section describes some implementation details.

7.1 Adding a storage service

GRANDET's `Actor` executes actions, such as `PUT`, `GET`, and `DELETE`, on the storage service. We implemented `Actor`s for EBS (SSD and magnetic) and S3 (standard, reduced redundancy, and infrequent access). Supporting a new storage service just requires adding a new subclass of `Actor` and implementing its interface methods.

Figure 8 shows the interface of the `Actor` class. The `put()`, `get()`, and `del()` are cloud storage operations. The `profile()` method, when called by GRANDET's `Profiler`, updates the cloud service's `Profile`, which includes pricing model and service conditions such as latency, availability, durability, and consistency.

The `Profiler` is a `cron` job that runs periodically. When triggered, it calls every `Actor`'s `profile()` method to update its profile. We implemented crawlers in our EBS and S3 `Actor`s to fetch and parse the pricing information from the Amazon Web Services website. `Profile`s are stored as JSON files so that users can also manually configure the pricing model or service levels.

```
class Actor {
 public:
  virtual void ~Actor()=default;

  // cloud storage operations
  virtual void put(const string& key, shared_ptr<Value> val)=0;
  virtual shared_ptr<Value> get(const string& key)=0;
  virtual void del(const string& key)=0;

  // updates pricing model, latency, availability, durability, etc.
  virtual void profile(shared_ptr<Profile> profile)=0;
};
```

Figure 8: GRANDET's *Actor class.*

7.2 Adding a prediction algorithm

We implemented the prediction algorithm as described in §5, and we believe that recent advancement of machine learning techniques may empower even better algorithms. Plugging a new prediction algorithm into the GRANDET framework is also as simple as subclassing the `Predictor` class with the following functions.

The `Predictor` has three listener functions, namely `notify_put()`, `notify_get()`, and `notify_del()`, which are called whenever there is a `PUT`, `GET` or `DELETE` request. The `Predictor` thus keeps track of the current workload. When making a decision, the decider calls the `Predictor`'s `predict_put()`, `predict_get()`, and `predict_lifetime()` functions to get the predicted future request frequency and expected lifetime.

7.3 Protocol and SDK

GRANDET's frontend and backend communicate through Unix domain socket IPC and all messages are serialized in Protocol Buffers [28]. GRANDET defines two types of protocol messages: `Request` and `Response`. A `Request` message is one of three types: `PUT`, `GET`, and `DELETE`. It also includes the key and value of the data object, the application's requirements such as minimum durability and maximum latency, and optionally hints for workload prediction and other metadata. The `Response` message contains a status code, and optionally the data object's value if it is response for a `GET` request.

Therefore, the SDK for a programming language is simply a wrapper over Protocol Buffer and socket programming. We have implemented the SDK for C++, PHP, and Python, with 70–170 lines of code each. We believe that supporting a new language would similarly require little programming effort.

7.4 Optimization

To further improve performance, we also implemented two optimizations to GRANDET's basic design.

Shortcut for file access. When `PUT`ting a file object that is already on disk, the request payload only includes the file name instead of the file content, and the GRANDET backend reads the file directly from disk. Therefore, it avoids sending the entire file from frontend to backend.

S3 authenticated URL. An application often `GET`s a data object from GRANDET only to send it verbatim to the user without any processing. For example, when a user clicks "download original image" on the Piwigo photo sharing application, Piwigo simply retrieves the data object for that original image and send it back to the user. Thus, if the data object is stored on S3, GRANDET incurs unnecessary overhead by acting as a proxy for the data transfer. To optimize for this scenario, the application can specify a special requirement in its `GET` request in the form of {`url=true, expire=600s`}; so the GRANDET backend sends the application a pre-authenticated URL for the S3 object with the specified expiration time (600s here). The application can thus redirect the user to download the image from the authenticated URL directly.

8 Evaluation

We evaluated GRANDET on four popular open source web applications: CumulusClips (video sharing) [14], Piwigo (photo sharing) [27], Elgg (social network) [17], and FileSender (file sharing) [19]. We modeled the usage data for each application according to the most popular website of its type, namely YouTube, Flickr, Facebook, and Dropbox. To make cost evaluations manageable, we scaled down the usage to 100 users in one month, while preserving real-world workload characteristics. Appendix A details how we modeled usage data. We ran all experiments on EC2 `m3.large` instances with EBS and S3 in the US east region, using Ubuntu 14.04.

Our experiments aim to answer four questions:
§8.1 Does GRANDET reduce cost?
§8.2 Is GRANDET fast?
§8.3 Is GRANDET scalable?
§8.4 Is GRANDET easy to use?

8.1 Cost savings

8.1.1 End-to-end cost savings

The overarching goal of GRANDET is to reduce cost used by web applications. Figure 9 shows a comparison of total storage cost of evaluated web applications with different storage backends.[2] For each application, the first five bars are the cost of placing all objects into a single storage service. The last bar is the result of GRANDET's dynamic placement. All numbers are normalized by the theoretical optimal placements, meaning that each object is placed at the best storage if the entire workload was known beforehand (*i.e.*, perfect prediction).

The results show that GRANDET always costs less than any single-storage option. It saves a geometric mean of

[2]Storage costs in this paper were reported by GRANDET based on the precise storage space used and number of requests recorded. We did not use Amazon's billing statement because it was too coarse-grained.

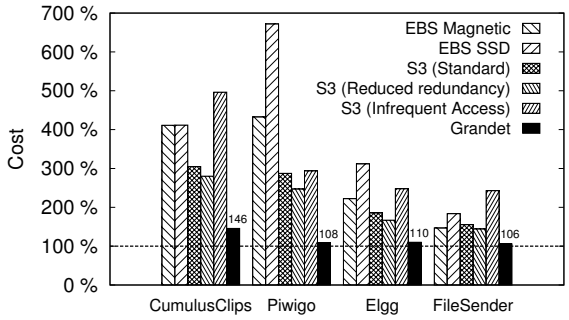


Figure 9: *Comparing the cost of different backends and* GRANDET. All costs are normalized to the optimal cost.

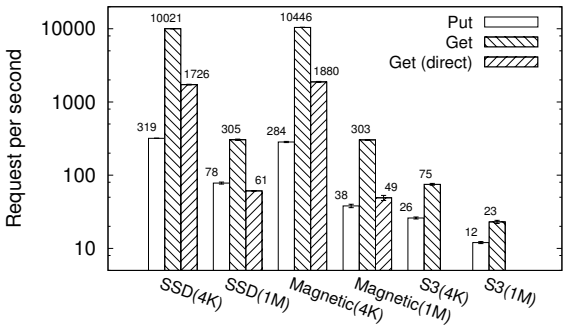


Figure 10: *The performance of* GRANDET *backend.* Each storage is evaluated with 4KB and 1MB requests. The error bar shows the standard deviation.

42.4% over the best single-storage setting. For example, GRANDET reduces Piwigo's cost by 56.2%. The reason is that Piwigo converts images into several resolutions, and images from different users and albums have distinct access patterns that are hard to be programmed statically but easy to be predicted dynamically by GRANDET.

Furthermore, for all but one applications, GRANDET's cost is within 10% of the optimal cost. For CumulusClips, although it costs 45.7% more than optimal, it is still 48.0% better than using any single storage backend.

It is worth noting that the cost saving ratio is independent of the number of users, because the cost is proportional to the workload, which in turn is proportional to the number of users. Therefore, GRANDET is effective in reducing cost for a broad spectrum of web applications.

8.1.2 Operational cost

To evaluate the operational cost that the GRANDET service itself incurs, we monitored its memory and CPU usage while running the Piwigo application. GRANDET only uses little memory; so we focus on CPU usage.

Assume that someone sets up a Piwigo instance to serve 100K users. Per our usage model (Appendix A), users would upload 120K photos and view 2.4M photos in one month. Meanwhile, 120K thumbnails would be generated and they would be viewed 64.8M times. Thus, there would be a total of 2.52M large requests (95% read) and 64.92M small requests (99% read) per month, or 0.972 large requests and 25.1 small requests per second.

We evaluated GRANDET to see how many requests per second (RPS) it can handle per percent of CPU usage. In the worst case, GRANDET can handle 1.54 RPS per percent of CPU usage with large requests (1MB, 95% read), and 29.5 RPS per percent of CPU usage with small requests (4KB, 99% read). Plugging it into the aforementioned scenario, GRANDET would consume 1.48% CPU to serve all requests. Since an EC2 `m3.large` instance costs $38 per month and it has two cores, GRANDET only costs $0.28 per month to serve 100K users in this case, negligible versus the storage cost it saves.

8.2 Performance

8.2.1 Microbenchmark

To evaluate the performance of basic operations of GRANDET, we evaluated each storage separately with two sizes of requests. Figure 10 shows the number of requests GRANDET can handle per second. We used one client in this experiment. Because the performance of `GET` requests are affected by the file system cache, we also measured the performance in the direct mode by specifying `O_DIRECT` in file system operations.

The performance of the EBS backends without cache matches the results of FIO [21], which measures the performance of the file system itself. Hence, GRANDET's performance is limited by the hardware and underlying OS, and GRANDET itself incurs little overhead.

One interesting property of EBS disks is that they have different burst and sustained performance. For example, EBS SSD disks can reach burst throughput of 150MB/s, close to Amazon's specification [4]. But after a few seconds, the throughput drops to ≈60MB/s and keeps stable.

The cached `GET` requests of EBS backends are obviously served from the cache. The major limiting factor here is the CPU speed. We can see that GRANDET can work with hardware that is much better than the ones available in the cloud currently.

The results are low for S3, because S3 has a high latency for any request. Our profiler usually records the latency to be 20–30ms, and this latency limits the number of requests S3 can handle per second. Because requests sent to S3 are much more expensive than other storages, S3 should not handle many requests, and it is also not designed to handle frequent requests.

8.2.2 End-to-end performance

We evaluated GRANDET's end-to-end performance on the same four web applications. Because we use FUSE to implement the file system interface, we also evaluated the overhead incurred by FUSE itself. For compari-

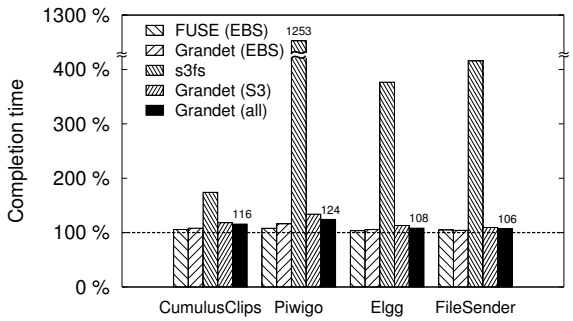


Figure 11: GRANDET*'s end-to-end performance.* Completion time is normalized to the baseline which uses EBS directly.

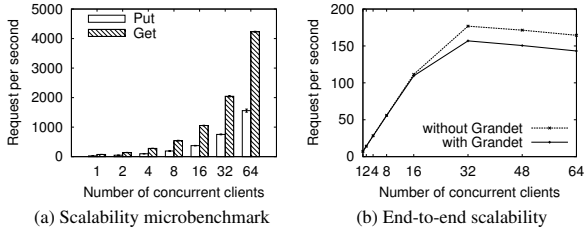
son, we also ran the evaluation on the state-of-the-art S3-based file system `s3fs` [31]. Figure 11 shows the time used to complete the workload of each application and storage setting. We normalized all results to the baseline where all files were stored directly on EBS SSD. For the first bar in each cluster, a folder on the EBS SSD volume was mounted with the loopback FUSE file system to the application's data folder. The second bar used GRANDET with only the EBS backend, so as to show GRANDET's overhead atop FUSE. The third bar used `s3fs`. To compare with it, the fourth bar used GRANDET with only the S3 backend. Finally, the last bar used GRANDET in the default configuration with all backends.

GRANDET's overhead comes from several parts. The first part is incurred by FUSE, which averages to 5.5% (the 1st bar). The second part is incurred by GRANDET itself. Because using GRANDET with only the EBS backend has an average overhead of 8.5% (the 2nd bar), the overhead incurred by GRANDET itself is less than 3%. The third part is incurred by the S3 backend, due to its higher latency than EBS. Mounting S3 as a file system with `s3fs` shows a prohibitive average overhead of 330% (the 3rd bar), whereas GRANDET's average overhead using only the S3 backend is 18.3% (the 4th bar). Overall, GRANDET incurs a geometric mean of 13.5% overhead (the last bar), which can be offset by the cost it saves.

8.3 Scalability

8.3.1 Microbenchmark

To check that whether GRANDET can scale up, we evaluated GRANDET with variable number of concurrent threads and variable request sizes on variable storages. The results are similar, and for brevity we show a typical one: S3 with request size of 4KB. Figure 12(a) shows the performance of the server when the number of concurrent clients increases. The number of requests the server can handle per second increases almost linearly. This implies that the number of requests one client can achieve is limited by the latency of the S3 service, and the server scales well with the number of clients.



(a) Scalability microbenchmark (b) End-to-end scalability

Figure 12: (a) *Scalability of* GRANDET *when using single S3 storages.* Evaluated with requests of 4KB in size. The error bar shows the standard deviation. (b) *End-to-end scalability.* Evaluated on the FileSender application with real workload.

8.3.2 End-to-end scalability

We evaluated the end-to-end scalability of GRANDET by measuring the number of end-to-end requests the system can handle when the number of clients increase. The requests go all the way through Nginx, PHP, FUSE and the GRANDET backend. We chose the most scalable application—FileSender—among all the applications we studied, so that if there were any scalability issues with our system, it would be revealed by the experiment. The FileSender application is the most scalable application because of its simplicity: it does not perform any operations on the files, but just lets other users download them.

Figure 12(b) shows the requests per second with variable number of concurrent clients. The results show that GRANDET scales as well as FileSender. Regardless of whether using GRANDET, FileSender does not scale past 32 concurrent clients. This is due to limited resource in the EC2 `m3.large` instance, not GRANDET's limitation.

8.4 Usability

GRANDET can run a web application unmodified and automatically save cost. We have also tested and confirmed that three of today's most popular web applications—MediaWiki, Wordpress, and Joomla—work seamlessly with GRANDET without any source code modification.

To further reduce cost, application developers can add hints to data objects. In all our evaluations, we did not add hints to CumulusClips and Elgg, added three hints to Piwigo, and added one hint to FileSender. We found that compared with using the default predictor, hints helped reduce cost by 9.3% for Piwigo and 9.4% for FileSender.

9 Discussion

We now discuss some design implications of GRANDET.

Persistence over server crash. If the GRANDET server crashes, all data objects that have been `PUT` onto EBS or S3 will persist. Metadata (*e.g.*, placement decisions) rely on the persistence of Redis, which can be configured as snapshot-based (RDB) or journal-based (AOF). In the worst case, they can be rebuilt by scanning all storage. For the file system interface, the local cache may not per-

sist, which would affect data objects in the async-upload queue that have not yet been PUT to the backend storage. GRANDET provides the same semantics as a file system by blocking on `fsync()` until the PUT is complete.

S3 consistency. S3 provides read-after-write consistency for new objects and eventual consistency for overwrites. There are two ways to work around it. First, the application can specify in each object's requirement to avoid S3. Second, GRANDET can use versioning in S3 placement decisions so that each PUT operates on a new object.

Data replication across cloud regions. Because EBS volumes can only be accessed within a cloud region, GRANDET's server must reside in the same cloud region as all EBS volumes. However, since GRANDET exposes a general key-value object store interface, it can be easily extended to multiple cloud regions by overlaying existing geo-replication solutions atop GRANDET.

Migration granularity. GRANDET migrates data lazily on a per-object basis; so data objects that are not accessed would not be migrated, even if better storage choices were available. One way to solve it is to have a thread periodically scan through all objects to find migration possibilities. In practice, the changes of workload on data objects are gradual, so that a cold object would already be migrated before its access drops to absolute zero.

EBS elasticity. Adjusting EBS volume size takes minutes to finish. GRANDET can leverage existing orthogonal strategies (*e.g.*, [29]), or rely on application developers for allocating EBS volumes. Amazon's recently-announced elastic file system (EFS) is fully elastic and does not have this issue. Once it is released, GRANDET can support it by simply adding an `Actor` class for it.

10 Related work

S3 lifecycle. Amazon has rudimentary support for moving S3 objects to the infrequent-access option or Glacier. However, such transitions are one-way and limited to S3, and developers must set rules manually. GRANDET supports automatic transitions across all storage options.

Cloud economics. Some recent work studies the economics of cloud computing. Much of the work is focused on reducing the cost of computing, not storage. For example, Tak *et al.* [35] discusses the cost factors for several cloud-based application deployment options, and Conductor [42] optimizes cloud service choices for MapReduces computations. Other work touches upon storage. CloudCmp [25] provides a microbenchmark suite for measuring the cost and performance of different cloud service providers. Developers can then inspect the benchmark results and pick a provider for their application. GRANDET may leverage this microbenchmark suite in its profiler implementation.

Cloud-backed file systems. Several systems provide a file system interface over a blob storage such as S3. Open source projects, such as `s3fs` [31] and `s3ql` [32], can mount an Amazon S3 bucket as a local file system. The BlueSky network file system [40] employs a log-structured design on the cloud storage. SCFS [11] enables sharing for cloud-backed file systems. These systems assume general file system workloads, and the main challenges they tackle are performance issues, such as how to implement random writes atop a blob storage that does not support partial updates. Unlike GRANDET, none of these systems exploits the characteristics of files used by web applications or reduces monetary cost.

Cloud-of-clouds. Several pieces of work propose the idea of storing data across multiple clouds. Some do so to replicate the same data multiple times for fault tolerance. For example, RACS [1] applies the RAID technology to cloud systems. DepSky [10] uses multiple services for dependability and security. MetaStorage [9] uses multiple services to manage consistency-latency trade-offs. NCCloud [23] applies network coding to cloud storages for fault tolerance. These systems aim to increase durability and availability, not to decrease monetary cost. In fact, by storing more copies of data, they increase monetary cost, which GRANDET can help reduce.

Other pieces of work, including FCFS [29], iCostale [2], Scalia [26], and SPANStore [44], store data across clouds for reducing cost, a goal similar to GRANDET's. FCFS only has simulations showing potential savings of storing objects across different cloud services, which serve as an excellent motivation for GRANDET. ICostale and Scalia also do simulations only, and they consider only blob storages which cannot support many popular web applications. To the best of our knowledge, none of FCFS, iCostale, or Scalia provide a system that developers can use. SPANStore also considers only blob storages; so it also requires modifications to many web applications. In addition, its coarse-grained placement decisions only consider geographical locations. In contrast to these systems, GRANDET makes fine-grained predictions and decisions based on each data object's own characteristics and access pattern, and it works seamlessly with today's web applications without modifications.

11 Conclusion

We presented GRANDET, a storage system that greatly reduces storage cost for web applications deployed in the cloud. It unifies multiple heterogeneous stores by placing each data object at the most economical store, and provides both a file system interface and a key-value SDK. Evaluation on a diverse set of four popular open-source web applications shows that it reduces cost by an average of 42.4%, and it is fast, scalable, and easy to use. Its source code is at http://columbia.github.io/grandet.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240. ACM, 2010.

[2] S. Agarwala, D. Jadav, and L. A. Bathen. iCostale: Adaptive cost optimization for storage clouds. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, 2011.

[3] Amazon EBS Product Details. http://aws.amazon.com/ebs/details/.

[4] Amazon EBS Volume Types. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html.

[5] Amazon speeds up its cloud with SSD block storage. http://www.networkworld.com/article/2364506/cloud-storage/amazon-speeds-up-its-cloud-with-ssd-block-storage.html.

[6] Amazon Web Services leads war on cloud price reductions. http://www.techrepublic.com/article/amazon-web-services-lead-the-war-on-cloud-price-reductions/.

[7] An Amazon Web Services C++ Library. http://libaws.sourceforge.net/.

[8] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[9] D. Bermbach, M. Klems, S. Tai, and M. Menzel. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 452–459. IEEE, 2011.

[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage Systems*, 9(4):12, 2013.

[11] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 169–180, 2014. ISBN 978-1-931971-10-2.

[12] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 229–238, June 2008.

[13] comScore Releases January 2014 U.S. Online Video Rankings. http://www.comscore.com/Insights/Press-Releases/2014/2/comScore-Releases-January-2014-US-Online-Video-Rankings.

[14] CumulusClips. http://cumulusclips.org/.

[15] Dropbox. https://www.dropbox.com.

[16] Dropbox Fact Sheet. https://www.dropbox.com/static/docs/DropboxFactSheet.pdf.

[17] Elgg. http://www.elgg.org/.

[18] Facebook. http://www.facebook.com.

[19] FileSender. http://www.filesender.org/.

[20] Filesystem in Userspace. http://fuse.sourceforge.net/.

[21] Flexible I/O Tester. https://github.com/axboe/fio.

[22] Flickr. http://www.flickr.com.

[23] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, page 21, 2012.

[24] K. Lerman and L. A. Jones. Social browsing on flickr. In *International Conference on Weblogs and Social Media*, 2007.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pages 1–14, 2010.

[26] T. G. Papaioannou, N. Bonvin, and K. Aberer. Scalia: An adaptive scheme for efficient multi-cloud storage. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 20:1–20:10, 2012. ISBN 978-1-4673-0804-5.

[27] Piwigo. http://piwigo.org/.

[28] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[29] K. P. Puttaswamy, T. Nandagopal, and M. Kodialam. Frugal storage for cloud file systems. In *Proceedings of the 2012 ACM European Conference on Computer Systems (EUROSYS '12)*, pages 71–84. ACM, 2012.

[30] Redis. http://redis.io/.

[31] s3fs. https://github.com/s3fs-fuse/s3fs-fuse.

[32] s3ql. https://bitbucket.org/nikratio/s3ql.

[33] Startups and Amazon Web Services. http://aws.amazon.com/start-ups/.

[34] Statistics - Youtube. https://www.youtube.com/yt/press/statistics.html.

[35] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam. To move or not to move: The economics of cloud computing. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, 2011.

[36] Y. Tang, G. Hu, X. Yuan, L. Weng, and J. Yang. Grandet: A unified, economical object store for web applications. Technical report, Columbia University, 2016.

[37] The Best Video Length for Different Videos on YouTube. http://www.minimatters.com/blog/youtube-best-video-length/.

[38] The man behind Flickr on making the service 'awesome again'. http://www.theverge.com/2013/3/20/4121574/flickr-chief-markus-spiering-talks-photos-and-marissa-mayer.

[39] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *Trans. Storage*, 5 (4):14:1–14:28, Dec. 2009.

[40] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[41] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, 2006.

[42] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[43] Wikimedia. Media storage. https://wikitech.wikimedia.org/wiki/Media_storage.

[44] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 292–308, 2013. ISBN 978-1-4503-2388-8.

[45] YouTube. http://www.youtube.com.

A Workload modeling

A.1 CumulusClips

We modeled the usage data of CumulusClips according to the popular video sharing website, YouTube [45].

YouTube has one billion users, and 300 hours of videos are uploaded per minute [34]. The average video length is four minutes [13, 37]. So on average each YouTube user uploads 0.19 videos per month, and hence 100 users would upload 19 videos. Each user views 76 videos per month on average [13], which translates to 7600 views for 100 users. Also, [12] mentions that the average video size of YouTube video is 8MB; so we use it as our average video size in our evaluation. Most thumbnails on the YouTube website have around 400×300 pixels; so we also use it as our thumbnail size. On the YouTube website, there are 20 recommended videos on the right side of each video; so we consider that for each video viewed, 20 thumbnails are also viewed.

A.2 Piwigo

We modeled the usage data of Piwigo according to the popular photo sharing website, Flickr [22].

Flickr has 87 million users and they upload 3.5 million new images per day [38]. So 100 users would upload 120 images per month. Lerman *et al.* [24] mentioned that the average view per photo on Flickr is 20 times. Each album on the Flickr website shows 27 thumbnails above it, meaning that when a user views one photo, she also downloads 27 thumbnails of other photos. On the Flickr website, large thumbnails have around 1600×1000 pixels and small thumbnails have around 640×400 pixels. A typical photo taken by a modern digital camera has 5120×3840 pixels. We use these parameters in our workload.

A.3 Elgg

We modeled the usage data of Elgg according to the popular social network website, Facebook [18].

For photos stored on Facebook, we observe that when a user clicks a photo, the dimension of the photo shown is 960×960 pixels. We use it as the photo size in our workload. This observation also matches [8], which mentioned that the average photo size on Facebook is 60KB. Facebook had 500 million users when [8] was published, and [8] mentioned that on average 120 million photos are uploaded every day. These numbers translate to 7.2 photo uploads per user per month; so 100 users would

upload 720 photos per month. We observe that photos in a typical Facebook user's timeline are thumbnails of 300×300 pixels; so we also use this in our workload. Each day, 10 billion photos are viewed on Facebook, including both thumbnails and original photos [8]. So each user views 20 photos per day on average. For 100 users, they view almost 60,000 photos per month. [8] also mentioned the ratio of views between thumbnails and original photos is 95% to 5%. We consider the nature of social networks in generating workloads: each user has a certain number of friends, and when friends post photos, she may see the photo. We already know that each user views a certain number of photos per month; so we distribute these views on her friends' photos.

A.4 FileSender

We modeled the usage data of FileSender according to the popular file sharing website, Dropbox [15].

Dropbox has 50 million users and they upload 500 million files each day [16]. So 100 users would upload 30,000 files per month. We use the average size of files from file sharing servers, 153KB [39]. The popularity of the shared files follows a Zipf distribution with $\alpha = 0.4$ [41].