Pixelcior: A Global Service for Personal Photo Management
Kyungho Jeon, Sharath Chandrashekhara, Karthik Dantu, Steven Y. Ko
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
{kyungho, sc296, kdantu, stevko}@buffalo.edu

ABSTRACT
Photos form a large fraction of the data that gets generated on mobile devices such as smart phones, wearables, action cameras, and other devices. To match this trend, a number of popular apps are either dedicated photo managers or support some form of photo management. Such apps support a variety of functions ranging from sharing and syncing, image manipulation, photo organization with advance image processing techniques among others. As these photos are made available on a wide range of platforms, the applications adapt the content for different platforms. As an example, the resolution of the image shared can depend on the screen size or the storage space available on that device. These methods of capture, storage, retrieval, and manipulation of images are built into every app that performs photo management resulting in duplicated images, repeated computation, and custom maintenance of metadata. This motivates the need of a dedicated service for this purpose which can not only manage the photos locally, but also can transparently manage content adaptation, image manipulation and propagation of the changes to all the devices in the system and the cloud. Pixelcior presents our study of the requirements of such a system as well as a preliminary design motivated by application requirements.

CCS Concepts
•Information systems → Mobile information processing systems;

1. INTRODUCTION
Spurred by the widespread use of camera-equipped devices, the number of photos taken everyday has been rapidly increasing over the last decade. While it’s estimated that about 80 billion photos were taken 2000, in 2014, 550 billion photos were shared on Facebook, WhatsApp and Snapchat alone [5]. With the advent of newer wearables such as Go-Pro cameras, smart glasses, and cameras embedded in household appliances, we believe that this trend is on the rise. As a result, photo apps have become one of the most sought-after app categories for regular users. For example, 5 of the top 10 installed free apps in Google Play are related to photos [7].

In general, photo management requires capabilities beyond simple photo capture and storage. A brief look at the functions provided by existing photo apps reveals how complicated photo management has become. Some of these functions include sharing through social media platforms and messaging, synchronization across multiple devices, and editing that transforms photos with a variety of filters. They also enable photo classification based on location, time, and other parameters, and more involved functions such as face and object recognition. In doing so, they also implement content adaptation; sharing a photo is typically done with a lower resolution copy to save on data transfer and storage; backing up a photo to a cloud storage service often deletes the original copy and only keeps a lower resolution copy locally, again to save on storage [3].

Given this state-of-the-art, we posit that many of these functionalities should be provided as a separate global service. Such a service should be able to manage all photo-related activities across apps and devices. Similar to a DBMS that frees application developers from the challenges of structured data management, a well-designed photo management service can potentially free photo app developers from worrying about the management of their photos. This will allow photo app developers to focus instead on innovative features that differentiate their apps from other apps. Given the increasing modalities through which photos can be consumed (phones, tablets, laptops, televisions, digital photo frames and others), such a service could also adapt photos to suit the device by adjusting the resolution and other image properties. By separating this as a service, such functionality can be availed uniformly by all applications instead of being implemented separately inside each one of them.

In this paper, we envision such a global photo management service called Pixelcior. Our goal is to identify the design requirements for Pixelcior and discuss potential design choices. Although our discussion is mostly centred on supporting apps for camera-equipped mobile devices and IoT (Internet of Things) devices, our general design also considers more powerful photo editing apps that run on bigger devices such as laptops and desktops, as well as Web photo services that run inside a browser. As described in Section 2, we envision Pixelcior providing the following functionalities:

• Photo data management: Another basic functionality related to photos is photo-storing; however, this
means not only storing raw photos but also supporting synchronization across multiple devices, sharing, search, and organizing photos.

- **Photo content adaptation**: Many photo apps adapt photos based on the devices they run on. For example, a photo app on a smartphone might use lower resolution photo stored locally (e.g., a thumbnail) and keep a full resolution copy in its backend server. A photo management service should support such content adaptation efficiently across devices and apps.

- **Photo manipulation and propagation**: One of the frequent photo-related activities is editing, ranging from simple operations such as applying a filter to more complicated ones such as background subtraction or high-dynamic-range imaging (HDR). When an operation is applied to a photo, the update should be propagated and visible on other devices as well.

- **App flexibility**: Many photo apps innovate with new features and a photo management service should not hinder such innovation. However, such a service should be cognizant of a set of features that can be applied on photos independent of the app.

- **App independence**: Since there are many photo apps developed and used, a photo management service should provide proper isolation between the apps that share the service. This includes not only access control but also backend support; for example, if an app wants to use a particular cloud service, the photo management service should allow the app to do so.

In the rest of the paper, we discuss these design requirements challenges in more detail by first surveying the application features (Section 2). We then briefly describe our design (Section 3). Finally, we discuss the related work (Section 4) and conclude the paper (Section 5).

## 2. THE PROBLEM OF PHOTO MANAGEMENT

To motivate a global service for photo management, we survey existing applications that are used for photo capture and management such as photo organizers (e.g., Google Photos and Apple iCloud Photo Library), social networking applications (e.g., Facebook and Instagram), messaging (e.g., WhatsApp). We describe our observations and identify the common requirements that drive these applications. We then discuss the challenges of implementing these features as a photo management service. In our discussion, we do not specifically discuss a particular application, since our focus is on categorizing the features and derive application requirements; these applications are popular and their features are well-understood in general.

### 2.1 Application Features

Popular photo applications provide features in three categories: capture and store, retrieve, and transform. Each category can involve many extra operations, and we discuss them below.

**Capture and store**: The most basic feature of photo applications is capturing of a photo, which includes capturing of not only a raw image but also relevant meta data such as location and date. This meta data can be used later for photo applications to organize photos in various ways.

Once a photo is captured, it is stored, and current photo applications implement many additional features while storing a photo. For example, an application might store photos locally and sync with a cloud service; it might generate and store a thumbnail along with the original photo; moreover, it might store additional meta data such as detected faces and objects.

**Retrieve**: Another basic feature of photo applications is retrieval. Along with storage, this is also done in a variety of ways in current photo applications. Often times, retrieval involves advanced operations such as search and organize; for example, a social media sharing application might retrieve photos by most recently captured first with the expectation that are likely to share the most recently captured photos. A photo management application might index photos by date, location, or people featured in the photos for easy access to its user.

**Transform**: Beyond storage and retrieval, many photo applications allow users to transform stored photos. The transform operations range in computing complexity from applying simple filters such as smoothing, sharpening, color adjustment etc. to more sophisticated operations such as HDR. It is important to note that a user can transform a photo on different devices; for example, a Google Photos user can use a smartphone frontend app or a Web frontend app to apply a filter.

### 2.2 Requirements and Challenges

As we mention in Section 1, we posit that these photo management features should be provided as a separate service. Once again, this is mainly because photos are one of the most popular data types nowadays and photo-related applications need a similar set of features as described above.

In the rest of the section, we discuss the challenges we need to overcome in order to design such a separate photo management service.

**Flexible interface**: As discussed above, applications have a variety of needs in managing photos. Thus, a photo management service should provide a flexible yet rich set of APIs so that applications can express their needs with ease. This includes all basic operations such as storage and retrieval, as well as more advanced operations such as search and organize. Not only that, transformation functionality is common for many applications; a photo management service should provide the APIs for photo transformation as well. Moreover, an application might need to be integrated with a specific cloud storage provider. Therefore, the service should give an option for an applications to choose. For this, a photo management service should provide APIs to plug-in various cloud service providers.

**Content adaptation**: Photo applications in general manage various resolutions of photos. Most mobile apps use thumbnails of photos for display in a collection view. Also, often times it is unnecessary to store a full-resolution photo on a mobile device, since its display does not support the resolution. For example, iPhone 6s can take $4000 \times 3000$ (12 megapixels), but its screen resolution is only $1334 \times 750$ [2]. Thus, to save device’s storage usage, Apple iCloud allows users to keep smaller resolution photos locally and to back up full sized photos in the cloud. However, other content adaptation policies are possible depending on what criteria an application is more concerned about. For example, Google Photos encourages users to backup limited sized
photos on the cloud and keeps full sized photos on local storage, to save on the cloud-side of the storage and data transfer cost. In extreme cases, the image might be stored only on the cloud, and retrieved every time it needs to be accessed. This presents a trade-off between storage, computing (to generate different versions of the image), and data transfer. Some applications implement optimization techniques such as pre-fetching of either lower-resolution copies or full-resolution copies based on the expected use for them. Thus, a photo management service should support content adaptation; in doing so, it should take into consideration that multiple devices access photos with different resolutions, and there is often times no need to retrieve full-resolution photos.

**Transformation and propagation:** As mentioned earlier, photos are not only taken, but also transformed by applying various photo transformation operations. These operations can be done by a user on different devices, e.g., a mobile device, a laptop, etc. This photo transformation gets more complicated when considering the fact that photos are synchronized and content-adapted across devices. Essentially, in order to enable cross-device access, a photo management service should handle the propagation of transformed photos.

An interesting aspect about this propagation of a transformed photo is that one can choose to propagate the photo itself or the transformation operation. For example, suppose that a user has a smartphone and a tablet, and installs a photo application on both devices. Also suppose that this photo application implements content adaptation, and the tablet has a full resolution photo while the smartphone only has a scaled-down version of the photo. If a user transformed a full resolution photo on the tablet, the propagation of this photo can be done either by sending the transformed photo itself, or by sending the transformation operation and applying it locally to the scaled-down version of the photo.

It is reasonable to assume that sending the transformed photo (a full resolution) can be expensive in terms of data transfer cost. However, sending a transformation operation and applying it locally might not produce the same quality photo. This is demonstrated in Figure 1. Each photo is transformed in different ways from the original photo. The resolution of the original photo is 512×512, and Figure 1a shows a lower resolution version with 256×256. From the above scenario, suppose that this lower resolution version is stored in the smartphone, and a blur operation is applied at the full resolution version in the tablet. Propagating this blurred photo to the smartphone has two options. The first option is sending the operation to the smartphone and applying it locally (i.e., applying the resize operation then the blur operation as shown in Figure 1b). The second option is applying the blur operation then the resize operation on the tablet, and send the photo itself to the smartphone (shown in Figure 1c). As we can see, they produce two different qualities of photos. Thus, a photo management service should be aware of the fact that different transformation options will not only have different costs but also produce different qualities of photos. We discuss in Section 3.3 on how this opens a number of opportunities for the internal design of a photo management service.

### 3. DESIGN SPACE

This section overviews the design space for our envisioned photo management service. We separate our discussion into three parts—an application’s point of view, internal photo representation, and implementation considerations. The discussion from the viewpoint of an application explains that our service is a single logical entity for an application. The discussion of our internal photo representation answers the question of what information needs to be managed in our service. The discussion of our implementation considerations examines the question of how the information can be managed. Our purpose is to discuss the factors we need to consider in designing a photo management service; it is not necessarily to present the final design. We leave the final design as our future work.

#### 3.1 An Application’s Point of View

From an application’s point of view, Pixelcior is a single logical entity that manages photos on behalf of the application. We use the term application in a collective sense; one application can consist of multiple front-end clients, e.g., a smartphone photo app, a Web app, etc., as well as back-end storage servers. In this sense, Google Photos and Instagram, for example, are all applications. Figure 2 illustrates this point; Pixelcior interfaces with front-end clients as well as back-end storage as a single logical service.

To simplify our discussion, we assume single user scenarios in this section, e.g., a single user who uses the Google Photo
app on a smartphone and the Google Photo Web app with a browser. However, extending our discussion to multiple user scenarios should be straightforward.

In its most basic form, Pixelcior is similar to a DBMS; it deals with not only storing of raw data, but also structuring and transforming the data. However, unlike DMBS, the focus of Pixelcior is on photos and it specializes all of its operations accordingly. These specialized operations include content adaptation operations and photo transformation operations as well as the management of those operations.

Table 1 shows the basic APIs we envision for both front-end clients and back-end storage. For front-end clients, our APIs support the basic get and put along with the search and organize features with tags. A tag is an application-defined string, and can be used to group multiple photos into a collection; some example tags include locations, (detected) faces, dates, etc. An application can use multiple tags for one photo.

In addition, our APIs support both manual and automatic content adaptation; we allow an application to store and retrieve a photo by either specifying a resolution (manual adaptation) or asking Pixelcior to pick a resolution (automatic adaptation). Automatic adaptation can be done based on various conditions, e.g., display resolution and size.

It is important to note that in Pixelcior, a photo is a logical object, consisting one or more physical photo files/objects of various resolutions along with its metadata. This means that (i) an application uses a single name for a photo even if there can be multiple resolutions of the same photo, (ii) an application retrieves a photo with its name and a resolution, and (iii) an application does not explicitly manage multiple files/objects for different resolutions.

For transformation operations, our APIs support apply that takes a photo and applies a built-in transformation operation. Table 2 shows an example of potential transformation operations Pixelcior can support. We will offer basic operations, so the applications can use the operations as basic building blocks. Whenever a transformation operation is applied to a photo, a new photo is created and stored. Moreover, our APIs include provenance APIs that allows an application to trace the transformation history of a photo back and forth. These APIs enable a large set of operations such as undo/redo of photo edits, restoration of a deleted photo, quick retrieval of all transformed photos from an original photo, etc.

### 3.2 Internal Photo Representation

As a single logical entity, Pixelcior internally uses a photo representation model to manage photos. This includes typical information for a photo, i.e., the name, the raw data, and the meta data (e.g., the file format). In addition, Pixelcior associates each photo to all additional meta data mentioned earlier in Section 3.1, such as tags and transformation provenance information.

In order to capture these different types of information, Pixelcior represents a photo as a node in a tree as shown in Figure 3. Each photo object can have multiple versions with different resolutions. If a photo is derived from another photo as the result of applying a transformation operation, they form a parent-child relationship. A new photo just taken creates a new tree, while a transformed photo from another one extends a tree.

The actual management of these trees, as well as the generation of transformed photos, should be done efficiently by the underlying implementation. In doing so, it should take multiple frontends into consideration (such as phones, tablets, and Web apps). For example, a phone frontend for a user might never retrieve a photo of the original resolution due to its small screen size or resolution. If this is the case, it makes sense for the phone frontend to only store scaled-down photos. However, if on another device the user edits a photo, this update has to be propagated to the phone frontend. Now this update propagation can be done in many
<table>
<thead>
<tr>
<th>Type</th>
<th>Interface</th>
<th>In</th>
<th>Out</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>get</td>
<td>key, QoS</td>
<td>image</td>
<td>Returns an image at the requested QoS. QoS can refer to a specific resolution of the image or a more generic request like the “best quality”, “fastest fetch”, etc.</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>key, photo</td>
<td>none</td>
<td>Creates a new image in the system.</td>
</tr>
<tr>
<td>Search</td>
<td>setTag</td>
<td>key, list:tags</td>
<td>none</td>
<td>Adds the list of tag to the photos, any of which can be used to retrieve the photo.</td>
</tr>
<tr>
<td></td>
<td>getTags</td>
<td>key</td>
<td>list:tags</td>
<td>Returns a list of tags associated with the photo.</td>
</tr>
<tr>
<td></td>
<td>getByTag</td>
<td>tag</td>
<td>list:photos</td>
<td>Returns a list of photos associated with the tag passed.</td>
</tr>
<tr>
<td>Update</td>
<td>apply</td>
<td>key, filter</td>
<td>none</td>
<td>Applies the specified filter and records the transformation. Filter can be a built in filter of an application specific filter.</td>
</tr>
<tr>
<td>Provenance</td>
<td>getTree</td>
<td>key, depth</td>
<td>version tree</td>
<td>Returns the version tree which can be used for undo-redo operations. Depth specifies the number of revisions to return.</td>
</tr>
</tbody>
</table>

Table 1: Basic APIs for Front-End Clients and Back-End Storage

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crop</td>
<td>Removes the outer part of the photo.</td>
</tr>
<tr>
<td>Intensity</td>
<td>Adjusts the image’s brightness and/or contrast.</td>
</tr>
<tr>
<td>Sharpness</td>
<td>Makes the image more or less blurry.</td>
</tr>
<tr>
<td>Tone</td>
<td>Applies color tone filters (e.g., Sephia)</td>
</tr>
<tr>
<td>De-noise</td>
<td>Remove noise from the image.</td>
</tr>
</tbody>
</table>

Table 2: Sample Transformation Operations. This is not an exhaustive list, but examples.

ways, and the underlying implementation needs to have a mechanism that is most efficient. We discuss these implementation considerations next.

### 3.3 Implementation Considerations

To illustrate the implementation considerations, Figure 4 shows a refined version of the tree shown previously in Figure 3. The key to understand the implementation considerations is that a transformed photo can be propagated to another device in one of two ways—either sending the photo directly or sending only the transformation operation and applying it locally. However, always preferring one way over another will not yield satisfactory results. This is because the cost of always sending transformed photos will be high; in addition, always locally applying transformation operations, especially a chain of operations, might result in too much divergence as explained earlier in Section 2. Thus, the underlying implementation should consider these factors and decide on what the best way is.

The refined tree in Figure 4 illustrates this further. The tree starts with 3 versions of a photo with different resolutions. For the sake of the illustration, it assumes that different frontends only access a particular resolution, e.g., the highest resolution on a laptop, a lower resolution on a tablet, etc. In this case, if a user edits the photo on different devices, it will be done over different resolutions. In Figure 4, solid arrows indicate the transformations done over different resolutions.

When synchronizing/transferring a transformed photo across different frontends, one way is to take the highest resolution copy, apply the transformation, generate multiple resolutions, and distribute those to appropriate frontends. It is likely that this will produce transformed photos with high quality. However, this is not the only way; for example, it is also possible to take a lower resolution copy and apply the transformation. This possibility is indicated by dotted arrows in Figure 4.

In fact, from the tree representation, we can see that there are many more possibilities for generating a transformed photo. Figure 4 illustrates this with the photo shown as grey. In the tree, there are many starting points to generate this photo, e.g., the original photo with the highest resolution (A:R1), the original with a lower resolution (A:R2), or the second-level transformed photo with the highest resolution (B:R1), etc. In addition, with each starting point, there are many paths that ultimately lead to the photo, e.g., A:R1 → B:R1 → B:R2 → C:R2 → C:R3, or A:R1 → B:R1 → C:R1 → C:R2 → C:R3, etc. Each path has different characteristics in terms of data transfer cost, final photo quality, etc. Thus, the underlying implementation needs to carefully evaluate these options and choose the most effective one.

### 4. RELATED WORK

With the explosion of the number of photos that are generated, it’s not surprising that a lot of work has been done in the recent years which deals with efficiently storing the photos on the cloud backend. There is also some previous work related to content adaptation. However, we are not aware of any work that aggregates photo management and content adaptation, providing a unified client side service. In this section, we discuss the related work in the areas of content adaptation and storage management systems.

**Content adaptation:** In the setting of content adaptation, previous work investigated methods for supporting updates on adapted, low fidelity items and their reconciliation with full versions, in a scenario with PowerPoint and e-mail messages with media attachments [8], textual document [14], XML-based document formats [12], and images [11]. quFile [13] provides an abstraction layer that encapsulates different representation of the same logical data. The particular representation returned by quFile is determined at the time, depending on context and policy.

**Storage systems:** Facebook’s f4 [10] and Haystack [4] are dedicated photo stores on the cloud which tackle the
problem of efficiently storing and retrieving billions of photos on Facebook. Pixelcior complements these systems by offering a solution to efficiently manage photos and updates at the client side.

Ori file system [9] is a distributed peer-to-peer file system that supports offline operations and version control. Though not directly related to our work, it provides a tree-based history management, similar to what we envision for Pixelcior.

Simba [6] tackles the issues of file inconsistencies that arise when syncing mobile data to the cloud and proposes a data-sync service with a unified interface for objects and tables. While Pixelcior also suggests a unified interface for accessing photos, our focus is mainly on content adaptation and picture management.

5. CONCLUSION

In this paper, we have motivated the need of global photo management service which greatly helps the development of applications dealing with photos. We have summarized the requirements and challenges involved in the development of such a service. We have also discussed the design space in general and suggested a practical system design. We have argued a case for the need of content adoption and suggests a novel way of propagating updates to images. In our preliminary work, we have explored various photo management applications and image manipulation techniques. Our next steps are to flush out the design of Pixelcior and demonstrate its utility across various platforms.

6. REFERENCES