Orchestrating Image Retrieval and Storage Over a Cloud System

Jannatun Noor, Md. Nazrul Huda Shanto, Joyanta Jyoti Mondal, Md. Golam Hossain, Sriram Chellappan, and A. B. M. Alim Al Islam

Abstract—Since massive numbers of images are now being communicated from, and stored in different cloud systems, faster retrieval has become extremely important. This is more relevant, especially after COVID-19 in bandwidth-constrained environments. However, to the best of our knowledge, a coherent solution to overcome this problem is yet to be investigated in the literature. In this article, by customizing the Progressive JPEG method, we propose a new Scan Script to ensure Faster Image Retrieval. Furthermore, we also propose a new lossy PJPEG architecture to reduce the file size as a solution to overcome our Scan Script's drawback. In order to achieve an orchestration between them, we improve the scanning of Progressive JPEG's picture payloads to ensure Faster Image Retrieval using the change in bit pixels of distinct Luma and Chroma components (Y, C_b, and C_r). The orchestration improves user experience even in bandwidth-constrained cases. We evaluate our proposed orchestration in a real-world setting across two continents encompassing a private cloud. Compared to existing alternatives, our proposed orchestration can improve user waiting time by up to 54% and decrease image size by up to 27%. Our proposed work is tested in cutting-edge cloud apps, ensuring up to 69% quicker loading time.

Index Terms—Scan script, progressive JPEG (PJPEG), faster image retrieval, image compression, discrete cosine transform, cloud computing

1 Introduction

With so many applications, multimedia communication over the cloud is gaining significant interest in recent times [1], [2]. These systems often leverage various open-source projects for faster and storage efficient access to image data, which is a critical component in multimedia communication over the Internet today.

There exists many formats to store images, among which JPEG is the most popular [3]. JPEG is used by almost all image-capturing devices today. In 2015, 7 billion images were produced in JPEG format every day [4], which is much higher now. The number of images stored in JPEG format from 2022 to 2023 is expected to increase by 10.7%. Besides, 74.2% of the websites use JPEG as their image format. Thus, as there is a huge amount of data stored in this format, and as such, optimizing retrieval of JPEG images is of utmost significance today.

JPEG performs lossy compression using an algorithm called Discrete Cosine Transform (DCT). For performing JPEG operations, baseline method is mostly used. This method works by encoding all the pixels sequentially. It produces the highest compression ratio and guarantees the best image quality. A less used method is Progressive JPEG (PJPEG). It works by loading lower frequency pixels of an image (or a low-quality presentation of the image) first. Later, it refers to the higher frequency pixels of the image. It shows a faster preview of the images. Hence, Progressive JPEG offers advantages under bandwidth-constrained environments.

Investigating the notion of Progressive image loading and retrieval has gained great interest in the research community in recent times [5]. Research studies [6], [7] in this regard mostly explore Progressive images' performance from the perspective of high-bandwidth network connections. However, slow Internet connections and limited bandwidths are a reality in many countries all over the world [8]. The user experience has become very important in recent years, as the Internet traffic keeps increasing (more so due to the recent COVID-19 pandemic [9]). Accordingly, to enhance the level of user experience, the performance of Progressive image loading and retrieval in a cloud environment needs to be improved even sustaining slow Internet connections and limited bandwidths.

In the case of Progressive Image Retrieval, the existing method for encoding PJPEG consists of loading 7 DC coefficient bits in the First Scan [10]. As the DC coefficient (pixel) usually contains high-magnitude values, it takes substantial

- *Jannatun Noor is with the Department of CSE, Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh, and also with the School of Data and Sciences, BRAC University, Dhaka 1212, Bangladesh. E-mail: jannatun.noor@bracu.ac.bd.*
- *Md. Nazrul Huda Shanto and Joyanta Jyoti Mondal are with the School of Data and Sciences, BRAC University, Dhaka 1212, Bangladesh. E-mail: {md.nazrul.huda.shanto, joyanta.jyoti.mondal}@g.bracu.ac.bd.*
- *Md. Golam Hossain is with Tirzok Private Limited, Dhaka 1205, Bangladesh. E-mail: mghhimu@gmail.com.*
- *Sriram Chellappan is with the University of South Florida, Tampa, FL 33620 USA. E-mail: sriramc@usf.edu.*
- *A. B. M. Alim Al Islam is with the Department of CSE, Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh. E-mail: alim_razi@cse.buet.ac.bd.*

Manuscript received 14 June 2021; revised 23 Mar. 2022; accepted 24 Mar. 2022. Date of publication 0 . 0000; date of current version 0 . 0000.
This work was supported in part by the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh and in part by the US National Science Foundation under Grant 2014547.
(Corresponding authors: Jannatun Noor, Md. Nazrul Huda Shanto, and Joyanta Jyoti Mondal.)
Recommended for acceptance by V. Piuri.
Digital Object Identifier no. 10.1109/TCC.2022.3162790

time to load the 7 bits. To decrease the loading time, one way is to load a lower number of DC coefficient bits that can provide a solution for Faster Image Retrieval. Till now, to the best of our knowledge, no research study focuses on this aspect towards achieving better user experiences through performing faster Progressive image loading even under bandwidth constraints. Besides, existing research studies are also yet to focus on this important realm in multimedia cloud operation and communication covering Faster Image Retrieval using progressing schemes sustaining the bandwidth limitation. This is equally applicable to popular OpenStack-like systems such as Secure Processing aware Media Storage (SPMS) [2].

To address this, in this paper, we propose a new Progressive Scan Script using fewer bits in the First Scan. We encode only 4 DC coefficient data bits in the First Scan without degradation in the image quality. Hence, it shows a much faster visualization of the image. User Waiting Time significantly decreases to 54% after using our new Script. A potential downside of the Scan Script is that it tends to make image size larger. Hence, we also propose a PJPEG lossy Architecture to overcome the drawback by reducing image file size.

Based on our study, we make the following set of contributions in this paper:

- We propose a new Scan Script for Faster Image Retrieval. Our proposal is inspired by a thorough investigation of the open-source libjpeg library [11] and optimization of scan scripts for Progressive JPEG.
- To overcome a potential downside of our proposed Scan Script of making image size larger, we propose a new lossy PJPEG architecture to produce smaller-sized image files.
- We implement our proposed architecture in a real testbed comprising a high-configuration server in Canada and a client in Bangladesh, which embraces the notion of a private cloud. Besides, our testbed setup realizes limited bandwidth and slow Internet connection perspectives. In the process of implementing the testbed, we elaborate system design and deployment details of the proposed architecture.
- We conduct rigorous experimentation over the testbed setup to evaluate the performance of our proposed architecture. We compare our experimental results against that of alternative solutions over various devices. The comparison confirms the better performance of our proposed architecture compared to that of the existing alternative solutions.
- Further, we compare the performance of our proposed work with that of other state-of-the-art cloud applications such as Dropbox and Google Drive. Our results demonstrate superior performance than the default image loading methods of the state-of-the-art cloud applications. Nonetheless, we also compare advantages of our proposed approach compared to other recent state-of-the-art research studies.

2 Related Work

Fetching large images from public storage systems to own processing systems and then processing those images in the own processing systems — both appear to be expensive and time-consuming. Our previous work [12] focus to retrieve image efficiently and securely in a private cloud. We integrate resizing and encryption–decryption algorithms as a secured proxy service combined with a cloud file sharing environment named Swift. High-resolution images often take a substantial amount of time to load with average network bandwidth speed. In cases, it even considerably takes longer on mobile devices over wireless connections. Hence, many research studies focus on partial visual contents for better user experience. Study [13] presents Content-Based Image Retrieval (CBIR) system that achieves coarse-to-fine progressive Remote Sensing (RS) image description and retrieval in the partially decoded JPEG-2000 compressed domain. Study [14] proposes a cloud-based face video retrieval system with deep learning. Studies [15], [16] proposed a progressive image transmission scheme based on strategic decomposition and block truncation coding, respectively.

To the best of our knowledge, our proposed methodology is the first to focus on faster and smoother progressive image retrieval for a bulk amount of images even in the presence of bandwidth-constrained scenarios. As we have discussed previously, to overcome our scan scripts drawback, we work with PJPEG compression. Researchers have always been trying to make JPEG compression more efficient in many different ways [17]. Study [18] proposes reducing redundant data in the DCT domain by performing selective quantization and optical encoding for Baseline JPEG. Study [19] suggests image pre-processing steps to improve standard JPEG compression ratio by increasing color repetition probability. Study [20] modify JPEG based on quick DCT that removes the majority of zeros. Moreover, Study [21] propose to use segmented entropy encoding. Lastly, study [22] shows that dynamic resizing with progressive JPEG saves $2.5\times$ read data over baseline JPEG at a Peak Signal-to-Noise Ratio (PSNR) of 32 dB.

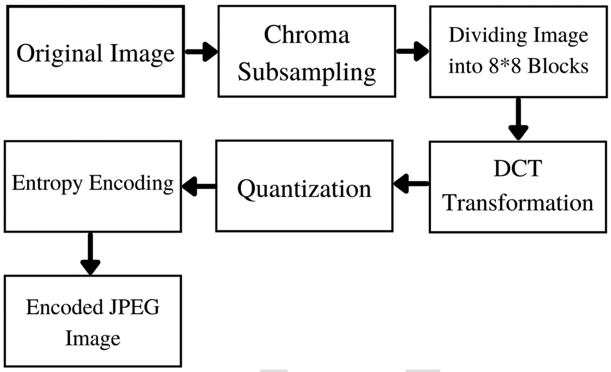


Fig. 1. JPEG encoding technique.

3 Background

JPEG compression is a lossy compression. JPEG deletes data bits while performing different processes like chroma subsampling, quantization, entropy encoding, etc. In Fig. 1, we see the encoding process of JPEG compression. To start, JPEG turns images from RGB to a different color space named $YCbCr$. JPEG uses this color space to delete specific data bits. Y or luminance is the light intensity. Cb and Cr

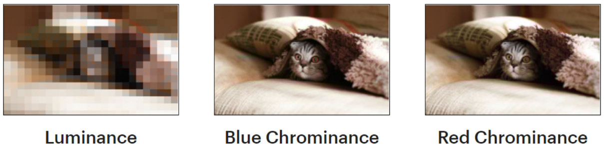


Fig. 2. Subsampling by 30% [23].

represents red chrominance, and blue chrominance respectively. Our eyes are more sensitive to luminance. Whereas, less sensitive to sudden changes in chrominance components [23], [24]. Fig. 2 shows changes in components after subsampling by 30%. Our eyes cannot detect sudden changes in chrominance. Hence, JPEG divides only the chrominance information by a factor of 2. This process is called chroma subsampling.

Next, JPEG divides a picture into chunks of 8×8 blocks. Sequence for pixels in a 8×8 is shown in Table 1. Every block contains $64(0-63)$ pixels and every pixels consist of 3 components(Y, Cb, Cr). Pixel values are from 0-255. JPEG subtracts every pixel value by 128.

Later, JPEG uses DCT to convert 8×8 block components to a frequency domain

$$\begin{aligned} F(u,v) &= \frac{1}{4} C(u)C(v) \sum\nolimits_{x=0}^{7} \sum\nolimits_{y=0}^{7} \\ & f(x,y) \cos\left[\frac{\pi(2x+1)u}{16}\right] \cos\left[\frac{\pi(2y+1)v}{16}\right] \\ & \text{for } u = 0,\ldots,7 \text{ and } v = 0,\ldots,7 \\ & \text{where } C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } k = 0 \\ 1 & \text{otherwise} \end{cases}. \end{aligned} \tag{1}$$

Equation (1) [26], [27] represents DCT. JPEG gets 64 new coefficients or pixel values after using DCT for all of the components. The First Coefficient of a block represents the DC coefficient. This coefficient shows the general intensity of the whole image block. AC coefficients change the intensity and have a much less magnitude than the DC coefficient.

In Fig. 3, we see, from the DC coefficient, as we go horizontally by moving right or vertically by moving down to AC coefficients, the frequency keeps increasing. DC coefficient has much more effective than AC coefficients as our eyes are not good at differentiating high-frequency data bits.

JPEG further reduces these coefficients by dividing these coefficients by quantization matrix. Quantization matrix

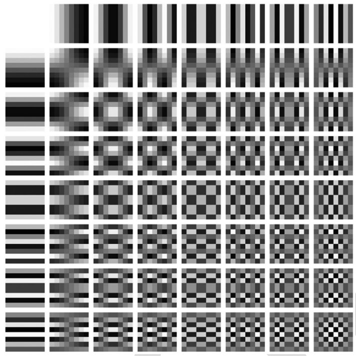


Fig. 3. Discrete cosine transform (DCT) [25].

values are lower for DC and its closer AC coefficients. There are separate quantization matrix tables for luminance and chrominance. In Table 2, we see the quantization table for luminance. As shown In Equation (2), JPEG only preserves the rounded values after the division. The data we lost in the process of rounding value is not renewable. That is why JPEG is a lossy compression. This process is called quantization. Quantization helps to get lower values for high-frequency AC coefficients

$$F_q(u,v) = \text{Round}\left(\frac{F(u,v)}{Q(u,v)}\right). \tag{2}$$

The last step for encoding JPEG is entropy encoding. Entropy encoding encodes coefficients with the same values in a zigzag format. The zigzag format is helpful to encode the image from a lower frequency to higher frequency data bits. Normally, Huffman Coding is used for entropy encoding. To decode the image, the processes are done again reversely.

The baseline method and the Progressive method encode pixels differently. The baseline method encodes images block by block. Where Progressive JPEG encodes specific pixels for every block script by script. Many social sites and websites are now using compressed and resized JPEG files to cover diversified remote devices [1], [2]. Hence, we briefly present the library of JPEG (libjpeg) [11] and OpenStack Swift-like media storage systems to provide a background related to our approach.

Libjpeg. Libjpeg library (written in C) is used in many platforms for handling JPEG image data format through implementing JPEG codec (encoding and decoding). It performs conversions between images inserting and exerting

TABLE 1
The Order to Scan DCT Coefficients [24]

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

TABLE 2
The Quality Factor [24]

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

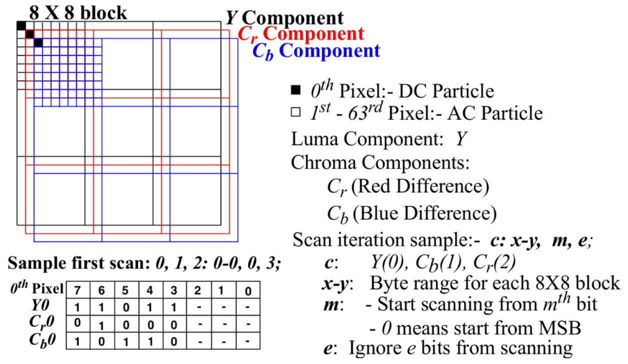


Fig. 4. An example of 8 x 8 block JPEG structure of Luma and Chroma components. Here, the scan iteration sample is explained for progressive JPEG type images.

textual comments and transforming JPEG files using libjpeg-turbo [11].

SPMS (Secure Processing Aware Media Storage). Recently, many media cloud storage such as SPMS are deployed using OpenStack Swift. Swift is an open-source object storage system having some special features. Such as eventual consistency, high availability, fault tolerance, replication, etc. It has two types of servers-proxy for management and processing and 3 storage servers (account, container, and object) for storing database and data objects [28]. Besides, the SPMS system has some special features of media securing, image data conversion to PJPEG, image resizing, video transcoding and resizing to various sizes, etc [2]. As SPMS-like media storage systems are used for multipurpose media management tasks (Such as video streaming and storing many versions of images), optimizing multimedia retrieval comes into play.

4 System Design and Implementation

We evaluate the performance of our proposed architectures through a real implementation. First, we briefly present our experimental testbed setup. Later, we present experimental results and findings for our architectures. Lastly, we compare our method with other existing studies.

4.1 Faster Image Retrieval

To ensure Faster Image Retrieval, partial loading is essential. Since Progressive JPEG allows partial encoding and decoding, we use Progressive JPEG. A Progressive JPEG is loaded Scan by Scan. The First Scan sets the parameter for the number of bits it will encode in the first partial loading. Hence, the less bit we use in the first Scan, the faster we load the first partial image. However, loading fewer bits can produce bad image quality. Our target is to encode a minimal number of bits for the first Scan while maintaining the visual quality same as the default Scans produced image.

For a better understanding of the architecture, here, we first present the structure of JPEG images in Fig. 4. The 0th pixel contains DC particle or coefficient and 1st to 63rd pixels contain AC coefficients [6]. Scan iterations over the pixels are represented with some variables. For example, each Scan Script can be represented by c: $x-y, m, e$. Here, c: $0, 1, 2$ ($0: Y$ component, 1: C_r component, and 2: C_b component). $x-y$ represents the pixel range that needs to be scanned for each 8 x 8 block. Thus, $0-0$ means scanning 0^{th} pixel for each block. Additionally, m: refers to Scan 'm' last bits, i.e., bits after this index need to be scanned. Here, '0' refers to the beginning or MSB. Nonetheless, e refers to skip 'e' bits counting from LSB. In Fig. 4, we present the sample of 0^{th} pixel for Y component. Here, we select all the DC coefficients.

Default Scan Script iterations are available online.[1] First Scan of Default Scan Script is $0, 1, 2 : 0-0, 0, 1$. Hence, the default Scan Script encodes 7 bits from the DC coefficient for all three components in the First Scan. Our target is to encode the lowest number of bits for the First Scan with maximum visual quality. Hence, we make eight different Scan Scripts (SS_1 to SS_8) by increasing bit by bit gradually for the First Scan. For example, we encode 1 bit from DC coefficient in the First Scan of SS_1, 2 bits for SS_2, and 8 bits SS_8, etc. For maximum visual quality, we select both luma (or luminance) and chroma (or chrominance) components; otherwise, the First Scan will be only black and white.

The First Scan for Scan Script 1 (SS_1) is $(0, 1, 2 : 0-0, 0, 7)$, the First Scan for Scan Script 2 (SS_2) is $(0, 1, 2 : 0-0, 0, 6)$, the First Scan for Scan Script 8 (SS_8) is $(0, 1, 2 : 0-0, 0, 0)$, etc. Out of these 8 Scan Scripts, to challenge the default Scan Script, we need a script that encodes fewer bits in the first Scan and produces image quality the same as the default Scan Script's First Scan. Our proposed First Scan Script will be as follows:

$$SS_{s1} = \min_{1<=i<=8} SS_{zi} \wedge \max_{1<=i<=8} V_{qi}. \tag{3}$$

We represent SS as the Scan Scripts and SS_s as the Scan number of the Scan Scripts. SS_{s1} represents the first Scan of the Scan Scripts. SS_{zi} represents the size of the image using the first Scan. V_{qi} represents the visual quality of the i^{th} Scan.

4.2 Lossy PJPEG Architecture

We modify our proposed Scan Script 4 (SS_4) and propose a lossy PJPEG architecture. First, We identify comparatively lower frequency coefficients. In Fig. 5a, we denote comparatively lower frequency pixels as LF, and comparatively higher frequency pixels as HF. We identify them by rigorously experimenting with the script. The pixels that have a huge impact on the image while skipping a data bit, we consider these as LF. Hence, we denote Coefficients $0-5$, $8-12$, $16-20$, $24-27$, $32-33$ as LF. Coefficients $6-7$, $13-15$, $21-23$, $28-31$, $34-63$ are HF.

We do not skip any bits for pixels $0-5$, $8-12$, $16-20$, $24-27$, and $32-33$. They have the highest impact on the image as it includes the DC and its closest AC coefficients. Later, We find $13-15$ and $21-23$; these pixels have a higher impact on the image compared to other AC coefficients. Hence, we skip only 1 bit from these pixels for all three components. For pixels $6-7, 28-31$ and $34-39$, we skip 2 bits for all of the components. C_b is the least sensitive color to our eyes. From $40-63$ pixels, we skip 3 bits for C_b. Only 2 bits for Y and C_r. Default Scan Script do not skip these bits and produce a larger image file size.

1. *https://github.com/libjpeg-turbo/libjpeg-turbo/blob/1.0.x/jcparam.c* (Line No. 508-526)

	0	1	2	3	4	5	6	7
0	LF	LF	LF	LF	LF	LF	HF	HF
1	LF	LF	LF	LF	LF	HF	HF	HF
2	LF	LF	LF	LF	LF	HF	HF	HF
3	LF	LF	LF	LF	HF	HF	HF	HF
4	LF	LF	HF	HF	HF	HF	HF	HF
5	HF	HF	HF	HF	HF	HF	HF	HF
6	HF	HF	HF	HF	HF	HF	HF	HF
7	HF	HF	HF	HF	HF	HF	HF	HF

(a) DCT Frequency coefficients

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	2	2
1	0	0	0	0	0	1	1	1
2	0	0	0	0	0	1	1	1
3	0	0	0	0	2	2	2	2
4	0	0	2	2	2	2	2	2
5	2	2	2	2	2	2	2	2
6	2	2	2	2	2	2	2	2
7	2	2	2	2	2	2	2	2

(b) Skipped Bits for Y and C_r component

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	2	2
1	0	0	0	0	0	1	1	1
2	0	0	0	0	0	1	1	1
3	0	0	0	0	2	2	2	2
4	0	0	2	2	2	2	2	2
5	3	3	3	3	3	3	3	3
6	3	3	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3

(c) Skipped Bits for C_b component

Fig. 5. In Fig. 5a identifies the lower frequency (LF) and higher frequency (HF) coefficients. (1,1) is the DC coefficient. Figs. 5b and 5c show the number of data bits we skip from each of the coefficients. Skipped bits are the same for component C_r and Y.

Figs. 5b and 5c show the number of data bits we skip each of the pixels. We skip 0 bits from 0th - 5th, 8th - 12th, 16th - 20th, 24th - 27th, 32nd - 33rd coefficients, 1 bit from 13th - 15th, 21st - 23rd coefficients, 2 bits from 6th - 7th, 28th - 31st, 34th - 39th coefficients for all the three components. Last, we skip 2 bits for Y and C_r, 3 bits for C_b from 40th - 63rd coefficients. We skip the bits from LSB. The bits we are skipping are deleted from the image

$$SS = \min_{1<=i<=512} SS_{xi} \wedge \max_{1<=i<=512} V_{gi}. \tag{4}$$

Here, SS represents the Scan Scripts. SS_{xi} represents the size of the image after loading i number of data bits. V_{gi} represents visual quality after loading i number of data bits. 8×8 block has $64(0-63)$ coefficients. Each of the coefficients carries 8 bits of data. Hence, in total, we have 512 data bits.

5 Performance Evaluation

We evaluate the performance of our proposed architectures through a real implementation.

5.1 Experimental Testbed Setup

We use real high-resource machines for deploying testbed servers in Canada. We create these servers using virtual machines, hosted in a physical data center. Here, we use two proxy servers, three account-container servers, three object servers for the media storage cluster. We use AMD Opteron 62xx class CPU, and OS Cent-OS 7. The memory and disk configurations of our Swift servers here cover- 1) two proxies each having one 8 GB memory and one 20 GB disk. 2) three account-containers each having one 8 GB memory and three disks each of 50 GB. 3) three objects having one 8 GB memory and three disks each of 700 GB. Each server has six 1 GB network interface cards. Fig. 6 and Table 13 present the experimental setup of our testbed. In addition, we deploy a private media cloud Secure Processing-aware Media Storage (SPMS) using OpenStack Swift (stable newton branch) with three replicas $(r=3)$ and 16384 partitions $(p=16384)$. There are nine devices for the account, container, and object ring files. Hence, each device has around 5461 partitions in $/srv/node/<server>$ folders (devices are mount in this location according to

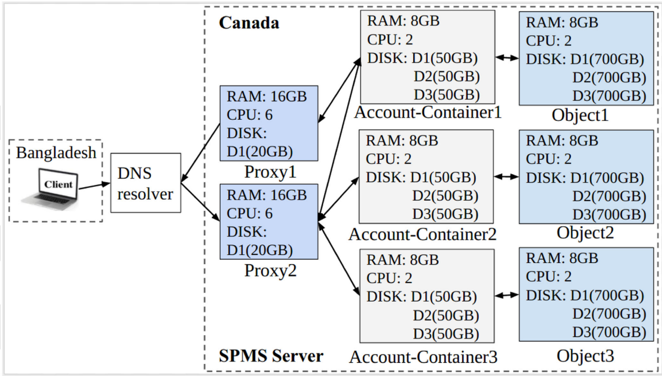


Fig. 6. Testbed setup comprising a server in Canada and a client in Bangladesh.

OpenStack Swift guide [28]). Moreover, we implement a social site for both mobile and web users. The mobile site contains different features for social interactions such as free video calls, chats, feeds, stickers, and so on. The site has already experienced more than 5 million downloads. The images that are saved and processed on this site leverage the architectures we propose in this paper.

In this setup, we upload different types of data from clients to the development server for around eight months.[2] Besides, we create 10,000 accounts and 10,000 containers in the Swift cluster. We upload around 1M images and video files in those accounts. Hence, the number of objects (n) is 1M for our test-bed server. We upload around 1.5TB data. Therefore, total data becomes 1.5TB $\times 3 = 4.5$ TB in our development server.

Moreover, we use another web hosting server (Fig. 8) for a different purpose. We use this server to test a real case scenario for the difference between the load time of a normal image and our proposed algorithms. This server is located in London, UK. The client is located in Dhaka, Bangladesh. It has 30 hops from the client to London through hopping over Kansas, USA. Note that, the performance will be affected depending on the distance between the locations of

2. The users have uploaded objects (images) according to their personal preferences and choice in their real usages. Thus, all the objects are mostly different as they come from real usages. We choose these images as they represent the real-life testing of our proposed architecture.

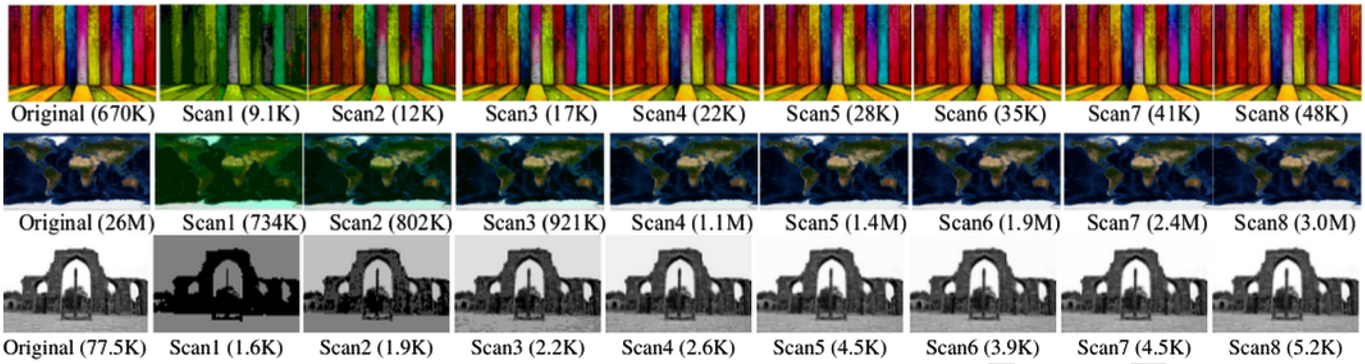


Fig. 7. Comparison of first scan images for eight combinations. Here, Scan1 is (SS_{s1} of SS_1), Scan2) is (SS_{s1} of SS_2, Scan3 is (SS_{s1} of SS_3), Scan4 is (SS_{s1} of SS_4), Scan5 is (SS_{s1} of SS_5), Scan6 is (SS_{s1} of SS_6), Scan7 is (SS_{s1} of SS_7), and Scan8 is (SS_{s1} of SS_8).

the client and the server. The loading time will increase by some milliseconds if the distance gets increased and vice versa. It will exhibit a similar effect in the case of the hop distance, i.e., the loading time will increase if the number of hops increases. To explore the impact, we change the location of the server to Singapore minimizing the number of hops from 30 to 11 while keeping the client in Dhaka, Bangladesh. After minimizing the number of hops, we observe a change of up to 25% difference in the loading time.

We create a custom dataset of 1333 pictures. Our selected dataset includes different sizes, resolutions, colorful, black and white images. We collect these pictures from datasets published in Kaggle [30], [31]. Table 14 shows the number of pictures of different sizes in the dataset.

To further evaluate our architectures, we use the MSCOCO2015 Test Dataset [32], which contains almost 81,000 images of various categories.

Furthermore, we use a local virtual machine (Cent-OS 7) to calculate the cumulative size of our proposed scan scripts. We install libjpeg, libjpeg-turbo, and libjpeg-turbo-utils in the virtual machine [11]. We use thousands of images of different sizes for testing our proposed Scan Scripts.

We use Structural Similarity Index (SSIM) to calculate image quality to perform an objective-based evaluation using QoE [33]. For calculating SSIM, we use VQMT software [29] and method available to calculate SSIM in Scikit-Learn library in Python [34]. A higher SSIM value means more similar to the original image. Also, we use Python script to find the difference in file size between the original and compressed image.

TABLE 3
Cumulative Size for Five Different Images Using Default Scan Script (SS) [10]

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 1;	41K	346.75K	271.71K	1.19M	2.4M
0: 1-5, 0, 2;	128K	713.06K	657.11K	2.56M	5.4M
2: 1-63, 0, 1;	145K	836.36K	803.46K	2.92M	5.8M
1: 1-63, 0, 1;	163K	976.48K	947.92K	3.42M	6.3M
0: 6-63, 0, 2;	267K	1.08M	1.09M	4.03M	8.4M
0: 1-63, 2, 1;	406K	1.45M	1.60M	5.40M	14M
0,1,2: 0-0, 1, 0;	413K	1.51M	1.64M	5.67M	15M
2: 1-63, 1, 0 ;	433K	1.63M	1.76M	6.42M	15M
1: 1-63, 1, 0 ;	455K	1.75M	1.89M	7.15M	16M
0: 1-63, 1, 0 ;	636K	1.89M	2.77M	9.39M	25M

5.2 Experimental Results

We describe the Experimental Results by our contributions separately.

5.2.1 Faster Image Retrieval

Tables 3, 4, 5, 6, 7, 8, 9, 10, and 11 present the cumulative size of each Scan files using default Scan Script and Scan Script $1-8$ for five images. For our benchmarking process, each table contains the combinations of scanning images while converting them from baseline to progressive. Furthermore, we upload them into our cloud. Later, comparing their sizes after each phase of the Scans.

In Fig. 7, we present three images (Image1 of 670 KB, Image5 of 26 MB, and Image6 of 77.5KB) implementing the First Scans(SS_{S1}) for 8 Scan Scripts (SS_1-SS_8). We find, Scan4 to Scan8 all the images look exactly the same. As we use fewer bits for Scan4, we choose Scan Script 4 to compare with the default Scan Script. Scan7 encodes 7 DC coefficient bits same as default Scan Script in the First Scan. Hence, we refer to Scan7 as the First Scan for default Scan Script.

For subjective-based evaluation, we use Mean Opinion Score (MOS) [33], [35] metric. We request 25 observers to differentiate among the images of Fig. 7 to perform a subjective evaluation. All of them confirm that visual quality (V_q) is the same for the First Scan of Scan Script 4 (SS_4) and Scan Script 7 (SS_7). For objective based evaluation, Table 12 shows the MOS and SSIM values of four images for the First

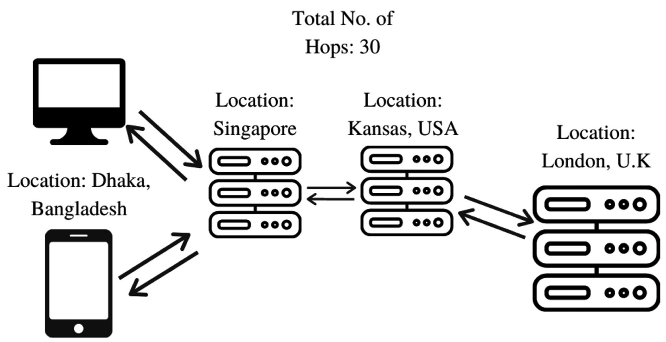


Fig. 8. Testbed server setup to obtain performance of diversified remote devices using the web hosting server.

TABLE 4
Cumulative Size for Five Images Using SS_1

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 7;	9.1K	84.24K	64.83K	317.19K	734K
0,1,2: 0-0, 7, 6;	17K	144.15K	107.01K	597.58K	1.4M
0,1,2: 0-0, 6, 5;	24K	204.05K	148.32K	878.17K	2.1M
0,1,2: 0-0, 5, 4;	30K	263.82K	189.39K	1.13M	2.9M
0,1,2: 0-0, 4, 3;	37K	323.33K	230.44K	1.40M	3.6M
0,1,2: 0-0, 3, 2;	43K	382.75K	271.46K	1.67M	4.2M
0,1,2: 0-0, 2, 1;	50K	442.05K	312.49K	1.94M	4.9M
0,1,2: 0-0, 1, 0;	57K	501.34K	353.49K	2.21M	5.5M
0: 1-27, 0, 1 ;	381K	1.34M	1.42M	5.63M	16M
2: 1-27, 0, 1 ;	397K	1.46M	1.56M	5.99M	16M
1: 1-27, 0, 1 ;	415K	1.60M	1.70M	6.49M	17M
0: 28-63, 0, 1 ;	431K	1.60M	1.70M	6.49M	17M
2: 28-63, 0, 1 ;	431K	1.60M	1.70M	6.49M	17M
1: 28-63, 0, 1 ;	431K	1.60M	1.70M	6.49M	17M
0: 1-63, 1, 0 ;	612K	1.74M	2.58M	8.73M	26M
2: 1-63, 1, 0 ;	632K	1.86M	2.71M	9.48M	27M
1: 1-63, 1, 0 ;	654K	1.99M	2.84M	10.21M	28M

TABLE 5
Cumulative Size for Five Images Using SS_2

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 6;	12K	105.72K	84.84K	362.15K	802K
0,1,2: 0-0, 6, 5;	19K	165.63K	126.15K	642.74K	1.5M
0,1,2: 0-0, 5, 4;	26K	225.40K	167.21K	923.05K	2.3M
0,1,2: 0-0, 4, 3;	32K	284.91K	208.27K	1.17M	3.0M
0,1,2: 0-0, 3, 2;	39K	344.32K	249.29K	1.44M	3.6M
0,1,2: 0-0, 2, 1;	46K	403.63K	290.32K	1.71M	4.3M
0,1,2: 0-0, 1, 0;	52K	462.91K	331.31K	1.98M	4.9M
0: 1-27, 0, 1;	377K	1.30M	1.39M	5.40M	15M
2: 1-27, 0, 1;	393K	1.42M	1.54M	5.76M	16M
1: 1-27, 0, 1;	411K	1.56M	1.68M	6.26M	16M
0: 28-63, 0, 1;	426K	1.56M	1.68M	6.26M	17M
2: 28-63, 0, 1;	427K	1.56M	1.68M	6.26M	17M
1: 28-63, 0, 1;	427K	1.56M	1.68M	6.26M	17M
0: 1-63, 1, 0;	607K	1.70M	2.56M	8.50M	26M
2: 1-63, 1, 0;	628K	1.83M	2.69M	9.25M	26M
1: 1-63, 1, 0;	650K	1.95M	2.82M	9.98M	27M

Scans of 8 Scan Scripts. In Table 12, We see the SSIM values are almost the same for Scan4 and Scan7.

Furthermore, we test First Scan of (SS_4) and (SS_7) in MSCOCO2015 Dataset. We find the average SSIM is 0.551 and 0.553 respectively. That verifies we get the same quality images for the First Scan's of Scan Script 4 and default Scan Script.

Lastly, we compare the load time of the actual picture and the picture generated by our proposed script in various State of The Art Cloud Applications such as Google Drive and Dropbox. Table 15 shows the load time difference between the pictures using the network section of Chrome DevTools [36]. We use 3 images (Image3, Image4, and Image5) to load our proposed First Scan of Scan Script 4. We load the images in different bandwidths. For example, Image3 in 0.125MBps, the original image loads in 66 seconds. Moreover, the image using our proposed Script takes

TABLE 6
Cumulative Size for Five Images Using SS_3

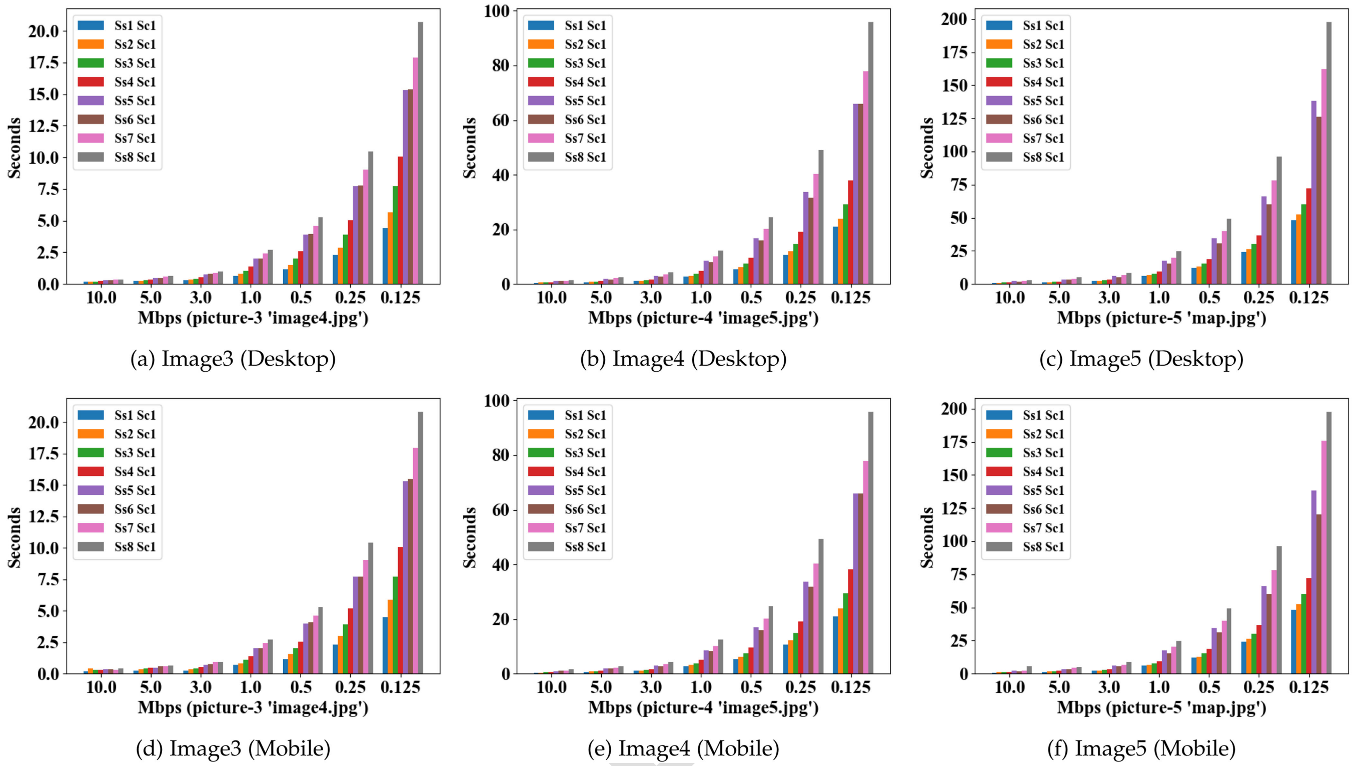
Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 5;	17K	138.02K	115.44K	445.45K	921K
0,1,2: 0-0, 5, 4;	24K	197.79K	156.51K	725.76K	1.7M
0,1,2: 0-0, 4, 3;	30K	257.29K	197.56K	0.98M	2.4M
0,1,2: 0-0, 3, 2;	37K	316.71K	238.58K	1.25M	3.0M
0,1,2: 0-0, 2, 1;	43K	376.02	279.61K	1.52M	3.7M
0,1,2: 0-0, 1, 0;	50K	435.30K	320.61K	1.79M	4.4M
0: 1-27, 0, 1;	374K	1.28M	1.38M	5.21M	15M
2: 1-27, 0, 1;	391K	1.40M	1.53M	5.56M	15M
1: 1-27, 0, 1;	408K	1.53M	1.67M	6.07M	16M
0: 28-63, 0, 1;	424K	1.54M	1.67M	6.07M	16M
2: 28-63, 0, 1;	424K	1.54M	1.67M	6.07M	16M
1: 28-63, 0, 1;	425K	1.54M	1.67M	6.07M	16M
0: 1-63, 1, 0;	605K	1.68M	2.55M	8.30M	25M
2: 1-63, 1, 0;	625K	1.80M	2.68M	9.06M	26M
1: 1-63, 1, 0;	647K	1.92M	2.81M	9.79M	27M

TABLE 7
Cumulative Size for Five Images Using SS_4 (Proposed)

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 4;	22K	186.71K	150.79K	577.73K	1.1M
0,1,2: 0-0, 4, 3;	29K	246.22K	191.85K	857.32K	1.8M
0,1,2: 0-0, 3, 2;	36K	305.64K	232.87K	1.10M	2.5M
0,1,2: 0-0, 2, 1;	42K	364.94K	273.90K	1.37M	2.1M
0,1,2: 0-0, 1, 0;	49K	424.23K	314.89K	1.65M	3.8M
0: 1-27, 0, 1;	373K	1.27M	1.38M	5.06M	14M
2: 1-27, 0, 1;	389K	1.39M	1.52M	5.42M	15M
1: 1-27, 0, 1;	407K	1.52M	1.66M	5.92M	15M
0: 28-63, 0, 1;	423K	1.52M	1.66M	5.92M	15M
2: 28-63, 0, 1;	423K	1.52M	1.66M	5.92M	15M
1: 28-63, 0, 1;	423K	1.52M	1.66M	5.92M	15M
0: 1-63, 1, 0;	604K	1.67M	2.54M	8.16M	25M
2: 1-63, 1, 0;	624K	1.79M	2.67M	8.91M	25M
1: 1-63, 1, 0;	646K	1.91M	2.80M	9.64M	26M

TABLE 8
Cumulative Size for Five Images Using SS_5

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 3;	28K	238.35K	190.20K	743.16K	1.4M
0,1,2: 0-0, 3, 2;	35K	297.77K	231.22K	0.99M	2.1M
0,1,2: 0-0, 2, 1;	42K	357.07K	272.25K	1.26M	2.8M
0,1,2: 0-0, 1, 0;	48K	416.36K	313.24K	1.53M	3.4M
0: 1-27, 0, 1;	373K	1.26M	1.38M	4.95M	14M
2: 1-27, 0, 1;	389K	1.38M	1.52M	5.31M	14M
1: 1-27, 0, 1;	407K	1.52M	1.66M	5.81M	15M
0: 28-63, 0, 1;	422K	1.52M	1.66M	5.81M	15M
2: 28-63, 0, 1;	423K	1.52M	1.66M	5.81M	15M
1: 28-63, 0, 1;	423K	1.52M	1.66M	5.81M	15M
0: 1-63, 1, 0;	603K	1.66M	2.54M	8.05M	24M
2: 1-63, 1, 0;	624K	1.78M	2.67M	8.80M	25M
1: 1-63, 1, 0;	646K	1.91M	2.80M	9.53M	26M



(a) Image3 (Desktop) (b) Image4 (Desktop) (c) Image5 (Desktop)

(d) Image3 (Mobile) (e) Image4 (Mobile) (f) Image5 (Mobile)

Fig. 9. Time needed to load different Scan Scripts under different bandwidth availability (Ss = Scan Script, Sc = Scan number).

only 34.06 seconds to load. It shows 48.39% improvement in our proposed Script. Table 15 confirms our images load faster on state-of-the-Art cloud applications as well.

5.2.2 Lossy PJPEG Architecture

In Table 14, we test our PJPEG Lossy Architecture using our custom dataset. We see 198 images are within the range of file size $1-30$kb. For these small-sized images, the average SSIM result is 0.96. On average, the image file size is reduced up to 10%. A slightly larger image produces a greater compression result. In the last group, 96 images within the file size of 3MB to 8MB produce the highest SSIM value of 0.98. It also produces the highest compression rate of reducing 25.31 percent more than regular JPEG standard compression.

Furthermore, we test our compression algorithm in the MSCOCO2015 dataset [32]. Our compression offers a 27.40% of reduction in file size than standard JPEG. The average SSIM result is 0.952.

5.2.3 System Resource Usage

We explore system resource usages by our proposed solutions and the default mechanism. As per our exploration, both our proposed solutions and the default mechanism consume nearly the same amount of resources. To be specific, as measured by System Monitor, memory usage is almost 100MB and CPU usage is close to $10-15\%$ in both cases.

5.3 Experimental Findings

Findings are discussed separately for both of our contributions as before.

TABLE 9
Cumulative Size for Five Images Using SS_6

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 2;	35K	294.39K	231.18K	0.94M	1.9M
0,1,2: 0-0, 2, 1;	41K	353.70K	272.21K	1.21M	2.5M
0,1,2: 0-0, 1, 0;	48K	412.98K	313.20K	1.48M	3.2M
0: 1-27, 0, 1;	372K	1.25M	1.38M	4.89M	14M
2: 1-27, 0, 1;	389K	1.38M	1.52M	5.25M	14M
1: 1-27, 0, 1;	406K	1.51M	1.66M	5.75M	14M
0: 28-63, 0, 1;	422K	1.51M	1.66M	5.75M	15M
2: 28-63, 0, 1;	422K	1.51M	1.66M	5.75M	15M
1: 28-63, 0, 1;	423K	1.51M	1.66M	5.75M	15M
0: 1-63, 1, 0;	603K	1.66M	2.54M	7.99M	24M
2: 1-63, 1, 0;	623K	1.78M	2.67M	8.75M	25M
1: 1-63, 1, 0;	645K	1.90M	2.80M	9.47M	26M

5.3.1 Faster Image Retrieval

We approach to balance the trade-off between the number of Scans and the size of the images in the first scan. We focus to ensure that viewers get an optimum view after the first partial image loading. Moreover, we ensure viewers do not wait for a long time to get the full image view because of a higher number of Scans. Considering these, after going through a rigorous bench-marking with some 50 images on our testbed, we observe that Scan Script 4 to Scan Script (SS_8) produce the same quality images in the First Scan. From them, Table 7 has a considerably fewer number of bits in the First Scan to generate a balanced view.

Fig. 9 shows the improvement of time and size of the candidate images in our test-bed with our proposed Scan

TABLE 10
Cumulative Size for Five Images Using SS_7

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 1;	41K	346.75K	271.71K	1.19M	2.4M
0,1,2: 0-0, 1, 0;	48K	406.03K	312.71K	1.46M	3.1M
0: 1-27, 0, 1;	372K	1.25M	1.38M	4.88M	13M
2: 1-27, 0, 1;	389K	1.37M	1.52M	5.23M	14M
1: 1-27, 0, 1;	406K	1.51M	1.66M	5.73M	14M
0: 28-63, 0, 1;	422K	1.51M	1.66M	5.73M	15M
2: 28-63, 0, 1;	422K	1.51M	1.66M	5.73M	15M
1: 28-63, 0, 1;	422K	1.51M	1.66M	5.73M	15M
0: 1-63, 1, 0;	603K	1.65M	2.54M	7.97M	24M
2: 1-63, 1, 0;	623K	1.77M	2.67M	8.73M	25M
1: 1-63, 1, 0;	645K	1.90M	2.80M	9.45M	25M

TABLE 11
Cumulative Size for Five Images Using SS_8

Scan Script	Size				
	Image1	Image2	Image3	Image4	Image5
0,1,2: 0-0, 0, 0;	48K	397.61K	312.96K	1.45M	3.0M
0: 1-27, 0, 1;	372K	1.24M	1.38M	4.87M	13M
2: 1-27, 0, 1;	389K	1.36M	1.52M	5.23M	14M
1: 1-27, 0, 1;	406K	1.50M	1.66M	5.73M	14M
0: 28-63, 0, 1;	422K	1.50M	1.66M	5.73M	15M
2: 28-63, 0, 1;	422K	1.50M	1.66M	5.73M	15M
1: 28-63, 0, 1;	423K	1.50M	1.66M	5.73M	15M
0: 1-63, 1, 0;	603K	1.64M	2.54M	7.96M	24M
2: 1-63, 1, 0;	623K	1.76M	2.67M	8.72M	24M
1: 1-63, 1, 0;	645K	1.89M	2.80M	9.45M	25M

TABLE 12
MOS and SSIM Values of Four Images for the First Scan of All the 8 Scan Scripts

First Scan	Avg. MOS	SSIM			
	Images	Image1	Image2	Image3	Image4
Scan1	0.51	0.29	0.62	0.60	0.54
Scan2	0.58	0.38	0.69	0.65	0.60
Scan3	0.65	0.42	0.82	0.69	0.65
Scan4	0.67	0.44	0.85	0.69	0.69
Scan5	0.68	0.45	0.87	0.70	0.70
Scan6	0.68	0.45	0.87	0.70	0.70
Scan7	0.68	0.45	0.87	0.70	0.70
Scan8	0.68	0.45	0.87	0.70	0.70

First Scan for different eight combinations are denoted as Scan1 - Scan8. MOS is calculated using 25 observers and SSIM is calculated using the VQMT tool [29].

TABLE 13
Configuration of Machines Used in Testbed Setup

Informations	Proxy Server	Object Server	Account-container Server	Client Machine
Architecture	x86_64	x86_64	x86_64	x86_64
CPU(s)	16	48	16	1
On-line CPU (s) list	0-15	0-47	0-15	0
Thread(s) per core	2	1	2	1
Core(s) per socket	4	12	4	1
Socket(s)	2	4	2	1
NUMA node (s)	2	8	2	1
CPU family	6	16	6	6
Model name	Intel(R) Xeon(R) CPU E5620 @2.40GHz	AMD Opteron (tm) Processor 6174	Intel(R) Xeon(R) CPU E5620 @2.40GHz	QEMU Virtual CPU version 1.5.3
CPU MHz	2394.141	2199.967	2394.103	2393.998
Virtualization Type	VT-x	AMD-V	VT-x	Full Storage

TABLE 14
Results After Applying Proposed PJPEG Lossy Architecture to the Custom Dataset

Size (kb)	1	30	100	500	1000	3000
	-30	-100	-500	-1000	-3000	-8500
Pictures	198	419	467	72	81	96
Reduced %	10.49	20.84	23.26	24.93	21.72	25.31
SSIM	0.96	0.96	0.96	0.97	0.98	0.98

Script, compared with the default Scan Script. Our proposed Scan Script gains over 50% improvement (54% to be exact) considering the time it takes for the first view of a progressive image to satisfy a viewer with an optimum view. Besides, for remote and local VM servers, the network hop is 16 and 2, respectively. Moreover, the average incoming and outgoing network speeds in a client machine is 400 Bit/s where Ttl is 43.61 MByte.

However, while using MSCOCO2015 Dataset, 73 out of 81,000 images are showing errors. These 73 images are black and white, and very small in size. The error does not occur for slightly larger-sized images. Later, we find that the default Scan Script also can not load these 73 images as well. We fix the error in our proposed script by removing chrominance components.

However, after loading all the scans, the image size is slightly larger for our proposed Scan Script 4. That is a minor drawback for our proposed Script.

5.3.2 Lossy PJPEG Architecture

While modifying our proposed Scan Script 4 (SS_4) to make a lossy architecture, we discover something unusual. we can not encode 32nd pixel alone. To solve this we had to encode 32nd and 33rd pixel together, despite the fact that 33rd pixel should be in the HF section. However, for making our scripts, we put 33rd pixel in LF.

To make the lossy compression, we try making many Scan Scripts. At first, we make a script that can reduce the file size up to 40% without even compromising image quality. However, it makes images a bit blurry while compressing a smaller image file size. The script produces good quality images for greater than 700kb file size. The median

TABLE 15
Load Time Comparison Between the Actual Picture and the Picture Generated by Our Proposed Faster Image Retrieval Scan Script in State of the Art Cloud Applications

Image3	Loading Time and Its Improvement in Google Drive			Loading Time and Its Improvement in Dropbox		
Speed	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)
10Mbps	1.17	0.832	28.89	1.86	1.73	6.99
5Mbps	1.83	1.18	35.52	2.62	1.79	31.68
3Mbps	3.52	1.36	61.36	6.21	5.66	8.86
1Mbps	10.05	5.99	40.40	19.79	17.37	12.23
0.5Mbps	17.44	9.43	45.93	28	18.76	33
0.25Mbps	34.29	17.16	49.95	72	72	0
0.125Mbps	66	34.06	48.39	114	96	15.79
Image4	Loading Time and Its Improvement in Google Drive			Loading Time and Its Improvement in Dropbox		
Speed	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)
10Mbps	1.95	0.928	52.41	2.45	1.91	22.04
5Mbps	4.16	1.3	68.75	3.73	3.56	4.56
3Mbps	2.92	2.48	15.07	6.34	6.04	4.73
1Mbps	9.61	8.55	11.03	19.78	19.07	3.59
0.5Mbps	15.91	12.15	23.63	39.11	38.25	2.20
0.25Mbps	54.06	33.56	37.92	66	57.53	12.83
0.125Mbps	114	66	42.11	150	144	4
Image5	Loading Time and Its Improvement in Google Drive			Loading Time and Its Improvement in Dropbox		
Speed	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)	Original Image (s)	Image using proposed Script (s)	Improvement in Proposed Script (%)
10Mbps	**No Preview Available due to a big Resolution size of 21600 x 10800**		**N/A**	1.6	1.57	1.88
5Mbps				3.01	2.99	0.66
3Mbps				5.08	5.07	0.197
1Mbps				16.69	16.67	0.12
0.5Mbps				33.68	32.52	3.44
0.25Mbps				66	66	0
0.125Mbps				138	138	0

Here, we counted the load time of only the image, not the UI.

size for images user usually consume is 200 to 2200kb on the Internet[53]. Hence, the script can not handle small size images. Therefore, we move forward to make another script that can maintain good quality for smaller images too. We come to know, the higher the image size is, the more we can delete data bits. Additionally, the more we delete data bits, the worse the image's quality becomes. Hence, we try removing fewer data bits to ensure the image quality. After experimenting more, we make a lossy PJPEG scan script that works for the smaller image file size. To use our lossy PJPEG architecture with having great results, we need a minimum image size of at least 6kb. Most used pictures on the Internet are greater than 6kb. Hence, it is not something that we should worry about. The bigger the image file size, the better SSIM we get, and the more we can reduce the image size. In our result Table 14, we see that our compression approach works better with larger images.

5.4 Comparison of Our Approach With Other Studies

We compare our proposed approach with other recent existing research studies in Tables 16 and 17. These tables compare the studies in qualitative and quantitative manners respectively. As shown in the Table 16, existing progressive JPEG based related research studies [13], [22], [38], [41], [42], [43], [44] use different technologies such as Dynamic Resizing, Segmented Compression, Trit-Planes Algorithm, Progressive Latent Ordering Nested Quantization (PLONQ), etc., to reduce file size for the overall image. Reducing file size leads to less retrieval time and transmission time for the full quality image. However, most of the existing studies do not focus on the notion of faster image preview even though faster image preview decreases user waiting time. A research study [42] significantly improves first preview time from JPEG2000 [54], BPG [55], Balle [56], WebP [57], and Toderici [58]. However, this study do not perform better than JPEG [26] and eventually have ended up with 26% increased user waiting time for JPEG. On the contrary, our proposed approach decreases user waiting time for JPEG by 54%. Here, our proposed approach adopts a new Scan Script for performing the first scan in road to ensuring a faster image preview. This, in turn, results in a faster loading of images using our proposed approach compared to other existing technologies.

Besides, storing images in public clouds can significantly increase retrieval delay. Using private clouds for managing images could present a remedy here, which is sparsely

TABLE 16
Comparison of Our Proposed Approach With Other Existing Research Studies

Name	Progressive Loading	Private Cloud	Retrieving Image Faster	Efficient Image Storage	User Waiting Time	Underlying Technology
Noor *et al.*, [12]	✓	✓	✓	✓		Bicubic interpolation in iBuck
Yan *et al.*, [22]	✓			✓		Dynamic Resizing
Abuzaher *et al.*, [37]				✓		RGB Percentage Replacement
Hussain *et al.*, [20]				✓		Modified Quantization and Arithmetic Encoding
Louie *et al.*, [38]	✓		✓	✓		Segmented Compression and Transmission
Iqbal *et al.*, [39]				✓		Modified Entropy Encoding
Mali *et al.*, [40]				✓		Sparse RNN Smoothing and Learned Quantization
Lee *et al.*, [41]	✓		✓			Trit-Planes Algorithm
Cai *et al.*, [42]	✓				✓	CNN Based Progressive Image Compression Framework
Byju *et al.*, [13]	✓		✓			Coarse Resolution and Wavelet Features
Lu *et al.*, [43]	✓			✓		PLONQ with Nested Quantization
Abdollahi *et al.*, [44]	✓					Recursive Least Squares(RLS) Adaptive Algorithm
Our Proposed Approach	✓	✓	✓	✓	✓	**Encoding Less Bits in The First Scan**

focused in the literature. In this regard, our previous study [12] works on a framework for secured image processing in a private cloud. Following our previous study, this paper attempts to fill up the gap in the literature by using a private cloud for the purposes of faster loading and retrieving images along with managing the storage efficiently. Thus, in summary, this paper realizes the notion of first scan to enable progressive loading and manages images over a private cloud, which in combination result in faster image retrieval as well as efficient image storage. Such a

TABLE 17
Quantitative Comparison Over Improvement in Performance Achieved by Our Proposed Approach and Other Existing Research Studies Along With Corresponding Datasets Under Experimentation as Reported in Respective Studies (CR Refers to Compression Rate and BPP Refers to Bits Per Pixel)

Name	Transmission Time Efficiency	Image Quality	Storage Efficiency	User Waiting Time	Dataset
Noor *et al.*, [12]	Upto 25%	SSIM: 0.9113	By 31.75%	-	3 Datasets; [45], [46] and a Custom Dataset
Yan *et al.*, [22]	-	PSNR: 32 dB	By 41%	-	MIR Flickr Dataset [47]
Abuzaher *et al.*, [37]	-	-	By 55%	-	Not Mentioned
Hussain *et al.*, [20]	-	PSNR: 38.9 dB	CR = 6.202 : 1	-	Custom Dataset
Louie *et al.*, [38]	Upto 50% .	-	By 50%	-	Not Mentioned
Iqbal *et al.*, [39]	-	SSIM: 0.999	1 BPP	-	Air Jet Image from JPEG AI Dataset [48]
Mali *et al.*, [40]	-	SSIM: 0.8413	0.371 BPP	-	Kodak Dataset [49], Div2K [50]
Lee *et al.*, [41]	-	PSNR: 35 dB	0.75 BPP	-	Kodak Dataset (For Verification) [49], and Vimeo90k Dataset [51]
Cai *et al.*, [42]	-	PSNR: 40 dB	1.72 BPP	26% More than JPEG	Kodak Dataset [49]
Byju *et al.*, [13]	Decoding Time 127.56s	-	-	-	Big Earth Dataset [52]
Lu *et al.*, [43]	-	PSNR: 39 dB	1.5 BPP	-	JPEG AI Testset [48]
Abdollahi *et al.*, [44]	-	PSNR: 21.7 dB	CR = 76 : 1	-	Custom Dataset
Our Proposed Approach	**Upto 69%**	**SSIM: 0.952**	**Upto 27%**	**54% Less than JPEG**	**MSCOCO2015 Dataset [32] and Custom Dataset**

combination is new in the literature to the best of our knowledge as shown in Table 16 when positioned against state-of-the-art. Nonetheless, Table 17 demonstrates that our proposed approach mostly works better than all other state-of-the art approaches in terms of transmission efficiency, image quality, storage efficiency, and user waiting time in combination.

6 Conclusion and Future Work

In this paper, we investigate an important problem in the realm of cloud-related image communication and storage from the perspective of its efficient retrieval. In this regard, we point to a significant gap in the literature on efficient retrieval and storage of progressive images - especially in bandwidth-constrained cases. Accordingly, we propose an orchestration methodology through a new image scanning technique and a new lossy compression technique. We implement the proposed orchestration in a real setup over two different continents, comprising a server in Canada and a client in Bangladesh enabling a private cloud architecture. We conduct rigorous experimentation to perform both system-level and subjective evaluations over the experimental setup. The evaluation results confirm that we can achieve substantial performance improvement using our proposed orchestration. Our future work includes system-level exploration of the next-generation JPEG images to improve image storage quality further.

Acknowledgments

Md. Nazrul Huda Shanto and Joyanta Jyoti Mondal have contributed equally to this work. Any opinions, conclusions, and findings are those of the authors alone and do not necessarily reflect the views of the funding agencies.

References

[1] J. Noor and A. B. M. A. Al Islam, "iBuck: Reliable and secured image processing middleware for OpenStack swift," in *Proc. IEEE Int. Conf. Netw. Syst. Secur.*, 2017, pp. 144–149.
[2] J. Noor, H. I. Akbar, R. A. Sujon, and A. B. M. A. Al Islam, "Secure processing-aware media storage (SPMS)," in *Proc. 36th IEEE Int. Perform. Comput. Commun. Conf.*, 2017, pp. 1–8.
[3] Understanding the most popular image file types and formats. Accessed: May 15, 2021. [Online]. Available: https://www.embedded-vision.com/sites/default/files/apress/computervisionmetrics/chapter2/9781430259299_Ch02.pdf
[4] JPEG - JPEG privacy & security abstract and executive summary. Accessed: Oct. 23, 2021. [Online]. Available: https://jpeg.org/items/20150910_privacy_security_summary.html
[5] P. Mullan, C. Riess, and F. Freiling, "Forensic source identification using JPEG image headers: The case of smartphones," *Digit. Investigation*, vol. 28, pp. S68–S76, 2019.
[6] J. Miano, *Compressed Image File Formats*. Reading, MA, USA: Addison Wesley, 1999.
[7] C. Harrison, A. K. Dey, and S. Hudson, "Evaluation of progressive image loading schemes," in *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, 2010, pp. 1549–1552.
[8] Internet speeds by country 2021, 2022, Accessed: Oct. 24, 2021. [Online]. Available: https://worldpopulationreview.com/country-rankings/internet-speeds-by-country
[9] How COVID-19 is affecting internet performance. Accessed: Oct. 26, 2021. [Online]. Available: https://www.fastly.com/blog/how-covid-19-is-affecting-internet-performance
[10] Libjpeg-turbo, "libjpeg-turbo default scan script." Accessed: Sep. 28, 2020. [Online]. Available: https://github.com/libjpeg-turbo/libjpeg-turbo/blob/master/wizard.txt
[11] Sourceforge, "Libjpeg." Accessed: Apr. 20, 2020. [Online]. Available: http://libjpeg.sourceforge.net/
[12] J. Noor, S. I. Salim, and A. B. M. A. Al Islam, "Strategizing secured image storing and efficient image retrieval through a new cloud framework," *J. Netw. Comput. Appl.*, vol. 192, no. 103167, 2021, Art. no. 103167.
[13] A. P. Byju, B. Demir, and L. Bruzzone, "A progressive content-based image retrieval in JPEG 2000 compressed remote sensing archives," *IEEE Trans. Geosci. Remote Sens.*, vol. 58, no. 8, pp. 5739–5751, Aug. 2020.
[14] F. Lin, H. Ngo, and C. Dow, "A cloud-based face video retrieval system with deep learning," *J. Supercomput.*, vol. 76, pp. 8473–8493, 2020.
[15] H.-H. Hsu, T. K. Shih, L. H. Lin, R.-C. Chang, and H.-F. Li, "Adaptive image transmission by strategic decomposition," in *Proc. 18th Int. Conf. Adv. Inf. Netw. Appl.*, 2004, pp. 525–530.
[16] C.-C. Chang, H.-C. Hsia, and T.-S. Chen, "A progressive image transmission scheme based on block truncation coding," in *Proc. Int. Conf. Hum. Soc. Internet*, 2001, pp. 383–397.
[17] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing Communications & Applications*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1995.
[18] M. Hasan, K. M. Nur, and H. B. Shakur, "An improved JPEG image compression technique based on selective quantization," *Int. J. Comput. Appl.*, vol. 55, pp. 9–14, 2012.
[19] J. A.-A. Mazen Abuzaher, "JPEG based compression algorithm," *Int. J. Eng. Appl. Sci.*, vol. 4, 2017, Art. no. 257481.
[20] A. A. Hussain, G. K. AL-Khafaji, and M. M. Siddeq, "Developed JPEG algorithm applied in image compression," in *Proc. Int. Sci. Conf. Al-Ayen Univ.*, vol. 928, no. 3, 2020, p. 032006.
[21] M. Stirner and G. Seelmann, "Improved redundancy reduction for JPEG files," in *Proc. Picture Coding Symp.*, 2007, pp. 1–4.
[22] E. Yan, K. Zhang, X. Wang, K. Strauss, and L. Ceze, "Customizing progressive JPEG for efficient image storage," in *Proc. 9th USENIX Conf. Hot Top. Storage File Syst.*, 2017, Art. no. 26.
[23] Unraveling the JPEG. Accessed: Oct. 24, 2021. [Online]. Available: https://parametric.press/issue-01/unraveling-the-jpeg/
[24] L. Tan and J. Jiang, "Chapter 13 - Image processing basics," in *Digital Signal Processing, 3rd ed.*, L. Tan and J. Jiang, Eds., 3rd ed. New York, NY, USA: Academic, 2019, pp. 649–726. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128150719000130
[25] R. Coyne, "Visualising 2D discrete cosine transforms." Accessed: Mar. 01, 2021. [Online]. Available: https://richardcoyne.com/2020/08/22/visualising-2d-discrete-cosine-transforms/
[26] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Trans. Consum. Electron.*, vol. 38, no. 1, pp. 18–34, Feb. 1992.
[27] S. Kunwar, "Image compression algorithm and JPEG standard," *Int. J. Sci. Res. Pub.*, vol. 7, no. 12, Dec. 2017.
[28] J. Arnold and Members of the SwiftStack Team, *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*, Sebastopol, CA, USA: O'Reilly Media, Inc., 2014.
[29] R. Hahling, "MSU video quality measurement tools." Accessed: Mar. 28, 2020. [Online]. Available: https://github.com/Rolinh/VQMT
[30] V. Sharma, "Qutub complex monuments' images dataset." Accessed: Feb. 16, 2021. [Online]. Available: https://www.kaggle.com/varunsharmaml/qutub-complex-monuments-images-dataset
[31] M. Oltean, "Fruits 360." Accessed: Feb. 16, 2021. [Online]. Available: https://www.kaggle.com/moltean/fruits
[32] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 740–755.
[33] F. Kuipers, R. Kooij, D. D. Vleeschauwer, and K. Brunnstrom, "Techniques for measuring quality of experience," in *Proc. 8th Int. Conf. Wired/Wireless Internet Commun.*, 2010, pp. 216–227.
[34] Structural similarity index—Skimage v0.19.0.dev0 docs. Accessed: Oct. 25, 2021. [Online]. Available: https://scikit-image.org/docs/dev/auto_examples/transform/plot_ssim.html
[35] Y. Wang, W. Zhou, and P. Zhang, "Implementation and Demonstration of QoE measurement platform," in *QoE Management in Wireless Networks*, Cham, Switzerland: Springer, Aug. 2016, pp. 45–57.
[36] Chrome DevTools. Accessed: Oct. 22, 2021. [Online]. Available: https://developer.chrome.com/docs/devtools/
[37] M. Abu-Zaher and J. Al-Azzeh, "JPEG based compression algorithm," *Int. J. Eng. Appl. Sci.*, vol. 4, 2017, Art. no. 257481.
[38] A. Louie and A. M. K. Cheng, "Work-in-progress: Designing a server-side progressive JPEG encoder for real-time applications," in *Proc. IEEE Real-Time Syst. Symp.*, 2020, pp. 379–382.

[39] Y. Iqbal and O.-J. Kwon, "Improved JPEG coding by filtering 8 × 8 DCT blocks," *J. Imag.*, vol. 7, no. 7, 2021, Art. no. 117.

[40] A. Mali, A. Ororbia, D. Kifer, and L. Giles, "Neural JPEG: End-to-end image compression leveraging a standard JPEG encoder-decoder," 2022, *arXiv:2201.11795*.

[41] J. Lee, S. Jeon, K. P. Choi, Y. Park, and C. Kim, "DPICT: Deep progressive image compression using trit-planes," *CoRR*, 2021. Accessed: Jan. 14, 2022. [Online]. Available: https://arxiv.org/abs/2112.06334

[42] C. Cai, L. Chen, X. Zhang, G. Lu, and Z. Gao, "A novel deep progressive image compression framework," in *Proc. Picture Coding Symp.*, 2019, pp. 1–5.

[43] Y. Lu, Y. Zhu, Y. Yang, A. Said, and T. S. Cohen, "Progressive neural image compression with nested quantization and latent ordering," in *Proc. IEEE Int. Conf. Image Process.*, 2021, pp. 539–543.

[44] N. Abdollahi, K. Shahtalebi, and M. F. Sabahi, "High compression rate, based on the RLS adaptive algorithm in progressive image transmission," *Signal Image Video Process.*, vol. 15, no. 4, pp. 835–842, 2021.

[45] R. Tyleček and R. Šára, "Spatial pattern templates for recognition of objects with regular structure," in *Proc. German Conf. Pattern Recognit.*, 2013, pp. 364–374.

[46] F. Korč and W. Förstner, "eTRIMS image database for interpreting images of man-made scenes," Dept. Photogrammetry, Univ. Bonn, May 2009.

[47] M. J. Huiskes and M. S. Lew, "The MIR flickr retrieval evaluation," in *Proc. 1st ACM Int. Conf. Multimedia Inf. Retrieval*, 2008, pp. 39–43.

[48] J. Ascenso and P. Akayzi, "JPEG AI image coding common test conditions," in *Proc. 84th Meeting ISO/IEC JTC 1/SC 29/WG 1 N84035*, 2019.

[49] R. W. Franzen, "Kodak lossless true color image suite." Accessed: Mar. 19, 2022. [Online]. Available: http://r0k.us/graphics/kodak.

[50] E. Agustsson and R. Timofte, "NTIRE 2017 challenge on single image Super-Resolution: Dataset and study," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2017, pp. 1122–1131.

[51] T. Xue, B. Chen, J. Wu, D. Wei, and W. T. Freeman, "Video enhancement with task-oriented flow," *Int. J. Comput. Vis.*, vol. 127, no. 8, pp. 1106–1125, 2019.

[52] BigEarth - Accurate and scalable processing of Big Data in earth observation. Accessed: Mar. 19, 2022. [Online]. Available: https://bigearth.eu/datasets.html

[53] Httparchive. Accessed: Sep. 28, 2020. [Online]. Available: http://httparchive.org/

[54] M. Rabbani, "JPEG2000: Image compression fundamentals, standards and practice," *J. Electron. Imag.*, vol. 11, no. 2, 2002, Art. no. 286.

[55] F. Bellard, "BPG image format." Accessed: Mar. 20, 2022. [Online]. Available: http://bellard.org/bpg/

[56] J. Balle, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston, "Variational image compression with a scale hyperprior," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–10.

[57] Google, "WebP: Compression techniques." Accessed: Mar. 20, 2022. [Online]. Available: http://developers.google.com/speed/webp/docs/compression

[58] G. Toderici *et al.*, "Full resolution image compression with recurrent neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5435–5443.

Jannatun Noor received the BSc and MSc degrees in computer science from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh. She has been working as a lecturer with the Department of CSE, BRAC University since September 2018. Besides, she is currently working as a graduate research assistant under the supervision of Prof Dr. A. B. M. Alim Al Islam with the Department of CSE, Bangladesh University of Engineering and Technology (BUET). Prior to BRACU, she worked as a team lead of the IPV-Cloud Team, IPvision Canada Inc. Her research work covers cloud computing, wireless networking, data mining, Big Data analysis, Internet of Things, etc.

Md. Nazrul Huda Shanto is currently working toward the bachelor's degree in computer science from the School of Data and Sciences, Brac University, Dhaka, Bangladesh. He is working as a research assistant with BRAC University. His area of research interests include HCI4D, cloud computing, cybersecurity, and image processing.

Joyanta Jyoti Mondal is currently working toward the bachelor's degree in the School of Data and Sciences, Brac University, Dhaka, Bangladesh. He is working as a research assistant with BRAC University. His research interests include machine learning, deep learning, computer vision, and cloud computing.

Md. Golam Hossain received the bachelor's degree in computer science and engineering from the Chittagong University of Engineering and Technology, Chittagong, Bangladesh. He is currently a software engineer with more than five years of work experience in coding, testing, integrating Linux-based services mostly written in C, C++, and Java along with provisioning, deploying, and maintaining the reliability of production servers. His research interests include cloud computing and networking.

Sriram Chellappan received the PhD degree in computer science and engineering from Ohio-State University, Columbus, Ohio, in 2007. He is currently a professor with the Department of Computer Science and Engineering, University of South Florida, where he directs the SCoRe (Social Computing Research) Lab. His primary interests lie in many aspects of how Society and Technology interact with each other, particularly within the realms of smart health, cyber safety, and privacy. His research is supported by grants from the National Science Foundation, Department of Education, Army Research Office, National Security Agency, DARPA, and more. Prior to joining USF, he was a faculty member with the Computer Science Department, Missouri University of Science and Technology. He received the NSF CAREER Award in 2013. He also received the Missouri S&T Faculty Excellence Award in 2014, the Missouri S&T Outstanding Teaching Commendation Award in 2014, and the Missouri S&T Faculty Research Award in 2015. His areas of research interests include mobile and wireless networking, cyber-physical systems, distributed, and cloud computing.

A. B. M. Alim Al Islam received the BSc and MSc degrees in computer science and engineering from the Bangladesh University of Engineering and Technology, Bangladesh, and the PhD degree in computer science and engineering from the School of ECE, Purdue University, West Lafayette, USA, in 2012. He is currently working as professor with the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh. His areas of research interests include wireless networks, embedded systems, modeling & simulation, computer networks security, and reliability analysis.