Big Data Coursework - Questions

Data Processing and Machine Learning in the Cloud

This is the **INM432 Big Data coursework 2025**. This coursework contains extended elements of **theory** and **practice**, mainly around parallelisation of tasks withSpark and a bit about parallel training using TensorFlow.

Code and Report

Your tasks parallelization of tasks in PySpark, extension, evaluation, and theoretical reflection. Please complete and submit the **coding tasks** in a copy of **this notebook**. Write your code in the **indicated cells** and **include** the **output** in the submitted notebook.
Make sure that **your code contains comments** on its **stucture** and explanations of its **purpose**.

Provide also a **report** with the **textual answers in a separate document**.
Include **screenshots** from the Google Cloud web interface (don't use the SCREENSHOT function that Google provides, but take a picture of the graphs you see for the VMs) and result tables, as well as written text about the analysis.

Submission

Download and submit **your version of this notebook** as an **.ipynb** file and also submit a **shareable link** to your notebook on Colab in your report (created with the Colab 'Share' function) (**and don't change the online version after submission**).

Further, provide your **report as a PDF document**. **State the number of words** in the document at the end. The report should **not have more than 2000 words**.

Please also submit **a PDF of your Jupyter notebook**.

Introduction and Description

This coursework focuses on parallelisation and scalability in the cloud with Spark and TesorFlow/Keras. We start with code based on **lessons 3 and 4** of the *Fast and Lean Data Science* course by Martin Gorner. The course is based on Tensorflow for data processing and MachineLearning. Tensorflow's data processing approach is somewhat similar to that of Spark, but you don't need to study Tensorflow, just make sure you understand the high-level structure.
What we will do here is **parallelising pre-processing**, and **measuring** performance, and we will perform **evaluation** and **analysis** on the cloud performance, as well as **theoretical discussion**.

This coursework contains **3 sections**.

Section 0

This section just contains some necessary code for setting up the environment. It has no tasks for you (but do read the code and comments).

Section 1

Section 1 is about preprocessing a set of image files. We will work with a public dataset "Flowers" (3600 images, 5 classes). This is not a vast dataset, but it keeps the tasks more manageable for development and you can scale up later, if you like.

In **'Getting Started'** we will work through the data preprocessing code from *Fast and Lean Data Science* which uses TensorFlow's `tf.data` package. There is no task for you here, but you will need to re-use some of this code later.

In **Task 1** you will **parallelise the data preprocessing in Spark**, using Google Cloud (GC) Dataproc. This involves adapting the code from 'Getting Started' to use Spark and running it in the cloud.

Section 2

In **Section 2** we are going to **measure the speed of reading data** in the cloud. In **Task 2** we will **paralellize the measuring** of different configurations **using Spark**.

Section 3

This section is about the theoretical discussion, based on one paper, in **Task 3**. The answers should be given in the PDF report.

General points

For **all coding tasks**, take the **time of the operations** and for the cloud operations, get performance **information from the web interfaces** for your reporting and analysis.

The **tasks** are **mostly independent** of each other. The later tasks can mostly be addressed without needing the solution to the earlier ones.

Section 0: Set-up

As usual, you need to run the **imports and authentication every time you work with this notebook**. Use the **local Spark** installation for development before you send jobs to the cloud.

Read through this section once and **fill in the project ID the first time**, then you can just step straight throught this at the beginning of each session - except for the two authentication cells.

Imports

We import some **packages that will be needed throughout**. For the **code that runs in the cloud**, we will need **separate import sections** that will need to be partly different from the one below.

In [1]:
```
import os, sys, math
import numpy as np
import scipy as sp
import scipy.stats
import time
import datetime
import string
import random
from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pickle
```

```
Tensorflow version 2.18.0
```

Cloud and Drive authentication

This is for **authenticating with with GCS Google Drive**, so that we can create and use our own buckets and access Dataproc and AI-Platform.

This section **starts with the two interactive authentications**.

First, we mount Google Drive for persistent local storage and create a directory `DB-CW` thay you can use for this work. Then we'll set up the cloud environment, including a storage bucket.

In [2]:
```
print('Mounting google drive...')
from google.colab import drive
drive.mount('/content/drive')
%cd "/content/drive/MyDrive"
!mkdir BD-CW
%cd "/content/drive/MyDrive/BD-CW"
```

```
Mounting google drive...
Mounted at /content/drive
/content/drive/MyDrive
mkdir: cannot create directory ‘BD-CW’: File exists
/content/drive/MyDrive/BD-CW
```

Next, we authenticate with the GCS to enable access to Dataproc and AI-Platform.

In [3]:
```
import sys
if 'google.colab' in sys.modules:
    from google.colab import auth
    auth.authenticate_user()
```

It is useful to **create a new Google Cloud project** for this coursework. You can do this on the GC Console page by clicking on the entry at the top, right of the *Google Cloud Platform* and choosing *New Project*. **Copy** the **generated project ID** to the next cell. Also **enable billing** and the **Compute, Storage and Dataproc** APIs like we did during the labs.

We also specify the **default project and region**. The REGION should be `europe-west2` as it is closest to us geographically. This way we don't have to specify this information every time we access the cloud.

In [4]:
```
PROJECT = 'big-data-coursework-457812'  ### USE YOUR GOOGLE CLOUD PROJECT ID HERE. ###
!gcloud config set project $PROJECT
REGION = 'us-central1'
CLUSTER = '{}-cluster'.format(PROJECT)
!gcloud config set compute/region $REGION
!gcloud config set dataproc/region $REGION

!gcloud config list # show some information
```

```
Updated property [core/project].
Updated property [compute/region].
Updated property [dataproc/region].
[component_manager]
disable_update_check = True
[compute]
region = us-central1
[core]
account = AryAnne1269@gmail.com
project = big-data-coursework-457812
[dataproc]
region = us-central1

Your active configuration is: [default]
```

With the cell below, we **create a storage bucket** that we will use later for **global storage**. If the bucket exists you will see a "ServiceException: 409 ...", which does not cause any problems. **You must create your own bucket to have write access.**

In [5]:
```
BUCKET = 'gs://{}-storage'.format(PROJECT)
!gsutil mb $BUCKET
```

```
Creating gs://big-data-coursework-457812-storage/...
ServiceException: 409 A Cloud Storage bucket named 'big-data-coursework-457812-storage' already exists. Try anoth
er name. Bucket names must be globally unique across all Google Cloud projects, including those outside of your o
rganization.
```

The cell below just **defines some routines for displaying images** that will be **used later**. You can see the code by double-clicking, but you don't need to study this.

In [6]:
```
#@title Utility functions for image display **[RUN THIS TO ACTIVATE]** { display-mode: "form" }
def display_9_images_from_dataset(dataset):
```

```
  plt.figure(figsize=(13,13))
  subplot=331
  for i, (image, label) in enumerate(dataset):
    plt.subplot(subplot)
    plt.axis('off')
    plt.imshow(image.numpy().astype(np.uint8))
    plt.title(str(label.numpy()), fontsize=16)
    # plt.title(label.numpy().decode(), fontsize=16)
    subplot += 1
    if i==8:
      break
  plt.tight_layout()
  plt.subplots_adjust(wspace=0.1, hspace=0.1)
  plt.show()

def display_training_curves(training, validation, title, subplot):
  if subplot%10==1: # set up the subplots on the first call
    plt.subplots(figsize=(10,10), facecolor='#F0F0F0')
    plt.tight_layout()
  ax = plt.subplot(subplot)
  ax.set_facecolor('#F8F8F8')
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['train', 'valid.'])

def dataset_to_numpy_util(dataset, N):
    dataset = dataset.batch(N)
    for images, labels in dataset:
        numpy_images = images.numpy()
        numpy_labels = labels.numpy()
        break;
    return numpy_images, numpy_labels

def title_from_label_and_target(label, correct_label):
  correct = (label == correct_label)
  return "{} [{}{}{}]".format(CLASSES[label], str(correct), ', shoud be ' if not correct else '',
                              CLASSES[correct_label] if not correct else ''), correct

def display_one_flower(image, title, subplot, red=False):
    plt.subplot(subplot)
    plt.axis('off')
    plt.imshow(image)
    plt.title(title, fontsize=16, color='red' if red else 'black')
    return subplot+1

def display_9_images_with_predictions(images, predictions, labels):
  subplot=331
  plt.figure(figsize=(13,13))
  classes = np.argmax(predictions, axis=-1)
  for i, image in enumerate(images):
    title, correct = title_from_label_and_target(classes[i], labels[i])
    subplot = display_one_flower(image, title, subplot, not correct)
    if i >= 8:
      break;

  plt.tight_layout()
  plt.subplots_adjust(wspace=0.1, hspace=0.1)
  plt.show()
```

Install Spark locally for quick testing

You can use the cell below to **install Spark locally on this Colab VM** (like in the labs), to do quicker small-scale interactive testing. Using Spark in the cloud with **Dataproc is still required for the final version**.

In [7]:
```
%cd
!apt-get update -qq
!apt-get install openjdk-8-jdk-headless -qq >> /dev/null # send any output to null device
!tar -xzf "/content/drive/My Drive/Big_Data/data/spark/spark-3.5.0-bin-hadoop3.tgz" # unpack

!pip install -q findspark
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/root/spark-3.5.0-bin-hadoop3"
# os.environ["SPARK_HOME"] = "/root/spark-2.4.8-bin-hadoop2.7"

import findspark
findspark.init()
import pyspark
print(pyspark.__version__)
sc = pyspark.SparkContext.getOrCreate()
print(sc)
```

```
/root
W: Skipping acquire of configured file 'main/source/Sources' as repository 'https://r2u.stat.illinois.edu/ubuntu
jammy InRelease' does not seem to provide it (sources.list entry misspelt?)
3.5.0
<SparkContext master=local[*] appName=pyspark-shell>
```

Section 1: Data pre-processing

This section is about the **pre-processing of a dataset** for deep learning. We first look at a ready-made solution using Tensorflow and then we build a implement the same process with Spark. The tasks are about **parallelisation** and **analysis** the performance of the cloud implementations.

1.1 Getting started

In this section, we get started with the data pre-processing. The code is based on lecture 3 of the 'Fast and Lean Data Science' course.

This code is using the TensorFlow `tf.data` package, which supports map functions, similar to Spark. Your **task** will be to **re-implement the same approach in Spark**.

We start by **setting some variables for the *Flowers* dataset**.

In [11]:
```
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
TARGET_SIZE = [192, 192] # target resolution for the images
CLASSES = [b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips']
    # labels for the data
```

We **read the image files** from the public GCS bucket that contains the *Flowers* dataset. **TensorFlow** has **functions** to execute glob patterns that we use to calculate the the number of images in total and per partition (rounded up as we cannont deal with parts of images).

In [12]:
```
nb_images = len(tf.io.gfile.glob(GCS_PATTERN)) # number of images
partition_size = math.ceil(1.0 * nb_images / PARTITIONS) # images per partition (float)
print("GCS_PATTERN matches {} images, to be divided into {} partitions with up to {} images each.".format(nb_ima
```

```
GCS_PATTERN matches 3670 images, to be divided into 16 partitions with up to 230 images each.
```

Map functions

In order to read use the images for learning, they need to be **preprocessed** (decoded, resized, cropped, and potentially recompressed). Below are **map functions** for these steps. You **don't need to study** the **internals of these functions** in detail.

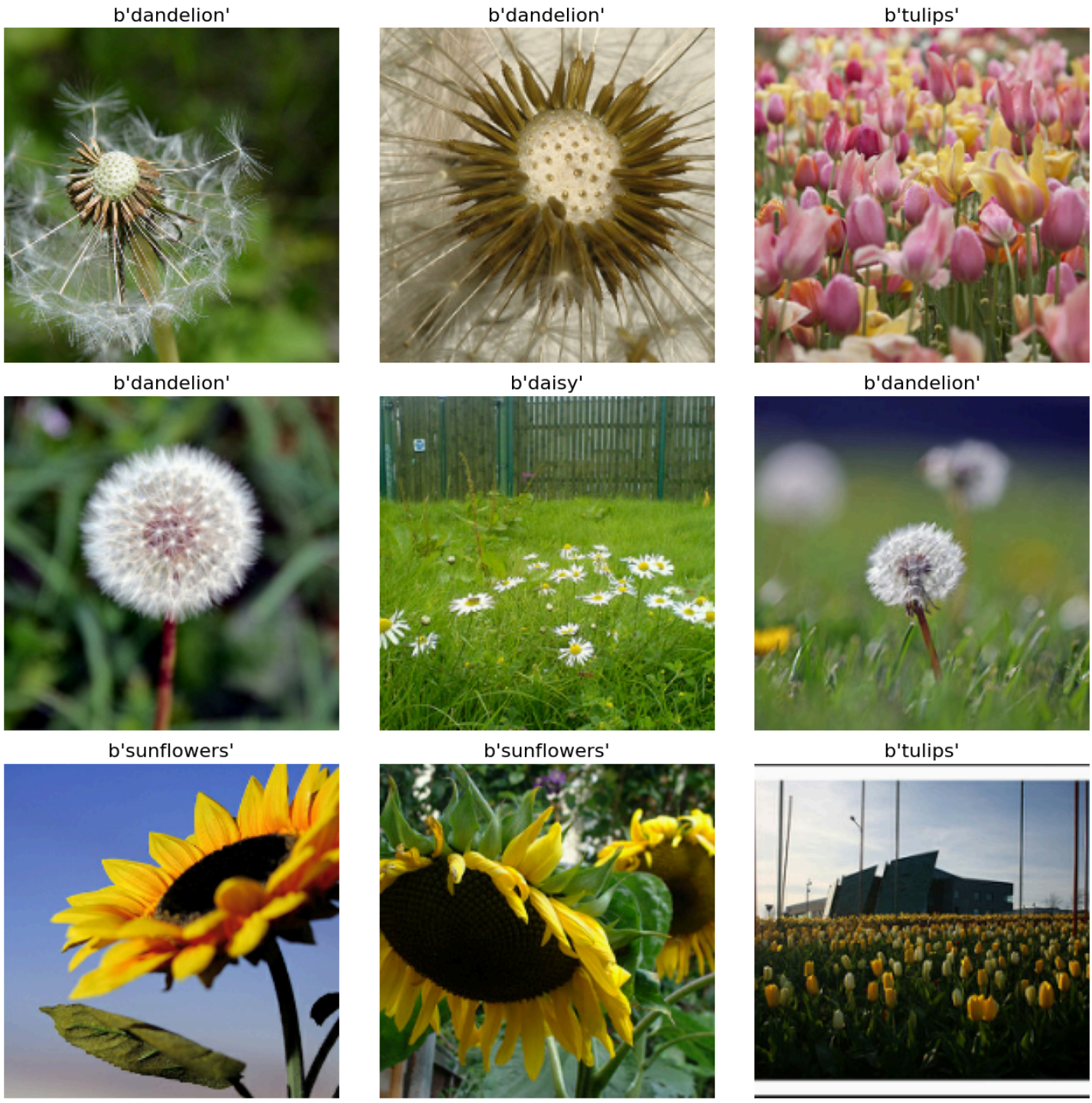
In [13]:
```
def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

def recompress_image(image, label):
    # this reduces the amount of data, but takes some time
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)
    return image, label
```

With `tf.data`, we can apply decoding and resizing as map functions.

In [14]:
```
dsetFiles = tf.data.Dataset.list_files(GCS_PATTERN) # This also shuffles the images
dsetDecoded = dsetFiles.map(decode_jpeg_and_label)
dsetResized = dsetDecoded.map(resize_and_crop_image)
```

We can also look at some images using the image display function defined above (the one with the hidden code).

In [15]:
```
display_9_images_from_dataset(dsetResized)
```



Now, let's test continuous reading from the dataset. We can see that reading the first 100 files already takes some time.

In [16]:
```
sample_set = dsetResized.batch(10).take(10) # take 10 batches of 10 images for testing
for image, label in sample_set:
    print("Image batch shape {}, {})".format(image.numpy().shape,
        [lbl.decode('utf8') for lbl in label.numpy()]))
```

```
Image batch shape (10, 192, 192, 3), ['sunflowers', 'dandelion', 'daisy', 'roses', 'dandelion', 'daisy', 'dandeli
on', 'tulips', 'tulips', 'dandelion'])
Image batch shape (10, 192, 192, 3), ['tulips', 'roses', 'sunflowers', 'dandelion', 'tulips', 'roses', 'tulips',
'dandelion', 'dandelion', 'sunflowers'])
Image batch shape (10, 192, 192, 3), ['dandelion', 'dandelion', 'dandelion', 'roses', 'tulips', 'roses', 'tulip
s', 'dandelion', 'dandelion', 'tulips'])
Image batch shape (10, 192, 192, 3), ['daisy', 'dandelion', 'roses', 'roses', 'sunflowers', 'tulips', 'sunflower
s', 'sunflowers', 'dandelion', 'roses'])
Image batch shape (10, 192, 192, 3), ['roses', 'dandelion', 'sunflowers', 'tulips', 'tulips', 'dandelion', 'dande
lion', 'dandelion', 'daisy', 'daisy'])
Image batch shape (10, 192, 192, 3), ['tulips', 'tulips', 'daisy', 'tulips', 'sunflowers', 'daisy', 'daisy', 'sun
flowers', 'sunflowers', 'sunflowers'])
Image batch shape (10, 192, 192, 3), ['roses', 'tulips', 'sunflowers', 'daisy', 'tulips', 'daisy', 'tulips', 'dan
delion', 'sunflowers', 'tulips'])
Image batch shape (10, 192, 192, 3), ['roses', 'roses', 'daisy', 'daisy', 'tulips', 'tulips', 'dandelion', 'dande
lion', 'daisy', 'daisy'])
Image batch shape (10, 192, 192, 3), ['roses', 'sunflowers', 'roses', 'sunflowers', 'tulips', 'tulips', 'sunflowe
rs', 'daisy', 'dandelion', 'daisy'])
Image batch shape (10, 192, 192, 3), ['sunflowers', 'sunflowers', 'tulips', 'daisy', 'roses', 'dandelion', 'rose
s', 'sunflowers', 'sunflowers', 'dandelion'])
```

1.2 Improving Speed

Using individual image files didn't look very fast. The 'Lean and Fast Data Science' course introduced **two techniques to improve the speed**.

Recompress the images

By **compressing** the images in the **reduced resolution** we save on the size. This **costs some CPU time** upfront, but **saves network and disk bandwith**, especially when the data are **read multiple times**.

In [17]:
```
# This is a quick test to get an idea how long recompressions takes.
dataset4 = dsetResized.map(recompress_image)
test_set = dataset4.batch(10).take(10)
for image, label in test_set:
    print("Image batch shape {}, {})".format(image.numpy().shape, [lbl.decode('utf8') for lbl in label.numpy()])
```

```
Image batch shape (10,), ['tulips', 'dandelion', 'sunflowers', 'daisy', 'sunflowers', 'roses', 'roses', 'tulips',
'tulips', 'roses'])
Image batch shape (10,), ['dandelion', 'dandelion', 'sunflowers', 'dandelion', 'tulips', 'daisy', 'dandelion', 'd
aisy', 'roses', 'tulips'])
Image batch shape (10,), ['dandelion', 'tulips', 'dandelion', 'dandelion', 'sunflowers', 'tulips', 'daisy', 'tuli
ps', 'daisy', 'sunflowers'])
Image batch shape (10,), ['dandelion', 'roses', 'dandelion', 'tulips', 'tulips', 'dandelion', 'tulips', 'roses',
'sunflowers', 'daisy'])
Image batch shape (10,), ['dandelion', 'roses', 'roses', 'daisy', 'tulips', 'tulips', 'sunflowers', 'roses', 'tul
ips', 'sunflowers'])
Image batch shape (10,), ['roses', 'dandelion', 'daisy', 'roses', 'sunflowers', 'sunflowers', 'roses', 'dandelio
n', 'dandelion', 'sunflowers'])
Image batch shape (10,), ['daisy', 'dandelion', 'sunflowers', 'sunflowers', 'daisy', 'daisy', 'dandelion', 'rose
s', 'sunflowers', 'tulips'])
Image batch shape (10,), ['dandelion', 'tulips', 'dandelion', 'roses', 'daisy', 'sunflowers', 'dandelion', 'dais
y', 'tulips', 'roses'])
Image batch shape (10,), ['dandelion', 'dandelion', 'daisy', 'roses', 'tulips', 'tulips', 'dandelion', 'dandelio
n', 'daisy', 'tulips'])
Image batch shape (10,), ['tulips', 'roses', 'daisy', 'daisy', 'roses', 'tulips', 'dandelion', 'dandelion', 'dand
elion', 'tulips'])
```

Write the dataset to TFRecord files

By writing **multiple preprocessed samples into a single file**, we can make further speed gains. We distribute the data over **partitions** to facilitate **parallelisation** when the data are used. First we need to **define a location** where we want to put the file.

In [18]:
```
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names
```

Now we can **write the TFRecord files** to the bucket.

Running the cell takes some time and **only needs to be done once** or not at all, as you can use the publicly available data for the next few cells. For convenience I have commented out the call to `write_tfrecords` at the end of the next cell. You don't need to run it (it takes some time), but you'll need to use the code below later (but there is no need to study it in detail).

There is a **ready-made pre-processed data** versions available here: `gs://cloud-samples-data/ai-platform/flowers_tfrec/tfrecords-jpeg-192x192-2/`, that we can use for testing.

In [19]:
```
# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints): # int64
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(tfrec_filewriter, img_bytes, label): # Create tf data records
    class_num = np.argmax(np.array(CLASSES)==label) # 'roses' => 2 (order defined in CLASSES)
    one_hot_class = np.eye(len(CLASSES))[class_num]     # [0, 0, 1, 0, 0] for class #2, roses
    feature = {
        "image": _bytestring_feature([img_bytes]), # one image in the list
        "class": _int_feature([class_num]) #,      # one class in the list
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

def write_tfrecords(GCS_PATTERN,GCS_OUTPUT,partition_size): # write the images to files.
    print("Writing TFRecords")
    tt0 = time.time()
    filenames = tf.data.Dataset.list_files(GCS_PATTERN)
    dataset1 = filenames.map(decode_jpeg_and_label)
    dataset2 = dataset1.map(resize_and_crop_image)
    dataset3 = dataset2.map(recompress_image)
    dataset4 = dataset3.batch(partition_size) # partitioning: there will be one "batch" of images per file
    for partition, (image, label) in enumerate(dataset4):
        # batch size used as partition size here
        partition_size = image.numpy().shape[0]
        # good practice to have the number of records in the filename
        filename = GCS_OUTPUT + "{:02d}-{}.tfrec".format(partition, partition_size)
        # You need to change GCS_OUTPUT to your own bucket to actually create new files
        with tf.io.TFRecordWriter(filename) as out_file:
            for i in range(partition_size):
                example = to_tfrecord(out_file,
                                      image.numpy()[i], # re-compressed image: already a byte string
                                      label.numpy()[i] #
                                      )
                out_file.write(example.SerializeToString())
        print("Wrote file {} containing {} records".format(filename, partition_size))
    print("Total time: "+str(time.time()-tt0))

#  write_tfrecords(GCS_PATTERN,GCS_OUTPUT,partition_size) # uncomment to run this cell
```

Test the TFRecord files

We can now **read from the TFRecord files**. By default, we use the files in the public bucket. Comment out the 1st line of the cell below to use the files written in the cell above.

In [20]:
```
GCS_OUTPUT = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/tfrecords-jpeg-192x192-2/'
# remove the line above to use your own files that you generated above

def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
```

```
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset


filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
datasetTfrec = load_dataset(filenames)
```

Let's have a look **if reading from the TFRecord** files is **quicker**.

In [21]:
```
batched_dataset = datasetTfrec.batch(10)
sample_set = batched_dataset.take(10)
for image, label in sample_set:
    print("Image batch shape {}, {})".format(image.numpy().shape, \
                        [str(lbl) for lbl in label.numpy()]))
```

```
Image batch shape (10, 192, 192, 3), ['1', '3', '3', '1', '1', '2', '4', '3', '4', '3'])
Image batch shape (10, 192, 192, 3), ['3', '0', '3', '4', '2', '2', '3', '2', '0', '3'])
Image batch shape (10, 192, 192, 3), ['4', '4', '4', '1', '3', '2', '4', '4', '4', '3'])
Image batch shape (10, 192, 192, 3), ['1', '3', '4', '1', '1', '4', '2', '2', '3', '2'])
Image batch shape (10, 192, 192, 3), ['0', '4', '3', '4', '0', '1', '2', '1', '2', '0'])
Image batch shape (10, 192, 192, 3), ['1', '1', '1', '2', '0', '0', '1', '4', '3', '1'])
Image batch shape (10, 192, 192, 3), ['1', '2', '0', '2', '3', '4', '2', '1', '1', '0'])
Image batch shape (10, 192, 192, 3), ['0', '1', '1', '3', '1', '0', '1', '3', '3', '3'])
Image batch shape (10, 192, 192, 3), ['3', '3', '3', '1', '1', '2', '0', '3', '0', '1'])
Image batch shape (10, 192, 192, 3), ['0', '0', '1', '1', '1', '0', '1', '4', '3', '2'])
```

Wow, we have a **massive speed-up**! The repackageing is worthwhile :-)

Task 1: Write TFRecord files to the cloud with Spark (40%)

Since recompressing and repackaging is very effective, we would like to be able to do it inparallel for large datasets. This is a relatively straightforward case of **parallelisation**. We will **use Spark to implement** the same process as above, but in parallel.

1a) Create the script (14%)

Re-implement the pre-processing in Spark, using Spark mechanisms for **distributing** the workload **over multiple machines**.

You need to:

i) **Copy** over the **mapping functions** (see section 1.1) and **adapt** the resizing and recompression functions **to Spark** (only one argument). (3%)

ii) **Replace** the TensorFlow **Dataset objects with RDDs**, starting with an RDD that contains the list of image filenames. (3%)

iii) **Sample** the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor of 0.02 for short tests. (1%)

iv) Then **use the functions from above** to write the TFRecord files. (3%)

v) The code for **writing to the TFRecord files** needs to be put into a function, that can be applied to every partition with the 'RDD.mapPartitionsWithIndex' function. The return value of that function is not used here, but you should return the filename, so that you have a list of the created TFRecord files. (4%)

In [ ]:
```
### CODING TASK ###

# Import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
# PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# i) Copy over the mapping functions (see section 1.1) and adapt the resizing and recompression function to Spar

# Function to decode an image file and extract the class label
def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)                    # Read file contents
    image = tf.image.decode_jpeg(bits)                   # Decode JPEG
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/').values[-2]
    return image, label

# Function to resize and crop image to the TARGET_SIZE
def resize_and_crop_image(data):
    image, label = data
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

# Function to recompress image for space savings
def recompress_image(data):
    image, label=data
    image = tf.cast(image, tf.uint8)  # Ensure image is uint8 format
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)  # Recompress
    image = image.numpy()
    return image, label
```

```
#ii) Replace the TensorFlow Dataset objects with RDDs, starting with an RDD that contains the list of image file
# retrieve a list of files that match the specified pattern.
filenames = tf.io.gfile.glob(GCS_PATTERN)
# create spark context
sc = pyspark.SparkContext.getOrCreate()
# create RDD for files
rdd_filenames = sc.parallelize(filenames)

# iii) Sample the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor
rdd_sampled = rdd_filenames.sample(False, SAMPLE_FRACTION)# Sample a small fraction of the images for quicker te

rdd_decoded = rdd_sampled.map(decode_jpeg_and_label)# Decode JPEG images and extract labels
rdd_resized = rdd_decoded.map(resize_and_crop_image)# Resize and crop images to fixed size
rdd_recompressed = rdd_resized.map(recompress_image)# Recompress images for smaller storage size


# iv) Then use the functions from above to write the TFRecord files, using an RDD as the vehicle for parallelisa
# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(img_bytes, label):
    class_num = np.argmax(CLASSES == label)  # Get class index
    one_hot_class = np.eye(len(CLASSES))[class_num]
    feature = {
        "image": _bytestring_feature([img_bytes]),   # Store image
        "class": _int_feature([class_num])            # Store class label
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

print("Writing TFRecords.")
def write_tfrecords(partition_index, iterator):
    filename = GCS_OUTPUT + f"/part-{partition_index:05d}.tfrec"  # Unique filename for each partition
    with tf.io.TFRecordWriter(filename) as writer:
        for image, label in iterator:
            example = to_tfrecord(image, label.numpy())  # Create TFRecord
            writer.write(example.SerializeToString())    # Write to file
    return [filename]

# v) The code for writing to the TFRecord files needs to be put into a function, that can be applied to every pa
# The return value of that function is not used here, but you should return the filename, so that you have a lis
rdd_final = rdd_recompressed.repartition(2)
written_files = (
    rdd_recompressed
      .mapPartitionsWithIndex(write_tfrecords)
      .collect()
)

# Final output
print(f"Finished writing {len(written_files)} TFRecord files.")
print("Files created:", written_files)
print(rdd_filenames.getNumPartitions())
```

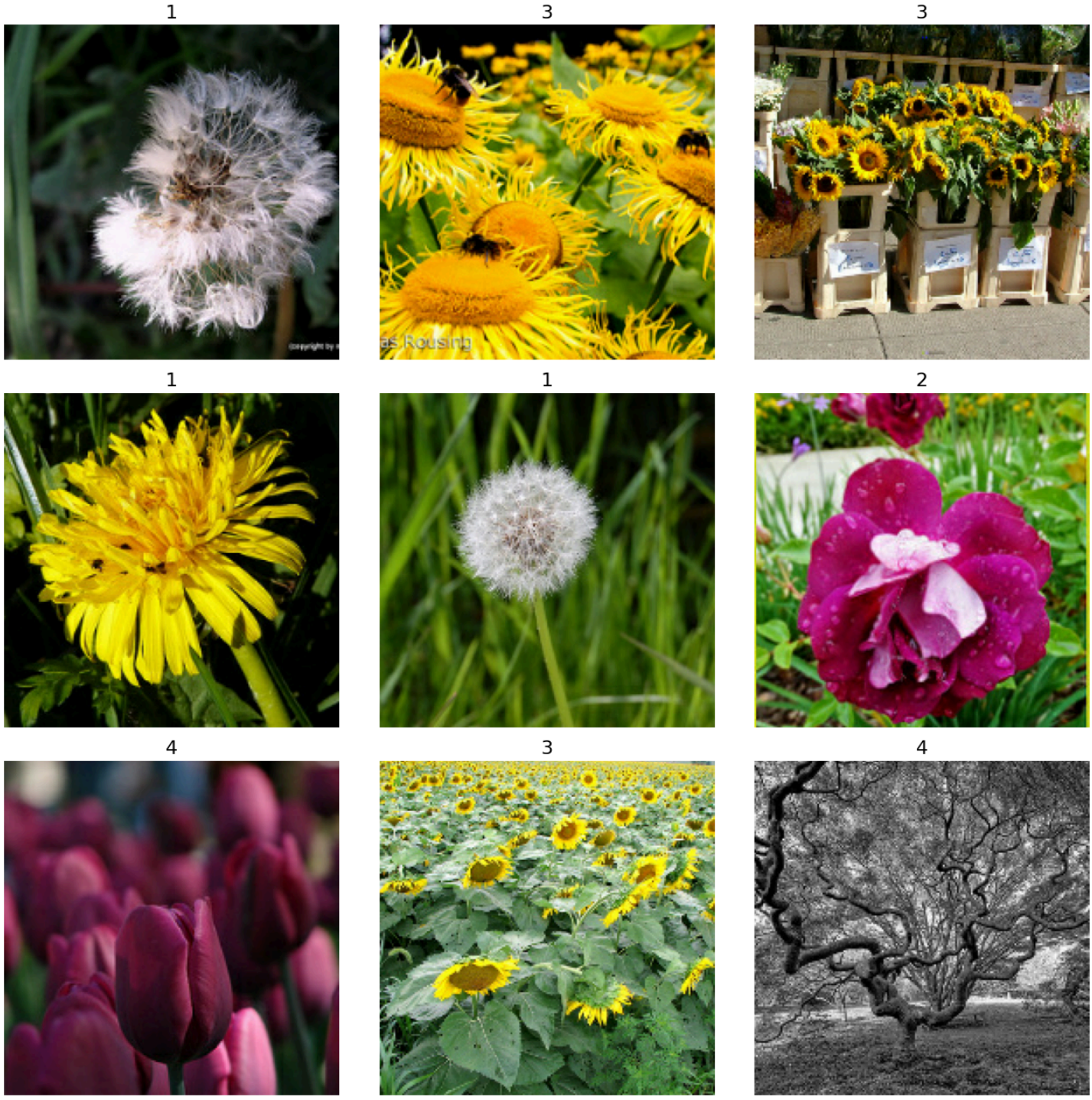
```
Tensorflow version 2.18.0
Writing TFRecords.
Finished writing 2 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec']
2
```

1b) Testing (3%)

i) Read from the TFRecord Dataset, using `load_dataset` and `display_9_images_from_dataset` to test.

In [ ]:

```
### CODING TASK ###
GCS_OUTPUT = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/tfrecords-jpeg-192x192-2/'

def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),
        "class": tf.io.FixedLenFeature([], tf.int64)
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
  option_no_order = tf.data.Options()
  option_no_order.experimental_deterministic = False

  dataset = tf.data.TFRecordDataset(filenames)
  dataset = dataset.with_options(option_no_order)
  dataset = dataset.map(read_tfrecord)
  return dataset

filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
datasetDecoded = load_dataset(filenames)
display_9_images_from_dataset(datasetDecoded)
```



ii) Write your code above into a file using the *cell magic* `%%writefile spark_write_tfrec.py` at the beginning of the file. Then, run the file locally in Spark.

In [ ]:

```
### CODING TASK ###
%%writefile spark_write_tfrec.py

# Import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pyspark
from pyspark.sql import SQLContext
```

```python
from pyspark.sql import Row

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
# PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# i) Copy over the mapping functions (see section 1.1) and adapt the resizing and recompression function to Spar

# Function to decode an image file and extract the class label
def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)                    # Read file contents
    image = tf.image.decode_jpeg(bits)                   # Decode JPEG
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/').values[-2]
    return image, label

# Function to resize and crop image to the TARGET_SIZE
def resize_and_crop_image(data):
    image, label = data
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

# Function to recompress image for space savings
def recompress_image(data):
    image, label=data
    image = tf.cast(image, tf.uint8)  # Ensure image is uint8 format
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)  # Recompress
    image = image.numpy()
    return image, label


#ii) Replace the TensorFlow Dataset objects with RDDs, starting with an RDD that contains the list of image file
# retrieve a list of files that match the specified pattern.
filenames = tf.io.gfile.glob(GCS_PATTERN)
# create spark context
sc = pyspark.SparkContext.getOrCreate()
# create RDD for files
rdd_filenames = sc.parallelize(filenames)

# iii) Sample the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor
rdd_sampled = rdd_filenames.sample(False, SAMPLE_FRACTION)# Sample a small fraction (2%) of the images for quick

rdd_decoded = rdd_sampled.map(decode_jpeg_and_label)# Decode JPEG images and extract labels
rdd_resized = rdd_decoded.map(resize_and_crop_image)# Resize and crop images to fixed size
rdd_recompressed = rdd_resized.map(recompress_image)# Recompress images for smaller storage size


# iv) Then use the functions from above to write the TFRecord files, using an RDD as the vehicle for parallelisa
# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
```

```
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(img_bytes, label):
    class_num = np.argmax(CLASSES == label)  # Get class index
    one_hot_class = np.eye(len(CLASSES))[class_num]
    feature = {
        "image": _bytestring_feature([img_bytes]),   # Store image
        "class": _int_feature([class_num])            # Store class label
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

print("Writing TFRecords.")
def write_tfrecords(partition_index, iterator):
    filename = GCS_OUTPUT + f"/part-{partition_index:05d}.tfrec"  # Unique filename for each partition
    with tf.io.TFRecordWriter(filename) as writer:
        for image, label in iterator:
            example = to_tfrecord(image, label.numpy())  # Create TFRecord
            writer.write(example.SerializeToString())    # Write to file
    return [filename]

# v) The code for writing to the TFRecord files needs to be put into a function, that can be applied to every pa
# The return value of that function is not used here, but you should return the filename, so that you have a lis
rdd_final = rdd_recompressed.repartition(2)
written_files = (
    rdd_recompressed
      .mapPartitionsWithIndex(write_tfrecords)
      .collect()
)

# Final output
print(f"Finished writing {len(written_files)} TFRecord files.")
print("Files created:", written_files)
print(rdd_filenames.getNumPartitions())
```

```
Overwriting spark_write_tfrec.py
```

1c) Set up a cluster and run the script. (6%)

Following the example from the labs, set up a cluster to run PySpark jobs in the cloud. You need to set up so that TensorFlow is installed on all nodes in the cluster.

i) Single machine cluster

Set up a cluster with a single machine using the maximal SSD size (100) and 8 vCPUs.

Enable **package installation** by passing a flag `--initialization-actions` with argument `gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh` (this is a public script that will read metadata to determine which packages to install). Then, the **packages are specified** by providing a `--metadata` flag with the argument `PIP_PACKAGES=tensorflow==2.4.0`.

Note: consider using `PIP_PACKAGES="tensorflow numpy"` or `PIP_PACKAGES=tensorflow` in case an older version of tensorflow is causing issues.

When the cluster is running, run your script to check that it works and keep the output cell output. (3%)

In [ ]:
```
### CODING TASK ###
# cluster with a single machine using the maximal SSD size (100) and 8 vCPUs.
!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --region=us-central1 \
    --image-version 1.5-ubuntu18 --single-node \
    --master-machine-type n1-standard-8 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100\
```

```
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata=PIP_PACKAGES="tensorflow==2.4.0 protobuf==3.20.3 numpy"
```

```
Waiting on operation [projects/big-data-coursework-457812/regions/us-central1/operations/f5f85e8b-4cc0-355f-afd0-9370ca15d217].

WARNING: Don't create production clusters that reference initialization actions located in the gs://goog-dataproc-initialization-actions-REGION public buckets. These scripts are provided as reference implementations, and they are synchronized with ongoing GitHub repository changes—a new version of a initialization action in public buckets may break your cluster creation. Instead, copy the following initialization actions from public buckets into your bucket : gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh
WARNING: The firewall rules for specified network or subnetwork would allow ingress traffic from 0.0.0.0/0, which could be a security risk.
WARNING: The specified custom staging bucket 'big-data-coursework-457812-storage' is not using uniform bucket level access IAM configuration. It is recommended to update bucket to enable the same. See https://cloud.google.com/storage/docs/uniform-bucket-level-access.
Created [https://dataproc.googleapis.com/v1/projects/big-data-coursework-457812/regions/us-central1/clusters/big-data-coursework-457812-cluster] Cluster placed in zone [us-central1-f].
```

Run the script in the cloud and test the output.

In [ ]:
```
### CODING TASK ###
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_write_tfrec.py
%time
```

```
Job [a96a6583c45449b0bf6374ae2f0662d1] submitted.
Waiting for job output...
2025-04-29 20:24:47.543994: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-04-29 20:24:47.544157: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/04/29 20:24:50 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/04/29 20:24:50 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/04/29 20:24:51 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/04/29 20:24:51 INFO org.spark_project.jetty.util.log: Logging initialized @7367ms to org.spark_project.jetty.u
til.log.Slf4jLog
25/04/29 20:24:51 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/04/29 20:24:51 INFO org.spark_project.jetty.server.Server: Started @7610ms
25/04/29 20:24:51 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@51f17779{HTTP/1.
1, (http/1.1)}{0.0.0.0:33723}
25/04/29 20:24:53 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.15.233:8032
25/04/29 20:24:54 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.15.233:10200
25/04/29 20:24:54 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/04/29 20:24:54 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/04/29 20:24:54 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/04/29 20:24:54 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/04/29 20:24:57 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
745953965784_0006
Writing TFRecords.
Writing TFRecords to GCS.
25/04/29 20:25:11 WARN org.apache.spark.scheduler.TaskSetManager: Stage 0 contains a task of very large size (193
KB). The maximum recommended task size is 100 KB.
Finished writing 2 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec']
2
25/04/29 20:25:22 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@51f17779{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [a96a6583c45449b0bf6374ae2f0662d1] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487c-
80fd-57ac1b5c7429/jobs/a96a6583c45449b0bf6374ae2f0662d1/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487
c-80fd-57ac1b5c7429/jobs/a96a6583c45449b0bf6374ae2f0662d1/driveroutput
jobUuid: 9e03759e-7010-33a9-b0b9-1cf5a1195bb3
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 2d8e2bba-8860-487c-80fd-57ac1b5c7429
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487c-80
fd-57ac1b5c7429/jobs/a96a6583c45449b0bf6374ae2f0662d1/staging/spark_write_tfrec.py
reference:
  jobId: a96a6583c45449b0bf6374ae2f0662d1
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-04-29T20:25:27.041651Z'
statusHistory:
- state: PENDING
  stateStartTime: '2025-04-29T20:24:42.754535Z'
- state: SETUP_DONE
  stateStartTime: '2025-04-29T20:24:42.781217Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-04-29T20:24:43.034754Z'
yarnApplications:
- name: spark_write_tfrec.py
  progress: 1.0
```

```
 state: FINISHED
 trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1745953965784_0006/
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 7.63 µs
```

In the free credit tier on Google Cloud, there are normally the following **restrictions** on compute machines:

- max 100GB of *SSD persistent disk*
- max 2000GB of *standard persistent disk*
- max 8 *vCPUs*
- no GPUs

See here for details The **disks are virtual** disks, where **I/O speed is limited in proportion to the size**, so we should allocate them evenly. This has mainly an effect on the **time the cluster needs to start**, as we are reading the data mainly from the bucket and we are not writing much to disk at all.

ii) Maximal cluster

Use the **largest possible cluster** within these constraints, i.e. **1 master and 7 worker nodes**. Each of them with 1 (virtual) CPU. The master should get the full *SSD* capacity and the 7 worker nodes should get equal shares of the *standard* disk capacity to maximise throughput.

Once the cluster is running, test your script. (3%)

In [ ]:
```
### CODING TASK ###
# cluster with a single machine using the maximal SSD size (100), 1 master with 1 vCPU + 7 workers with 1 vCPU.
import math

# calculate the size of each worker node
max_workers_disk = 2000
worker_nodes = 7
worker_disk_size = str(int(math.floor(2000/worker_nodes))) + 'GB'

!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --region=us-central1 \
    --image-version 1.5-ubuntu18 \
    --master-machine-type n1-standard-1 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
    --num-workers 7 --worker-machine-type n1-standard-1 --worker-boot-disk-size 285 \
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata PIP_PACKAGES="tensorflow==2.4.0 protobuf==3.20.3 numpy"
```

```
Waiting on operation [projects/big-data-coursework-457812/regions/us-central1/operations/e50efb36-be72-3d7f-9b2a-4cc735acff37].

WARNING: Creating clusters using the n1-standard-1 machine type is not recommended. Consider using a machine type with higher memory.
WARNING: Don't create production clusters that reference initialization actions located in the gs://goog-dataproc-initialization-actions-REGION public buckets. These scripts are provided as reference implementations, and they are synchronized with ongoing GitHub repository changes—a new version of a initialization action in public buckets may break your cluster creation. Instead, copy the following initialization actions from public buckets into your bucket : gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh
WARNING: For PD-Standard without local SSDs, we strongly recommend provisioning 1TB or larger to ensure consistently high I/O performance. See https://cloud.google.com/compute/docs/disks/performance for information on disk I/O performance.
WARNING: The firewall rules for specified network or subnetwork would allow ingress traffic from 0.0.0.0/0, which could be a security risk.
WARNING: The specified custom staging bucket 'big-data-coursework-457812-storage' is not using uniform bucket level access IAM configuration. It is recommended to update bucket to enable the same. See https://cloud.google.com/storage/docs/uniform-bucket-level-access.
Created [https://dataproc.googleapis.com/v1/projects/big-data-coursework-457812/regions/us-central1/clusters/big-data-coursework-457812-cluster] Cluster placed in zone [us-central1-f].
```

In [ ]:
```
### CODING TASK ###
# submitting spark job
```

```
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_write_tfrec.py
%time
```

```
Job [72cc03a1d6f74bf88f2028c09543844b] submitted.
Waiting for job output...
2025-04-30 10:21:26.868904: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-04-30 10:21:26.869125: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/04/30 10:21:33 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/04/30 10:21:33 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/04/30 10:21:33 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/04/30 10:21:33 INFO org.spark_project.jetty.util.log: Logging initialized @10895ms to org.spark_project.jetty.
util.log.Slf4jLog
25/04/30 10:21:33 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/04/30 10:21:33 INFO org.spark_project.jetty.server.Server: Started @11169ms
25/04/30 10:21:34 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@7d7e2198{HTTP/1.
1, (http/1.1)}{0.0.0.0:42455}
25/04/30 10:21:36 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.15.233:8032
25/04/30 10:21:36 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.15.233:10200
25/04/30 10:21:36 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/04/30 10:21:36 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/04/30 10:21:36 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/04/30 10:21:36 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/04/30 10:21:37 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Thread.join(Thread.java:1257)
        at java.lang.Thread.join(Thread.java:1331)
        at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:980)
        at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:630)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:807)
25/04/30 10:21:39 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
745953965784_0008
Writing TFRecords.
25/04/30 10:21:57 WARN org.apache.spark.scheduler.TaskSetManager: Stage 0 contains a task of very large size (193
KB). The maximum recommended task size is 100 KB.
Finished writing 2 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec']
2
25/04/30 10:22:13 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@7d7e2198{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [72cc03a1d6f74bf88f2028c09543844b] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487c-
80fd-57ac1b5c7429/jobs/72cc03a1d6f74bf88f2028c09543844b/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487
c-80fd-57ac1b5c7429/jobs/72cc03a1d6f74bf88f2028c09543844b/driveroutput
jobUuid: 82296c13-81c7-3946-a880-61e8c831e4be
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 2d8e2bba-8860-487c-80fd-57ac1b5c7429
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487c-80
fd-57ac1b5c7429/jobs/72cc03a1d6f74bf88f2028c09543844b/staging/spark_write_tfrec.py
reference:
  jobId: 72cc03a1d6f74bf88f2028c09543844b
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-04-30T10:22:17.633022Z'
statusHistory:
- state: PENDING
  stateStartTime: '2025-04-30T10:21:21.024107Z'
- state: SETUP_DONE
```

```
  stateStartTime: '2025-04-30T10:21:21.050739Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-04-30T10:21:21.412443Z'
yarnApplications:
- name: spark_write_tfrec.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1745953965784_0008/
CPU times: user 5 µs, sys: 0 ns, total: 5 µs
Wall time: 18.1 µs
```

1d) Optimisation, experiments, and discussion (17%)

i) Improve parallelisation

If you implemented a straightfoward version, you will **probably** observe that **all the computation** is done on only **two nodes**. This can be adressed by using the **second parameter** in the initial call to **parallelize**. Make the **suitable change** in the code you have written above and mark it up in comments as `### TASK 1d ###`.

Demonstrate the difference in cluster utilisation before and after the change based on different parameter values with **screenshots from Google Cloud** and measure the **difference in the processing time**. (6%)

ii) Experiment with cluster configurations.

In addition to the experiments above (using 8 VMs),test your program with 4 machines with double the resources each (2 vCPUs, memory, disk) and 1 machine with eightfold resources. Discuss the results in terms of disk I/O and network bandwidth allocation in the cloud. (7%)

iii) Explain the difference between this use of Spark and most standard applications like e.g. in our labs in terms of where the data is stored. What kind of parallelisation approach is used here? (4%)

Write the code below and your answers in the report.

In [ ]:
```python
# i) Improve parallelisation
# Import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# i) Copy over the mapping functions (see section 1.1) and adapt the resizing and recompression function to Spar
```

```python
# Function to decode an image file and extract the class label
def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)                    # Read file contents
    image = tf.image.decode_jpeg(bits)                   # Decode JPEG
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/').values[-2]
    return image, label

# Function to resize and crop image to the TARGET_SIZE
def resize_and_crop_image(data):
    image, label = data
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

# Function to recompress image for space savings
def recompress_image(data):
    image, label=data
    image = tf.cast(image, tf.uint8)  # Ensure image is uint8 format
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)  # Recompress
    image = image.numpy()
    return image, label


#ii) Replace the TensorFlow Dataset objects with RDDs, starting with an RDD that contains the list of image file
# retrieve a list of files that match the specified pattern.
filenames = tf.io.gfile.glob(GCS_PATTERN)
# create spark context
sc = pyspark.SparkContext.getOrCreate()
# create RDD for files
rdd_filenames = sc.parallelize(filenames,16)
print(">>> Num partitions =", rdd_filenames.getNumPartitions())

# iii) Sample the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor
rdd_sampled = rdd_filenames.sample(False, SAMPLE_FRACTION)# Sample a small fraction (2%) of the images for quick

rdd_decoded = rdd_sampled.map(decode_jpeg_and_label)# Decode JPEG images and extract labels
rdd_resized = rdd_decoded.map(resize_and_crop_image)# Resize and crop images to fixed size
rdd_recompressed = rdd_resized.map(recompress_image)# Recompress images for smaller storage size

# iv) Then use the functions from above to write the TFRecord files, using an RDD as the vehicle for parallelisa
# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(img_bytes, label):
    class_num = np.argmax(CLASSES == label)  # Get class index
    one_hot_class = np.eye(len(CLASSES))[class_num]
    feature = {
        "image": _bytestring_feature([img_bytes]),   # Store image
        "class": _int_feature([class_num])            # Store class label
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

print("Writing TFRecords.")
def write_tfrecords(partition_index, iterator):
```

```
    filename = GCS_OUTPUT + f"/part-{partition_index:05d}.tfrec"  # Unique filename for each partition
    with tf.io.TFRecordWriter(filename) as writer:
        for image, label in iterator:
            example = to_tfrecord(image, label.numpy())  # Create TFRecord
            writer.write(example.SerializeToString())    # Write to file
    return [filename]

# v) The code for writing to the TFRecord files needs to be put into a function, that can be applied to every pa
# The return value of that function is not used here, but you should return the filename, so that you have a lis
rdd_final = rdd_recompressed.repartition(PARTITIONS)# Repartition the data to improve parallelism during TFRecor
written_files = rdd_final.mapPartitionsWithIndex(write_tfrecords).collect()

# Final output
print(f"Finished writing {len(written_files)} TFRecord files.")
print("Files created:", written_files)
```

```
Tensorflow version 2.18.0
>>> Num partitions = 16
Writing TFRecords.
Finished writing 16 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec', 'gs://big-data-coursew
ork-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00002.tfrec', 'gs://big-data-coursework-457812-storage/t
frecords-jpeg-192x192-2/flowers/part-00003.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x19
2-2/flowers/part-00004.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-000
05.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00006.tfrec', 'gs://big
-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00007.tfrec', 'gs://big-data-coursework-457
812-storage/tfrecords-jpeg-192x192-2/flowers/part-00008.tfrec', 'gs://big-data-coursework-457812-storage/tfrecord
s-jpeg-192x192-2/flowers/part-00009.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flo
wers/part-00010.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00011.tfre
c', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00012.tfrec', 'gs://big-data-c
oursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00013.tfrec', 'gs://big-data-coursework-457812-sto
rage/tfrecords-jpeg-192x192-2/flowers/part-00014.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-
192x192-2/flowers/part-00015.tfrec']
```

In [ ]:

```
# i) Improve parallelisation
%%writefile task_1d.py

# Import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# i) Copy over the mapping functions (see section 1.1) and adapt the resizing and recompression function to Spar

# Function to decode an image file and extract the class label
def decode_jpeg_and_label(filepath):
```

```
    bits = tf.io.read_file(filepath)                    # Read file contents
    image = tf.image.decode_jpeg(bits)                   # Decode JPEG
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/').values[-2]
    return image, label

# Function to resize and crop image to the TARGET_SIZE
def resize_and_crop_image(data):
    image, label = data
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

# Function to recompress image for space savings
def recompress_image(data):
    image, label=data
    image = tf.cast(image, tf.uint8)  # Ensure image is uint8 format
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)  # Recompress
    image = image.numpy()
    return image, label


#ii) Replace the TensorFlow Dataset objects with RDDs, starting with an RDD that contains the list of image file
# retrieve a list of files that match the specified pattern.
filenames = tf.io.gfile.glob(GCS_PATTERN)
# create spark context
sc = pyspark.SparkContext.getOrCreate()
# create RDD for files
rdd_filenames = sc.parallelize(filenames,16)
print(">>> Num partitions =", rdd_filenames.getNumPartitions())

# iii) Sample the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor
rdd_sampled = rdd_filenames.sample(False, SAMPLE_FRACTION)# Sample a small fraction (2%) of the images for quick

rdd_decoded = rdd_sampled.map(decode_jpeg_and_label)# Decode JPEG images and extract labels
rdd_resized = rdd_decoded.map(resize_and_crop_image)# Resize and crop images to fixed size
rdd_recompressed = rdd_resized.map(recompress_image)# Recompress images for smaller storage size

# iv) Then use the functions from above to write the TFRecord files, using an RDD as the vehicle for parallelisa
# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(img_bytes, label):
    class_num = np.argmax(CLASSES == label)  # Get class index
    one_hot_class = np.eye(len(CLASSES))[class_num]
    feature = {
        "image": _bytestring_feature([img_bytes]),   # Store image
        "class": _int_feature([class_num])            # Store class label
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

print("Writing TFRecords.")
def write_tfrecords(partition_index, iterator):
    filename = GCS_OUTPUT + f"/part-{partition_index:05d}.tfrec"  # Unique filename for each partition
    with tf.io.TFRecordWriter(filename) as writer:
```

```
        for image, label in iterator:
            example = to_tfrecord(image, label.numpy())  # Create TFRecord
            writer.write(example.SerializeToString())    # Write to file
    return [filename]

# v) The code for writing to the TFRecord files needs to be put into a function, that can be applied to every pa
# The return value of that function is not used here, but you should return the filename, so that you have a lis
print("Writing TFRecords to GCS.")
rdd_final = rdd_recompressed.repartition(PARTITIONS)# Repartition the data to improve parallelism during TFRecor
written_files = rdd_final.mapPartitionsWithIndex(write_tfrecords).collect()

# Final output
print(f"Finished writing {len(written_files)} TFRecord files.")
print("Files created:", written_files)
```

```
Overwriting task_1d.py
```

In [ ]:

```
# submitting spark job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1d.py
%time
```

```
Job [9d7e21b3248b44979e7f6be2a684ea8e] submitted.
Waiting for job output...
2025-04-29 19:53:12.088652: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-04-29 19:53:12.088834: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/04/29 19:53:15 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/04/29 19:53:15 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/04/29 19:53:15 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/04/29 19:53:15 INFO org.spark_project.jetty.util.log: Logging initialized @7080ms to org.spark_project.jetty.u
til.log.Slf4jLog
25/04/29 19:53:15 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/04/29 19:53:16 INFO org.spark_project.jetty.server.Server: Started @7308ms
25/04/29 19:53:16 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@7d7e2198{HTTP/1.
1, (http/1.1)}{0.0.0.0:39771}
25/04/29 19:53:17 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.15.233:8032
25/04/29 19:53:18 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.15.233:10200
25/04/29 19:53:18 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/04/29 19:53:18 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/04/29 19:53:18 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/04/29 19:53:18 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/04/29 19:53:21 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
745953965784_0004
>>> Num partitions = 16
Writing TFRecords.
Writing TFRecords to GCS.
Finished writing 16 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec', 'gs://big-data-coursew
ork-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00002.tfrec', 'gs://big-data-coursework-457812-storage/t
frecords-jpeg-192x192-2/flowers/part-00003.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x19
2-2/flowers/part-00004.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-000
05.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00006.tfrec', 'gs://big
-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00007.tfrec', 'gs://big-data-coursework-457
812-storage/tfrecords-jpeg-192x192-2/flowers/part-00008.tfrec', 'gs://big-data-coursework-457812-storage/tfrecord
s-jpeg-192x192-2/flowers/part-00009.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flo
wers/part-00010.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00011.tfre
c', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00012.tfrec', 'gs://big-data-c
oursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00013.tfrec', 'gs://big-data-coursework-457812-sto
rage/tfrecords-jpeg-192x192-2/flowers/part-00014.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-
192x192-2/flowers/part-00015.tfrec']
16
25/04/29 19:53:52 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@7d7e2198{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [9d7e21b3248b44979e7f6be2a684ea8e] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487c-
80fd-57ac1b5c7429/jobs/9d7e21b3248b44979e7f6be2a684ea8e/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487
c-80fd-57ac1b5c7429/jobs/9d7e21b3248b44979e7f6be2a684ea8e/driveroutput
jobUuid: 35d84457-91cd-382a-b5d9-0b3a9feb7fd8
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 2d8e2bba-8860-487c-80fd-57ac1b5c7429
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/2d8e2bba-8860-487c-80
fd-57ac1b5c7429/jobs/9d7e21b3248b44979e7f6be2a684ea8e/staging/task_1d.py
reference:
  jobId: 9d7e21b3248b44979e7f6be2a684ea8e
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-04-29T19:53:56.277010Z'
```

```
statusHistory:
- state: PENDING
  stateStartTime: '2025-04-29T19:53:07.626695Z'
- state: SETUP_DONE
  stateStartTime: '2025-04-29T19:53:07.661777Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-04-29T19:53:07.917499Z'
yarnApplications:
- name: task_1d.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1745953965784_0004/
CPU times: user 0 ns, sys: 10 µs, total: 10 µs
Wall time: 71.3 µs
```

In [ ]:
```
# ii) Experiment with cluster configurations
# cluster with a single machine using the maximal SSD size (100) and  4 machines with double the resources each
import math
max_workers_disk = 2000
worker_nodes = 3
worker_disk_size = str(int(math.floor(max_workers_disk/worker_nodes))) + 'GB'

!gcloud dataproc clusters create $CLUSTER \
    --region=us-central1 \
    --bucket=$PROJECT-storage \
    --image-version 1.5-ubuntu18 \
    --master-machine-type n1-standard-2 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
    --num-workers 3 --worker-machine-type n1-standard-2 --worker-boot-disk-size $worker_disk_size \
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata=PIP_PACKAGES="tensorflow==2.4.0 protobuf==3.20.3 numpy"
```

```
Waiting on operation [projects/big-data-coursework-457812/regions/us-central1/operations/1a4023f7-d1b8-3acf-a8ff-d8e271ce7363].

WARNING: Don't create production clusters that reference initialization actions located in the gs://goog-dataproc-initialization-actions-REGION public buckets. These scripts are provided as reference implementations, and they are synchronized with ongoing GitHub repository changes—a new version of a initialization action in public buckets may break your cluster creation. Instead, copy the following initialization actions from public buckets into your bucket : gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh
WARNING: For PD-Standard without local SSDs, we strongly recommend provisioning 1TB or larger to ensure consistently high I/O performance. See https://cloud.google.com/compute/docs/disks/performance for information on disk I/O performance.
WARNING: The firewall rules for specified network or subnetwork would allow ingress traffic from 0.0.0.0/0, which could be a security risk.
WARNING: The specified custom staging bucket 'big-data-coursework-457812-storage' is not using uniform bucket level access IAM configuration. It is recommended to update bucket to enable the same. See https://cloud.google.com/storage/docs/uniform-bucket-level-access.
Created [https://dataproc.googleapis.com/v1/projects/big-data-coursework-457812/regions/us-central1/clusters/big-data-coursework-457812-cluster] Cluster placed in zone [us-central1-f].
```

In [ ]:
```
# submit spark job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1d.py
%time
```

```
Job [bfa0f21f50d64cccaa046a04987dc0cd] submitted.
Waiting for job output...
2025-04-30 13:13:23.035355: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-04-30 13:13:23.035416: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/04/30 13:13:26 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/04/30 13:13:26 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/04/30 13:13:26 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/04/30 13:13:26 INFO org.spark_project.jetty.util.log: Logging initialized @6683ms to org.spark_project.jetty.u
til.log.Slf4jLog
25/04/30 13:13:26 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/04/30 13:13:26 INFO org.spark_project.jetty.server.Server: Started @6841ms
25/04/30 13:13:26 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@6cf1109b{HTTP/1.
1, (http/1.1)}{0.0.0.0:40907}
25/04/30 13:13:28 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.0.18:8032
25/04/30 13:13:28 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.0.18:10200
25/04/30 13:13:28 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/04/30 13:13:28 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/04/30 13:13:28 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/04/30 13:13:28 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/04/30 13:13:31 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
746017145425_0002
>>> Num partitions = 16
Writing TFRecords.
Writing TFRecords to GCS.
Finished writing 16 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec', 'gs://big-data-coursew
ork-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00002.tfrec', 'gs://big-data-coursework-457812-storage/t
frecords-jpeg-192x192-2/flowers/part-00003.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x19
2-2/flowers/part-00004.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-000
05.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00006.tfrec', 'gs://big
-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00007.tfrec', 'gs://big-data-coursework-457
812-storage/tfrecords-jpeg-192x192-2/flowers/part-00008.tfrec', 'gs://big-data-coursework-457812-storage/tfrecord
s-jpeg-192x192-2/flowers/part-00009.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flo
wers/part-00010.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00011.tfre
c', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00012.tfrec', 'gs://big-data-c
oursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00013.tfrec', 'gs://big-data-coursework-457812-sto
rage/tfrecords-jpeg-192x192-2/flowers/part-00014.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-
192x192-2/flowers/part-00015.tfrec']
25/04/30 13:14:00 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@6cf1109b{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [bfa0f21f50d64cccaa046a04987dc0cd] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-4840-
9932-781588378450/jobs/bfa0f21f50d64cccaa046a04987dc0cd/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-484
0-9932-781588378450/jobs/bfa0f21f50d64cccaa046a04987dc0cd/driveroutput
jobUuid: 0a5dff5e-88ef-325b-8e3e-d5aa685e98aa
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 9e5c9736-be4a-4840-9932-781588378450
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-4840-99
32-781588378450/jobs/bfa0f21f50d64cccaa046a04987dc0cd/staging/task_1d.py
reference:
  jobId: bfa0f21f50d64cccaa046a04987dc0cd
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-04-30T13:14:05.394321Z'
statusHistory:
```

```
- state: PENDING
  stateStartTime: '2025-04-30T13:13:18.298233Z'
- state: SETUP_DONE
  stateStartTime: '2025-04-30T13:13:18.443952Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-04-30T13:13:18.737056Z'
yarnApplications:
- name: task_1d.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1746017145425_0002/
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 8.11 µs
```

In [ ]:
```
# delete the cluster
!gcloud dataproc clusters delete $CLUSTER -q
```

```
ERROR: (gcloud.dataproc.clusters.delete) FAILED_PRECONDITION: Cannot delete cluster 'big-data-coursework-457812-cluster' while it has other pending delete operations.
```

In [ ]:
```
# cluster with a single machine using the maximal SSD size (100) and  1 machine with eightfold resources.
!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --image-version 1.5-ubuntu18 \
    --master-machine-type n1-standard-8 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100\
    --num-workers 0\
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata=PIP_PACKAGES="tensorflow==2.4.0 protobuf==3.20.3 numpy"
```

```
Waiting on operation [projects/big-data-coursework-457812/regions/us-central1/operations/890f4d13-7da7-3b68-a0f1-a8d0a442a45e].

WARNING: Don't create production clusters that reference initialization actions located in the gs://goog-dataproc-initialization-actions-REGION public buckets. These scripts are provided as reference implementations, and they are synchronized with ongoing GitHub repository changes—a new version of a initialization action in public buckets may break your cluster creation. Instead, copy the following initialization actions from public buckets into your bucket : gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh
WARNING: The firewall rules for specified network or subnetwork would allow ingress traffic from 0.0.0.0/0, which could be a security risk.
WARNING: The specified custom staging bucket 'big-data-coursework-457812-storage' is not using uniform bucket level access IAM configuration. It is recommended to update bucket to enable the same. See https://cloud.google.com/storage/docs/uniform-bucket-level-access.
Created [https://dataproc.googleapis.com/v1/projects/big-data-coursework-457812/regions/us-central1/clusters/big-data-coursework-457812-cluster] Cluster placed in zone [us-central1-f].
```

In [ ]:
```
# submit cluster job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1d.py
```

```
Job [646465bd456042dc8e70c003b8fed9b3] submitted.
Waiting for job output...
2025-04-30 11:50:02.882541: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-04-30 11:50:02.882580: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/04/30 11:50:06 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/04/30 11:50:06 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/04/30 11:50:06 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/04/30 11:50:06 INFO org.spark_project.jetty.util.log: Logging initialized @5726ms to org.spark_project.jetty.u
til.log.Slf4jLog
25/04/30 11:50:06 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/04/30 11:50:06 INFO org.spark_project.jetty.server.Server: Started @5845ms
25/04/30 11:50:06 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@55bbd3d1{HTTP/1.
1, (http/1.1)}{0.0.0.0:38365}
25/04/30 11:50:07 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.0.17:8032
25/04/30 11:50:08 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.0.17:10200
25/04/30 11:50:08 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/04/30 11:50:08 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/04/30 11:50:08 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/04/30 11:50:08 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/04/30 11:50:10 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
746013150954_0001
>>> Num partitions = 16
Writing TFRecords.
Writing TFRecords to GCS.
Finished writing 16 TFRecord files.
Files created: ['gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00000.tfrec', 'g
s://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00001.tfrec', 'gs://big-data-coursew
ork-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00002.tfrec', 'gs://big-data-coursework-457812-storage/t
frecords-jpeg-192x192-2/flowers/part-00003.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x19
2-2/flowers/part-00004.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-000
05.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00006.tfrec', 'gs://big
-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00007.tfrec', 'gs://big-data-coursework-457
812-storage/tfrecords-jpeg-192x192-2/flowers/part-00008.tfrec', 'gs://big-data-coursework-457812-storage/tfrecord
s-jpeg-192x192-2/flowers/part-00009.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flo
wers/part-00010.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00011.tfre
c', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00012.tfrec', 'gs://big-data-c
oursework-457812-storage/tfrecords-jpeg-192x192-2/flowers/part-00013.tfrec', 'gs://big-data-coursework-457812-sto
rage/tfrecords-jpeg-192x192-2/flowers/part-00014.tfrec', 'gs://big-data-coursework-457812-storage/tfrecords-jpeg-
192x192-2/flowers/part-00015.tfrec']
25/04/30 11:50:29 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@55bbd3d1{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [646465bd456042dc8e70c003b8fed9b3] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/5ef3e82e-19a2-4ea3-
a83e-f5fb5ab36b72/jobs/646465bd456042dc8e70c003b8fed9b3/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/5ef3e82e-19a2-4ea
3-a83e-f5fb5ab36b72/jobs/646465bd456042dc8e70c003b8fed9b3/driveroutput
jobUuid: 3b1e51ee-fab2-32ec-8857-a7fcf907fbd6
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 5ef3e82e-19a2-4ea3-a83e-f5fb5ab36b72
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/5ef3e82e-19a2-4ea3-a8
3e-f5fb5ab36b72/jobs/646465bd456042dc8e70c003b8fed9b3/staging/task_1d.py
reference:
  jobId: 646465bd456042dc8e70c003b8fed9b3
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-04-30T11:50:32.678021Z'
statusHistory:
```

```
- state: PENDING
  stateStartTime: '2025-04-30T11:49:58.397686Z'
- state: SETUP_DONE
  stateStartTime: '2025-04-30T11:49:58.426450Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-04-30T11:49:58.904164Z'
yarnApplications:
- name: task_1d.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1746013150954_0001/
```

In [ ]:
```
# delete the cluster
!gcloud dataproc clusters delete $CLUSTER -q
```

```
Waiting on operation [projects/big-data-coursework-457812/regions/us-central1/operations/406037d2-38ac-3528-9ab6-98e179fee28f].
Deleted [https://dataproc.googleapis.com/v1/projects/big-data-coursework-457812/regions/us-central1/clusters/big-data-coursework-457812-cluster].
```

Section 2: Speed tests

We have seen that **reading from the pre-processed TFRecord files** is **faster** than reading individual image files and decoding on the fly. This task is about **measuring this effect** and **parallelizing the tests with PySpark**.

2.1 Speed test implementation

Here is **code for time measurement** to determine the **throughput in images per second**. It doesn't render the images but extracts and prints some basic information in order to make sure the image data are read. We write the information to the null device for longer measurements `null_file=open("/dev/null", mode='w')`. That way it will not clutter our cell output.

We use batches (`dset2 = dset1.batch(batch_size)`) and select a number of batches with (`dset3 = dset2.take(batch_number)`). Then we use the `time.time()` to take the **time measurement** and take it multiple times, reading from the same dataset to see if reading speed changes with mutiple readings.

We then **vary** the size of the batch (`batch_size`) and the number of batches (`batch_number`) and **store the results for different values**. Store also the **results for each repetition** over the same dataset (repeat 2 or 3 times).

The speed test should be combined in a **function** `time_configs()` that takes a configuration, i.e. a dataset and arrays of `batch_sizes` , `batch_numbers` , and `repetitions` (an array of integers starting from 1), as **arguments** and runs the time measurement for each combination of batch_size and batch_number for the requested number of repetitions.

In [19]:
```
# Here are some useful values for testing your code, use higher values later for actually testing throughput
batch_sizes = [2,4]
batch_numbers = [3,6]
repetitions = [1]

def time_configs(dataset, batch_sizes, batch_numbers, repetitions):
    dims = [len(batch_sizes),len(batch_numbers),len(repetitions)]
    print(dims)
    results = np.zeros(dims)
    params = np.zeros(dims + [3])
    print( results.shape )
    with open("/dev/null",mode='w') as null_file: # for printing the output without showing it
        tt = time.time() # for overall time taking
        for bsi,bs in enumerate(batch_sizes):
            for dsi, ds in enumerate(batch_numbers):
                batched_dataset = dataset.batch(bs)
                timing_set = batched_dataset.take(ds)
                for ri,rep in enumerate(repetitions):
                    print("bs: {}, ds: {}, rep: {}".format(bs,ds,rep))
                    t0 = time.time()
```

```
                    for image, label in timing_set:
                        #print("Image batch shape {}".format(image.numpy().shape),
                        print("Image batch shape {}, {})".format(image.numpy().shape,
                            [str(lbl) for lbl in label.numpy()]), null_file)
                    td = time.time() - t0 # duration for reading images
                    results[bsi,dsi,ri] = ( bs * ds) / td
                    params[bsi,dsi,ri] = [ bs, ds, rep ]
    print("total time: "+str(time.time()-tt))
    return results, params
```

Let's try this function with a **small number** of configurations of batch_sizes batch_numbers and repetions, so that we get a set of parameter combinations and corresponding reading speeds. Try reading from the image files (dataset4) and the TFRecord files (datasetTfrec).

In [20]:
```
[res,par] = time_configs(dataset4, batch_sizes, batch_numbers, repetitions)
print(res)
print(par)

print("=============")

[res,par] = time_configs(datasetTfrec, batch_sizes, batch_numbers, repetitions)
print(res)
print(par)
```

```
[2, 2, 1]
(2, 2, 1)
bs: 2, ds: 3, rep: 1
Image batch shape (2,), ["b'sunflowers'", "b'roses'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf
-8'>
Image batch shape (2,), ["b'dandelion'", "b'dandelion'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (2,), ["b'tulips'", "b'tulips'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-
8'>
bs: 2, ds: 6, rep: 1
Image batch shape (2,), ["b'sunflowers'", "b'roses'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf
-8'>
Image batch shape (2,), ["b'roses'", "b'tulips'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2,), ["b'roses'", "b'sunflowers'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf
-8'>
Image batch shape (2,), ["b'sunflowers'", "b'daisy'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf
-8'>
Image batch shape (2,), ["b'sunflowers'", "b'tulips'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='ut
f-8'>
Image batch shape (2,), ["b'roses'", "b'dandelion'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-
8'>
bs: 4, ds: 3, rep: 1
Image batch shape (4,), ["b'dandelion'", "b'daisy'", "b'daisy'", "b'dandelion'"]) <_io.TextIOWrapper name='/dev/n
ull' mode='w' encoding='utf-8'>
Image batch shape (4,), ["b'sunflowers'", "b'tulips'", "b'sunflowers'", "b'tulips'"]) <_io.TextIOWrapper name='/d
ev/null' mode='w' encoding='utf-8'>
Image batch shape (4,), ["b'dandelion'", "b'daisy'", "b'roses'", "b'daisy'"]) <_io.TextIOWrapper name='/dev/null'
mode='w' encoding='utf-8'>
bs: 4, ds: 6, rep: 1
Image batch shape (4,), ["b'dandelion'", "b'tulips'", "b'tulips'", "b'tulips'"]) <_io.TextIOWrapper name='/dev/nu
ll' mode='w' encoding='utf-8'>
Image batch shape (4,), ["b'daisy'", "b'dandelion'", "b'dandelion'", "b'tulips'"]) <_io.TextIOWrapper name='/dev/
null' mode='w' encoding='utf-8'>
Image batch shape (4,), ["b'dandelion'", "b'daisy'", "b'roses'", "b'dandelion'"]) <_io.TextIOWrapper name='/dev/n
ull' mode='w' encoding='utf-8'>
Image batch shape (4,), ["b'sunflowers'", "b'dandelion'", "b'dandelion'", "b'tulips'"]) <_io.TextIOWrapper name
='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (4,), ["b'daisy'", "b'roses'", "b'tulips'", "b'roses'"]) <_io.TextIOWrapper name='/dev/null' mo
de='w' encoding='utf-8'>
Image batch shape (4,), ["b'sunflowers'", "b'dandelion'", "b'dandelion'", "b'roses'"]) <_io.TextIOWrapper name='/
dev/null' mode='w' encoding='utf-8'>
total time: 3.1053614616394043
[[[11.73231888]
  [17.14619059]]

 [[17.89808984]
  [19.95966084]]]
[[[[2. 3. 1.]]

  [[2. 6. 1.]]]


 [[[4. 3. 1.]]

  [[4. 6. 1.]]]]
=============
[2, 2, 1]
(2, 2, 1)
bs: 2, ds: 3, rep: 1
Image batch shape (2, 192, 192, 3), ['1', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['3', '1']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['1', '2']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
bs: 2, ds: 6, rep: 1
Image batch shape (2, 192, 192, 3), ['1', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['3', '1']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['1', '2']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['4', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['4', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
Image batch shape (2, 192, 192, 3), ['3', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='utf-8'>
bs: 4, ds: 3, rep: 1
```

```
Image batch shape (4, 192, 192, 3), ['1', '3', '3', '1']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['1', '2', '4', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['4', '3', '3', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
bs: 4, ds: 6, rep: 1
Image batch shape (4, 192, 192, 3), ['1', '3', '3', '1']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['1', '2', '4', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['4', '3', '3', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['3', '4', '2', '2']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['3', '2', '0', '3']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
Image batch shape (4, 192, 192, 3), ['4', '4', '4', '1']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding
='utf-8'>
total time: 0.6160838603973389
[[[ 26.45450216]
  [ 72.23100536]]

 [[116.27723577]
  [227.19193998]]]
[[[[2. 3. 1.]]

  [[2. 6. 1.]]]


 [[[4. 3. 1.]]

  [[4. 6. 1.]]]]
```

Task 2: Parallelising the speed test with Spark in the cloud. (36%)

As an exercise in **Spark programming and optimisation** as well as **performance analysis**, we will now implement the **speed test** with multiple parameters in parallel with Spark. Runing multiple tests in parallel would **not be a useful approach on a single machine, but it can be in the cloud** (you will be asked to reason about this later).

2a) Create the script (14%)

Your task is now to **port the speed test above to Spark** for running it in the cloud in Dataproc. **Adapt the speed testing** as a Spark program that performs the same actions as above, but **with Spark RDDs in a distributed way**. The distribution should be such that **each parameter combination (except repetition)** is processed in a separate Spark task.

More specifically:

- i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations in an RDD (2%)
- ii) get a Spark context and create the dataset and run timing test for each combination in parallel (2%)
- iii) transform the resulting RDD to the structure (parameter_combination, images_per_second) and save these values in an array (2%)
- iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the result for each parameter (2%)
- v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associativity in mind when implementing the average. (3%)
- vi) write the results to a pickle file in your bucket (2%)
- vii) Write your code it into a file using the *cell magic* `%%writefile spark_job.py` (1%)

Important: The task here is not to parallelize the pre-processing, but to run multiple speed tests in parallel using Spark.

In [ ]:

```
### CODING TASK
# import required libraries
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row
from pyspark.sql import SparkSession

# read TFrecord functions
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# create function for tfr files
def time_configs_tfr(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Reload the TFRecord dataset
    filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
    dset = load_dataset(filenames)

    # Batch and take a subset
    batch = dset.batch(b_size)
    sample_set = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:  # Ignore outputs
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample_set:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations

# list of parameter combinations
batch_sizes = [2, 4, 6, 8]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]
```

```
parameter_list = []

for b_size in batch_sizes:
    for b_num in batch_numbers:
        for rep in repetitions:
            parameter_list.append([b_size, b_num, rep])

# ii) get a Spark context and create the dataset and run timing test for each combination in parallel

# create spark context for rdds
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_tfr_files = sc.parallelize(parameter_list)

ss_tfr_files = SparkSession(sc)

# obtaining a simple list of lists by flattening using flatmap
tfr_files = rdd_tfr_files.flatMap(time_configs_tfr)

# create dataframe for tfr files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

df_tfr_files = tfr_files.toDF(columns)


def resize_and_crop_image(image, label):
    # image: a Tensor, label: a Tensor
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(
        resize_crit < 1,
        lambda: tf.image.resize(image, [w * tw / w, h * tw / w]),
        lambda: tf.image.resize(image, [w * th / h, h * th / h])
    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(
        image,
        (nw - tw) // 2,
        (nh - th) // 2,
        tw,
        th
    )
    return image, label


def recompress_image(image, label):
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(
        image,
        optimize_size=True,
        chroma_downsampling=False
    )
    return image, label

# create function for image files
# load the image files
def load_dataset_decoded():
    dataset_filenames = tf.data.Dataset.list_files(GCS_PATTERN)  # List all image files
    dataset_decoded = dataset_filenames.map(decode_jpeg_and_label)  # Decode images + get labels
    dataset_resized = dataset_decoded.map(resize_and_crop_image)    # Resize + crop to 192x192
    return dataset_resized

def img_configs_new(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
```

```
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Load dataset (raw images)
    dset = load_dataset_decoded()

    # Batch and sample
    batch = dset.batch(b_size)
    sample = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# create spark context
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_img_files = sc.parallelize(parameter_list)

ss_img_files = SparkSession(sc)

# run timing tests for image files
img_files = rdd_img_files.flatMap(img_configs_new)

# create DataFrame for image files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

df_img_files = img_files.toDF(columns)
```

In [ ]:
```
### CODING TASK
# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these
# for tfr files
rdd_tfr_array = df_tfr_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_tfr_array.take(9)
```

Out[ ]:
```
[((2, 6, 0), 528.238785919691),
 ((2, 6, 0), 356.65597607726704),
 ((2, 6, 1), 780.2509495093556),
 ((2, 6, 0), 503.07500399808094),
 ((2, 6, 1), 875.1960214922882),
 ((2, 6, 2), 852.717458703939),
 ((2, 9, 0), 854.6141882025333),
 ((2, 9, 0), 900.8169908125521),
 ((2, 9, 1), 1348.4821833639953)]
```

In [ ]:
```
### CODING TASK
# for image files
rdd_img_array = df_img_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_img_array.take(9)
```

Out[ ]:
```
[((2, 6, 0), 17.645677423878556),
 ((2, 6, 0), 13.657278917656061),
 ((2, 6, 1), 12.434730426193656),
 ((2, 6, 0), 13.628696388689717),
 ((2, 6, 1), 15.270843958808603),
 ((2, 6, 2), 14.854222955973265),
 ((2, 9, 0), 19.59255769293152),
 ((2, 9, 0), 18.716197415640423),
 ((2, 9, 1), 18.493933909040603)]
```

In [ ]:
```
### CODING TASK
# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the r
# Batch size vs images per second
# tfr files
rdd_tfr_bsizes_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
tfr_bsizes_speed = rdd_tfr_bsizes_speed.collect()

# Batch number vs images per second
rdd_tfr_bnums_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
tfr_bnums_speed = rdd_tfr_bnums_speed.collect()

# Repetitions vs images per second
rdd_tfr_repetitions_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
tfr_repetitions_speed = rdd_tfr_repetitions_speed.collect()

# Dataset size vs images per second
rdd_tfr_datasetsize_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
tfr_datasetsize_speed = rdd_tfr_datasetsize_speed.collect()

# image files
rdd_img_bsizes_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
img_bsizes_speed = rdd_img_bsizes_speed.collect()

# Batch number vs images per second
rdd_img_bnums_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
img_bnums_speed = rdd_img_bnums_speed.collect()

# Repetitions vs images per second
rdd_img_repetitions_speed = df_img_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
img_repetitions_speed = rdd_img_repetitions_speed.collect()

# Dataset size vs images per second
rdd_img_datasetsize_speed = df_img_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
img_datasetsize_speed = rdd_img_datasetsize_speed.collect()
```

In [ ]:
```
### CODING TASK
# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associ
# tfr files
rdd_tfr_bsizes_avg_speed = rdd_tfr_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
tfr_bsizes_avg_speed = rdd_tfr_bsizes_avg_speed.collect()

rdd_tfr_bnums_avg_speed = rdd_tfr_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
tfr_bnums_avg_speed = rdd_tfr_bnums_avg_speed.collect()
```

```
rdd_tfr_repetitions_avg_speed = rdd_tfr_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_repetitions_avg_speed = rdd_tfr_repetitions_avg_speed.collect()

rdd_tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_avg_speed.collect()

# image files
rdd_img_bsizes_avg_speed = rdd_img_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
img_bsizes_avg_speed = rdd_img_bsizes_avg_speed.collect()

rdd_img_bnums_avg_speed = rdd_img_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
img_bnums_avg_speed = rdd_img_bnums_avg_speed.collect()

rdd_img_repetitions_avg_speed = rdd_img_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_repetitions_avg_speed = rdd_img_repetitions_avg_speed.collect()

rdd_img_datasetsize_avg_speed = rdd_img_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_datasetsize_avg_speed = rdd_img_datasetsize_avg_speed.collect()
```

In [ ]:
```
### CODING TASK
# vi) write the results to a pickle file in your bucket
import pickle
import subprocess

PROJECT = "big-data-coursework-457812"

def save_to_pickle_and_upload(object_list, filename, bucket):
    # Save a list of objects to a pickle file and upload it to a GCS bucket.

    # Save locally
    with open(filename, mode='wb') as f:
        for obj in object_list:
            pickle.dump(obj, f)

    print(f"Saved {len(object_list)} objects to local file {filename}")

    # Upload to GCS
    bucket_path = f"{bucket}/{filename}"
    proc = subprocess.run(["gsutil", "cp", filename, bucket_path], stderr=subprocess.PIPE)

    if proc.returncode == 0:
        print(f"Uploaded {filename} successfully to {bucket_path}")
    else:
        print(f"Error uploading {filename} to {bucket_path}")
        print(proc.stderr.decode())

results_list = [
    tfr_bsizes_speed,
    tfr_bnums_speed,
    tfr_repetitions_speed,
    tfr_datasetsize_speed,
    tfr_bsizes_avg_speed,
    tfr_bnums_avg_speed,
    tfr_repetitions_avg_speed,
    tfr_datasetsize_avg_speed,
    img_bsizes_speed,
    img_bnums_speed,
    img_repetitions_speed,
    img_datasetsize_speed,
    img_bsizes_avg_speed,
    img_bnums_avg_speed,
    img_repetitions_avg_speed,
    img_datasetsize_avg_speed
]

save_to_pickle_and_upload(
    object_list=results_list,
    filename="results-task2a.pkl",
    bucket=f"gs://{PROJECT}-storage"
)
```

```
Saved 16 objects to local file results-task2a.pkl
Uploaded results-task2a.pkl successfully to gs://big-data-coursework-457812-storage/results-task2a.pkl
```

In [ ]:
```
### CODING TASK
# vii) Write your code it into a file using the cell magic %%writefile spark_job.py (1%)
%%writefile spark_job_task2a.py
# import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
```

```
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row
from pyspark.sql import SparkSession
import pickle
import subprocess

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# read TFrecord functions
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# create function for tfr files
def time_configs_tfr(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Reload the TFRecord dataset
    filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
    dset = load_dataset(filenames)

    # Batch and take a subset
    batch = dset.batch(b_size)
    sample_set = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:  # Ignore outputs
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample_set:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
```

```
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations

# list of parameter combinations
batch_sizes = [2, 4, 6, 8]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []

for b_size in batch_sizes:
    for b_num in batch_numbers:
        for rep in repetitions:
            parameter_list.append([b_size, b_num, rep])

# ii) get a Spark context and create the dataset and run timing test for each combination in parallel

# create spark context for rdds
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_tfr_files = sc.parallelize(parameter_list)

ss_tfr_files = SparkSession(sc)

# obtaining a simple list of lists by flattening using flatmap
tfr_files = rdd_tfr_files.flatMap(time_configs_tfr)

# create dataframe for tfr files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

df_tfr_files = tfr_files.toDF(columns)

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits, channels=3)
    # label is the folder name, e.g. 'daisy'
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep="/").values[-2]
    return image, label

def resize_and_crop_image(image, label):
    # image: a Tensor, label: a Tensor
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(
        resize_crit < 1,
        lambda: tf.image.resize(image, [w * tw / w, h * tw / w]),
        lambda: tf.image.resize(image, [w * th / h, h * th / h])
    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(
        image,
        (nw - tw) // 2,
        (nh - th) // 2,
        tw,
        th
    )
    return image, label


def recompress_image(image, label):
```

```
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(
        image,
        optimize_size=True,
        chroma_downsampling=False
    )
    return image, label

# create function for image files
# load the image files
def load_dataset_decoded():
    dataset_filenames = tf.data.Dataset.list_files(GCS_PATTERN)  # List all image files
    dataset_decoded = dataset_filenames.map(decode_jpeg_and_label)  # Decode images + get labels
    dataset_resized = dataset_decoded.map(resize_and_crop_image)    # Resize + crop to 192x192
    return dataset_resized

def img_configs_new(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Load dataset (raw images)
    dset = load_dataset_decoded()

    # Batch and sample
    batch = dset.batch(b_size)
    sample = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# create spark context
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_img_files = sc.parallelize(parameter_list)

ss_img_files = SparkSession(sc)

# Run timing tests for image files
img_files = rdd_img_files.flatMap(img_configs_new)

# Create DataFrame for image files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

df_img_files = img_files.toDF(columns)

# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these
# for tfr files
rdd_tfr_array = df_tfr_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_tfr_array.take(9)

# for image files
```

```
rdd_img_array = df_img_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_img_array.take(9)

# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the r
# Batch size vs images per second
# tfr files
rdd_tfr_bsizes_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
tfr_bsizes_speed = rdd_tfr_bsizes_speed.collect()

# Batch number vs images per second
rdd_tfr_bnums_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
tfr_bnums_speed = rdd_tfr_bnums_speed.collect()

# Repetitions vs images per second
rdd_tfr_repetitions_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
tfr_repetitions_speed = rdd_tfr_repetitions_speed.collect()

# Dataset size vs images per second
rdd_tfr_datasetsize_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
tfr_datasetsize_speed = rdd_tfr_datasetsize_speed.collect()

# image files
rdd_img_bsizes_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
img_bsizes_speed = rdd_img_bsizes_speed.collect()

# Batch number vs images per second
rdd_img_bnums_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
img_bnums_speed = rdd_img_bnums_speed.collect()

# Repetitions vs images per second
rdd_img_repetitions_speed = df_img_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
img_repetitions_speed = rdd_img_repetitions_speed.collect()

# Dataset size vs images per second
rdd_img_datasetsize_speed = df_img_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
img_datasetsize_speed = rdd_img_datasetsize_speed.collect()

# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associ
# tfr files
rdd_tfr_bsizes_avg_speed = rdd_tfr_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
tfr_bsizes_avg_speed = rdd_tfr_bsizes_avg_speed.collect()

rdd_tfr_bnums_avg_speed = rdd_tfr_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
tfr_bnums_avg_speed = rdd_tfr_bnums_avg_speed.collect()

rdd_tfr_repetitions_avg_speed = rdd_tfr_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_repetitions_avg_speed = rdd_tfr_repetitions_avg_speed.collect()

rdd_tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_avg_speed.collect()
```

```
# image files
rdd_img_bsizes_avg_speed = rdd_img_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
img_bsizes_avg_speed = rdd_img_bsizes_avg_speed.collect()

rdd_img_bnums_avg_speed = rdd_img_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
img_bnums_avg_speed = rdd_img_bnums_avg_speed.collect()

rdd_img_repetitions_avg_speed = rdd_img_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_repetitions_avg_speed = rdd_img_repetitions_avg_speed.collect()

rdd_img_datasetsize_avg_speed = rdd_img_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_datasetsize_avg_speed = rdd_img_datasetsize_avg_speed.collect()

# vi) write the results to a pickle file in your bucket

def save_to_pickle_and_upload(object_list, filename, bucket):
    # Save a list of objects to a pickle file and upload it to a GCS bucket.

    # Save locally
    with open(filename, mode='wb') as f:
        for obj in object_list:
            pickle.dump(obj, f)

    print(f"Saved {len(object_list)} objects to local file {filename}")

    # Upload to GCS
    bucket_path = f"{bucket}/{filename}"
    proc = subprocess.run(["gsutil", "cp", filename, bucket_path], stderr=subprocess.PIPE)

    if proc.returncode == 0:
        print(f"Uploaded {filename} successfully to {bucket_path}")
    else:
        print(f"Error uploading {filename} to {bucket_path}")
        print(proc.stderr.decode())

results_list = [
    tfr_bsizes_speed,
    tfr_bnums_speed,
    tfr_repetitions_speed,
    tfr_datasetsize_speed,
    tfr_bsizes_avg_speed,
    tfr_bnums_avg_speed,
    tfr_repetitions_avg_speed,
    tfr_datasetsize_avg_speed,
    img_bsizes_speed,
    img_bnums_speed,
    img_repetitions_speed,
    img_datasetsize_speed,
    img_bsizes_avg_speed,
    img_bnums_avg_speed,
    img_repetitions_avg_speed,
    img_datasetsize_avg_speed
]

save_to_pickle_and_upload(
    object_list=results_list,
    filename="results-task2a.pkl",
    bucket=f"gs://{PROJECT}-storage"
)
```

```
Overwriting spark_job_task2a.py
```

2b) Testing the code and collecting results (4%)

i) First, test locally with `%run`.

It is useful to create a **new filename argument**, so that old results don't get overwritten.

You can for instance use `datetime.datetime.now().strftime("%y%m%d-%H%M")` to get a string with the current date and time and use that in the file name.

In [8]:
```
# new filename argument

%%writefile spark_job_task2b.py

# import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row
from pyspark.sql import SparkSession
import pickle
import subprocess

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# read TFrecord functions
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# create function for tfr files
def time_configs_tfr(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
```

```
    repetition = parameters_rdd[2]

    # Reload the TFRecord dataset
    filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
    dset = load_dataset(filenames)

    # Batch and take a subset
    batch = dset.batch(b_size)
    sample_set = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:  # Ignore outputs
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample_set:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations

# list of parameter combinations
batch_sizes = [2, 4, 6, 8]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []

for b_size in batch_sizes:
    for b_num in batch_numbers:
        for rep in repetitions:
            parameter_list.append([b_size, b_num, rep])

# ii) get a Spark context and create the dataset and run timing test for each combination in parallel

# create spark context for rdds
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_tfr_files = sc.parallelize(parameter_list)

ss_tfr_files = SparkSession(sc)

# obtaining a simple list of lists by flattening using flatmap
tfr_files = rdd_tfr_files.flatMap(time_configs_tfr)

# create dataframe for tfr files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

df_tfr_files = tfr_files.toDF(columns)

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits, channels=3)
    # label is the folder name, e.g. 'daisy'
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep="/").values[-2]
    return image, label

def resize_and_crop_image(image, label):
    # image: a Tensor, label: a Tensor
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
```

```
    image = tf.cond(
        resize_crit < 1,
        lambda: tf.image.resize(image, [w * tw / w, h * tw / w]),
        lambda: tf.image.resize(image, [w * th / h, h * th / h])
    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(
        image,
        (nw - tw) // 2,
        (nh - th) // 2,
        tw,
        th
    )
    return image, label


def recompress_image(image, label):
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(
        image,
        optimize_size=True,
        chroma_downsampling=False
    )
    return image, label

# create function for image files

# load the image files
def load_dataset_decoded():
    dataset_filenames = tf.data.Dataset.list_files(GCS_PATTERN)  # List all image files
    dataset_decoded = dataset_filenames.map(decode_jpeg_and_label)  # Decode images + get labels
    dataset_resized = dataset_decoded.map(resize_and_crop_image)    # Resize + crop to 192x192
    return dataset_resized

def img_configs_new(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Load dataset (raw images)
    dset = load_dataset_decoded()

    # Batch and sample
    batch = dset.batch(b_size)
    sample = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# create spark context
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_img_files = sc.parallelize(parameter_list)
```

```
ss_img_files = SparkSession(sc)

# Run timing tests for image files
img_files = rdd_img_files.flatMap(img_configs_new)

# Create DataFrame for image files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

df_img_files = img_files.toDF(columns)

# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these
# for tfr files
rdd_tfr_array = df_tfr_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_tfr_array.take(9)

# for image files
rdd_img_array = df_img_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_img_array.take(9)

# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the r
# Batch size vs images per second
# tfr files
rdd_tfr_bsizes_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
tfr_bsizes_speed = rdd_tfr_bsizes_speed.collect()

# Batch number vs images per second
rdd_tfr_bnums_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
tfr_bnums_speed = rdd_tfr_bnums_speed.collect()

# Repetitions vs images per second
rdd_tfr_repetitions_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
tfr_repetitions_speed = rdd_tfr_repetitions_speed.collect()

# Dataset size vs images per second
rdd_tfr_datasetsize_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
tfr_datasetsize_speed = rdd_tfr_datasetsize_speed.collect()

# image files
rdd_img_bsizes_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
img_bsizes_speed = rdd_img_bsizes_speed.collect()

# Batch number vs images per second
rdd_img_bnums_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
img_bnums_speed = rdd_img_bnums_speed.collect()

# Repetitions vs images per second
rdd_img_repetitions_speed = df_img_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
```

```
img_repetitions_speed = rdd_img_repetitions_speed.collect()

# Dataset size vs images per second
rdd_img_datasetsize_speed = df_img_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
img_datasetsize_speed = rdd_img_datasetsize_speed.collect()

# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associ
# tfr files
rdd_tfr_bsizes_avg_speed = rdd_tfr_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
tfr_bsizes_avg_speed = rdd_tfr_bsizes_avg_speed.collect()

print(">>> tfr_bsizes_avg_speed:", tfr_bsizes_avg_speed)

rdd_tfr_bnums_avg_speed = rdd_tfr_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
tfr_bnums_avg_speed = rdd_tfr_bnums_avg_speed.collect()

rdd_tfr_repetitions_avg_speed = rdd_tfr_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_repetitions_avg_speed = rdd_tfr_repetitions_avg_speed.collect()

rdd_tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_avg_speed.collect()

# image files
rdd_img_bsizes_avg_speed = rdd_img_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
img_bsizes_avg_speed = rdd_img_bsizes_avg_speed.collect()

rdd_img_bnums_avg_speed = rdd_img_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
img_bnums_avg_speed = rdd_img_bnums_avg_speed.collect()

rdd_img_repetitions_avg_speed = rdd_img_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_repetitions_avg_speed = rdd_img_repetitions_avg_speed.collect()

rdd_img_datasetsize_avg_speed = rdd_img_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_datasetsize_avg_speed = rdd_img_datasetsize_avg_speed.collect()

# vi) write the results to a pickle file in your bucket

def save_to_pickle_and_upload(object_list, filename, bucket):
    # Save a list of objects to a pickle file and upload it to a GCS bucket.

    # Save locally
    with open(filename, mode='wb') as f:
        for obj in object_list:
            pickle.dump(obj, f)

    print(f"Saved {len(object_list)} objects to local file {filename}")

    # Upload to GCS
    bucket_path = f"{bucket}/{filename}"
    proc = subprocess.run(["gsutil", "cp", filename, bucket_path], stderr=subprocess.PIPE)

    if proc.returncode == 0:
        print(f"Uploaded {filename} successfully to {bucket_path}")
    else:
        print(f"Error uploading {filename} to {bucket_path}")
        print(proc.stderr.decode())

results_list = [
    tfr_bsizes_speed,
    tfr_bnums_speed,
    tfr_repetitions_speed,
    tfr_datasetsize_speed,
    tfr_bsizes_avg_speed,
    tfr_bnums_avg_speed,
    tfr_repetitions_avg_speed,
    tfr_datasetsize_avg_speed,
    img_bsizes_speed,
    img_bnums_speed,
```

```
    img_repetitions_speed,
    img_datasetsize_speed,
    img_bsizes_avg_speed,
    img_bnums_avg_speed,
    img_repetitions_avg_speed,
    img_datasetsize_avg_speed
]

save_to_pickle_and_upload(
    object_list=results_list,
    filename="results-task2b.pkl",
    bucket=f"gs://{PROJECT}-storage"
)
```

```
Writing spark_job_task2b.py
```

In [10]:
```
### CODING TASK ###
%run ./spark_job_task2b.py
```

```
Tensorflow version 2.18.0
>>> tfr_bsizes_avg_speed: [(2, 568.1703805904534), (4, 2294.989016557173), (6, 5430.931319451093), (8, 7124.88185
0011814)]
Saved 16 objects to local file results-task2b.pkl
Uploaded results-task2b.pkl successfully to gs://big-data-coursework-457812-storage/results-task2b.pkl
```

ii) Cloud

If you have a cluster running, you can run the speed test job in the cloud.

While you run this job, switch to the Dataproc web page and take **screenshots of the CPU and network load** over time. They are displayed with some delay, so you may need to wait a little. These images will be useful in the next task. Again, don't use the SCREENSHOT function that Google provides, but just take a picture of the graphs you see for the VMs.

In [ ]:
```
# # cluster with a single machine using the maximal SSD size (100) and  4 machines with double the resources eac
# import math
# max_workers_disk = 2000
# worker_nodes = 3
# worker_disk_size = str(int(math.floor(max_workers_disk/worker_nodes))) + 'GB'

# !gcloud dataproc clusters create $CLUSTER \
#     --region=us-central1 \
#     --bucket=$PROJECT-storage \
#     --image-version 1.5-ubuntu18 \
#     --master-machine-type n1-standard-2 \
#     --master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
#     --num-workers 3 --worker-machine-type n1-standard-2 --worker-boot-disk-size $worker_disk_size \
#     --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
#     --metadata=PIP_PACKAGES="tensorflow==2.4.0 protobuf==3.20.3 numpy"
```

In [12]:
```
## CODING TASK ###
# submit the spark job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_job_task2b.py
```

```
Job [a6ff3b128d8d46909a726f062ae4a845] submitted.
Waiting for job output...
2025-05-02 10:24:41.991943: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-05-02 10:24:41.992007: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/05/02 10:24:45 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/05/02 10:24:45 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/05/02 10:24:45 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/05/02 10:24:45 INFO org.spark_project.jetty.util.log: Logging initialized @7600ms to org.spark_project.jetty.u
til.log.Slf4jLog
25/05/02 10:24:46 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/05/02 10:24:46 INFO org.spark_project.jetty.server.Server: Started @7847ms
25/05/02 10:24:46 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@6cf1109b{HTTP/1.
1, (http/1.1)}{0.0.0.0:34377}
25/05/02 10:24:48 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.0.18:8032
25/05/02 10:24:48 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.0.18:10200
25/05/02 10:24:48 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/05/02 10:24:48 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/05/02 10:24:48 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/05/02 10:24:48 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/05/02 10:24:49 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Thread.join(Thread.java:1257)
        at java.lang.Thread.join(Thread.java:1331)
        at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:980)
        at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:630)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:807)
25/05/02 10:24:51 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
746017145425_0007
>>> tfr_bsizes_avg_speed: [(2, 1335.2341763829547), (4, 2577.214204259268), (6, 3217.2268870283447), (8, 5194.108
789874017)]
Saved 16 objects to local file results-task2b.pkl
Error uploading results-task2b.pkl to gs://big-data-coursework-457812-storage/results-task2b.pkl
WARNING:  Python 3.6.x is no longer officially supported by the Google Cloud CLI
and may not function correctly. Please use Python version 3.8 and up.

If you have a compatible Python interpreter installed, you can use it by setting
the CLOUDSDK_PYTHON environment variable to point to it.

Traceback (most recent call last):
  File "/snap/google-cloud-cli/328/bin/bootstrapping/gsutil.py", line 15, in <module>
    import bootstrapping
  File "/snap/google-cloud-cli/328/bin/bootstrapping/bootstrapping.py", line 50, in <module>
    from googlecloudsdk.core.credentials import store as c_store
  File "/snap/google-cloud-cli/328/lib/googlecloudsdk/core/credentials/store.py", line 34, in <module>
    from googlecloudsdk.api_lib.auth import external_account as auth_external_account
  File "/snap/google-cloud-cli/328/lib/googlecloudsdk/api_lib/auth/external_account.py", line 24, in <module>
    from googlecloudsdk.core.credentials import creds as c_creds
  File "/snap/google-cloud-cli/328/lib/googlecloudsdk/core/credentials/creds.py", line 36, in <module>
    from google.auth import external_account as google_auth_external_account
  File "/snap/google-cloud-cli/328/lib/third_party/google/auth/external_account.py", line 32, in <module>
    from dataclasses import dataclass
ModuleNotFoundError: No module named 'dataclasses'

25/05/02 11:26:57 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@6cf1109b{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [a6ff3b128d8d46909a726f062ae4a845] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-4840-
9932-781588378450/jobs/a6ff3b128d8d46909a726f062ae4a845/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-484
```

```
0-9932-781588378450/jobs/a6ff3b128d8d46909a726f062ae4a845/driveroutput
jobUuid: 2f32eb12-87af-38ba-91d6-434f46be0629
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 9e5c9736-be4a-4840-9932-781588378450
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-4840-99
32-781588378450/jobs/a6ff3b128d8d46909a726f062ae4a845/staging/spark_job_task2b.py
reference:
  jobId: a6ff3b128d8d46909a726f062ae4a845
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-05-02T11:27:02.217340Z'
statusHistory:
- state: PENDING
  stateStartTime: '2025-05-02T10:24:36.361657Z'
- state: SETUP_DONE
  stateStartTime: '2025-05-02T10:24:36.390731Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-05-02T10:24:36.663192Z'
yarnApplications:
- name: spark_job_task2b.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1746017145425_0007/
```

2c) Improve efficiency (6%)

If you implemented a straightfoward version of 2a), you will **probably have an inefficiency** in your code.

Because we are reading multiple times from an RDD to read the values for the different parameters and their averages, caching existing results is important. Explain **where in the process caching can help**, and **add a call to** `RDD.cache()` to your code, if you haven't yet. Measure the the effect of using caching or not using it.

Make the **suitable change** in the code you have written above and mark them up in comments as `### TASK 2c ###`.

Explain in your report what the **reasons for this change** are and **demonstrate and interpret its effect**

In [29]:
```python
# import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row
from pyspark.sql import SparkSession
import pickle
import subprocess

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
```

```
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# read TFrecord functions
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# create function for tfr files
def time_configs_tfr(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Reload the TFRecord dataset
    filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
    dset = load_dataset(filenames)

    # Batch and take a subset
    batch = dset.batch(b_size)
    sample_set = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:  # Ignore outputs
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample_set:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations

# list of parameter combinations
batch_sizes = [2, 4, 6, 8]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []

for b_size in batch_sizes:
    for b_num in batch_numbers:
```

```
        for rep in repetitions:
            parameter_list.append([b_size, b_num, rep])

# ii) get a Spark context and create the dataset and run timing test for each combination in parallel

# create spark context for rdds
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_tfr_files = sc.parallelize(parameter_list)

ss_tfr_files = SparkSession(sc)

# obtaining a simple list of lists by flattening using flatmap
tfr_files = rdd_tfr_files.flatMap(time_configs_tfr)

# create dataframe for tfr files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

### TASK 2c ###
# cache TFRecord results
tfr_files.cache()

df_tfr_files = tfr_files.toDF(columns)

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits, channels=3)
    # label is the folder name, e.g. 'daisy'
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep="/").values[-2]
    return image, label

def resize_and_crop_image(image, label):
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(
        resize_crit < 1,
        lambda: tf.image.resize(image, [w * tw / w, h * tw / w]),
        lambda: tf.image.resize(image, [w * th / h, h * th / h])
    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(
        image,
        (nw - tw) // 2,
        (nh - th) // 2,
        tw,
        th
    )
    return image, label


def recompress_image(image, label):
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(
        image,
        optimize_size=True,
        chroma_downsampling=False
    )
    return image, label

# create function for image files

# load the image files
def load_dataset_decoded():
    dataset_filenames = tf.data.Dataset.list_files(GCS_PATTERN)  # List all image files
```

```
    dataset_decoded = dataset_filenames.map(decode_jpeg_and_label)  # Decode images + get labels
    dataset_resized = dataset_decoded.map(resize_and_crop_image)    # Resize + crop to 192x192
    return dataset_resized

def img_configs_new(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Load dataset (raw images)
    dset = load_dataset_decoded()

    # Batch and sample
    batch = dset.batch(b_size)
    sample = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# create spark context
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_img_files = sc.parallelize(parameter_list)

ss_img_files = SparkSession(sc)

# Run timing tests for image files
img_files = rdd_img_files.flatMap(img_configs_new)

# Create DataFrame for image files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

### TASK 2c ###
# cache TFRecord results
img_files.cache()

df_img_files = img_files.toDF(columns)

# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these
# for tfr files
rdd_tfr_array = df_tfr_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_tfr_array.take(9)

# for image files
rdd_img_array = df_img_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_img_array.take(9)

# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the r
```

```
# Batch size vs images per second
# tfr files
rdd_tfr_bsizes_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
tfr_bsizes_speed = rdd_tfr_bsizes_speed.collect()

# Batch number vs images per second
rdd_tfr_bnums_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
tfr_bnums_speed = rdd_tfr_bnums_speed.collect()

# Repetitions vs images per second
rdd_tfr_repetitions_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
tfr_repetitions_speed = rdd_tfr_repetitions_speed.collect()

# Dataset size vs images per second
rdd_tfr_datasetsize_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
tfr_datasetsize_speed = rdd_tfr_datasetsize_speed.collect()

# image files
rdd_img_bsizes_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
img_bsizes_speed = rdd_img_bsizes_speed.collect()

# Batch number vs images per second
rdd_img_bnums_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
img_bnums_speed = rdd_img_bnums_speed.collect()

# Repetitions vs images per second
rdd_img_repetitions_speed = df_img_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
img_repetitions_speed = rdd_img_repetitions_speed.collect()

# Dataset size vs images per second
rdd_img_datasetsize_speed = df_img_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
img_datasetsize_speed = rdd_img_datasetsize_speed.collect()

# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associ
# tfr files
rdd_tfr_bsizes_avg_speed = rdd_tfr_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
tfr_bsizes_avg_speed = rdd_tfr_bsizes_avg_speed.collect()

rdd_tfr_bnums_avg_speed = rdd_tfr_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
tfr_bnums_avg_speed = rdd_tfr_bnums_avg_speed.collect()

rdd_tfr_repetitions_avg_speed = rdd_tfr_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_repetitions_avg_speed = rdd_tfr_repetitions_avg_speed.collect()

rdd_tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_avg_speed.collect()

# image files
rdd_img_bsizes_avg_speed = rdd_img_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
img_bsizes_avg_speed = rdd_img_bsizes_avg_speed.collect()

rdd_img_bnums_avg_speed = rdd_img_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
img_bnums_avg_speed = rdd_img_bnums_avg_speed.collect()
```

```
rdd_img_repetitions_avg_speed = rdd_img_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_repetitions_avg_speed = rdd_img_repetitions_avg_speed.collect()

rdd_img_datasetsize_avg_speed = rdd_img_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_datasetsize_avg_speed = rdd_img_datasetsize_avg_speed.collect()

# vi) write the results to a pickle file in your bucket

def save_to_pickle_and_upload(object_list, filename, bucket):
    # Save a list of objects to a pickle file and upload it to a GCS bucket.

    # Save locally
    with open(filename, mode='wb') as f:
        for obj in object_list:
            pickle.dump(obj, f)

    print(f"Saved {len(object_list)} objects to local file {filename}")

    # Upload to GCS
    bucket_path = f"{bucket}/{filename}"
    proc = subprocess.run(["gsutil", "cp", filename, bucket_path], stderr=subprocess.PIPE)

    if proc.returncode == 0:
        print(f"Uploaded {filename} successfully to {bucket_path}")
    else:
        print(f"Error uploading {filename} to {bucket_path}")
        print(proc.stderr.decode())

results_list = [
    tfr_bsizes_speed,
    tfr_bnums_speed,
    tfr_repetitions_speed,
    tfr_datasetsize_speed,
    tfr_bsizes_avg_speed,
    tfr_bnums_avg_speed,
    tfr_repetitions_avg_speed,
    tfr_datasetsize_avg_speed,
    img_bsizes_speed,
    img_bnums_speed,
    img_repetitions_speed,
    img_datasetsize_speed,
    img_bsizes_avg_speed,
    img_bnums_avg_speed,
    img_repetitions_avg_speed,
    img_datasetsize_avg_speed
]

save_to_pickle_and_upload(
    object_list=results_list,
    filename="results-task2c.pkl",
    bucket=f"gs://{PROJECT}-storage"
)
```

```
Tensorflow version 2.18.0
Saved 16 objects to local file results-task2c.pkl
Uploaded results-task2c.pkl successfully to gs://big-data-coursework-457812-storage/results-task2c.pkl
```

In [38]:

```
%%writefile spark_job_task2c.py

# import required libraries
import os, sys, math
import numpy as np
# import scipy as sp
# import scipy.stats
import time
import datetime
import string
import random
# from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
```

```
import pyspark
from pyspark.sql import SQLContext
from pyspark.sql import Row
from pyspark.sql import SparkSession
import pickle
import subprocess

# Pattern to load all image file paths (GCS public dataset)
GCS_PATTERN = 'gs://cloud-samples-data/ai-platform/flowers_tfrec/*/*.jpg'

# Project and Storage settings
PROJECT = 'big-data-coursework-457812'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'
PARTITIONS = 16
SAMPLE_FRACTION = 0.02 # for iii)
TARGET_SIZE = (192, 192)
# List of class names (labels)
CLASSES = np.array([b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips'])

# read TFrecord functions
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# create function for tfr files
def time_configs_tfr(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Reload the TFRecord dataset
    filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
    dset = load_dataset(filenames)

    # Batch and take a subset
    batch = dset.batch(b_size)
    sample_set = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:  # Ignore outputs
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample_set:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
```

```
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations

# list of parameter combinations
batch_sizes = [2, 4, 6, 8]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []

for b_size in batch_sizes:
    for b_num in batch_numbers:
        for rep in repetitions:
            parameter_list.append([b_size, b_num, rep])

# ii) get a Spark context and create the dataset and run timing test for each combination in parallel

# create spark context for rdds
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_tfr_files = sc.parallelize(parameter_list)

ss_tfr_files = SparkSession(sc)

# obtaining a simple list of lists by flattening using flatmap
tfr_files = rdd_tfr_files.flatMap(time_configs_tfr)

# create dataframe for tfr files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

### TASK 2c ###
# cache TFRecord results
tfr_files.cache()

df_tfr_files = tfr_files.toDF(columns)

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits, channels=3)
    # label is the folder name, e.g. 'daisy'
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep="/").values[-2]
    return image, label

def resize_and_crop_image(image, label):
    # image: a Tensor, label: a Tensor
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw, th = TARGET_SIZE[1], TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)

    image = tf.cond(
        resize_crit < 1,
        lambda: tf.image.resize(image, [w * tw / w, h * tw / w]),
        lambda: tf.image.resize(image, [w * th / h, h * th / h])
    )

    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(
        image,
        (nw - tw) // 2,
        (nh - th) // 2,
        tw,
        th
    )
```

```
    return image, label


def recompress_image(image, label):
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(
        image,
        optimize_size=True,
        chroma_downsampling=False
    )
    return image, label

# create function for image files

# load the image files
def load_dataset_decoded():
    dataset_filenames = tf.data.Dataset.list_files(GCS_PATTERN)  # List all image files
    dataset_decoded = dataset_filenames.map(decode_jpeg_and_label)  # Decode images + get labels
    dataset_resized = dataset_decoded.map(resize_and_crop_image)    # Resize + crop to 192x192
    return dataset_resized

def img_configs_new(parameters_rdd):
    # parameters_rdd = [batch_size, batch_number, repetition]
    b_size = parameters_rdd[0]
    b_num = parameters_rdd[1]
    repetition = parameters_rdd[2]

    # Load dataset (raw images)
    dset = load_dataset_decoded()

    # Batch and sample
    batch = dset.batch(b_size)
    sample = batch.take(b_num)

    results = []

    with open("/dev/null", mode='w') as null_file:
        for rep in range(repetition):
            s_time = time.time()
            for image, label in sample:
                print("Image batch shape {}, {})".format(image.numpy().shape, [str(lbl) for lbl in label.numpy()
            e_time = time.time()
            elapsed = e_time - s_time
            throughput = (b_size * b_num) / elapsed
            datasetsize = b_size * b_num
            results.append([b_size, b_num, rep, datasetsize, elapsed, throughput])

    return results

# create spark context
sc = pyspark.SparkContext.getOrCreate()

# parallelize parameter list
rdd_img_files = sc.parallelize(parameter_list)

ss_img_files = SparkSession(sc)

# Run timing tests for image files
img_files = rdd_img_files.flatMap(img_configs_new)

# Create DataFrame for image files
columns = ["b_sizes", "b_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

### TASK 2c ###
# cache TFRecord results
img_files.cache()

df_img_files = img_files.toDF(columns)

# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these
```

```python
# for tfr files
rdd_tfr_array = df_tfr_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_tfr_array.take(9)

# for image files
rdd_img_array = df_img_files.rdd.map(lambda row: (
    (int(row['b_sizes']), int(row['b_nums']), int(row['repetitions'])),
    float(row['throughput'])
))

rdd_img_array.take(9)

# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the r
# Batch size vs images per second
# tfr files
rdd_tfr_bsizes_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
tfr_bsizes_speed = rdd_tfr_bsizes_speed.collect()

# Batch number vs images per second
rdd_tfr_bnums_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
tfr_bnums_speed = rdd_tfr_bnums_speed.collect()

# Repetitions vs images per second
rdd_tfr_repetitions_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
tfr_repetitions_speed = rdd_tfr_repetitions_speed.collect()

# Dataset size vs images per second
rdd_tfr_datasetsize_speed = df_tfr_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
tfr_datasetsize_speed = rdd_tfr_datasetsize_speed.collect()

# image files
rdd_img_bsizes_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_sizes']), float(row['throughput'])
))
img_bsizes_speed = rdd_img_bsizes_speed.collect()

# Batch number vs images per second
rdd_img_bnums_speed = df_img_files.rdd.map(lambda row: (
    int(row['b_nums']), float(row['throughput'])
))
img_bnums_speed = rdd_img_bnums_speed.collect()

# Repetitions vs images per second
rdd_img_repetitions_speed = df_img_files.rdd.map(lambda row: (
    int(row['repetitions']), float(row['throughput'])
))
img_repetitions_speed = rdd_img_repetitions_speed.collect()

# Dataset size vs images per second
rdd_img_datasetsize_speed = df_img_files.rdd.map(lambda row: (
    int(row['datasetsize']), float(row['throughput'])
))
img_datasetsize_speed = rdd_img_datasetsize_speed.collect()

# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associ
# tfr files
rdd_tfr_bsizes_avg_speed = rdd_tfr_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
tfr_bsizes_avg_speed = rdd_tfr_bsizes_avg_speed.collect()
```

```
rdd_tfr_bnums_avg_speed = rdd_tfr_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
tfr_bnums_avg_speed = rdd_tfr_bnums_avg_speed.collect()

rdd_tfr_repetitions_avg_speed = rdd_tfr_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_repetitions_avg_speed = rdd_tfr_repetitions_avg_speed.collect()

rdd_tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
tfr_datasetsize_avg_speed = rdd_tfr_datasetsize_avg_speed.collect()

# image files
rdd_img_bsizes_avg_speed = rdd_img_bsizes_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0],
img_bsizes_avg_speed = rdd_img_bsizes_avg_speed.collect()

rdd_img_bnums_avg_speed = rdd_img_bnums_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a[0]+b[0], a[
img_bnums_avg_speed = rdd_img_bnums_avg_speed.collect()

rdd_img_repetitions_avg_speed = rdd_img_repetitions_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_repetitions_avg_speed = rdd_img_repetitions_avg_speed.collect()

rdd_img_datasetsize_avg_speed = rdd_img_datasetsize_speed.mapValues(lambda z: (z, 1)).reduceByKey(lambda a,b: (a
img_datasetsize_avg_speed = rdd_img_datasetsize_avg_speed.collect()

# vi) write the results to a pickle file in your bucket

def save_to_pickle_and_upload(object_list, filename, bucket):
    # Save a list of objects to a pickle file and upload it to a GCS bucket.

    # Save locally
    with open(filename, mode='wb') as f:
        for obj in object_list:
            pickle.dump(obj, f)

    print(f"Saved {len(object_list)} objects to local file {filename}")

    # Upload to GCS
    bucket_path = f"{bucket}/{filename}"
    proc = subprocess.run(["gsutil", "cp", filename, bucket_path], stderr=subprocess.PIPE)

    if proc.returncode == 0:
        print(f"Uploaded {filename} successfully to {bucket_path}")
    else:
        print(f"Error uploading {filename} to {bucket_path}")
        print(proc.stderr.decode())

results_list = [
    tfr_bsizes_speed,
    tfr_bnums_speed,
    tfr_repetitions_speed,
    tfr_datasetsize_speed,
    tfr_bsizes_avg_speed,
    tfr_bnums_avg_speed,
    tfr_repetitions_avg_speed,
    tfr_datasetsize_avg_speed,
    img_bsizes_speed,
    img_bnums_speed,
    img_repetitions_speed,
    img_datasetsize_speed,
    img_bsizes_avg_speed,
    img_bnums_avg_speed,
    img_repetitions_avg_speed,
    img_datasetsize_avg_speed
]

save_to_pickle_and_upload(
    object_list=results_list,
    filename="results-task2c.pkl",
    bucket=f"gs://{PROJECT}-storage"
)
```

```
Writing spark_job_task2c.py
```

In [37]:
```
!ls
```

```
results-task2c.pkl       spark_job2c.py
spark-3.5.0-bin-hadoop3  spark_job_task2b.py
```

In [39]:
```
### CODING TASK ###
%run ./spark_job_task2c.py
```

```
Tensorflow version 2.18.0
Saved 16 objects to local file results-task2c.pkl
Uploaded results-task2c.pkl successfully to gs://big-data-coursework-457812-storage/results-task2c.pkl
<Figure size 640x480 with 0 Axes>
```

In [40]:
```
## CODING TASK ###
# submit the spark job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_job_task2c.py
```

```
Job [1645da81298b4fde996b6df02a377eb8] submitted.
Waiting for job output...
2025-04-30 20:12:59.106821: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynami
c library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or direct
ory; LD_LIBRARY_PATH: :/usr/lib/hadoop/lib/native
2025-04-30 20:12:59.106866: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if y
ou do not have a GPU set up on your machine.
Tensorflow version 2.4.0
25/04/30 20:13:02 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
25/04/30 20:13:02 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
25/04/30 20:13:02 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
25/04/30 20:13:02 INFO org.spark_project.jetty.util.log: Logging initialized @6344ms to org.spark_project.jetty.u
til.log.Slf4jLog
25/04/30 20:13:02 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown;
jvm 1.8.0_382-b05
25/04/30 20:13:02 INFO org.spark_project.jetty.server.Server: Started @6496ms
25/04/30 20:13:02 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@59e779be{HTTP/1.
1, (http/1.1)}{0.0.0.0:40729}
25/04/30 20:13:04 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-coursewor
k-457812-cluster-m/10.128.0.18:8032
25/04/30 20:13:04 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-da
ta-coursework-457812-cluster-m/10.128.0.18:10200
25/04/30 20:13:04 INFO org.apache.hadoop.conf.Configuration: resource-types.xml not found
25/04/30 20:13:04 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Unable to find 'resource-types.xml'.
25/04/30 20:13:04 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = memory-m
b, units = Mi, type = COUNTABLE
25/04/30 20:13:04 INFO org.apache.hadoop.yarn.util.resource.ResourceUtils: Adding resource type - name = vcores,
units = , type = COUNTABLE
25/04/30 20:13:05 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Thread.join(Thread.java:1257)
        at java.lang.Thread.join(Thread.java:1331)
        at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:980)
        at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:630)
        at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:807)
25/04/30 20:13:07 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1
746017145425_0006
Saved 16 objects to local file results-task2c.pkl
Uploaded results-task2c.pkl successfully to gs://big-data-coursework-457812-storage/results-task2c.pkl
25/04/30 20:25:19 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@59e779be{HTTP/1.1, (http/
1.1)}{0.0.0.0:0}
Job [1645da81298b4fde996b6df02a377eb8] finished successfully.
done: true
driverControlFilesUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-4840-
9932-781588378450/jobs/1645da81298b4fde996b6df02a377eb8/
driverOutputResourceUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-484
0-9932-781588378450/jobs/1645da81298b4fde996b6df02a377eb8/driveroutput
jobUuid: c3df8a80-083c-3df2-b984-b087961d37e1
placement:
  clusterName: big-data-coursework-457812-cluster
  clusterUuid: 9e5c9736-be4a-4840-9932-781588378450
pysparkJob:
  mainPythonFileUri: gs://big-data-coursework-457812-storage/google-cloud-dataproc-metainfo/9e5c9736-be4a-4840-99
32-781588378450/jobs/1645da81298b4fde996b6df02a377eb8/staging/spark_job_task2c.py
reference:
  jobId: 1645da81298b4fde996b6df02a377eb8
  projectId: big-data-coursework-457812
status:
  state: DONE
  stateStartTime: '2025-04-30T20:25:24.965760Z'
statusHistory:
- state: PENDING
  stateStartTime: '2025-04-30T20:12:54.809511Z'
- state: SETUP_DONE
  stateStartTime: '2025-04-30T20:12:54.838310Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2025-04-30T20:12:55.092382Z'
yarnApplications:
```

```
- name: spark_job_task2c.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-coursework-457812-cluster-m:8088/proxy/application_1746017145425_0006/
```

2d) Retrieve, analyse and discuss the output (12%)

Run the tests over a wide range of different paramters and list the results in a table.

Perform a **linear regression** (e.g. using scikit-learn) over **the values for each parameter** and for the **two cases** (reading from image files/reading TFRecord files). List a **table** with the output and interpret the results in terms of the effects of overall. Also, **plot** the output values, the averages per parameter value and the regression lines for each parameter and for the product of batch_size and batch_number

Discuss the **implications** of this result for **applications** like large-scale machine learning. Keep in mind that cloud data may be stored in distant physical locations. Use the numbers provided in the PDF latency-numbers document available on Moodle or here for your arguments.

How is the **observed** behaviour **similar or different** from what you'd expect from a **single machine**? Why would cloud providers tie throughput to capacity of disk resources?

By **parallelising** the speed test we are making **assumptions** about the limits of the bucket reading speeds. See here for more information. Discuss, **what we need to consider** in **speed tests** in parallel on the cloud, which bottlenecks we might be identifying, and how this relates to your results.

Discuss to what extent **linear modelling** reflects the **effects** we are observing. Discuss what could be expected from a theoretical perspective and what can be useful in practice.

Write your **code below** and **include the output** in your submitted `ipynb` file. Provide the answer **text in your report**.

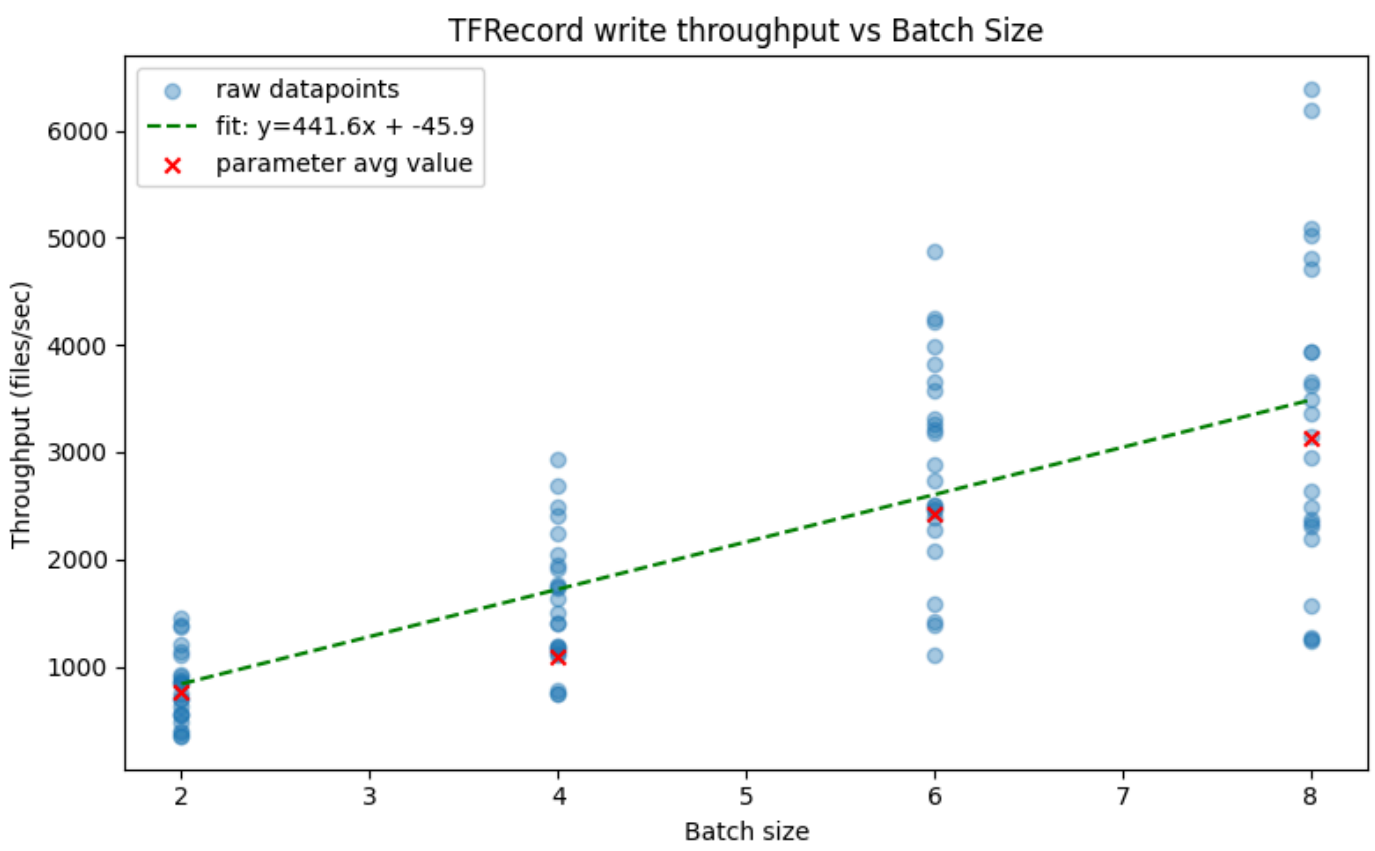
In [22]:
```
# pull down the pickle
!gsutil cp $BUCKET/results-task2b.pkl .

# load the pickle
import pickle

with open("results-task2b.pkl", "rb") as f:
    tfr_bsizes_speed      = pickle.load(f)
    tfr_bnums_speed       = pickle.load(f)
    tfr_repetitions_speed = pickle.load(f)
    tfr_datasetsize_speed = pickle.load(f)
    tfr_bsizes_avg_speed  = pickle.load(f)
    tfr_bnums_avg_speed   = pickle.load(f)
    tfr_repetitions_avg_speed = pickle.load(f)
    tfr_datasetsize_avg_speed = pickle.load(f)
    img_bsizes_speed        = pickle.load(f)
    img_bnums_speed        = pickle.load(f)
    img_repetitions_speed  = pickle.load(f)
    img_datasetsize_speed  = pickle.load(f)
    img_bsizes_avg_speed   = pickle.load(f)
    img_bnums_avg_speed    = pickle.load(f)
    img_repetitions_avg_speed = pickle.load(f)
    img_datasetsize_avg_speed = pickle.load(f)
```

```
Copying gs://big-data-coursework-457812-storage/results-task2b.pkl...
/ [1 files][ 10.6 KiB/ 10.6 KiB]
Operation completed over 1 objects/10.6 KiB.
```

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress
```

In [26]:
```
# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
```

```
    tfr_bsizes_speed,
    columns=["bsizes", "reading_speed"]
)
df_avg = pd.DataFrame(
    tfr_bsizes_avg_speed,
    columns=["bsizes", "reading_speed"]
)

# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["bsizes"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["bsizes"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("bsizes")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["bsizes"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
    x_fit, y_fit,
    linestyle="--",
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["bsizes"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("TFRecord write throughput vs Batch Size")
plt.xlabel("Batch size")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```

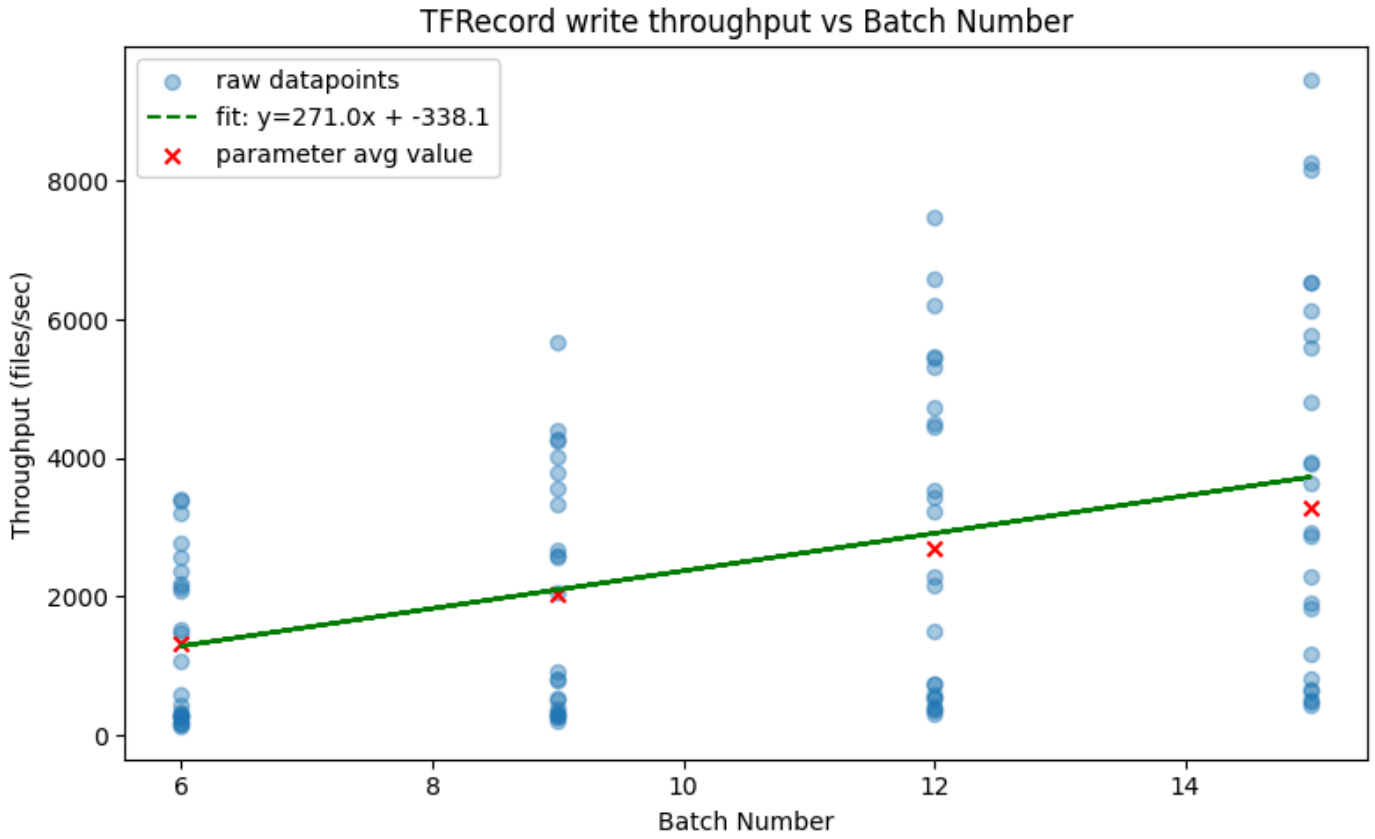



```
Coefficient (slope): 441.58
Intercept          : -45.89
P-value            : 1.22e-16
```

In [28]:

```python
# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    tfr_bnums_speed,
    columns=["bnums", "reading_speed"]
)
df_avg = pd.DataFrame(
    tfr_bnums_avg_speed,
    columns=["bnums", "reading_speed"]
)

# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["bnums"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["bnums"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("bnums")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["bnums"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
    x_fit, y_fit,
    linestyle="--",
```

```
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["bnums"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("TFRecord write throughput vs Batch Number")
plt.xlabel("Batch Number")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```

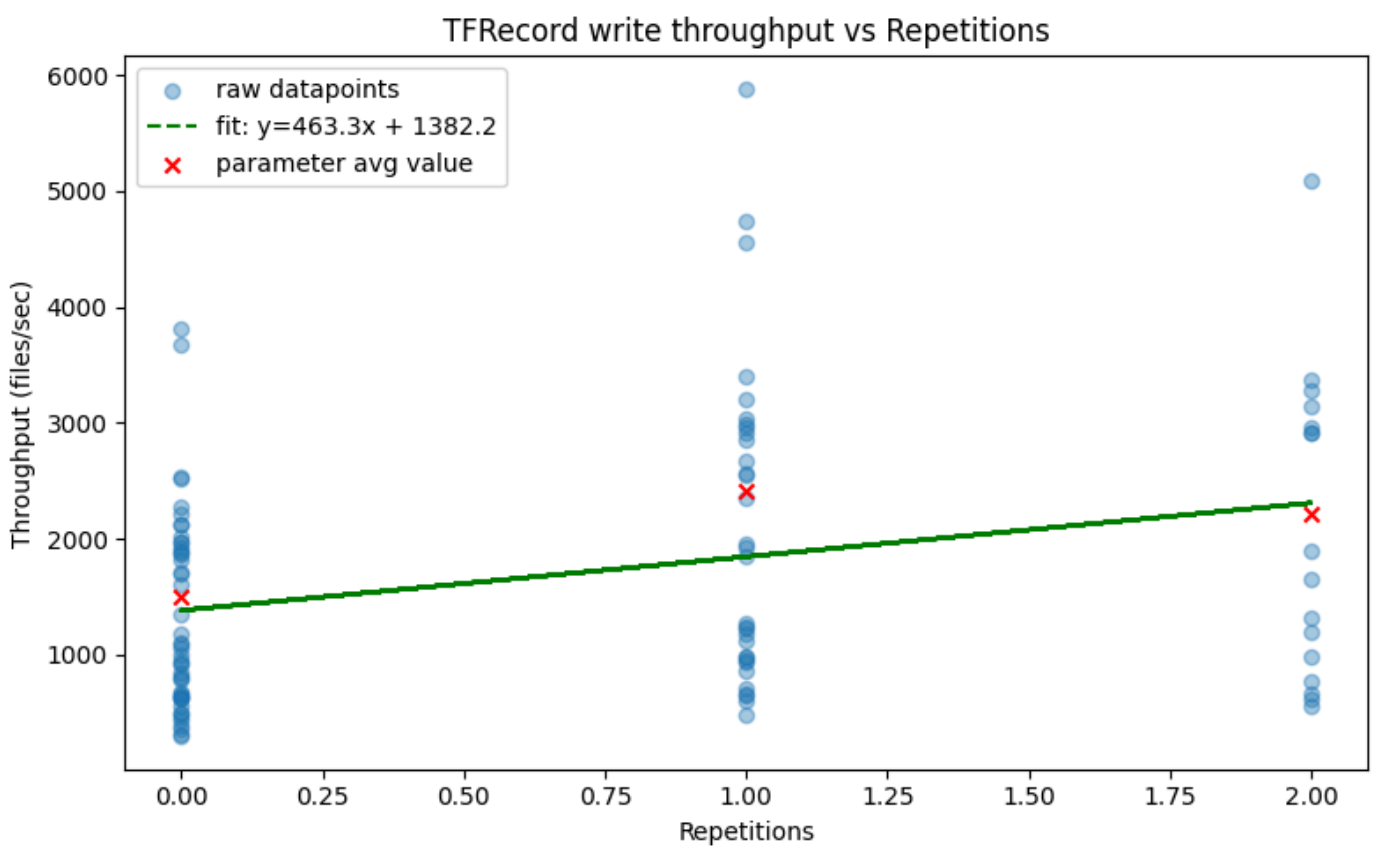



```
Coefficient (slope): 271.05
Intercept          : -338.11
P-value            : 4.81e-05
```

In [30]:

```
### CODING TASK ###

# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    tfr_repetitions_speed,
    columns=["repetitions", "reading_speed"]
)
df_avg = pd.DataFrame(
    tfr_repetitions_avg_speed,
    columns=["repetitions", "reading_speed"]
)
```

```python
# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["repetitions"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["repetitions"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("repetitions")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["repetitions"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
    x_fit, y_fit,
    linestyle="--",
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["repetitions"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("TFRecord write throughput vs Repetitions")
plt.xlabel("Repetitions")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```

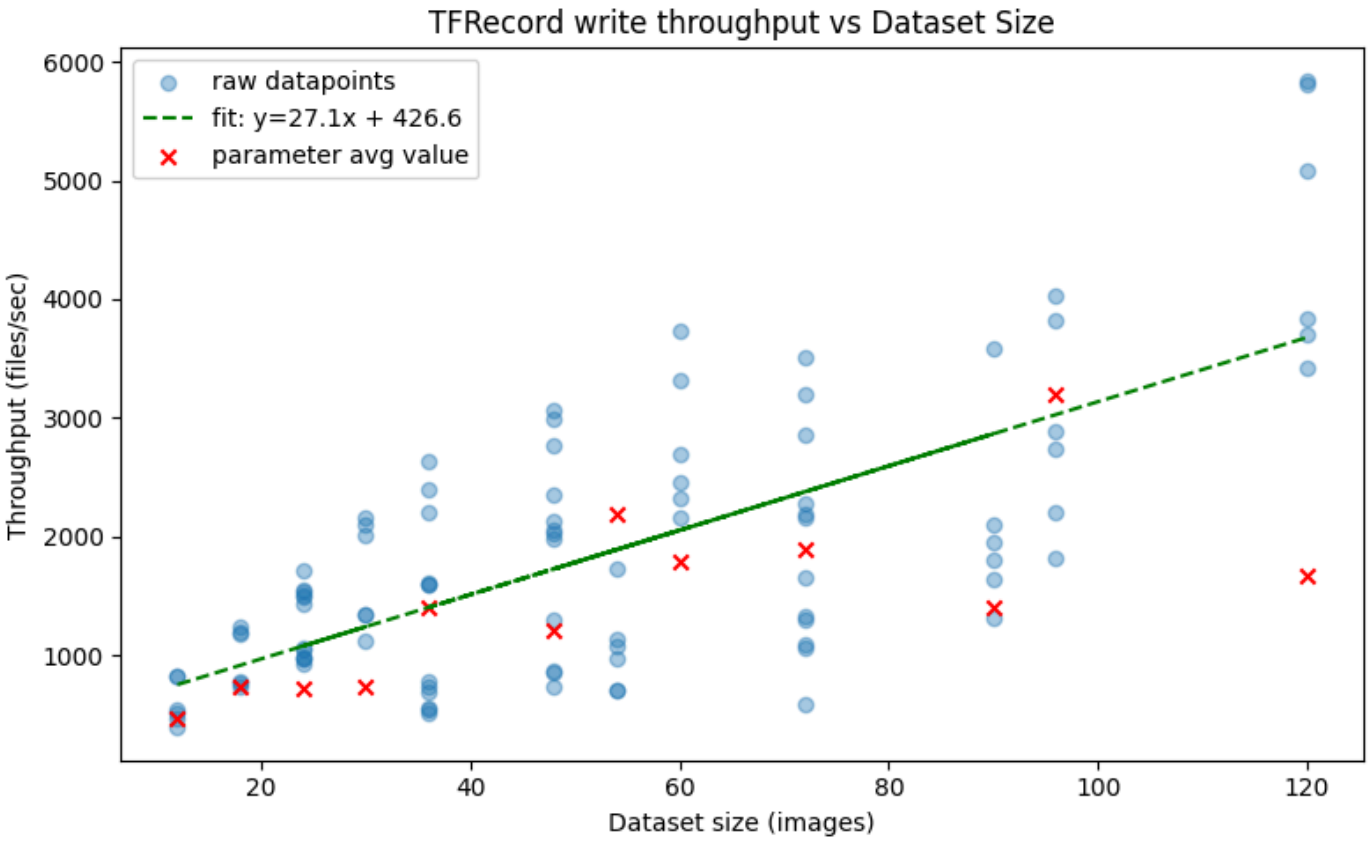



```
Coefficient (slope): 463.26
Intercept          : 1382.23
P-value            : 0.00336
```

In [31]:

```
### CODING TASK ###

# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    tfr_datasetsize_speed,
    columns=["datasetsize", "reading_speed"]
)
df_avg = pd.DataFrame(
    tfr_datasetsize_avg_speed,
    columns=["datasetsize", "reading_speed"]
)

# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["datasetsize"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["datasetsize"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("datasetsize")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["datasetsize"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
```

```
    x_fit, y_fit,
    linestyle="--",
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["datasetsize"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("TFRecord write throughput vs Dataset Size")
plt.xlabel("Dataset size (images)")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```

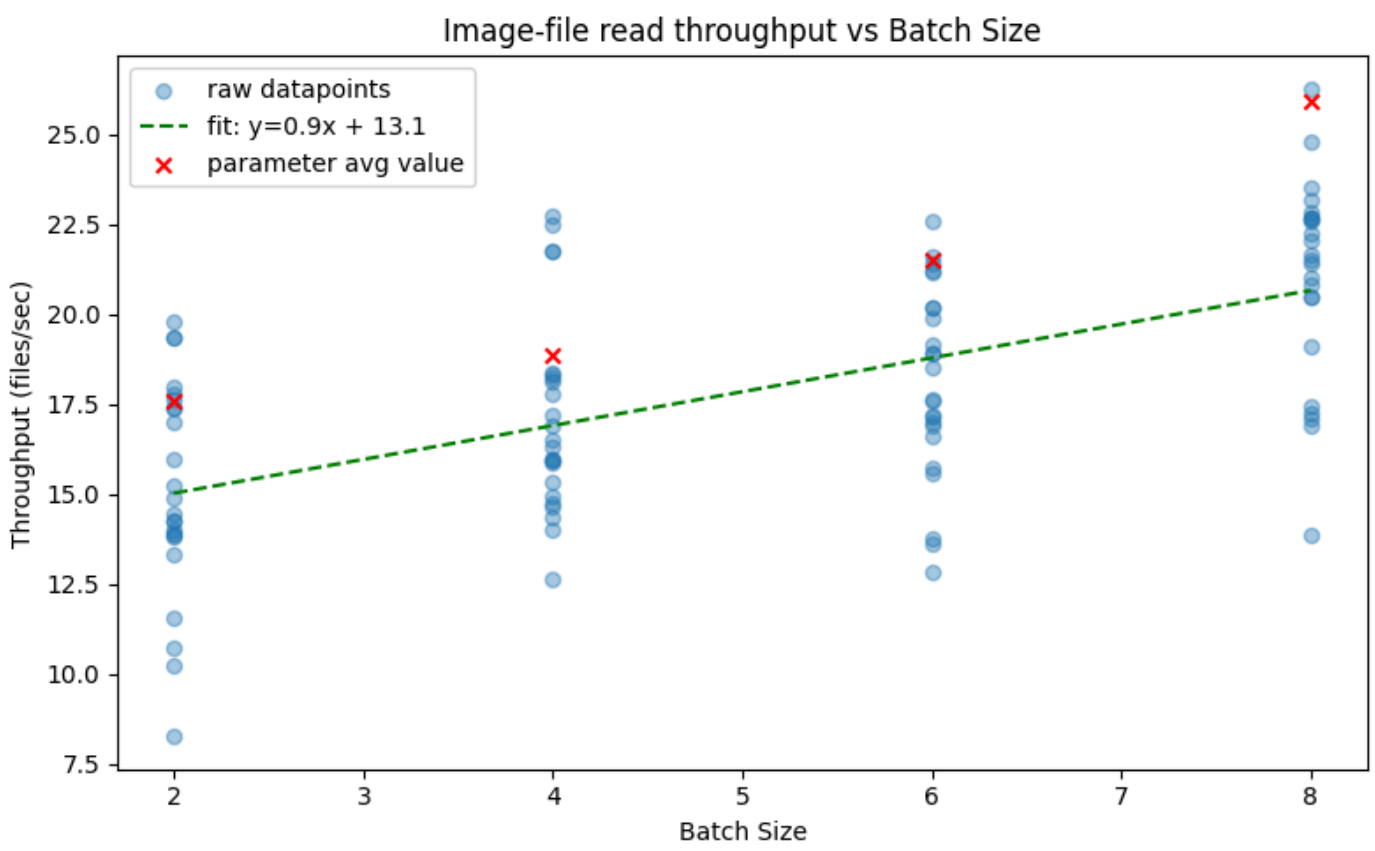



```
Coefficient (slope): 27.08
Intercept          : 426.58
P-value            : 9.17e-16
```

In [32]:

```
# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    img_bsizes_speed,
    columns=["bsizes", "reading_speed"]
)
df_avg = pd.DataFrame(
    img_bsizes_avg_speed,
    columns=["bsizes", "reading_speed"]
)
```

```python
# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["bsizes"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["bsizes"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("bsizes")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["bsizes"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
    x_fit, y_fit,
    linestyle="--",
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["bsizes"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("Image-file read throughput vs Batch Size")
plt.xlabel("Batch Size")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```




```
Coefficient (slope): 0.94
Intercept          : 13.14
P-value            : 8.17e-11
```

In [33]:

```python
# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    img_bnums_speed,
    columns=["bnums", "reading_speed"]
)
df_avg = pd.DataFrame(
    img_bnums_avg_speed,
    columns=["bnums", "reading_speed"]
)

# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["bnums"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["bnums"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("bnums")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["bnums"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
    x_fit, y_fit,
    linestyle="--",
```

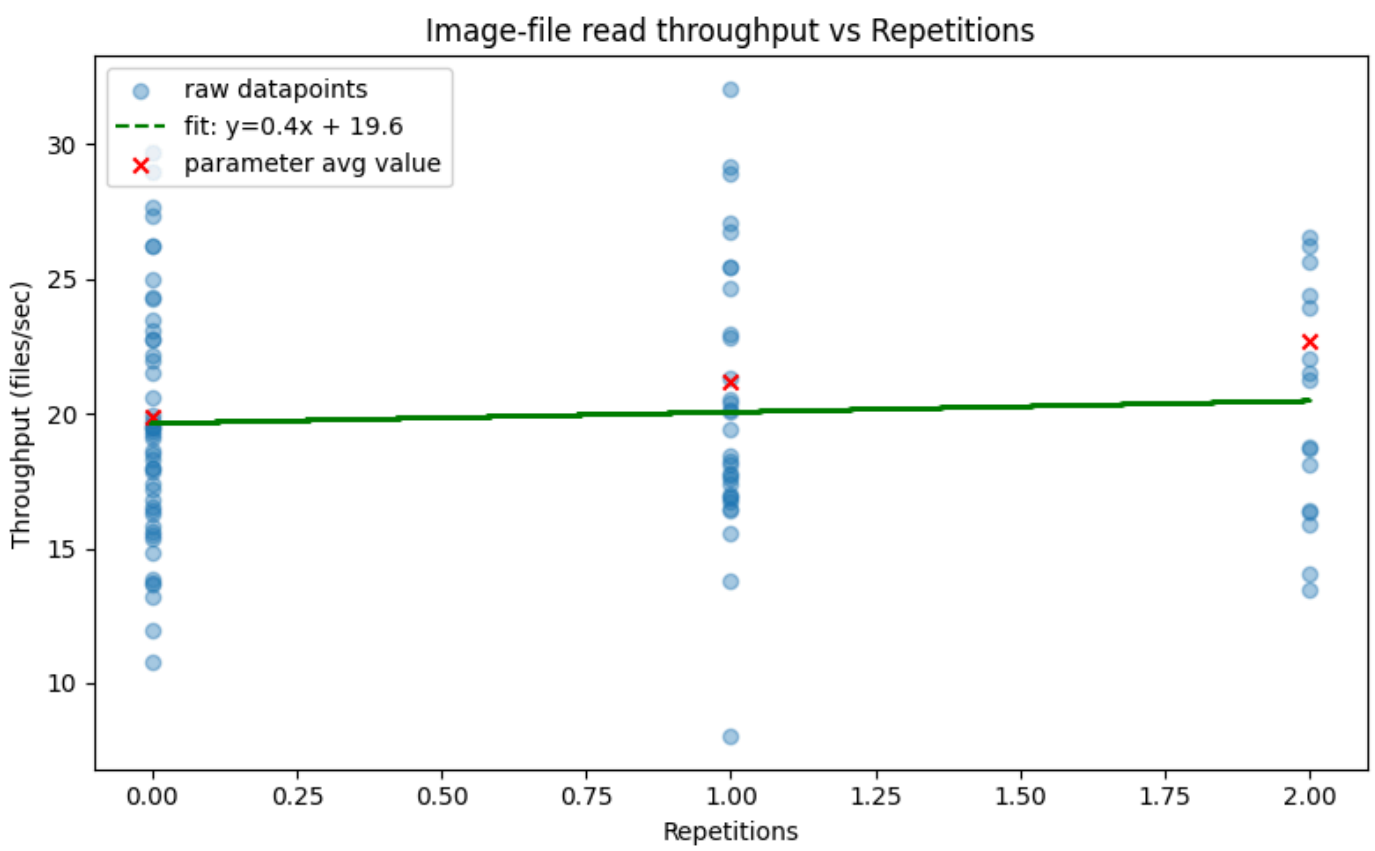
```
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["bnums"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("Image-file read throughput vs Batch Number")
plt.xlabel("Batch Number")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```


```
Coefficient (slope): 0.39
Intercept          : 14.51
P-value            : 0.00117
```

In [34]:

```
### CODING TASK ###

# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    img_repetitions_speed,
    columns=["repetitions", "reading_speed"]
)
df_avg = pd.DataFrame(
    img_repetitions_avg_speed,
    columns=["repetitions", "reading_speed"]
)
```

```python
# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["repetitions"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["repetitions"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("repetitions")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["repetitions"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
    x_fit, y_fit,
    linestyle="--",
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["repetitions"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("Image-file read throughput vs Repetitions")
plt.xlabel("Repetitions")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```




```
Coefficient (slope): 0.42
Intercept          : 19.64
P-value            : 0.515
```

In [35]:

```
### CODING TASK ###

# rebuilding the two DataFrames from the lists saved in task2b
df_raw = pd.DataFrame(
    img_datasetsize_speed,
    columns=["datasetsize", "reading_speed"]
)
df_avg = pd.DataFrame(
    img_datasetsize_avg_speed,
    columns=["datasetsize", "reading_speed"]
)

# running the regression on the raw datapoints
slope, intercept, r_value, p_value, stderr = linregress(
    df_raw["datasetsize"],
    df_raw["reading_speed"]
)

# building the fit line
x_fit = df_raw["datasetsize"]
y_fit = slope * x_fit + intercept

# sorting the averages so they plot in the correct X order
df_avg = df_avg.sort_values("datasetsize")

plt.figure(figsize=(8,5))

# raw points
plt.scatter(
    df_raw["datasetsize"],
    df_raw["reading_speed"],
    alpha=0.4,
    label="raw datapoints"
)

# regression line (green dashed)
plt.plot(
```

```
    x_fit, y_fit,
    linestyle="--",
    color="green",
    label=f"fit: y={slope:.1f}x + {intercept:.1f}"
)

# parameter-averages (red X's)
plt.scatter(
    df_avg["datasetsize"],
    df_avg["reading_speed"],
    marker="x",
    color="red",
    label="parameter avg value"
)

plt.title("Image-file read throughput vs Dataset Size")
plt.xlabel("Dataset size (images)")
plt.ylabel("Throughput (files/sec)")
plt.legend(loc="upper left")
plt.tight_layout()
plt.show()

# printing the values
print(f"Coefficient (slope): {slope:.2f}")
print(f"Intercept          : {intercept:.2f}")
print(f"P-value            : {p_value:.3g}")
```


```
Coefficient (slope): 0.12
Intercept          : 14.85
P-value            : 1.15e-14
```

Section 3. Theoretical discussion

Task 3: Discussion in context. (24%)

In this task we refer an idea that is introduced in this paper:

- Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., & Zhang, M. (2017). Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics.. In USENIX NSDI 17 (pp. 469-482).

Alipourfard et al (2017) introduce the prediction an optimal or near-optimal cloud configuration for a given compute task.

3a) Contextualise

Relate the previous tasks and the results to this concept. (It is not necessary to work through the full details of the paper, focus just on the main ideas). To what extent and under what conditions do the concepts and techniques in the paper apply to the task in this coursework? (12%)

3b) Strategise

Define - as far as possible - concrete strategies for different application scenarios (batch, stream) and discuss the general relationship with the concepts above. (12%)

Provide the answers to these questions in your report.

Final cleanup

Once you have finshed the work, you can delete the buckets, to stop incurring cost that depletes your credit.

In [ ]:
```
!gsutil -m rm -r $BUCKET/* # Empty your bucket
!gsutil rb $BUCKET # delete the bucket
```