```
from pyspark.sql import SparkSession
spark = SparkSession.builder.enableHiveSupport().getOrCreate()

from hdfs3 import HDFileSystem
from pyspark.sql import Row
from pyspark.sql.types import *
from pyspark.sql.functions import *
from datetime import datetime
from datetime import date
import os
import math
import holidays
import pandas as pd
import numpy as np
from host_name import *

"""
    run 'run_collector()' to collect IP's in unprocessed files and do preliminary processing
    (date, time)
    run 'run_processor()' to get holidays, visitor/session ID, duration etc
"""

hdfs = HDFileSystem(host=HOST_NAME, port=8888)


def flatten(pair):
    """
    :param pair: file name and content text from sc.wholeTextFiles(files)
    :return: RDD of filename(yyyymmddhhmm) and each line of content(ip)
    """
    f, text = pair
    f = f[-12:]  # file name
    return [line.split(', ') + [f] for line in text.splitlines()]


def get_last_file(file='/iplog_last_file.tx'):
    """
    :param file: text file name of the last processed file(string)
    :return: last processed file (yyyymmddhhmm) as integer
    """
    if hdfs.exists(file):
        return int(sc.textFile(file).collect()[0])
    else:
        print
        'File does not exist.'


def save_last_file(last_processed_file, file='/iplog_last_file.tx'):
    """
    :param last_processed_file: last processed file (yyyymmddhhmm)
    :param file: text file name of the last processed file(string)
    """
    if hdfs.exists(file):
        # remove current file
        hdfs.rm(file, recursive=True)

    # save the new one
    sc.parallelize([last_processed_file]).saveAsTextFile(file)


def get_all_file(hdfs_path, keyword='iplog'):
    """
    :param hdfs_path: directory where the files are stored
    :param keyword: keyword to filter files, if any
    :return: a list of last 12 character of files(yyyymmddhhmm)
```

```
    """
    files = hdfs.ls(hdfs_path)
    # only keep files with hadoop in the name and only keep last 12 characters
    files = [x[-12:] for x in files if keyword in x]
    return files


def get_holidays(year=[2017, 2018], add_on=None):
    """
    year: int, a year or a list of years
    add_on: a list of holidays(ex. university bonus day) that are not included in the holidays
    library,
            in the format of string 'yyyy-mm-dd', or 'yyyy/mm/dd'.
            Names of these days will be the default "Holiday"
    return: a dictionary of US holidays, including add_on holidays
    """
    holidays_us = holidays.US(years=year)
    if add_on is not None:
        holidays_us.append(add_on)
    return holidays_us

def ip_collector_parquet(path='/new_logs/', file_key='access-*'):
    if hdfs.exists(path):
        files = path + file_key
        log = sc.wholeTextFiles(files).flatMap(flatten)
        if log.count() > 0:
            log_df = log.toDF(['ip', 'file'])
            func = udf(lambda x: datetime.strptime(x, '%Y%m%d%H%M'), TimestampType())
            log_df = log_df.withColumn('datetime', func(col('file')))
            log_df = log_df.withColumn('date', log_df['datetime'].cast('date')) \
                .withColumn('year', year('datetime').cast('integer')) \
                .withColumn('month', month('datetime').cast('integer')) \
                .withColumn('day', dayofmonth('datetime').cast('integer')) \
                .withColumn('hour', hour('datetime').cast('integer')) \
                .withColumn('minute', minute('datetime').cast('integer')) \
                .withColumn('dow', date_format('datetime', 'E').cast('string')) \
                .withColumn('dow_num', date_format('datetime', 'u').cast('integer'))
            log_df = log_df.withColumn('file', log_df['file'].cast(LongType()))
            log_df = log_df.orderBy(log_df.datetime, ascending=True)
            newest = log_df.groupBy().max('file').collect()[0]['max(file)']

            new_file = '/ip_collection_new.parquet'
            log_df.write.mode('append').parquet(new_file)

            last_file_f = '/iplog_last_file.tx'
            save_last_file(last_processed_file=newest, file=last_file_f)

            nip = log_df.count()
            nfile = log_df.select('file').distinct().count()
            print nip, 'new IPs from', nfile, 'new files have been added to the master file.'
            print 'Latest file:', newest

        else:
            print 'No new file to be processed.'

        hdfs.rm(path, recursive=True)


def run_collector():
    last_file = get_last_file()
    hdfs_path = '/iplog_dump/'
    all_files = get_all_file(hdfs_path, keyword='access')
    new_files = [x for x in all_files if int(x) > last_file]
    n = len(new_files)
    k = 50
```

```
    temp_dir = '/new_logs/'
    if hdfs.exists(temp_dir):
        print 'Directory "new_logs" already exists.'
        print 'There are ' + str(len(hdfs.ls(temp_dir))) + ' files in this directory'

    else:
        for i in xrange(int(math.ceil(float(n) / float(k)))):
            # create directory
            hdfs.mkdir(temp_dir)
            print
            'Temporary directory "new_logs" has been created.'

            m = __builtins__.min(k * (i + 1), n)
            for f in new_files[k * i:m]:
                dest = '/new_logs/access-' + f
                src = '/iplog_dump/access-' + f
                if hdfs.du(src).values()[0] > 0:
                    os.system('hdfs dfs -cp ' + src + ' ' + dest)

            if len(hdfs.ls(temp_dir)) > 0:
                ip_collector_parquet()

def run_processor():
    if hdfs.exists('/ip_collection_new.parquet'):
        print 'There are new data.'

        # last rows data

        last_rows_file = '/iplog_last_rows.parquet'
        last_rows_df = spark.read.parquet(last_rows_file)
        last_rows_df = last_rows_df.withColumn('datetime',
        last_rows_df['datetime'].cast(TimestampType()))
        last_rows_df = last_rows_df.orderBy(last_rows_df.datetime, ascending=True)
        last_rows_pd = last_rows_df.toPandas()

        # new logs

        new_file = '/ip_collection_new.parquet'
        new_df = spark.read.parquet(new_file)
        new_df = new_df.orderBy(new_df.datetime, ascending=True)
        new_pd = new_df.toPandas()
        last_rows_pd.file.max()
        new_pd = new_pd[new_pd.file > last_rows_pd.file.max()].reset_index(drop=True)

    #      Add holiday

        if 'holiday' not in new_pd.columns:
            new_pd['holiday'] = new_pd.date.apply(lambda d: d in holidays_us)
            new_pd['dow_2'] = new_pd['dow']
            new_pd['dow_num_2'] = new_pd['dow_num']
            new_pd.loc[(new_pd.holiday) | (new_pd.dow_num_2 >= 6), 'dow_num_2'] = 6
            new_pd.loc[new_pd.dow_num_2 == 6, 'dow_2'] = 'weekend_holidays'

        # add new visitor id to new visitor (ip)
        # get "ses_base", which is the latest session number of each visitor. For new visitor,
        it's 1.
        existing_visitor = last_rows_pd[['ip', 'visitor_id',
        'session']].rename(columns={'session': 'ses_base'})
        new_visitor = pd.DataFrame({'ip':
        new_pd[~new_pd.ip.isin(existing_visitor.ip)].ip.unique()})
        vid_base = existing_visitor.visitor_id.max()
        new_visitor['visitor_id'] = list(range(vid_base + 1, len(new_visitor) + vid_base + 1))
        new_visitor['ses_base'] = 1  # ses
        comb_visitor = existing_visitor.append(new_visitor, ignore_index=True)
```

```
new_pd = pd.merge(left=new_pd, right=comb_visitor.drop('ses_base', axis=1), on='ip',
how='left')

comb_pd = last_rows_pd.append(new_pd, ignore_index=True)[
    last_rows_pd.columns.tolist()]  # column order gets shifted (alphabetic order)
    after appending
comb_pd = pd.merge(left=comb_pd, right=comb_visitor.drop('visitor_id', axis=1),
on='ip', how='left')

# Time Delta (temporarily for identifying sessions)

# create time delta in minute
grouped = comb_pd.groupby('visitor_id')
comb_pd['timedelta'] = grouped.datetime.transform(pd.Series.diff) / np.timedelta64(1,
'm')
# At this point, it's the time delta with the previous row, therefore the value should be
# the duration of the previous page
# Right now it's useful for identifying visitors (if idle time > 30mins). This will be
corrected later

# Session
# each visitor, if idle time >=30 mins, consider it a new session
comb_pd['check_session'] = np.array(grouped.timedelta.apply(lambda x: x >= 30))
comb_pd['temp_session'] = grouped.check_session.cumsum()
comb_pd['session'] = comb_pd.temp_session + comb_pd.ses_base
comb_pd['session'] = comb_pd['session'].astype('int')
comb_pd = comb_pd.drop(['check_session', 'temp_session', 'ses_base'], axis=1)
# session id (visitor_session)
comb_pd['session_id'] = comb_pd.visitor_id.astype('str') + '_' +
comb_pd.session.astype('str')

# Duration

# Re-calculate time delta by session _id
comb_pd = comb_pd.sort_values(['session_id', 'datetime'],
ascending=True).reset_index(drop=True)
grouped = comb_pd.groupby('session_id')

# duration (time delta) in minute
comb_pd['duration'] = grouped.datetime.transform(pd.Series.diff) / np.timedelta64(1, 'm')
# At this point, duration is the time delta with the previous row, therefore the value
should be the duration of the previous page

# Shift duration one row up
comb_pd['duration'] = list(comb_pd['duration'][1:]) + [np.nan]

# Sort by datetime
comb_pd = comb_pd.sort_values(['datetime', 'visitor_id'],
ascending=True).reset_index(drop=True)

new_last_rows_pd = comb_pd.groupby('ip').tail(1).reset_index(drop=True)
no_last_rows_pd = comb_pd[
    ((comb_pd.session_id.isin(new_last_rows_pd.session_id)) &
    (~comb_pd.duration.isnull())) |
    (~comb_pd.session_id.isin(new_last_rows_pd.session_id))].reset_index(drop=True)

# Set schema - the default will convert panda int64 to biginit, and timestamp as biginit
# * It's necessary to convert the datetime column (dtype = timestamp) to string before
converting to spark DF, otherwise
#   it will be read as biginit (serial number datetime(1970, 1, 1) +
timedelta(microseconds=tstampl/1000))
#
https://stackoverflow.com/questions/45308406/how-does-spark-handle-timestamp-types-during
-pandas-dataframe-conversion
# * The order of the StructField MUST be the same as the columns order
```

```
    new_last_rows_pd['datetime'] = new_last_rows_pd.datetime.astype('str')
    no_last_rows_pd['datetime'] = no_last_rows_pd.datetime.astype('str')

    sch = StructType([StructField('ip', StringType(), True), StructField('file',
    LongType(), True),
                      StructField('datetime', StringType(), True), StructField('date',
                      DateType(), True),
                      StructField('year', IntegerType(), True), StructField('month',
                      IntegerType(), True),
                      StructField('day', IntegerType(), True), StructField('hour',
                      IntegerType(), True),
                      StructField('minute', IntegerType(), True), StructField('dow',
                      StringType(), True),
                      StructField('dow_num', IntegerType(), True), StructField('holiday',
                      BooleanType(), True),
                      StructField('dow_2', StringType(), True), StructField('dow_num_2',
                      IntegerType(), True),
                      StructField('visitor_id', IntegerType(), True),
                      StructField('timedelta', DoubleType(), True),
                      StructField('session', IntegerType(), True),
                      StructField('session_id', StringType(), True),
                      StructField('duration', DoubleType(), True)])

    new_file = '/ip_collection_new.parquet'
    hdfs.rm(new_file, recursive=True)

else:
    print 'There is no new data to be processed'
```