Till now whatever we have covered in batch processing

Real time streaming - streaming data.

Batch processing -
==========

doing analysis on top of files
your processing might take few minutes, hours, days
mainly reporting kind of requirement

Real time processing
==========

we will have continously flowing data and we have to calculate the results instantly.

credit card fraud detection
finding the trending hashtag
track website failures using the server logs.

when we talk about hadoop

HDFS + MR + YARN

MR - batch

mapreduce cannot handle real time streaming.

how to handle streaming data?

Apache Spark - batch / streaming

spark is a general purpose compute engine

In spark there is a module called as spark structured streaming

there are 2 ways to handle streaming data using spark...

RDD - spark streaming API's

Dataframe / Spark SQL - Spark structured streaming API's

continuous stream of data...

producer - application which is generating continous data
consumer - spark streaming application

twitter
for example if we have to compute the trending hashtags
twitter is the producer

step 1: I will create a dummy producer
nc -lk 9998

you want to create a data on a particular socket

socket = IP + port (localhost + 9998)

step 2: start a consumer

```
from pyspark.sql.functions import *
# Create DataFrame representing the stream of input lines from connection to
localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9998) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()


# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

==========

Batch processing

capture the data and save it in a file.

we used to process this file at a regular interval lets say

every 12 hours

every 3 hours

every 1 hour

every 15 mins

every 1 min

streaming job - we want to increase the frequency of processing

Big Retail Chain

500 different stores

in every store there are 10 billing counters

5000 billing counters

10 am - 9 pm

we want to calculate total sales across all the store every 15 mins.

you are getting the data in data lake - hdfs / adls gen2 / s3

in a folder you are getting new files

10:00 am - 10:15 am - 2M batch 1 2M
10:15 am - 10:30 am - 5M batch 2 3M
10:30 am - 10:45 am - 9M batch 3

you are using batch processing and scheduling your job every 15 mins

whatever transaction happened between 10 am to 10:15 am - there is no guarantee that by 10:30 am it will be processed..

=> backpressure - you were supposed to run batch 2 now and your batch 1 is still running.

=> late arriving records

a record which was generated (event time - 10:08 am but it arrived late)

10:20 am

=> incremental style

=> what if one of the batch fails for some reason

=> fault tolerance

=> how would you manage the state - aggregations

=> how to schedule the job for continuous listening

spark structured streaming takes care of all these issues

=> automatic looping between the batches

=> storing intermediate results

=> combining results to the previous batch result

=> restart from same place in case of failure - fault tolerance.

internally spark engine is a micro batch engine

near real time

RDD

Structured API's
- offers unified model for batch and stream processing
- runs over the spark sql engine. takes advantages of sql based optimizations.

Code for word count
==========

1. local (visual studio)

2. itversity labs

3. databricks community edition

```
+--------+
|   value|
+--------+
|hi there|
+--------+
```

big data is really interesting

```
lines.select(explode(split(lines.value," ")).alias("word"))
```

[big , data, is, really, interesting]

words

```
word
======
big
data
is
really
interesting
```

```
from pyspark.sql.functions import *

lines = spark \
    .readStream \
    .format("socket") \
    .option("host","localhost") \
    .option("port",9989) \
    .load()


words =  lines.select(explode(split(lines.value," ")).alias("word"))
wordCounts = words.groupBy("word").count()

query = wordCounts \
```

```
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .option("checkpointLocation","checkpointdir5") \
    .start()

from pyspark.sql.functions import *
from pyspark.sql import SparkSession

if __name__ == '__main__':

    print("Creating Spark Session ")

    spark =  SparkSession.builder \
            .appName("streaming application") \
            .config('spark.sql.shuffle.partitions',3) \
            .master("local[2]") \
            .getOrCreate()


# 1. read the data
    lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9993) \
    .load()

# 2. processing logic
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

wordCounts = words.groupBy("word").count()


# 3. write to the sink
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .option("checkpointLocation","checkpointlocation1") \
    .start()
```

query.awaitTermination()

==========

streaming df can be created through the data stream reader

input sources
==========

there are a few built in sources..

=> File source - reads files written in a directory as a stream of data.

files will be processed in the order of the file modification time. if latestFirst is set then the order will be reversed.

supported file formats - text, csv, json, orc, parquet

=> kafka source - reads the data from kafka topic.

=> socket source (testing) - the listening server socket is on the driver. does not provide end to end fault tolerance grarantees.

as soon as the streaming query starts it will prepare the checkpoint location.

checkpoint directory
=> fault tolerance
=> ??

the streaming query will keep checking for new data socket / in the directory location.

once it finds new data/files it will trigger the microbatch.

inputdir
 - file1

before the microbatch starts, streaming query updates the checkpoint that file1 is going to be processed.

once the microbatch finishes it will update this in checkpoint location.

streaming query wont stop, it will keep looking for new data.

in case of complete if I move file2

CLOSED 2
PENDING_PAYMENT 3
COMPLETE 4

UPDATE MODE (UPSERT) updates + inserts

AnalysisException: Append output mode not supported when there are streaming aggregations on streaming DataFrames/DataSets without watermark;

3 output modes
==========

1. append - only the new records

2. complete - the entire updated result table will be written to the external storage.

3. update - only the rows that are updates or newly inserted will be written to the external storage.

how spark streaming is able to maintain the state?

state store - in memory

same thing is even persisted in checkpoint location..

state store is a key-value store which provides both the read and write operations. In structured streaming we use the state store to handle stateful operations across the batches

groupBy, aggregations

stateful and stateless

state is maintained in state store (memory)

select, filter, map

since spark 3.1 we also have DataStreamReader.table() to read tables as streaming dataframes

DataStreamWrite.toTable() to write streaming dataframes as tables.

```
format("delta")

checkpoint
- incremental processing
- fault tolerance

from pyspark.sql.functions import *
from pyspark.sql import SparkSession

if __name__ == '__main__':

    print("Creating Spark Session ")

    spark =  SparkSession.builder \
            .appName("streaming application") \
            .config("spark.sql.shuffle.partitions",3) \
            .master("local[2]") \
            .getOrCreate()

    orders_schema = 'order_id long, order_date date, order_customer_id long,
order_status string'

# 1. read the data
    orders_df = spark \
    .readStream \
    .format("json") \
    .schema(orders_schema) \
    .option("path","data/inputdir") \
    .load()

# 2. processing logic
    orders_df.createOrReplaceTempView("orders")
    completed_orders = spark.sql("select * from orders where order_status =
'COMPLETE'")

# 3. write to the sink
query = completed_orders \
    .writeStream \
    .format("csv") \
    .outputMode("append") \
    .option("path","data/outputdir") \
    .option("checkpointLocation","checkpointdir1") \
    .start()

query.awaitTermination()
```

Running Notes
==========

append
complete
update

In this session we will talk about generating the input data

Json data...

- Complete output mode

- update mode

orders
customers
order_items

/public/retail_db

we need to join
orders
customers
order_items

Running Notes
==========

Databricks

Delta format - Complete
table

Parquet, CSV - Complete mode does not work
table

========

Update Mode
============

toTable will work with append mode, complete mode

with update mode it wont work

target table (final result table) t orders_final_result

source table (mirco batch table) s orders_result

if t.customer_id == s.customer_id

when matched then

update all the columns...

when not matched then

insert *

target table - 100000 records

source table - 2 inserts and 1 update

merge