Apache Spark
==============

Apache spark is a
====================

general purpose
in memory
compute engine

compute engine
================

hadoop provides 3 things:
1. hdfs - Storage
2. mapreduce - Computation
3. YARN - Resource Manager

Spark is a replacement/alternative of mapreduce.

Spark is a plug and play compute engine which needs 2 things to work with.

1. Storage - local storage, hdfs, Amazon S3

2. Resource Manager - YARN, Mesos, Kubernetes

in-memory
==========

mr1   mr2   mr3   mr4   mr5

HDFS

for each mapreduce job we require 2 disk access

one is for reading and other is for writing.

Spark

V1 V2 V3 v4 v5

HDFS

only 2 disk IO's are required.

spark is said to be
10 to 100 times faster than mapreduce

General Purpose
================

pig for cleaning

hive for querying

mahout

sqoop

only bound to use map and reduce

learn just one style of writing the code and all the things like cleaning, querying, machine learning, data ingestion all these can happen with that.

filter

map

reduce

Spark Session - 2
==================

The basic unit which holds the data in spark is called as RDD

resilient distributed dataset

List

RDD is nothing but in-memory distributed collection.

```
rdd1 = load file1 from hdfs
rdd1 = rdd1.map
rdd1 = rdd2.filter
rdd1.collect()
```

DAG - Directed Acyclic graph

There are 2 kind of operations in spark
==

1. Transformation
2. Action

Transformations are lazy

Action are not.

whenever you call transformations an entry to the execution plan is added.

RDD's are:

Distributed

in memory

Resilient (fault tolerant) - if we loose an rdd we can again recover it back.

rdds are resilient to failures.

```
RDD1
 | MAP
 |
RDD2
 | FILTER
 |
RDD3
```

IF RDD3 IS LOST THEN IT WILL CHECK FOR ITS PARENT RDD USING THE LINEAGE GRAPH AND IT WILL QUICKLY APPLY THE FILTER TRANSFORMATION ON RDD2

Immutable

==========

once we load rdd with data the data cannot be changed.

why immutable?

why transformations are lazy?

assume that transformations are not lazy.

consider you have a 1 gb file in hdfs.

rdd1 = load file1 from hdfs

rdd1.print(line1)

to print just 1 line we ended up loading 1 gb file in memory.

consider the fact, spark is lazy.

```
rdd1 = load file1 from hdfs
```

```
rdd1.print(line1)
```

file1 is in hdfs with 10 lakh rows

```
rdd1 = load file1 from hdfs
rdd2 = rdd1.map
rdd3 = rdd2.filter
rdd3.collect()
```

consider if spark is not lazy.

rdd1 will be materialized.

10 lakh lines will be processes

in case of a map the number of input records is the same as number of output records.

consider after applying filter we are just interested in 5 records..

but consider the fact that spark is lazy:

=======

word count in spark
=====================

we need to find the frequency of each word in a file which resides in hdfs.

we have created a file in local and then moved it to hdfs

now we want to process this file in hdfs using apache spark

spark-shell (scala)

pyspark (python)

sc is nothing but the spark context

and it is the entry point to the spark cluster the basic unit which holds the data in spark is called as an rdd

```
val rdd1 = sc.textFile("/user/cloudera/sparkinput/file1")
```

```
val rdd2 = rdd1.flatMap(x => x.split(" "))
```

flatmap basically takes each line as input

Array(spark,is,very,interesting,spark,is,in,memory,compute,engine)

```
spark           (spark,1)
is              (is,1)
very       (very,1)
interesting (interesting,1)
spark.     (spark,1)
is
in
```

in a map if we have n inputs then we will definitely have n outputs.

```
val rdd3 = rdd2.map(x => (x,1))

(spark,4)
(is,1)
(very,1)
(interesting,2)


val rdd4 = rdd3.reduceByKey((x,y) => x+y)

rdd4.collect()


localhost:4040  (this gives you the spark ui)
```

spark with scala code
======================

```
val rdd1 = sc.textFile("/user/cloudera/sparkinput/file1")

val rdd2 = rdd1.flatMap(x => x.split(" "))
val rdd3 = rdd2.map(x => (x, 1))

val rdd4 = rdd3.reduceByKey( x,y => x + y)

rdd4.collect()
```

pyspark
=========

```
rdd1 = sc.textFile("file:///home/cloudera/file1")

rdd2 = rdd1.flatMap(lambda x : x.split(" "))

rdd3 = rdd2.map(lambda x : (x, 1))

rdd4 = rdd3.reduceByKey(lambda x,y : x + y)

rdd4.collect

.saveAsTextFile("<hdfs path>")
```

in scala we have anonymous functions and same thing is called as lambdas in your python

Spark practical - 4
====================

word count problem - spark-shell (terminal)

word count problem in ide with a better dataset

we improved the word count by normalizing the case

so we sorted the data and get the top 10.

customer_id , product_id, amount_spent

we need to find out top 10 customers who spent the maximum amount.

```
44,8602,37.19            44,37.19
35,5368,65.89   -> map   35,65.89
2,3391,40.64             2,40.64
```

(44,94)

(35,165)
(2,40)

map

reduceByKey((x,y) => x+y)

sortBy(x => x._2)

collect

x 44,8602,37.19

x.split(",")

whenever rdd contains tuple of 2 elements it is called as a pair rdd.

here the 1st element can be treated like key and second element can be treated like value.

Spark practical - 5
=====================

user_id   movie_idrating_given timestamp

how many times movies were rated 5 star
how many times movies were rated 4 star
how many times movies were rated 3 star
how many times movies were rated 2 star
how many times movies were rated 1 star

3
3
1
2
1
3
2
4

(3,1)
(3,1)
(1,1)

(2,1)
(1,1)

instead of using map where we say (x,1) and doing reduceByKey later.

map + reduceByKey is a tranformation -> rdd

countByValue is an action -> local variable

so if you feel that countByValue is the last thing you are doing and there are no more operations after that then it's ok to have countByValue.

but if we feel that we need more operations after this then we should not use countByValue because we wont get parallelism.

Spark practical - 6
====================

row_id, name, age, number_of_connections we need to find average number of connections for each age

33 , 100
33 , 200
33, 300

output 33,200

42, 200
42, 400
42, 500
42 ,700

output 42, 450

input 0,Will,33,385

output (33,385)

```
  //input
  //(33,100)
```

```
//(33,200)
//(33,300)

x._1 33
x._2 100

//output
//(33,(100,1))
//(33,(200,1))
//(33,(300,1))


mappedInput.map(x => (x._1,(x._2,1)))

  input
      x //(33,(100,1))
  y //(33,(200,1))
  //(33,(300,1))

   x._1
   y._1



  output
  //(33,(600,3))
```

```
//(34,(800,4))


reduceBykey((x,y) => (x._1 + y._1 , x._2 + y._2))

input
//(33,(600,3))
//(34,(800,4))


output
(33,200)
(34,200)


   totalsByAge.map(x => (x._1,x._2._1/x._2._2))
```

Spark practical - 7
====================

1,11

1st column is the search word
11th is the total cost for the search word.

big data contents 24.06
learning big data 34.98

to get just 2 columns from all the columns

map

```
val initial_rdd =
sc.textFile("/Users/trendytech/Downloads/bigdata-campai
gn-data.csv")

val mappedInput = initial_rdd.map(x =>
(x.split(",")(10).toFloat,x.split(",")(0)))
```

```
24.06, big data contents
34.98, learning big data
```

flatMapValues

input: 24.06, big data contents

```
val words = mappedInput.flatMapValues(x => x.split(" "))
```

```
(24.06,big)
(24.06,data)
(24.06,contents)
```

```
val finalMapped = words.map(x =>
(x._2.toLowerCase(),x._1))
```

```
(big,24.06)
(data, 24.06)
(contents, 24.06)
```

output:
```
(big,24.06)
(data,24.06)
(contents,24.06)
(learning,34.98)
(big,34.98)
(data,34.98)
```

```
val total = finalMapped.reduceByKey((x,y) => x+y)

val sorted = total.sortBy(x => x._2,false)
```

Spark practical - 8
====================

big data training in bangalore

we created a file with all boring words and stored it on desktop

boringwords is small data

we will broadcast this on all machines..

and the campaign search keywords will be distributed across the cluster.

map side join in hive
======================

broadcast join in spark
==============================

broadcast variable

map

list

arrays

set

Spark practical - 9
========================

Accumulators

there is a shared copy kept in your driver machine.

each of the executors will update it.

however none of the executors can read the value of accumulator, they can only change the value.

this is same as counters in your mapreduce.

2 kinds of shared variable
============================

1. accumulator (we have single copy on driver machine)

2. broadcast variable (we have a separate copy on each machine)

accumulator is similar to counters in mapreduce broadcast variable plays the same role as map side join in hive

=======

spark practical - 10
========================

```
"WARN: Tuesday 4 September 0405"
"ERROR: Tuesday 4 September 0408"
"ERROR: Tuesday 4 September 0408"
"ERROR: Tuesday 4 September 0408"
"ERROR: Tuesday 4 September 0408"
"ERROR: Tuesday 4 September 0408"
```

```
(WARN,1)
(ERROR,1)
(ERROR,1)
(ERROR,1)
(ERROR,1)
(ERROR,1)
```

reduceByKey((x,y) => x+y)

step 1: I want to first create a list in scala using above data.

step 2: create an rdd out of the above list.

step 3: I want to calculate the count of WARN AND ERROR

spark practical - 11
======================

1. narrow and wide transformations

Narrow transformations - no shuffling is involved
========================
map
flatMap
filter

wide transformation - where shuffling is involved
=====================
reduceByKey
groupByKey

500 mb file in hdfs

val rdd1 = sc.textFile("path of the file")

4 partitions because your hdfs file has 4 blocks.

there is a 1 to 1 mapping between your file blocks and rdd partitions.

rdd1.map(x => x.length)

p1 p2 p3 p4

o1 o2 o3 o4

hello how are you

hello
how
are
you

reduceByKey

p1
(hello,1)
(how,1)

p2
(hello,1)
(is,1)

p3
(is,1)
(how,1)

p4
(world,1)
(how,1)

(hello,2)
(how,3)
(is,2)
(world,1)

2. Stages in spark

stages are marked by shuffle boundaries.

whenever we encounter a shuffle, a new stage gets created.

whenever we call a wide transformation a new stage gets created.

if I use 3 wide tranformations how many stages get created?

it will be 4 stages..

if we have 2 stages..

then there is 1 shuffling involved.

output of stage 1 is sent to disk

and stage 2 reads it back from disk

we can atleast try to make sure we use wide transformations later

1. narrow vs wide transformation
2. stages in spark

spark practical - 12
======================

Difference between reduceByKey and reduce

reduceByKey is a transformation
reduce is an action

whenever you call a transformation on a rdd you get resultant rdd.

whenever you call an action on a rdd you get local variable.

reduceByKey only works on pair rdd's (tuple with 2 elements)

(hello,1)
(how,1)
(how,1)

reduce is an action

why spark developers gave reduceByKey as transformation and reduce as an action?

reduce gives you a single output which is very small.

reduceByKey

(hello,49)
(hi,23)

we can still have huge amount of data and we might be willing to do further operations in parallel.

spark practical - 13
======================

groupByKey vs reduceByKey

both of them are wide transformations.

reduceByKey
=============
we will get advantage of local aggregation

1. more work in parallel
2. less shuffling required

you can think this same as combiner acting at the mapper end.

groupByKey
============
we do not get any local aggregation

all the key value pairs are sent (shuffled) to another machine.

so we have to shuffle more data
and we get less parallelism.

always prefer reduceByKey and never use groupByKey

consider you have 1 TB data in hdfs

1000 node cluster

how many partitions will be there in your rdd?

1 TB / 128 mb - 8000 blocks

so your rdd will have 8000 partitions.

on each node we might end up getting 8 partitions.

```
NODE 1
WARN: Tuesday 4 September 0405
WARN: Tuesday 4 September 0405
WARN: Tuesday 4 September 0405
ERROR: Tuesday 4 September 0405

NODE 2
ERROR: Tuesday 4 September 0405
ERROR: Tuesday 4 September 0405
```

ERROR: Tuesday 4 September 0405
INFO: Tuesday 4 September 0405

NODE 3
INFO: Tuesday 4 September 0405
INFO: Tuesday 4 September 0405
INFO: Tuesday 4 September 0405
INFO: Tuesday 4 September 0405
INFO: Tuesday 4 September 0405
INFO: Tuesday 4 September 0405

NODE 4

NODE 5

groupByKey

all the warns will go on one machine
all the errors will go on one machine
and all the info will go on one machine

at the max if we have 3 distinct keys.. then we will have maximum 3 machines which all our data.

earlier before applying groupByKey we have our data well distributed across 1000 machines

but now after using groupByKey we have data distributed across at the max 3 machines.

3 machines hold 1 TB data in memory that means we have a huge possibility of out of memory error.

we will get only 3 partitions (maximum) which are full and it can lead to out of memory error.

even if we do not get out of memory error then also it is not suggested to use groupByKey

because we are restricting our parallelism.

so we should never use groupByKey.

hello,1

hello,1
hello,1

(hello,{1,1,1}) x

x._1 , x._2.size

number of jobs is equal to number of actions.

whenever you use wide transformation then new stage is created.

a task corresponds to each partition.

350 mb file.

350/128 - 3 blocks (wrong answer)

and the local block size is 32 mb.

350/32 - 11 blocks

and your rdd should have 11 partitions.

both of them are wide transformations

reduceByKey do local aggregation

groupByKey do not perform local aggregation. can lead to out of memory error.

spark practical - 14
======================

1. pair rdd - tuple of 2 elements

("hello",1)
("hi",1)
("how",1)

transformations like groupByKey, reduceByKey etc..

can only work on a pair rdd.

2. tuple of 2 elements , is this same as a map ?

It is not same.

In a map we can only have distinct keys.. the same key cannot repeat again.

```
("hello",1)
("hello",1)
("hello",1)
("hello",1)
```

in a pair rdd the keys can repeat..

3. to save the output we can use

```
rdd.saveAsTextFile("<the output folder path>")
```

this is a action just like collect.

because the execution plan is executed when we call saveAsTextFile

In the spark UI

inside jobs you will see the actions that you have called.

that means each action is a job in spark UI.

sortByKey is a transformation but still it shows in the jobs.

whenever you call an action..

All the transformations from the very beginning are executed..

what happens when you call next actions

All the transformations from the very beginning are executed..

action3

again all the transformations from the very beginning are executed..

spark practical - 15
====================

sc.defaultParallelism will tell the default parallelism - 8

sc.parallelize

1. sc.defaultParallelism is to check the parallelism level

2. to check the number of partitions

rdd.getNumPartitions

3. sc.defaultMinPartitions which determine the minimum number of partitions rdd has, when we load from file.

what is the difference between repartition and coalesce?

you have a 500 mb file in hdfs.

and you have a spark cluster of 20 machines..

if file size is 500 mb and default block size is 128 mb

we will have 4 blocks in hdfs.

and thats why we will have 4 partitions in your rdd.

rdd1.repartition(10)

we saw repartition can increase the number of partitions.

can it decrease?

you have a 1 tb file in hdfs

8000 blocks

1000 node cluster and you are creating a rdd for this.

8000 partitions

with each node holding around 8 partitions.

map

filter

filter

map

reduce

when we start each partition has 128 mb data.

but when we apply transformation like filter

inside each partition we are left with just few kb's of data..

8000 partitions with each holding few kbs of data...

rdd.repartition(50)

repartition
============

conclusion: repartition can be used to both increase as well decrease the number of partitions in a rdd.

repartition is a wide transformation because shuffling is involved.

coalesce
=========
it can only decrease the number of partitions.

it cannot increase the number of partitions. however if you try increasing it wont give an error. but it wont change the number of partitions.

coalesce is a tranformation

if you want to decrease the number of partitions
==
=======

coalesce or repartition

to decrease the number of partitions coalesce is preferred as it will try to minimize the shuffling.

N1 - p1 , p2

N2 - p3 , p4

N3 - p5 , p6

N4 - p7 , p8

rdd1 has 16 partitions

rdd1.repartition(8)

repartition has a intention to have final partitions of exactly equal size and for this it has to go through complete shuffling.

rdd1.coalesce(8)

coalesce has a intention to minimize the shuffling and combines existing partitions on each machine to avoid a full shuffle.

if you want to increase the number of partitions
===

repartition

spark practical - 16
======================

cache & persist

consider if you have a rdd which you have generated by doing bunch of transformations..

rdd1

rdd2

rdd3

rdd4.cache

rdd5

rdd5.collect

rdd5.count

cache and persist both have same purpose.

if you want to reuse the results of existing rdd. then you can use them.

speed up the applications that access the same rdd multiple times.

an rdd that is not cached is re evaluated again each time an action is invoked.

the difference is that cache will cache the rdd in memory.

however persist comes with various storage levels.

in memory
in disk
off heap

persist() this is equivalent to cache() and this will persist in memory.

persist(StorageLevel.DISK_ONLY)

MEMORY_ONLY - data is cached in memory.

non serialized format.

serialization is in bytes format. - it takes less storage
non serialized means its in object format. - it takes more storage

DISK_ONLY - data is cached on disk in serialized format. in the form of bytes. and takes less storage.

MEMORY_AND_DISK - data is cached in memory. if enough memory is not available, evicted blocks from memory are serialized to disk.

this mode of operation is recommended when re-evaluation is expensive and memory resources are scarse.

OFF_HEAP - blocks are cached off-heap.

outside the JVM, grabing a piece of raw memory outside the executor.

problem with storing the objects in jvm is that it uses garbage collection for freeing up. garbage collection to free up the space is a time taking process.

but off_heap is a unsafe thing as we have to deal with raw memory outside your jvm.

Block eviction
=================

consider the situation that some of the block partitions are so large (skew) that they will quickly fill up the storage memory used for caching.

when the storage memory becomes full, an eviction policy will be used to make up the space for new blocks.

LRU (least recently used)

serialization increases the processing cost but reduces the memory foot print.

non serialized the processing can be a bit fast but it uses more memory foot print.

DISK_ONLY - SERIALIZED
MEMORY_ONLY - BY DEFAULT THESE ARE NON SERIALIZED
MEMORY_AND_DISK - BY DEFAULT THESE ARE NON SERIALIZED
OFF_HEAP - SERIALIZED
MEMORY_ONLY_SER - SERIALIZED
MEMORY_AND_DISK_SER - SERIALIZED

MEMORY_ONLY_2 - THIS NUMBER 2 INDICATES 2 REPLICAS STORED ON 2 DIFFERENT WORKER NODES.

REPLICATION IS USEFUL FOR SPEEDING UP RECOVERY IN CASE ONE NODE OF THE CLUSTER FAILS.

rdd.toDebugString is to check the lineage graph.
and we need to read it from bottom to top.

if we use cache() and we dont have enough memory then it will skip caching it. it wont give any error.

persist(StorageLevel.MEMORY_AND_DISK)

DO NOT CACHE OR PERSIST YOUR BASE RDD.

spark practical - 17
=====================

1. difference between a DAG and a lineage

2. how to create a jar for your spark project and run the jar.

lineage is nothing but the dependency graph.

shows dependency of various rdd. it's a logical plan.

DAG is a acyclic graph.

jobs, stages and tasks.

how to run the jar
===================

spark-submit --class <class_name> <complete path of the jar>

```
./spark-submit --class WordCount
/Users/trendytech/Desktop/wordcount.jar
```

spark practical - 18
======================

Top movies

1. Atleast 1000 people should have rated for that movie..

2. average rating > 4.5

(101,Toy Story)
(101,4.7)

//input
(101,(Toy Story,4.7)) x

//output

x._2._1
Toy Story

spark practical - 19
=====================

Spark Ecosystem

Structured API's - dataframes, datasets and spark sql
spark streaming to process real time data.

Map vs Map partition

this rdd has 10000 rows and 10 partitions.

each partition holds 1000 records.

func1

rdd.map(func1) - 10000 times

rdd.mapPartitions() - 10 times

http://apachesparkbook.blogspot.com/2015/11/mappartiti
on-example.html

what all transformations and actions you have used?

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

===========

Spark core
===========

rdd's

Structured API's
=================

Dataframes and datasets

A dataframe is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relation database.

employee - 100000

dataframe

4 partitions

each parition holding 25k records..

richer optimizations are possible.

rdd till now.. spark 1 style..

spark 2 style of writing code.

are dataframes and datasets launched in spark 2?

df/ds were available in spark 1 as well.

from spark 2 onwards we got a better support for these 2 and both of these are merged into a single API datasets api..

sc (spark context)

separate separate context for each and every thing. spark context ,hive context, sql context spark session - is a unified entry point of a spark application.

it provides a way to interact with various spark functionilty with a lesser number of constructs.

Instead of having a spark context , hive context, sql context now all of it is encapsulatted in a spark session.

spark session is a singleton object

.builder() it returns a builder object

this will help us to configure the spark session

.appName
.master

treat your spark session like your driver.

spark.stop()

Structured API's session - 2

=================================

spark session

Structured API's session - 3
=================================

number of jobs is ideally equal to the number of actions.

each job will have stages - number of shuffle boundaries

each stage will have tasks - number of partitions

Structured API's session - 4
=================================

whenever we are working with dataframes or datasets
we are dealing with higher level programming constructs..

when we were working with raw rdd that was a low level code.

your spark compiler will convert your high level code (dataframe code) to low level rdd code.

driver   executors

driver will convert your high level code into low level rdd code and then it will send the low level code to the executors..

```
rdd.map( x => {
        //anonymous function (low level code)
})
```

Structured API's session - 5
=========================

rdd vs dataframe vs datasets

rdd
====

when we deal with raw rdd. we deal with low level code

map
filter
flatMap
reduceByKey

this low level code is not developer friendly.
rdd lacks some of the basic optimizations.

dataframes
==========

spark 1.3 version

higher level constructs which makes the developer life easy.

challenges with dataframes
==========================

1. dataframes do not offer strongly typed code..
   type errors wont be caught at compile time rather we get surprise at runtime.

2. developers felt that there flexibility has become limited..

that the dataframes can be converved to rdd

df.rdd (whenever we want more flexibility and type safety)

1. this conversion from dataframes to rdd is not seamless.

2. if we work with raw rdd by converting dataframe to rdd. we will miss out on some of the major optimizations.

catalyst optimizer / tungsten engine

Dataset API
============

spark 1.6

1. compile time safety

2. we get more flexibility in terms of using lower level code.

conversion from dataframes to datasets is seamless.

we wont lose on any of the optimizations.

before spark 2 both dataframes and datasets are 2 different things..

Spark 2

in spark 2 they merged these 2 into a single unified spark dataset API (Structured API)

Dataframe is nothing but a Dataset[Row]

Row is nothing but a generic type which will be bound at runtime.

in case of dataframes the datatypes are bound at runtime however Dataset[Employee]

the type will be bound at compile time.

Dataset[Row] -> dataframe (type errors are caught at runtime)

Dataset[Employee] -> dataset (compile time type safely)

how to convert dataframe to a dataset

if we replace generic Row with specific object then it becomes a Dataset.

Structured API's session - 6
=============================

with dataframes you will get runtime errors.

with datasets you will be able to catch errors at compile time.

we saw that types are resolved at runtime in case of data frames

now let us convert our dataframe to a dataset

how to convert a dataframe to a dataset?

this is where you require a case class.

create case class

and create dataset[OrdersData]

Dataframes are more preferred over Datasets..

to convert from dataframes to datasets there is an overhead involved and this is for casting it to a particular type

Serialization - converting data into a binary form..

when we are dealing with dataframes then the serialization is managed by tungsten binary format.. (encoders) when we are dealing with datasets then the serialization is managed by java serialization (slow)

using datasets will help us to cut down on developer mistakes..
but it comes with an extra cost of casting and expensive serialization.

Structured API's session - 7
================================

1. Read the data from a Data Source and create a dataframe/datasets.

- external data source (mysql database, redshift, mongodb)
- internal data source (hdfs, s3, azure blob, google storage)

so we have the flexibility in spark to create a dataframe directly from an external data source.

spark is very good at processing but is not that efficient at ingesting data.

spark gives you a jdbc connector to ingest the data from mysql database directly.

sqoop to get the data to my hdfs and then I will load the data from hdfs to spark dataframe.

use tools like sqoop which are specially made for data ingestion to get your data from external data source to internal data source.

2. Performing a bunch of transformations and actions.

transformations/actions using higher level constructs.

3. writing the data to the target (Sink)

internal/external

Read the data from source (internal data source)

do bunch of transformations/actions

dump the output to the sink

3 read modes
================

1. PERMISSIVE (it sets all the fields to null when it encouters a corrupted record) - default

 _corrupt_record

2. DROPMALFORMED (will ignore the malformed record)

3. FAILFAST (whenever a malformed record is encountered a exception is raised)

file formats
===============
csv

json

parquet

read modes
===========
PERMISSIVE
DROPMALFORMED
FAILFAST

SCHEMA
========
1. INFER (INFER SCHEMA OPTION)
2. IMPLICIT (PARQUET, AVRO ETC..)
3. EXPLICIT

Structured API's session - 8
===============================

3 options to have the schema for a dataframe.

1. infer schema (inferSchema as true)
2. implicit schema (reading parquet, avro etc...)
3. explicit schema (manually defining the schema)

explicit schema
==================

1. Programatically using StructType
2. DDL String

Programatic approach
======================

```
val ordersSchema = StructType(List(
StructField("orderid", IntegerType),
StructField("orderdate", TimestampType),
StructField("customerid", IntegerType),
StructField("status", StringType)
))
```

Int - scala
IntegerType - spark

Long - scala

LongType - spark

Float	FloatType
Double	DoubleType
String	StringType
Date	DateType
Timestamp	TimestampType

DDL String
===========

val ordersSchemaDDL = "orderid Int, orderdate String, custid Int, ordstatus String"

How to convert dataframe to dataset.. it is by using a case class..

dataframe is a dataset[Row]

dataset is a dataset[Orders]

Structured API's session - 9
==============================

1. we read the data from a source and create a dataframe

2. we do bunch of transformations and actions - processing

3. we write the output to target location - sink

saveModes
==========

1. append (putting the file in the existing folder)

2. overwrite (first delete the existing folder, and then it will create a new one)

3. errorIfExists (will give error if output folder already exist)

4. ignore (if folder exist it will ignore)

normally when we are writing a dataframe to our target.

then we have few options to control the file layout

spark file layout
===================

1. Number of files and file size

2. partitioning and bucketing

3. sorted data - sortBy

Note: number of output files is equal to the number of partitions in your dataframe.

1. simple repartiton
=====================

it can help you increase the parallelism df.repartition(4)

with a normal repartition you wont be able to skip some of the partitions for performance improvement

partition pruning is not possible.

2. partitionBy

is equivalent to your partitioning in hive.

it provides partition pruning

3. bucketBy(4,"order_id")

maxRecordsPerFile

csv, parquet, json ...

avro is external and not supported by default.

we need to add a jar.

spark 2.4.4   2.11

spark avro 2.4.4 2.11

Structured API's session - 11
==================================

sometimes we have a requirement to save the data in a persistent manner in the form of table.

when data is stored in the form of table then we can connect tableau, power bi etc... for reporting purpose.

table has 2 parts
====================

data	metadata
spark warehouse	catalog metastore
spark.sql.warehouse.dir	in memory (on terminating application it is gone)

we can use hive metastore to handle spark metadata

spark hive 2.4.4 2.11

bucketBy works when we say saveAsTable

Structured API's session - 12
=================================

1. Dataframe reader - taking the data from source

2. tranformations to process your data

3. Dataframe writer - to write your data to target location

Transformations
=================

1. Low level Transformations
   ==========================

map

filter

groupByKey

Note: we can perform low level tranformations using raw rdds

some of these are even possible with dataframes and datasets..

2. High level Transformations
   ===========================

   select

   where

   groupBy

Note: These are supported by Dataframes and Datasets..

since this is an unstructured file.

I will load this file as a rdd (raw rdd)

each line of the rdd is of string type..

use a map transformation which is low level transformation.

input to the map tranformation is:

1 2013-07-25   11599,CLOSED

output:

1,2013-07-25,11599,CLOSED

In my map tranformation I will use a regular expression

I will associate the output with the case class

1,2013-07-25,11599,CLOSED

so that we have structure associated..

input to the map is a raw line.

output from the map will be structured line.

if we have schema associated/structure associated we can convert our rdd to a dataset

rdd

then we imposed structure on top of rdd

on structured rdd we call .toDS method to convert it to dataset..

I can do whatever higher level transformations I want to use.

Idea is to give structure to your data and then use high level transformations.

do this as early as possible..

Structured API's session - 13
==================================

how to refer a column in a dataframe/dataset

1. column string
================

ordersDf.select("order_id","order_status").show

2. column object
=================

ordersDf.select(column("order_id"),col("order_date"),$"order_customer_id", 'order_status).show

ordersDf.select(column("order_id"),col("order_status")).show

column
col

both of these can be used in pyspark, spark with scala.

scala specific
===============

$"order_id"
'order_id

syntactic sugar but available only for scala

we cannot mix both columns strings and column object in the same statement.

column expression
===================

Note: we cannot mix columns strings with column expression

nor we can mix column object with column expression

column string - select("order_id")

column object - select(column("order_id"))

column expression - concat(x,y)

there is a way to convert column expression to a column object

Structured API's session - 14
================================

UDF (user defined functions)

Structured API's

whenever we want to add a new column we use .withColumn

df.withColumn("adult",)

column object expression udf
==============================

df.withColumn("adult",parseAgeFunction(col("age")))

basically we register the function with the driver.

the driver will serialize the function and will send it to each executor.

sql/string expression udf
=============================

session takeaways
===================

if you want to add a new column to a dataframe then use .withColumn transformation

how to convert dataframe to dataset , it is by using case class val ds = df.as[Person]

how to convert dataset to a dataframe, using .toDF()
val df1 = ds.toDf()

creating our own user defined function is spark

1. column object expression
it is not registered in catalog.

2. sql expression (easier)
the function is registered in catalog.
so that we will be able to use it with spark sql also.

whenever we register a UDF with driver.

driver will serialize it (convert it into bytes) and will send it to each executor.

Structured API's session - 15
=============================

```
1,"2013-07-25",11599,"CLOSED"
2,"2014-07-25",256,"PENDING_PAYMENT"
3,"2013-07-25",11599,"COMPLETE"
4,"2019-07-25",8827,"CLOSED"
```

1. I want to create a scala list - done
2. from the scala list I want to create a dataframe
   orderid, orderdate, customerid, status - done

3. I want to convert orderdate field to epoch timestamp (unixtimestamp) - number of seconds after 1st january 1970 - done

4. create a new column with the name "newid" and make sure it has unique id's
   - done

5. drop duplicates - (orderdate , customerid) - done

6. I want to drop the orderid column - done

7. sort it based on orderdate -

if I want to add a new column or if I want to change the content of a column I should be using .withColumn

Structured API's session - 16
================================

Aggregate transformations

1. Simple aggregations

2. grouping aggregates

3. window aggregates

order_data.csv it is 46 mb file

Simple aggregations
=====================

when after doing the aggregations we get a single row.

total number of records, sum of all quantities.

grouping aggregates
=====================

in this we will be doing a group by

in the output there can be more than one record.

window aggregates
===================

so we will be dealing with a fixed size window.

Simple aggregations
=====================

1. load the file and create a dataframe. I should do it using standard dataframe reader api. - done

Simple Aggregate

totalNumberOfRows, totalQuantity, avgUnitPrice, numberOfUniqueInvoices

2. calculate this using column object expression - done

3. do the same using string expression - done

4. Do it using spark sql - done.

Structured API's session - 17
===========================

Grouping Aggregates

group the data based on Country and Invoice Number

I want total quantity for each group, sum of invoice value

1. do it using column object expression - done

2. do it using string expression - done

3. do it using spark sql - done

Structured API's session - 18
==============================

simple aggregations

grouping aggregations

window aggregations..

1. parition column - country
2. ordering column - weeknum
3. the window size - from 1st row to the current row

Structured API's session - 19

====================================

there are 2 kind of joins

1. Simple join (Shuffle sort merge join)

2. Broadcast join

we have 2 datasets
=======================

orders - order_customer_id

customers - customer_id

kind of joins which are possible
===

1. inner (matching records from both the tables)

we wont see the customer who never placed a order.

2. outer - matching records + non matching records from left table + non matching records from right table

3. left - matching records + non matching records from the left table

4. right - matching records + non matching records from the right table

Lets a some customers never placed a order.. but I do not want to miss on these customers details.

Structured API's session - 20
================================

1. showcasing how your code can lead to ambiguous column names.

this happens when we try to select a column name which is coming from 2 different dataframes..

how to solve this issue..
============================
there are 2 ways to solve this problem

1. this is before the join

you rename the ambiguous column in one of the dataframe

.withColumnRenamed("old_column_name","new_column_name")

2. once the join is done we can drop one of those columns.

.drop
======================

2. how to deal with null's

problem statement
==================
whenever order_id is null show -1

coalesce

Structured API's session - 21
==============================

internals of a normal join operation

shuffle..

Simple join involves - Shuffle sort merge join

executor1 - node 1
==========
orders
15192,2013-10-29 00:00:00.0,2,PENDING_PAYMENT
33865,2014-02-18 00:00:00.0,2,COMPLETE

(2,{15192,2013-10-29
00:00:00.0,PENDING_PAYMENT})
(2,{33865,2014-02-18 00:00:00.0,COMPLETE})

customers
3,Ann,Smith,XXXXXXXXX,XXXXXXXXX,3422 Blue
Pioneer Bend,Caguas,PR,00725

(3,{Ann,Smith,XXXXXXXXX,XXXXXXXXX,3422 Blue
Pioneer Bend,Caguas,PR,00725})

it will write the output into the exchange.

exchange is nothing but like a buffer in the executor..

from this exchange spark framework can read it and do the shuffle.

exchange

executor2 - node 2
==========

2,Mary,Barrett,XXXXXXXXX,XXXXXXXXX,9526 Noble Embers Ridge,Littleton,CO,80126

orders
35158,2014-02-26 00:00:00.0,3,COMPLETE
15192,2013-10-29 00:00:00.0,2,PENDING_PAYMENT

exchange

executor3 - node 3
==========

exchange

15192,2013-10-29 00:00:00.0,2,PENDING_PAYMENT
2,Mary,Barrett,XXXXXXXXX,XXXXXXXXX,9526 Noble Embers

all the records with the same key go to the same reduce exchange.

Structured API's session - 22
==========================

1. simple - shuffle
2. broadcast - this does not require a shuffle.

whenever we are joining 2 large dataframes then it will invoke a simple join and shuffle will be required.

when you have one large dataframe and the other dataframe is smaller. in that case you can go with the broadcast join.