Memory Management in Apache Spark
==========

Executor memory - 2 GB

--executor-memory 2G (during spark-submit)

2GB (heap memory)
max (10% or 384 mb) - overhead memory (non heap memory)

non heap is outside JVM.

```
spark-submit \
--deploy-mode cluster \
--master yarn \
--num-executors 1 \
--executor-cores 4 \
--executor-memory 8G \
--conf spark.dynamicAlllocation.enabled=false \
prog1.py
```

Required executor memory (8192),
overhead (819 MB),
and PySpark memory (0 MB)
is above the max threshold (8192 MB) of this cluster! Please check the values of

yarn.scheduler.maximum-allocation-mb

yarn.nodemanager.resource.memory-mb

2GB - executor (jvm)

300 mb - reserved for spark engine

1.7 GB
- 60% unified area (Storage memory & execution memory) - 1 GB
- 40% user memory - 700 mb

384 mb - overhead memory
300 mb - reserved
1 gb - storage & execution memory
- 500 mb - storage memory
- 500 mb - execution memory

700 mb - user memory

384 mb - overhead memory (outside jvm)

overhead memory - VM related overheads
reserved - 300 mb (spark engine)
storage memory - cache,persist
execution memory - shuffle, sort, join, aggregation
user memory - rdd related operations, user defined datastructures

execution memory can take some memory from storage memory provided it is free

storage memory can take some memory from execution memory provided it is free

1 GB - storage and execution (unified area)
-500 mb (storage)
-500 mb (execution)

caching 700 mb data

if execution requires memory it can evict storage memory to a certain threshold.

1 GB - storage and execution (unified area)
-500 mb (storage)
-500 mb (execution)

the execution requires 700 mb memory

300 mb of storage memory is available - caching

storage cannot evict the execution

execution can evict storage
but storage cannnot evict the execution.

384 - overhead memory
300 mb is reserved
1000 - 300 = 700 mb

420 mb (unified area)
280 mb (user memory)

4 Gb of executor memory

4 cpu cores

28% of it is storage memory
28% of it is execution memory

2.4 GB / 4 cpu cores

per core...

600 mb

=======

off heap memory - outside the JVM (no garbage collection)

pyspark memory - if we are using python related libraries, then it initiates a python worker and it will need some memory. (scala, java)

heap memory (JVM) - when we create some objects in memory...
garbage collection takes times...

2 GB off heap memory -

2 gb executor memory - 912 mb (unified area)

storage and execution memory - 2 gb + 912 mb

overhead - spark.executor.memoryOverhead
heap - spark.executor.memory
off-heap - spark.memory.offHeap.size
pyspark - spark.executor.pyspark.memory

you requested for X - spark.executor.memory

X - 300 mb (.6) unified area
(.4) user memory

spark.memory.fraction = .6
spark.memory.storageFraction = .5

```
spark-submit \
--deploy-mode cluster \
--master yarn \
--num-executors 1 \
--executor-cores 4 \
--executor-memory 8G \
--conf spark.dynamicAlllocation.enabled=false \
--conf spark.memory.fraction=.8 \
prog1.py
```

==========

Sort Aggregate vs Hash Aggregate
==========

grouping based on customer_id , month -
sorting based on month

3807, august, 300 (I also want to sort it based on month)

the query which took 1.2 min - sort Aggregate

the query which took 15 seconds - hash aggregate

question 1 - why in query 1 - sort aggregate was used and in query 2 - hash aggreagate was used?

question 2 - why the query 2 which has hash aggregate ran faster.

10 initial partitions

1	january
1	january
2	febuary
1	january
2	febuary
3	january
1	january

1 january
1 january

1 january
1 january
2 febuary
2 febuary
3 january

1 january 4
2 febuary 2
3 january 1

1000 numbers

O(nlogn)

1000 * 10 = 10000

2000 numbers

2000 * 11 = 22000

Hash Aggregate

1 january
1 january
2 febuary
1 january
2 febuary
3 january
1 january

hash table - off heap memory

key value
1-january 2,01 (string is immutable)

sort aggregate - O(nlogn)
hash aggreage - O(n)

key value
1-january 2,1

File formats & Compression Techniques
==========

fundamental question when designing a solution architecture - how my data will be stored.

we talk about 2 things

1. file formats
2. compression techniques

==========

why do we need different file formats

=> we want to save storage
=> we want faster processing
=> we want less time for I/O operations

There are a lot of choices available for file formats

=> faster reads
=> faster writes
=> splittable
=> schema evolution support
=> should support advanced compression techniques
=> most compatible platform

The file formats have been divided in 2 broad categories -

1. Row based file formats

=> writing is easy, faster writes

select order_id, customer_id from orders

in a row based file format when you want to read subset of columns then you have to read the entire row...

select * from orders

=> reading a subset of columns is not efficient

=> less compression

2. column based file formats -

select order_id, customer_id from orders

=> efficient reads

=> slower writes

=> very good compression

==========

=> row based

=> column based

text file formats
==========

for simplicity we generally practise with text file formats

csv - raw files

xml, json - slight structure

human readable

csv file - everything is stored as a text

90123490 - string (16 bytes)

if this would have been stored as a integer - 4 bytes

so text files will take a lot of storage as everything is stored as text/string.

568

789

this required to convert from string to int so that we can perform necessary operations..

and this conversion will take time..

processing on text files can be time taking because of type conversions.

data size is also huge, IO operations will also take a lot of time

when we talk about a csv file format

=> storage (more space)
=> processing (slow and takes more time)
=> I/O (a lot of I/O is involved which takes more time)

XML & JSON - whatever bad we talked about CSV is still valid and there are some extra issues...

file size is really bulky
+
not splittable

[{"order_id":1,"order_date":"2013-07-25","customer_id":11599,"order_status":"CLOSED"},{"order_id":2,"order_date":"2013-07-25","customer_id":256,"order_status":"PENDING_PAYMENT"},{"order_id":3,"order_date":"2013-07-25","customer_id":12111,"order_status":"COMPLETE"},{"order_id":4,"order_date":"2013-07-25","customer_id":8827,"order_status":"CLOSED"}

=========

csv, json , xml

Specialized file formats
==========

3 main file formats

avro, orc, parquet

avro - row based file format

orc (optimized row columnar),
parquet - column based file format

avro - general purpose (row based)

orc - hive (columnar)

parquet - spark (columnar)

all of the above file formats are splittable
all of the above file formats support schema evolution
we can use any kind of compression techniques with the above file formats

orders

4 columns
order_id order_date customer_id order_status

5 columns
order_id order_date customer_id order_status, order_amount

when talking about csv, json (text file formats) we are restricted to use certain compression techniques only...

but with orc, parquet, avro we can use any kind of compression..

lzo, snappy

orc, parquet, avro - along with data the metadata is also embedded, the compression codec is also mentioned..

avro is row based -

=> it will support faster writes but slower reads
=> the schema for avro is stored along with data, self describing
=> compression codec will be mentioned in the metadata
=> avro is quite a general file format
=> schema evolution
=> avro can be best fit when storing data in landing zone of a datalake
=> splittable

==========

ORC and Parquet
==========

=> column based file formats
=> not efficient for writing

=> optimized for reads
=> highly efficient for storage
=> orc best fits with hive
=> parquet is best fit with spark

we get some lightweight compression techniques

dictionary encoding
bit packing
delta encoding
run length encoding

apart from above we can get generalized compression like snappy, lzo, gzip, bzip2

dictionary encoding
==========

1 million records - 50 different states

it will create a dictionary

50 entries
[karnataka,1]
[andhrapradesh,2]
[maharashtra,3]
.
.

"The Democratic Republic of Congo"

Bit packing
==========

25 entries in your dictionary

16 8 4 2 1
1 0 0 0 1

4 bytes for integer - 32 bits

"The Democratic Republic of Congo" 17 - 10001

Delta Encoding
==========

timestamp

12:49:00 00:00:00
12:49:01 00:00:00
12:49:02 00:00:00

12:49:00 00:00:00
1
2

run length encoding
==========

sssssssggggggggghhhhhhhhhh

s8g7h9

lets say you have a column with 1 million values, but all the values are 0

2 numbers

(0,1000000)

parquet takes 87% less space & queries run 34x faster (1 TB of data, S3)

orc even takes lesser space than parquet...

orc and parquet supports schema evolution

both are splittable

csv - 128 mb
avro - 30 mb
parquet - 13 mb
orc - 7 mb

less I/O

less storage

header - par1

body - row groups -> column chunks -> pages

footer - metadata

you have a folder with 4 parquet files

500 mb - 80000 rows

1st row group - 20000 -
column chunk1 - orderid -> multiple pages - 1 mb (20 pages which hold actual data with some metadata)
min and max orderid in a page
column chunk2 - orderdate - 40 pages
column chunk3 - customerid
column chunk4 - orderstatus

2nd row group - 20000
column chunk1 - orderid
column chunk2 - orderdate
column chunk3 - customerid
column chunk4 - orderstatus

3rd row group - 20000
column chunk1 - orderid
column chunk2 - orderdate
column chunk3 - customerid
column chunk4 - orderstatus

4th row group - 20000
column chunk1 - orderid
column chunk2 - orderdate
column chunk3 - customerid
column chunk4 - orderstatus

a parquet file -> row groups (128 mb) -> column chunks -> pages
metadata
metadata

row group 1 -> 20000 records
min order_id 28
max order_id 20045

select * from orders where order_id = 45

it will help you to skip the data...

predicate pushdown...

row group - 5 records

column chunks

1 2 3 4 5 | 2013-07-25 00:00:00.0 2013-07-25 00:00:00.0 2013-07-25 00:00:00.0 2013-07-25 00:00:00.0 2013-07-25 00:00:00.0 | 11599 256 12111 8827 11318 | CLOSED PENDING_PAYMENT COMPLETE CLOSED COMPLETE

order_id min 1
order_id max 5
order_date min
order_date max
customer_id min 256
customer_id max 12111

predicate pushdown
column pushdown

select order_id,customer_id from orders

aws - athena
azure - synapse serverless

you are charged for the amount of data scanned...

1 tb of data scanned $5

20 tb data - 5 gb of it

header - par1

body -> row groups (128 mb) -> column chunks -> pages

footer - metadata

==========

schema evolution
==========

input folder
orders1
orderid long, order_date date

orders2
orderid long,order_date date, customerid long

orders3
orderid long,order_date date,order_status string,customer_id long

/public/trendytech/datasets/parquet-schema-evol-demo1/parquet

/public/trendytech/datasets/parquet-schema-evol-demo/csv

orders1.csv
orderid, orderdate
1,2013-07-25 00:00:00.0
2,2013-07-25 00:00:00.0

orders2.csv
orderid, orderdate, customerid
3,2013-07-25 00:00:00.0,12111
4,2013-07-25 00:00:00.0,8827

orders3.csv
orderid, orderdate, customerid, order_status
5,2013-07-25 00:00:00.0,11318,COMPLETE

6,2013-07-25 00:00:00.0,7130,COMPLETE

orders4.csv
orderid, orderdate, order_status, customerid
5,2013-07-25 00:00:00.0,COMPLETE,11318
6,2013-07-25 00:00:00.0,COMPLETE,7130

==========

compression techniques
==========

3 copied in your datalake

you want to save some storage space...

1 tb file - csv

200 gb - compressed

=> we want to save storage space
=> we want to reduce the I/O cost

compression comes with some additional cost

cpu cycles + time

4 compression techniques
==========

1. snappy - optimized for speed and gives moderate level of compression
very fast
moderate compression
parquet, orc - snappy is the default compression technique.
snappy when used with csv or text file formats it is not splittable.
orc, avro, parquet - container based file formats so that it can be splittable.

2. lzo - optimized for speed just like snappy
moderate compression.
it requires a separate licence as its not generally distributed along with hadoop
it is splittable

3. gzip -

good compression
slow in processing
not splittable, you should use it with container based file formats.
gzip provides 2.5 times the compression offered by snappy
1 gb - 100 mb (1 partition)
- 200 mb (2 partitions)

its better to decrease the partition size

4. bzip2 - optimized for storage and provides very very good compression
super slow in terms of processing
splittable
bzip2 will compress around 9% better than gzip, but it makes things 10 times slower than gzip
archival purpose

	compression-ratio	Speed	splittabily
snappy	moderate	fast	no
lzo	moderate	fast	yes
gzip	good	moderate	no
bzip2	very good	slow	yes

csv raw - 1.1 gb (9 partitions)

csv + snappy - 313 mb (1 partition) not splittable

csv + gzip - 168 mb (1 partition) not splittable

csv + bzip2 - 139 mb (2 partitions) splittable

parquet + snappy - 109 mb (2 partitions) splittable

parquet + gzip - 101 mb (2 partitions) splittable

orc + snappy - 59 mb (2 partitions) splittable

orc + lzo - 52 mb (2 partitions) splittable

=>speed
=>compression ratio

moderate level of compression with faster speed is a good thing...
really good compression with slow speed can be good for archival of data.