Disclaimer: **These slides are copyrighted and strictly for personal use only**

- This document is reserved for people enrolled into the [Ultimate Big Data Masters Program (Cloud Focused) by Sumit Sir]

- **Please do not share this document,** it is intended for personal use and exam preparation only, thank you.

- If you've obtained these slides for free on a website that is not the course's website, please reach out to legal@trendytech.in . Thanks!

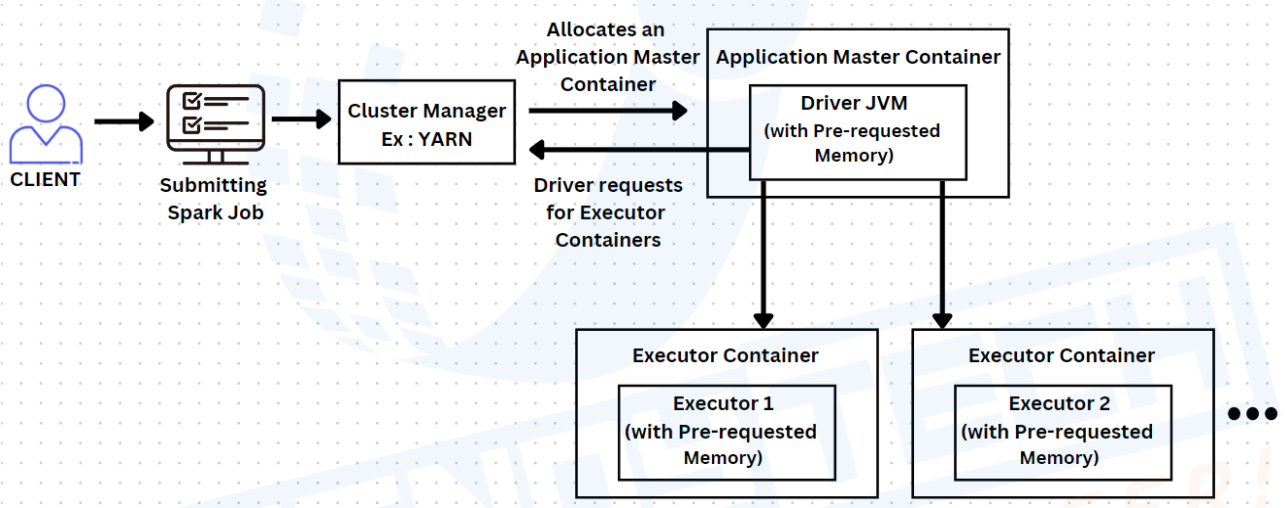
- All the Best and Happy Learning!

Memory Management in Apache Spark

Requesting for executor memory

Example : Requesting for executor memory while submitting the job through spark-submit command

```
spark-submit \
--deploy-mode cluster \
--master yarn \
--num-executors 1 \
--executor-cores 4 \
--executor-memory 2G \
--conf spark.dynamicAllocation.enabled = false \
program1.py
```



Memory allocated across various segments

Heap Memory / JVM Memory - Example : requested for 2GB

Overhead memory (outside the JVM):

max(10% of executor memory or 384MB)

Off-Heap memory(outside the JVM):

Memory managed directly by the application but outside the regular Java heap. It is used for specific data structures and caching purposes. It is allocated for storage and execution memory.

Eg: Consider a scenario where memory allocated for executor memory is 2G

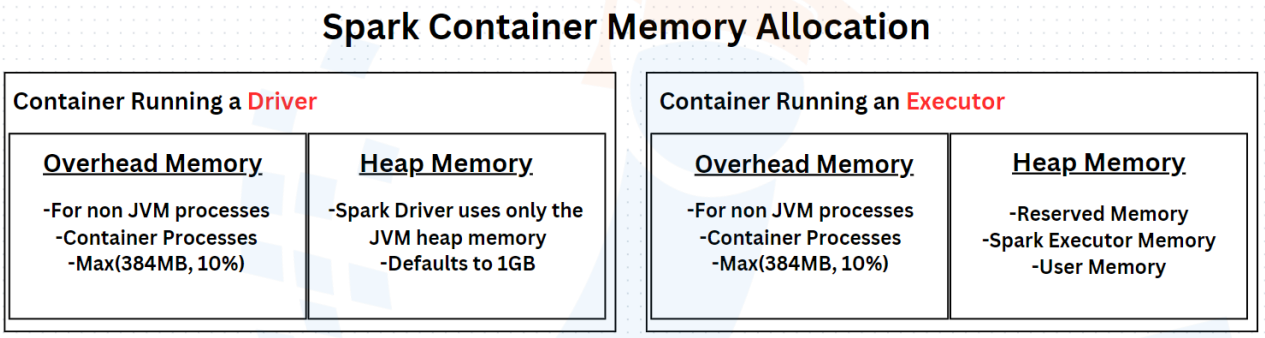
So, Unified memory(storage and execution memory) = 912MB

now, if we request for off heap memory of 2G

Eg: config('spark.memory.offHeap.size', '2G')
config('spark.memory.offHeap.enabled', True)

Note: Until & unless the offHeap isn't enabled it won't be considered/added to unified memory.

So, now the unified Area(storage + execution memory) = 912MB(unified area) + 2GB(off-Heap Memory)



Spark's Memory Centric Architecture -

Driver Memory Allocation

Say 2GB driver memory was requested using the conf " spark.driver.memory = 2GB "

Driver JVM Memory = 2GB (as requested using spark.driver.memory = 2GB)	Driver Overhead Memory = Max(384MB , 10% of driver memory) i.e., Max(384MB,204.8MB) = 384MB The overhead memory value can be modified using the conf - spark.driver.memoryOverhead

Total Memory = 2GB + 384MB

Memory Types

Overhead Memory - Memory allocated for VM related overhead.

Reserved Memory - Memory allocated for Spark Engine.

Storage Memory - Memory for Cache and Persist operations.

Execution Memory - Temporary memory required for execution of Join | Sort | Shuffle | Aggregations ...

User Memory - Memory for RDD related operations, and user defined data-structures

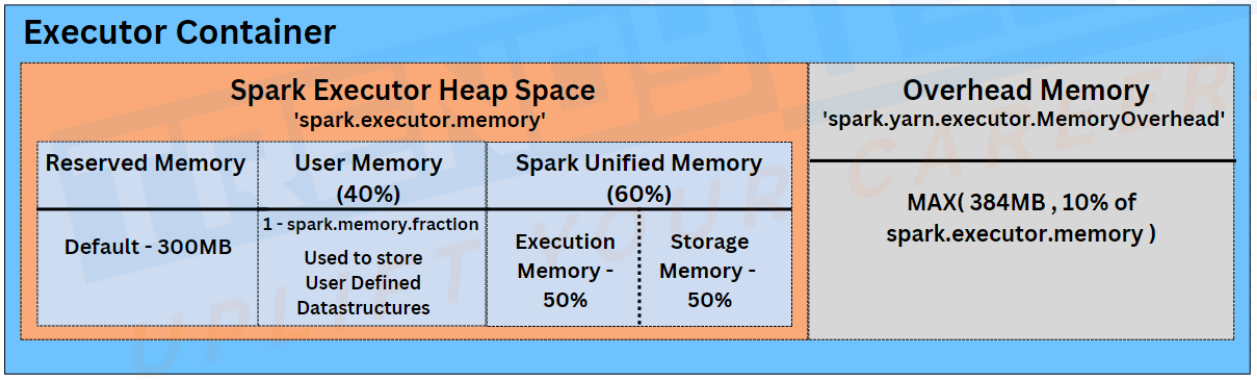
How is the executor memory divided

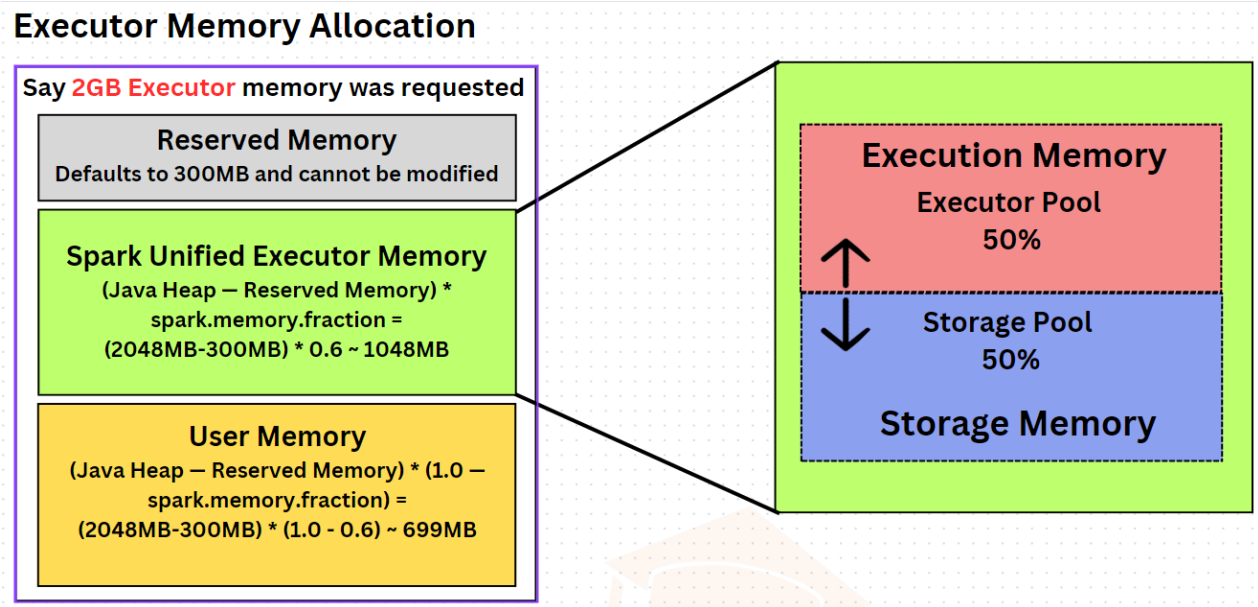
1. Memory Reserved for Spark Engine - 300MB
2. Remaining 1.7GB (2GB - 300MB)

   60% for Unified Area(Storage and Execution memory)

   40% for User Memory

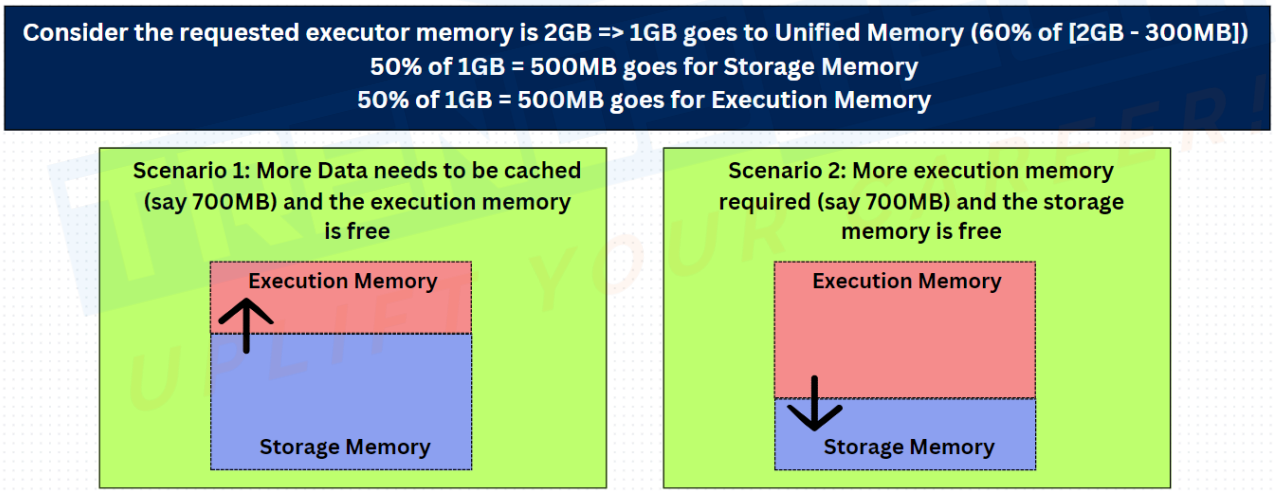
3. Overhead / Off-Heap Memory - Max(10% of executor memory, 384MB)





The demarcation between the Execution Memory and the Storage Memory is not rigid. Based on the requirement and available free memory, the execution memory can extend into the storage memory and vice-versa. This flexibility allows for effective utilization of the available memory.

Scenarios explaining the Eviction process in case of Execution and Storage memory



Key Points :

- **Eviction process happens when there is no free memory available.**
- **Execution can evict Storage memory within the threshold limits.**
- **However, Storage cannot evict Execution memory.**

Off-Heap Memory and PySpark Memory

Off Heap Memory - It is the overhead memory available outside JVM and does not require garbage collection for cleaning up the objects, thereby making it faster in comparison to the heap memory inside JVM that requires garbage collection to free the memory of unused objects.

PySpark Memory - When using python related libraries, then a python worker will be initiated which will require some memory for execution. Required only for Python based Spark code (Not required for Scala or Java based spark code)

Configuration properties used to set the values for following categories of memory

overhead - spark.executor.memoryOverhead

heap - spark.executor.memory

offHeap - spark.memory.offHeap.size

pyspark - spark.executor.pyspark.memory

Other Important Configuration properties

spark.memory.fraction - To set the required value for spark executor's unified memory(default 0.6 i.e., 60%)

spark.memory.storageFraction - To set the required ratio of memory utilized between the Execution memory and Storage memory(default 0.5 i.e., 50%)

Sort Aggregate Vs Hash Aggregate

Consider the following Example :
-Create order_df by loading the Orders dataset
-Create a temporary view on orders_df
-Group based on order_id and month
-Sort based on month

Scenario 1 (Execution time~ 1min)

```
spark.sql("""select customer_id, date_format(order_date,
'MMMM') as order_month,
count(1) as total_cnt, first(date_format(order_date,'MM')) as
month_num from orders
group by customer_id, order_month order by
month_num""").write.format("noop").mode("overwrite").save()
```

Scenario 2 (Execution time~ 15sec)

```
spark.sql("""select customer_id, date_format(order_date,
'MMMM') as order_month,
count(1) as total_cnt, first(int(date_format(order_date,'MM')))
as month_num from orders
group by customer_id, order_month order by
month_num""").write.format("noop").mode("overwrite").save()
```

Sort Aggregate takes place in Scenario 1 (Takes more time)

- First sorts the data and then aggregates . Sorting is a costly operation and takes a considerable amount of execution time.
- Time complexity of sorting is O(nlogn)

- If there are 1000 records, then

  Time taken to sort = 1000 * log(1000) = 1000 * 10 = 10000

  (Sorting is an exponential process)

```
[9]: spark.sql("""select customer_id, date_format(order_date, 'MMMM') as order_month,
     count(1) as total_cnt, first(date_format(order_date,'MM')) as month_num from orders
     group by customer_id, order_month order by month_num""").explain(True)

     == Parsed Logical Plan ==
     'Sort ['month_num ASC NULLS FIRST], true
     +- 'Aggregate ['customer_id, 'order_month], ['customer_id, 'date_format('order_date, MMMM) AS order_month#42, 'count(1) AS total_cnt#43, first
     ('date_format('order_date, MM), false) AS month_num#45]
        +- 'UnresolvedRelation [orders], [], false

     == Analyzed Logical Plan ==
     customer_id: bigint, order_month: string, total_cnt: bigint, month_num: string
     Sort [month_num#45 ASC NULLS FIRST], true
     +- Aggregate [customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))], [customer_id#2L, date_format(cast(ord
     er_date#1 as timestamp), MMMM, Some(America/Toronto)) AS order_month#42, count(1) AS total_cnt#43L, first(date_format(cast(order_date#1 as time
     stamp), MM, Some(America/Toronto)), false) AS month_num#45]
        +- SubqueryAlias orders
           +- Relation[order_id#0L,order_date#1,customer_id#2L,order_status#3] csv

     == Optimized Logical Plan ==
     Sort [month_num#45 ASC NULLS FIRST], true
     +- Aggregate [customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))], [customer_id#2L, date_format(cast(ord
     er_date#1 as timestamp), MMMM, Some(America/Toronto)) AS order_month#42, count(1) AS total_cnt#43L, first(date_format(cast(order_date#1 as time
     stamp), MM, Some(America/Toronto)), false) AS month_num#45]
        +- Project [order_date#1, customer_id#2L]
           +- Relation[order_id#0L,order_date#1,customer_id#2L,order_status#3] csv

     == Physical Plan ==
     *(3) Sort [month_num#45 ASC NULLS FIRST], true, 0
     +- Exchange rangepartitioning(month_num#45 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [id=#83]
        +- SortAggregate(key=[customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#52], functions=[count(1), fi
     rst(date_format(cast(order_date#1 as timestamp), MM, Some(America/Toronto)), false)], output=[customer_id#2L, order_month#42, total_cnt#43L, mo
     nth_num#45])
           +- *(2) Sort [customer_id#2L ASC NULLS FIRST, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#52 ASC NULLS FIRS
     T], false, 0
              +- Exchange hashpartitioning(customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#52, 200), ENSUR
     E_REQUIREMENTS, [id=#78]
                 +- SortAggregate(key=[customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto)) AS date_format(cast
     (order_date#1 as timestamp), MMMM, Some(America/Toronto))#52], functions=[partial_count(1), partial_first(date_format(cast(order_date#1 as time
     stamp), MM, Some(America/Toronto)), false)], output=[customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#
```

Hash Aggregate takes place in Scenario 2 (Takes relatively very less time)

- **Creates a hash table and keeps updating the table**
    1. **If a new key is encountered, it will be added to the hash table.**
    2. **If an existing key is encountered, it will add to the value of the key in the hash table.**
- **Time complexity of Hash aggregate is O(n)**
- **It requires an additional memory for the hash table creation.**

```
spark.sql("""select customer_id, date_format(order_date, 'MMMM') as order_month,
count(1) as total_cnt, first(int(date_format(order_date,'MM'))) as month_num from orders
group by customer_id, order_month order by month_num""").explain(True)
```

```
== Parsed Logical Plan ==
'Sort ['month_num ASC NULLS FIRST], true
+- 'Aggregate ['customer_id, 'order_month], ['customer_id, 'date_format('order_date, MMMM) AS order_month#59, 'count(1) AS total_cnt#60, first
('int('date_format('order_date, MM)), false) AS month_num#62]
   +- 'UnresolvedRelation [orders], [], false

== Analyzed Logical Plan ==
customer_id: bigint, order_month: string, total_cnt: bigint, month_num: int
Sort [month_num#62 ASC NULLS FIRST], true
+- Aggregate [customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))], [customer_id#2L, date_format(cast(ord
er_date#1 as timestamp), MMMM, Some(America/Toronto)) AS order_month#59, count(1) AS total_cnt#60L, first(cast(date_format(cast(order_date#1 as
timestamp), MM, Some(America/Toronto)) as int), false) AS month_num#62]
   +- SubqueryAlias orders
      +- Relation[order_id#0L,order_date#1,customer_id#2L,order_status#3] csv

== Optimized Logical Plan ==
Sort [month_num#62 ASC NULLS FIRST], true
+- Aggregate [customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))], [customer_id#2L, date_format(cast(ord
er_date#1 as timestamp), MMMM, Some(America/Toronto)) AS order_month#59, count(1) AS total_cnt#60L, first(cast(date_format(cast(order_date#1 as
timestamp), MM, Some(America/Toronto)) as int), false) AS month_num#62]
   +- Project [order_date#1, customer_id#2L]
      +- Relation[order_id#0L,order_date#1,customer_id#2L,order_status#3] csv

== Physical Plan ==
*(3) Sort [month_num#62 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(month_num#62 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [id=#116]
   +- *(2) HashAggregate(keys=[customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#69], functions=[count
(1), first(cast(date_format(cast(order_date#1 as timestamp), MM, Some(America/Toronto)) as int), false)], output=[customer_id#2L, order_month#5
9, total_cnt#60L, month_num#62])
      +- Exchange hashpartitioning(customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#69, 200), ENSURE_R
EQUIREMENTS, [id=#112]
         +- *(1) HashAggregate(keys=[customer_id#2L, date_format(cast(order_date#1 as timestamp), MMMM, Some(America/Toronto)) AS date_format(c
ast(order_date#1 as timestamp), MMMM, Some(America/Toronto))#69], functions=[partial_count(1), partial_first(cast(date_format(cast(order_date#1
```

Key Points of Hash Vs Sort Aggregate

- **Query in Scenario 2 ran faster as it used Hash Aggregate which creates a hash table with a simple algorithm that has a time complexity of O(n)**
- **Whereas query in Scenario 1 took longer time for execution as it used Sort Aggregate. Sorting is a costly operation as it requires shuffling of data and implements an algorithm with a time complexity of O(nlogn)**

- **System could not implement Hash Aggregate in Scenario 1 as the date datatype was String and String is an immutable datatype.**

Apache Spark Logical and Physical Plan

Example :

```
spark.sql("""select * from orders inner join customers
on orders.customer_id == customers.customerid where
order_status = 'CLOSED'""").explain(True)
```

```
== Parsed Logical Plan ==
'Project [*]
+- 'Filter ('order_status = CLOSED)
   +- 'Join Inner, ('orders.customer_id = 'customers.customerid)
      :- 'UnresolvedRelation [orders], [], false
      +- 'UnresolvedRelation [customers], [], false

== Analyzed Logical Plan ==
order_id: bigint, order_date: string, customer_id: bigint, order_status: string, customerid: bigint, customer_fname: string, customer_lname: string, username: strin
g, password: string, address: string, city: string, state: string, pincode: bigint
Project [order_id#72L, order_date#73, customer_id#74L, order_status#75, customerid#98L, customer_fname#99, customer_lname#100, username#101, password#102, address#10
3, city#104, state#105, pincode#106L]
+- Filter (order_status#75 = CLOSED)
   +- Join Inner, (customer_id#74L = customerid#98L)
      :- SubqueryAlias orders
      :  +- Relation[order_id#72L,order_date#73,customer_id#74L,order_status#75] csv
      +- SubqueryAlias customers
         +- Relation[customerid#98L,customer_fname#99,customer_lname#100,username#101,password#102,address#103,city#104,state#105,pincode#106L] csv

== Optimized Logical Plan ==
Join Inner, (customer_id#74L = customerid#98L)
:- Filter ((isnotnull(order_status#75) AND (order_status#75 = CLOSED)) AND isnotnull(customer_id#74L))
:  +- Relation[order_id#72L,order_date#73,customer_id#74L,order_status#75] csv
+- Filter isnotnull(customerid#98L)
   +- Relation[customerid#98L,customer_fname#99,customer_lname#100,username#101,password#102,address#103,city#104,state#105,pincode#106L] csv

== Physical Plan ==
*(2) BroadcastHashJoin [customer_id#74L], [customerid#98L], Inner, BuildRight, false
:- *(2) Filter ((isnotnull(order_status#75) AND (order_status#75 = CLOSED)) AND isnotnull(customer_id#74L))
:  +- FileScan csv [order_id#72L,order_date#73,customer_id#74L,order_status#75] Batched: false, DataFilters: [isnotnull(order_status#75), (order_status#75 = CLOSED),
isnotnull(customer_id#74L)], Format: CSV, Location: InMemoryFileIndex[hdfs://m01.itversity.com:9000/public/trendytech/orders/orders_1gb.csv], PartitionFilters: [], P
ushedFilters: [IsNotNull(order_status), EqualTo(order_status,CLOSED), IsNotNull(customer_id)], ReadSchema: struct<order_id:bigint,order_date:string,customer_id:bigin
t,order_status:string>
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]),false), [id=#83]
   +- *(1) Filter isnotnull(customerid#98L)
      +- FileScan csv [customerid#98L,customer_fname#99,customer_lname#100,username#101,password#102,address#103,city#104,state#105,pincode#106L] Batched: false, Dat
aFilters: [isnotnull(customerid#98L)], Format: CSV, Location: InMemoryFileIndex[hdfs://m01.itversity.com:9000/public/trendytech/retail_db/customers], PartitionFilter
s: [], PushedFilters: [IsNotNull(customerid)], ReadSchema: struct<customerid:bigint,customer_fname:string,customer_lname:string,username:string,password:str...
```

Parsed Logical Plan (Unresolved) - Parse the query to check the correctness of the query syntax. It will throw a ParseException if there are any syntax errors.

Parsed logical plan only checks for the correctness of syntax and cannot identify if the entities like table name / column name used in the query exists or not. Therefore, it is unresolved.

Analysed Logical Plan (Resolved) - Analyses if all the entities like table names, column names, views, etc used in the query exists or not. It will throw an Analysis Exception if for instance a table with the name mentioned in the query doesn't exist.

After checking for the syntax correctness in the previous stage, the system then checks for any analysis exception by cross-checking with the Catalog leading to a Resolved Logical Plan.

Optimised Logical Plan - Certain sets of predefined rules are used to optimize the query execution plan at the early stages.

Example :

1. Predicate Pushdown - In this case the filters are pushed down or applied at the very early stages. This ensures that operations are performed on only relevant data.
2. Combining multiple projections into a single projection.
3. Combining multiple filters into a single operation.

Physical Plan - Used to identify / decide what kind of joins or aggregations strategies can be chosen for optimal query performance.

Example :

1. Whether to use Hash Aggregate or Sort Aggregate
2. Which type of Join to be used - Broadcast Hash Join | Sort-Merge Join | Shuffle-Hash Join.

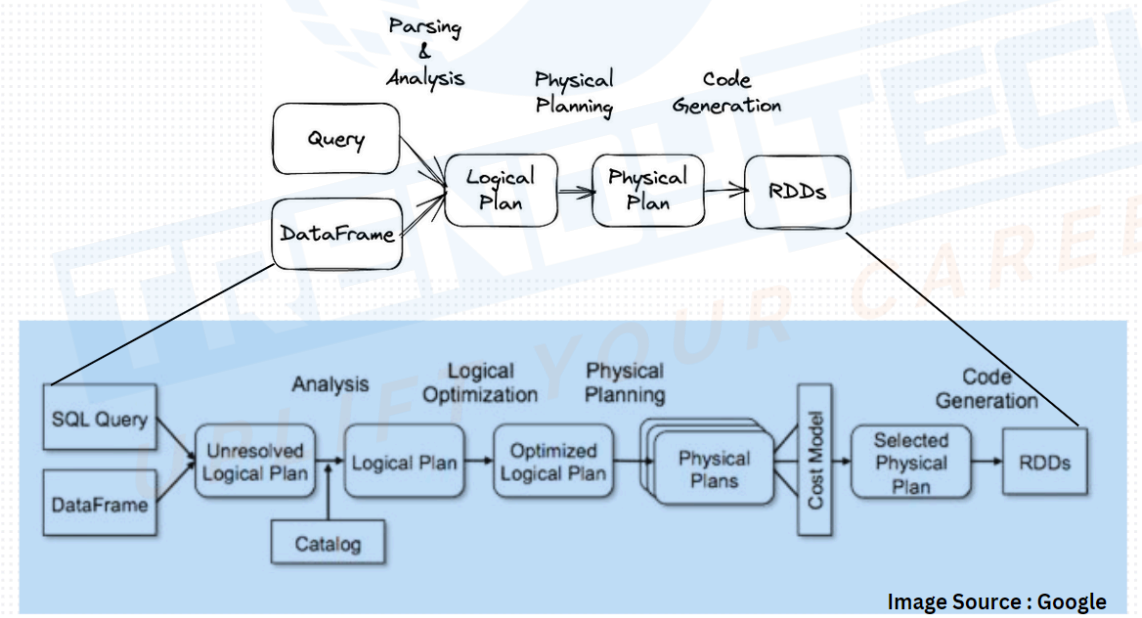


Image Source : Google

Catalyst Optimizer

It is an optimization process where a query execution goes through a set of pre-configured rules or custom-defined rules in the optimization layer to improve query performance.

File Formats and Compression Techniques

How data is stored in the backend is one of the important concepts to be considered while designing a solution architecture. There are 2 important factors that play a major role in data storage.

1. File formats
2. Compression techniques

What is the need for different file formats

1. Saving storage
2. Faster processing
3. Reduced time for I/O operations

There are several file formats available that provide one or more of the following features :

- Faster reads
- Faster writes
- Splittable
- Schema Evolution Support
- Supports Advanced Compression Techniques

The respective file formats that meet the project requirements will be used for processing.

2 Broad Categories of Files Formats

1. **Row Based**

   The entire record, all the column values of a row are stored together followed by the values of the subsequent records.

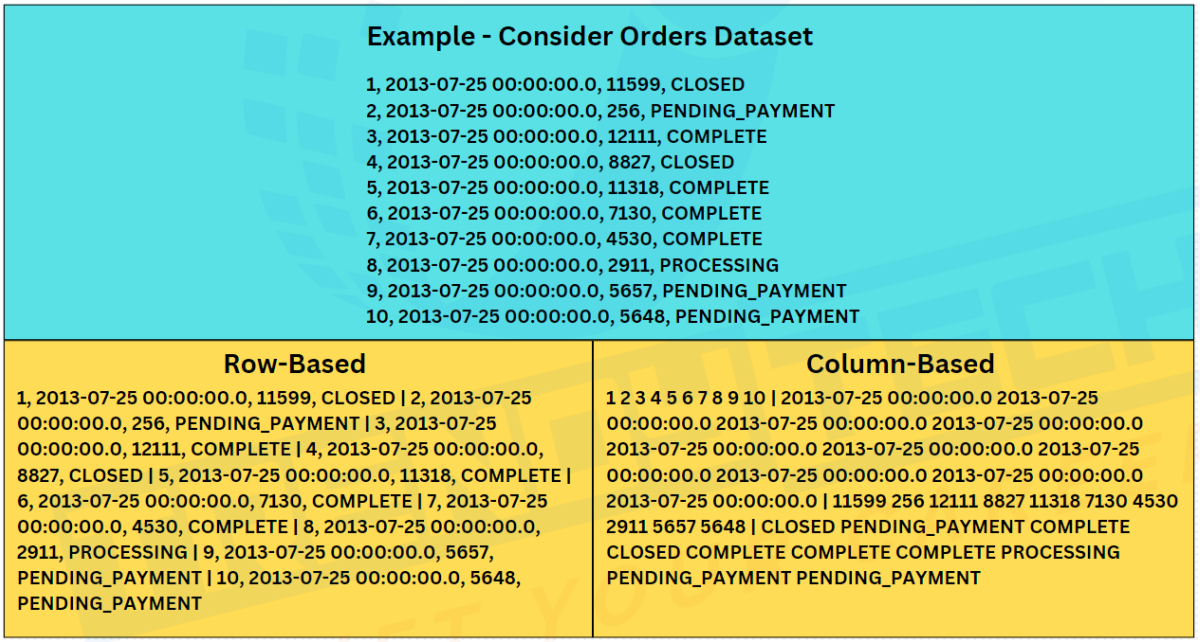
   - Used when **faster writes** is a requirement as appending new rows is easy in case of row based.

- **Slower reads**, reading a subset of columns is not efficient as the entire dataset has to be scanned.
- Provides **Less Compression**

2. **Column Based**

   The values of a single column of all the records are stored together. Likewise the subsequent column values of the next column for all the records are stored together.

   - **Efficient reads** when a subset of columns has to be read because of the way data is stored underneath. Only the relevant data can be read without having to go through the entire dataset.
   - **Slower writes** as the data has to be updated at different places to write even a single new record.
   - Provides very **Good Compression** as all the values of the same datatypes are stored together.



File Formats that are not a best fit for Big Data Processing

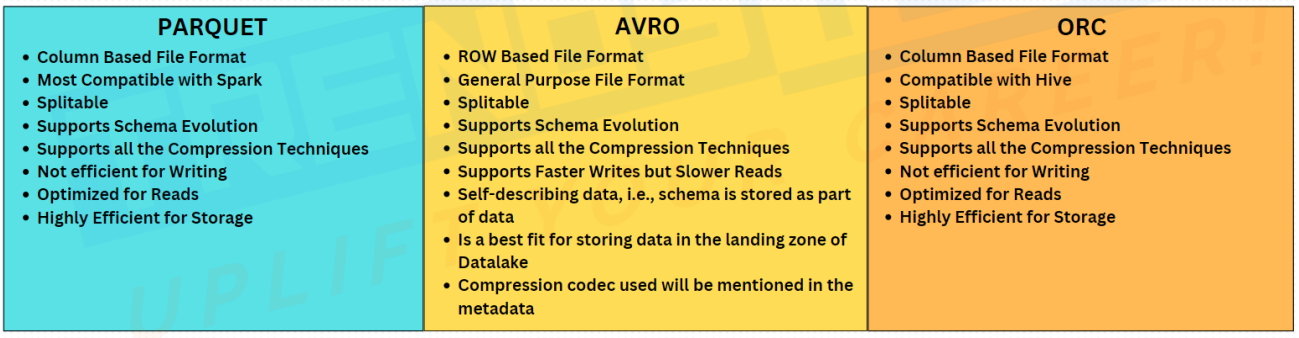
1. Text files like CSV
   - Stores the values as strings/text internally and thereby consumes a lot of memory for storing and processing.
   - If any numeric operations like addition, subtraction, etc have to be performed on numeric like values that are internally stored as

string in case of text files, then they have to be casted to desired types like integer, date, long,etc.
    - Casting / conversion is a costly and a time consuming operation.
    - Data size is more and therefore the network bandwidth required to transfer the data is also high.
    - Since the data size is more, I/O operations will also take a lot of time.
2. XML and JSON - These files format have partial schema associated.
    - All the disadvantages of the text file format is also applicable for XML and JSON files
    - Since they have an inbuilt schema associated along with the data, these file formats are bulky
    - These file formats are not splittable, which implies no parallelism can be achieved.
    - Lot of I/O is involved.

Specialized File Formats for Big Data Domain

There are 3 main File Formats well suited for Big Data Problems.

1. **PARQUET**
2. **AVRO**
3. **ORC**



Light-Weight Compression Techniques

1. **Dictionary Encoding**

    When the values are too large, then a dictionary will be created that contains the mapping information of the large values to some numeric key. This will reduce the memory utilisation drastically.

Example :

2. **Bit Packing**

   Is a simple compression technique that uses as few bits as possible to store a piece of data. If done the right way, it can significantly reduce the data size. When it is used along with dictionary encoding, it gives the best compression.

   Example :

   Consider the following dictionary encoding

   [Andaman and Nicobar Island, 17]

   In the above case, it would ideally require 32 bits/ 4 bytes to store an integer 17. But with Bit-packing the same value can be stored in less than 5 bits. Binary equivalent of 17 = 10001

3. **Delta Encoding**

   Also known as delta compression is a compression technique that stores the data in the form of deltas(differences) between sequential data rather than the complete value.

   Example :

   Consider timestamp

   12:49:00 00:00:00

   12:49:01 00:00:00

   12:49:02 00:00:00

With Delta Encoding, it stores only the difference values and thereby drastically reduces the storage memory required.

12:49:00 00:00:00

1

2

4. **Run-length Encoding**

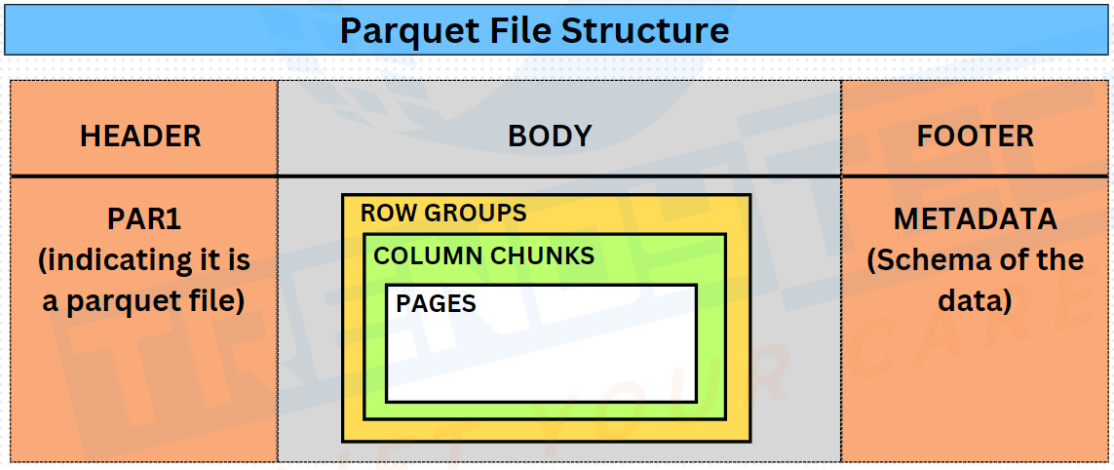
It is a loss-less compression technique where the data sequences having redundant data are stored as a single value along with the count of the number of times the redundant data appears in the sequence.

Example :

Consider the redundant data sequence -

ssssssssgggggggghhhhhhhhh

After Run Length encoding, it is stored as s8g7h9 and it can be easily reconstructed while decompression.



Lets consider a parquet file (orders data) of 500MB with 80000 rows
HEADER : PAR1
BODY :

1st Row Group : consists of 20000 records
- Column Chunk 1 - orderid
  - Pages (1MB each - Holds Actual data + Metadata)
- Column Chunk 2 - orderdate
  - Pages
- Column Chunk 3 - customerid
  - Pages
- Column Chunk 4 - orderstatus

Pages
2nd Row Group : consists of 20000 records
Column Chunk 1 - orderid
Pages (1MB each - Holds Actual data + Metadata)
Column Chunk 2 - orderdate
Pages
Column Chunk 3 - customerid
Pages
Column Chunk 4 - orderstatus
Pages
3rd Row Group : consists of 20000 records
Column Chunk 1 - orderid
Pages (1MB each - Holds Actual data + Metadata)
Column Chunk 2 - orderdate
Pages
Column Chunk 3 - customerid
Pages
Column Chunk 4 - orderstatus
Pages
...

FOOTER : Consists of Metadata

Schema Evolution

Schema evolution allows for easy incorporation of the changes in the schema of evolving data. With time, data might evolve requiring corresponding changes in the schema to be updated.

Events which brings about schema change are :

- Adding new columns/ fields
- Dropping existing columns/ fields
- Changing the datatypes …

Example :

- Consider the orders dataset with the columns - **order_id, order_date** loaded into the dataframe and written in parquet file format.
- Consider another new orders dataset with a newly added column of customer_id. The columns of this new orders data are - **order_id, order_date, customer_id.** This new data is loaded to the dataframe and written in the parquet file format.

- Now, if we try to read this data to display the resultant data, where there are changes in the schema of the old data and newly added data, schema merge would not have taken place as it is disabled by default.
- To enable and incorporate the schema evolution feature, the property **mergeSchema** has to be enabled by using :

  option("mergeSchema", True)

Compression Techniques

What is the need for compression?

- To save Storage Space
- To reduce I/O cost

Compression involves additional cost

- CPU cycles
- Time to compress and uncompress the files, specifically if complex algorithms are implemented.

Generalised Compression Techniques

1. **Snappy**
   - Optimized for speed and gives moderate level of compression. It is highly preferred as it is very fast and provides moderate compression.
   - It is the default compression technique for Parquet and ORC.
   - Snappy by default is not splittable when used with CSV or text file formats.
   - Snappy has to be used with container based file formats like ORC and Parquet to make it splittable.
2. **LZO**

   Optimized for speed with moderate compression. It requires a separate licence as it is not generally distributed along with hadoop. It is Splittable by default

3. **Gzip**

   Provides high compression ratio and therefore it is comparatively slow in processing. It is not splittable by default and has to be used along with container based file formats to make it splittable.

4. **Bzip2**

   Very optimized for storage and provides the best compression and thereby very slow in terms of processing. It is inherently splittable.

Note : Some compressions are optimized for

1. **Speed**
2. **Compression Ratio**

(If the requirement is a higher compression ratio then the speed of compression will be slower as it would involve more CPU Cycles to implement complex algorithms. However, for quick compressions, the compression ratio will be less.)

- **Moderate compressions with high speed is preferred mostly.**
- **Data archival would require a high compression ratio.**