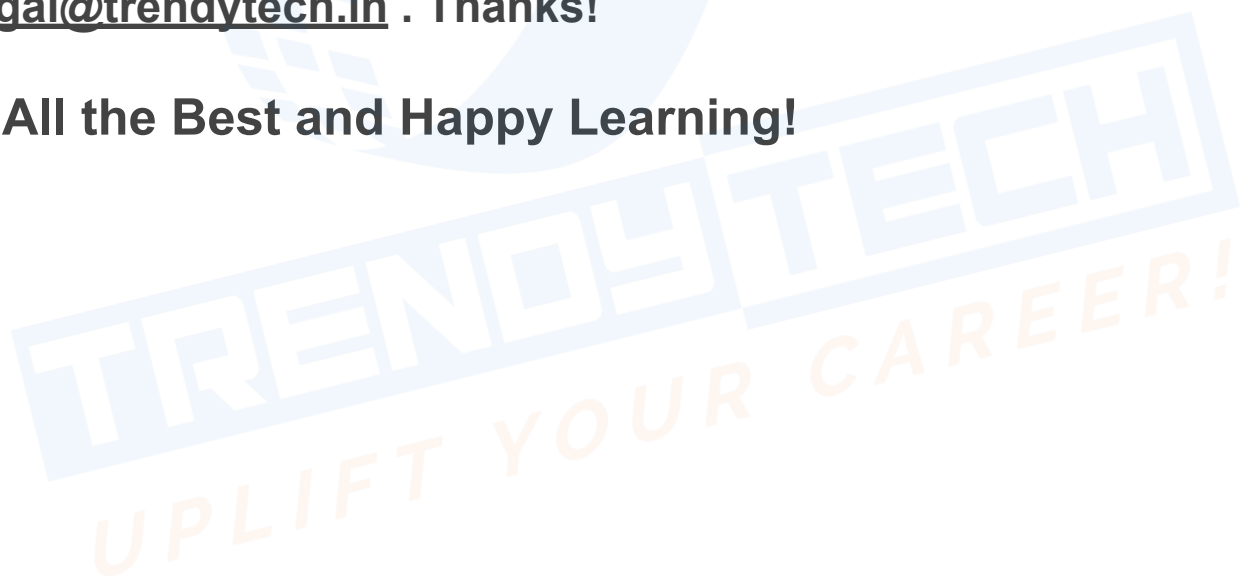


Disclaimer: These slides are copyrighted and strictly for personal use only

- 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!



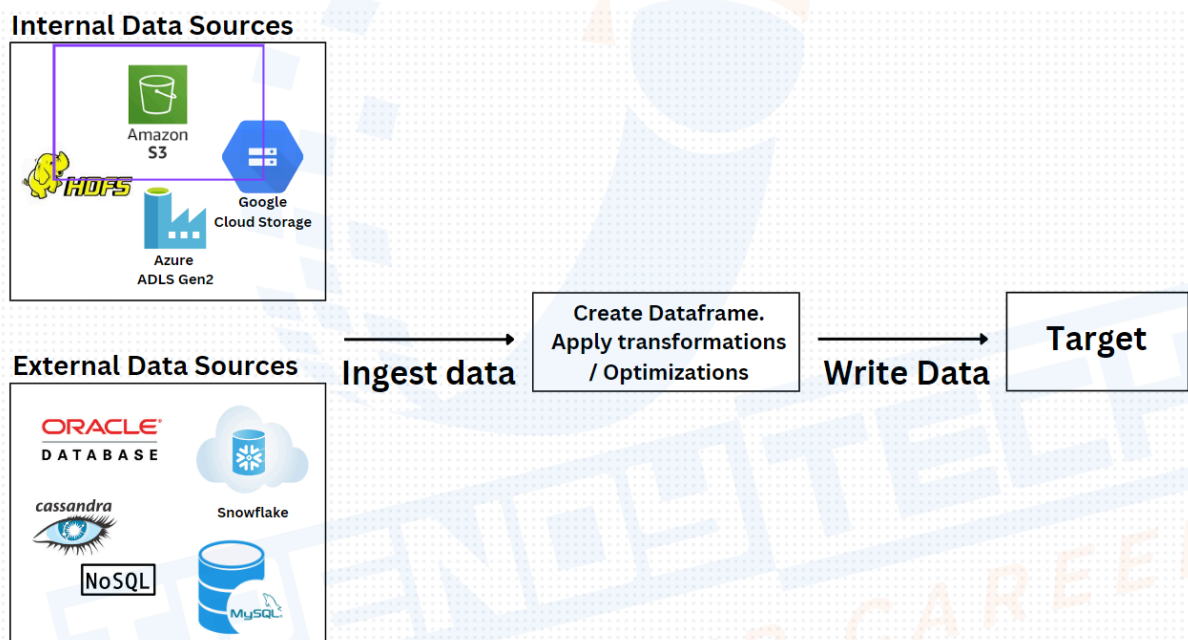
Spark Workflow -

1. Create a Dataframe and load the data from a Data Source.
2. Performing various Transformations.
3. Write the results to the Target.


Data Sources

Internal - Data stored in Data Lake that is native to spark. Spark can directly interact with this data. Ex - HDFS, Amazon S3, Azure data lake storage Gen2 / Google Cloud Storage

External - Certain connections need to be configured for spark to interact with the data. Ex - Oracle Database, MySQL, Cassandra, MongoDB



2 Approaches of processing External Data in Spark

1. Ingest Data from DB to Datalake using ingestion framework 

2. Connect to the external data sources using spark directly and then process. 

DataFrame Writer API

```
orders_df. write \  
.format("csv") \  
.mode("overwrite") \  
.option("path", "/user/itv006763/sparkwritedemo") \  
.save()
```

While writing the results using the dataframe writer, the path to the folder where the results needs to be written has to be specified.

Different file formats while writing to the dataframe

- **CSV** is not the best file format.
- **Parquet** is the most optimized and compatible file format for spark.
- **JSON** is a bulky file format that consumes a large amount of space as it has to embed the column names for each record (like Key-Value pairs)
- **ORC** is also a good file format in terms of optimization.
- **AVRO** is an external data source that requires certain cluster configurations to be set-up before using this format.

Note : File formats play a very critical role in terms of optimizing the storage and computation cost. It is advisable to choose the most optimized and compatible file format for best performance benefits.

Write modes in Spark

1. **overwrite** - If the folder where the results needs to be written already exists, then it will be overwritten.
2. **ignore** - If the folder already exists, writing files will be ignored.
3. **append** - If the folder already exists, new files will be appended to the existing folder
4. **errorIfExists** - If the folder already exists, the write operation would throw an error.

partitionBy Clause

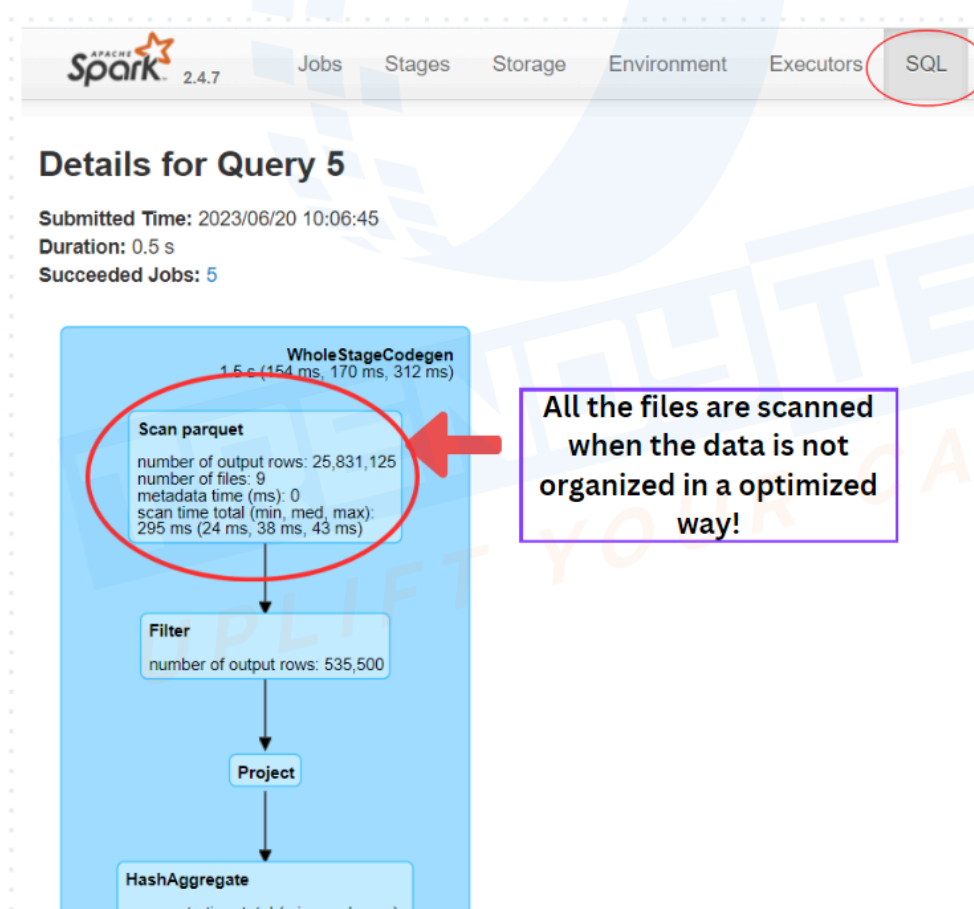
Consider an example scenario : Working with orders dataset as shown in the image below,

Suppose we run a query on this randomly stored data

```
spark.sql("select count(*) from orders where order_status = 'CLOSED'")
```

Partitioning is a process of structuring the data in the underlying file system in an efficient way that only a subset of data will be scanned whenever a query is fired instead of scanning the entire data leading to performance gains.

The query execution time will be noticeably high as all the files have to be scanned for processing the query.



In order to optimize the query performance, the data needs to be partitioned, in which case only a subset of data will be scanned for processing and thereby improving the performance significantly.

```
orders_df.write \
  .format("csv") \
  .mode("overwrite") \
  .partitionBy("order_status") \
  .option("path", "/user/itv006763/partition_demo_output1") \
  .save()
```

- partitionBy should be applied on the column which has low cardinality (less number of distinct columns)
- partitionBy should be applied on the column where filtering is used frequently in the query on that particular column.
- **Partition pruning** is scanning only the necessary partitions and skipping the other partitions.
- In a query, if filtering is applied on a non-partitioned column, then it doesn't provide any performance gains. Example

```
spark.sql ("select count(*) from orders where cust_id = 12345")
```

This query doesn't give any performance benefit as the filtering is applied on cust_id which is not the column on which partitioning is done.

- **Partitioning can be applied simultaneously on more than one column by specifying the comma separated list of columns in the partitionBy clause.**

```
customer_df.write \
  .format("csv") \
  .mode("overwrite") \
  .partitionBy("customer_state", "customer_city") \
  .option("path", "/user/itv006753/partition_demo") \
  .save()
```

The above code creates 2 levels of folders (nested)

Top-level folder : State (if there are X states, X top-level folders will be created.)

Sub folder : Cities (if there are Y cities in a State, then Y subfolders are created for that particular state.)

bucketBy Clause

When there are a large number of distinct values (High Cardinality), then Bucketing is a better choice over Partitioning. In case of bucketing, the **number of buckets** and the **bucketing column** has to be defined upfront and passed as parameters to the bucketBy clause

Bucketing helps in 2 ways

1. Skipping the irrelevant data
2. Join Optimizations

Every partition will have files equal to the number of buckets mentioned while creating the dataframe.

Bucketing using bucketBy clause

```
customer_final_df.write \
    .format("parquet") \
    .mode("overwrite") \
    .bucketBy(4,"customer_id") \
    .saveAsTable("itv006753_bucketing_db.customersnew")
```

//Running query on a non-bucketed column to check the performance

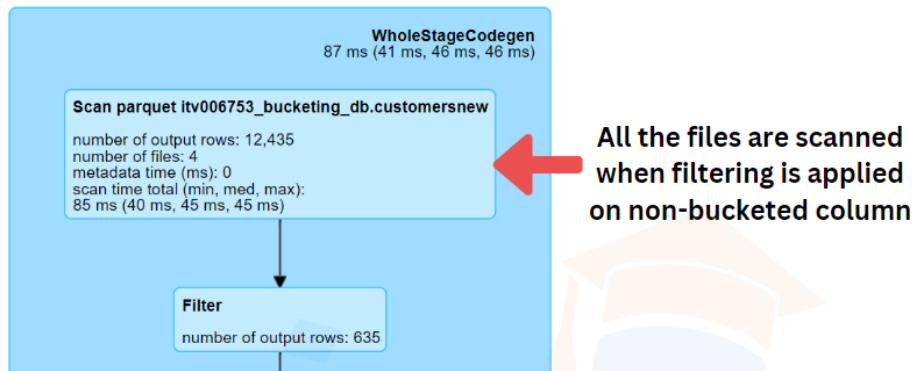
```
spark.sql("select * from itv006753_bucketing_db.customersnew where  
customer_state='TX'")
```

Details for Query 36

Submitted Time: 2023/06/21 02:02:41

Duration: 99 ms

Succeeded Jobs: 6



//Running query on a bucketed table on the bucketed column to check the performance

```
spark.sql("select * from itv006753_bucketing_db.customersnew where customer_id=10")
```

This results in significant performance gains as only one file will be scanned to get the desired results.

Key Points:

- Based on a hash function, the records will be moved to the different buckets/ files.
- In case of bucketing, a managed spark table has to be created to save the bucketed data.
- A combination of Partitioning followed by Bucketing is possible, where there will be two level filtering.
- However, Bucketing followed by Partitioning is not possible. Because, partitioning results in folders and bucketing will result in files. Having files inside a folder is possible but we cannot have a folder inside a file.

Storing and retrieving the data using bucketing approach

Consider an example where the number of fixed buckets = 4

Storing the data - hash function used is modulo (since there are 4 buckets, modulo 4 is applied) Ex : $1\%4$, $2\%4$, $3\%4$, $4\%4$,....

Bucket 0	4, 8, 12
Bucket 1	1, 5, 9
Bucket 2	2, 6, 10
Bucket 3	3, 7, 11

Retrieving the data - hash function used is modulo. Say you are required to retrieve record 9, then $9\%4 = 1$, implies data is present in Bucket 1 (Only Bucket 1 is scanned and the rest are skipped)

Partitioning	Bucketing
<ul style="list-style-type: none"> • When the number of distinct values are less - Low Cardinality • Number of partitions = Number of distinct values in a column on which partitioning is applied. • Example : <code>partitionBy("order_status")</code> • partitioned data is saved in folders 	<ul style="list-style-type: none"> • When the number of distinct values are high - High Cardinality • Since there are large number of distinct values, Number of buckets have to be defined while writing to the dataframe. • Example : <code>bucketBy(8,"customer_id")</code> • bucketed data is saved in a managed spark table

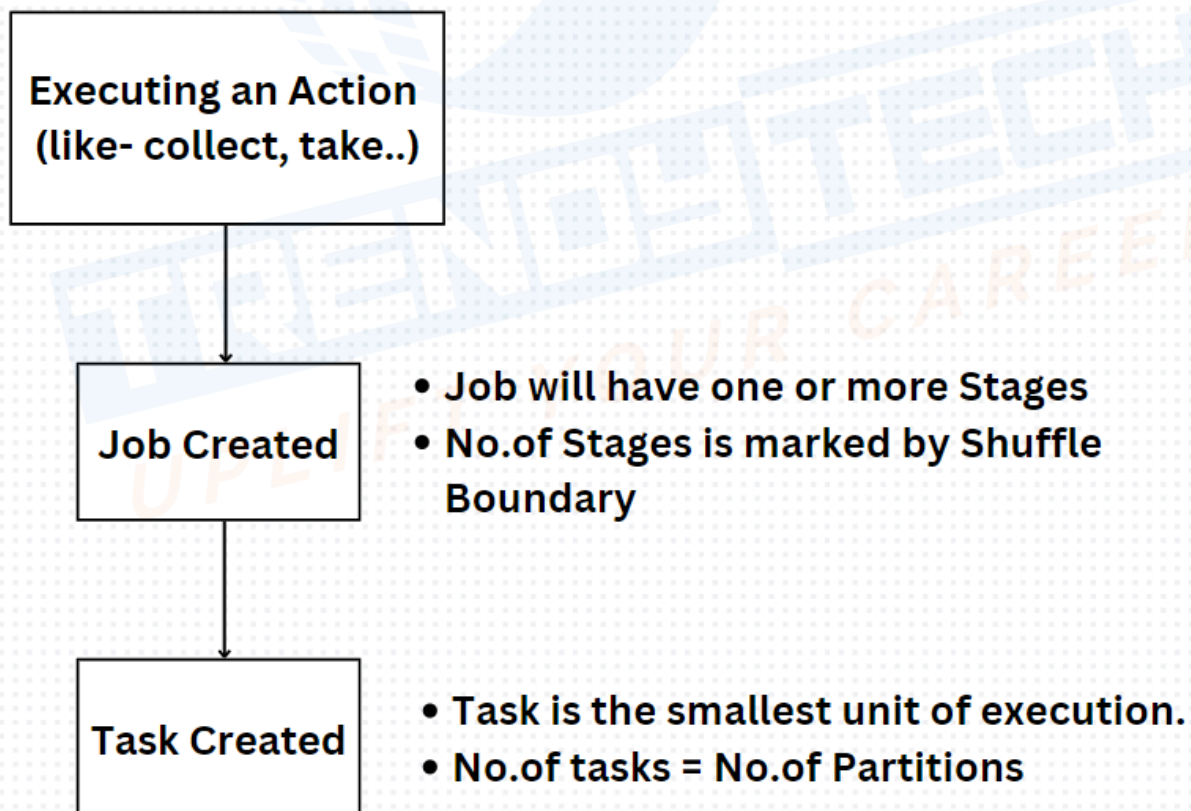
Databricks community edition to visualize Spark UI by setting up a cluster!!

1. Sign up for the Databricks Community Edition

<https://www.databricks.com/try-databricks>

2. Create a Cluster
3. Open a notebook (will be associated to the newly created cluster) to execute the code
4. To browse the data files in DBFS (Databricks file system), enable DBFS [Settings -> Admin Console -> Workspace settings -> Enable DBFS file browser under Advanced]
5. Execute the code on notebook.
6. To view the spark UI, click on the drop down arrow near the cluster name on the right top corner -> click on Spark UI option.
7. The Spark UI opens in a new tab that is cleaner and more intuitive.

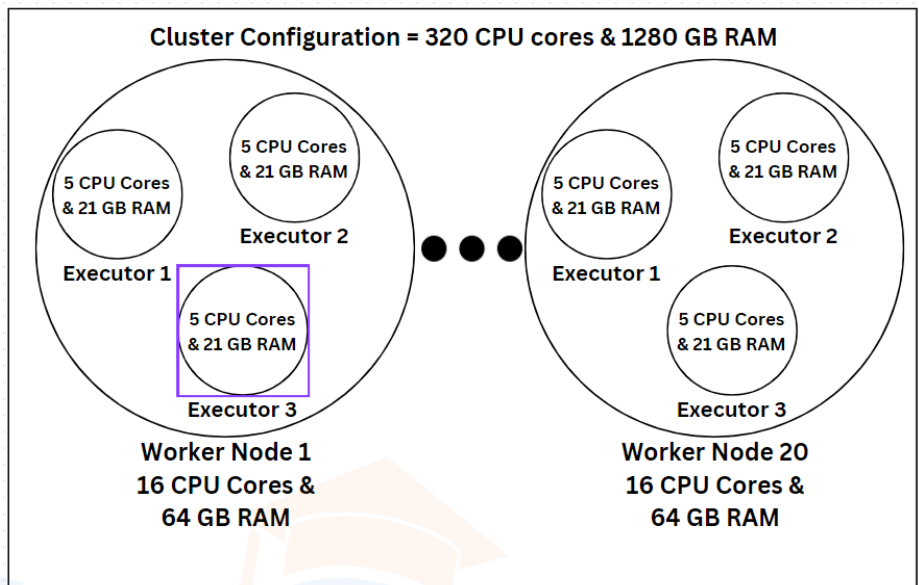
Internals of Spark





1.1 GB File in HDFS
-> 9 Partitions / Blocks
-> 9 Tasks

20 Node Cluster => 20 Worker Nodes



No. of tasks that can run parallelly on this cluster = 300 (There are 320 cpu cores but some cores are used for background activities, remaining 300 cores are used for task execution)

Key Points

- Executor or container size = 5 cpu cores, 21 GB RAM (there can be 3 such executors on every node of the cluster considering the given executor configurations.)
- Consider that 1 core and 1 GB RAM is consumed for background activities in every node of the cluster. Then, the remaining 15 cpu cores and 63 GB RAM for execution
- 20 node cluster => 60 executors of 5cpu & 21GB RAM each.

Example Scenario

A file of size 10.1 GB is present in HDFS. This file needs to be processed using dataframe.

10.1 GB => 81 partitions in the dataframe

Resources available :

4 executors (each executor has 5 CPU Cores and 21 GB RAM)

- Overall, 4 executors will have 20 cores and 84 GB RAM

- Each CPU core can process 1.2 GB of data (30% of available memory will be used for execution and the rest is utilized for memory management)

No.of tasks that can run in parallel ?

Only 20 tasks can run in Parallel (as there are only 4 executors)

Out of 81 Partitions to be processed,

First level execution - 20 partitions in 10 sec

Second level execution - 20 partitions in 10 sec

Third level execution - 20 partitions in 10 sec

Fourth level execution - 20 partitions in 10 sec (total of 80 tasks executed until fourth level)

Fifth level execution - 1 partitions in 8 sec (Remaining 1 task)

Key Points

- To check the default parallelism in spark

```
spark.sparkContext.defaultParallelism
```

2

- Default parallelism is 2 because there are 2 executors available with 1 Core and 1 GB RAM each.
- By default dynamic allocation of executors is enabled for the cluster
- Spark application is executed in Client mode on the cluster.

APACHE Spark 2.4.7 Jobs Stages Storage Environment **Executors** SQL

Executors

▶ Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time
Active(3)	0	300.8 KB / 1.2 GB	0.0 B	2	0	0	4	4	3 s (0.1 s)
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 m)
Total(3)	0	300.8 KB / 1.2 GB	0.0 B	2	0	0	4	4	3 s (0.1 s)

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks
driver	g01.itversity.com:40317	Active	0	122 KB / 384.1 MB	0.0 B	0	0	0	0	0
1	w02.itversity.com:41605	Active	0	56.9 KB / 384.1 MB	0.0 B	1	0	0	1	1
2	w03.itversity.com:41905	Active	0	122 KB / 384.1 MB	0.0 B	1	0	0	3	3

Client Mode of Spark

Driver Is running on a local gateway node

- Parallelism depends on the amount of resources that is allocated. If an application holds 8 cores then the parallelism will be 8. In the above diagram, there are 2 cores, therefore the parallelism is 2.

How distinct() works?

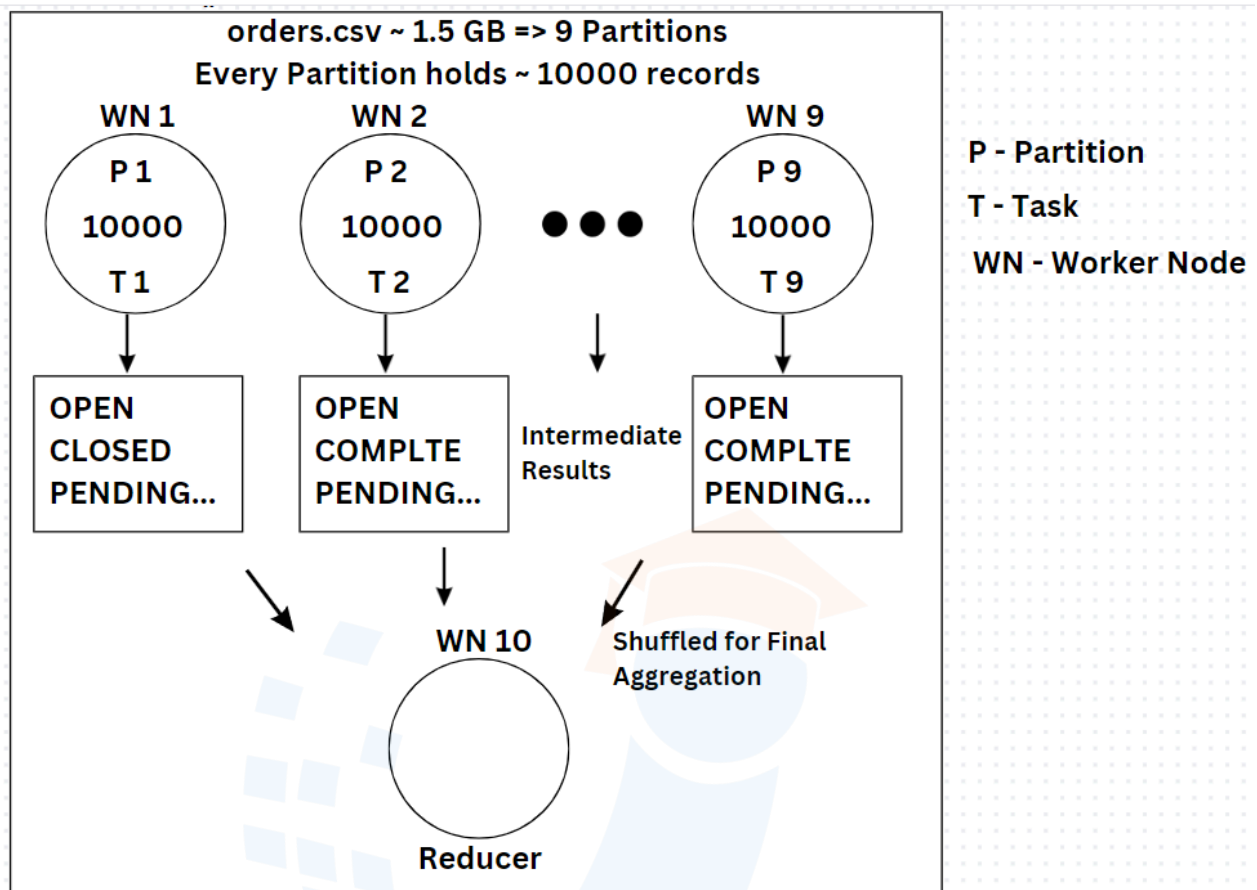
Considering orders.csv dataset of ~ 1.5 GB, there would be 9 partitions created for this dataset based on the default partition-size of 128 MB.

```
orders_df.createOrReplaceTempView("orders")

spark.sql("select count(distinct(order_status)) from orders")

count(DISTINCT order_status)
```

9



- Each partition would handle an approximate of 10000 records to provide the distinct order_status. All of the local results calculated on each partition are then shuffled to a common single node where the final aggregation is carried out to give the distinct order_status.
- When a **distinct()**, **wide transformation** is invoked, 200 partitions are created and 200 tasks get launched.

Stage 1 - Initially **9 tasks** are launched to load the dataframe.

Stage 2 - distinct() is a wide transformation and 200 partitions are created and **200 tasks** are launched.

Stage 3 - Final aggregations where **1 task** is launched and final distinct values are evaluated.


- Parallelism depends on the number of executors available at any given point in time. In case of Dynamic allocation, executors will be dynamically allocated to the tasks as per the availability.
- Dynamic allocation is enabled by default at the cluster level in the lab. This feature can be disabled if a static amount of resources are required.

How to turn off the Dynamic Allocation and request a static amount of resources?

3 ways of defining the amount of needed resources

1. Number of Executors / Containers
2. Number of CPU Cores per Executor
3. Amount of Memory per Executor

```
from pyspark.sql import SparkSession
import getpass
username = getpass.getuser()
spark= SparkSession. \
builder. \
config('spark.shuffle.useOldFetchProtocol', 'true'). \
config('spark.dynamicAllocation.enabled', 'false'). \
config('spark.executor.instances', '2'). \
config('spark.executor.cores', '2'). \
config('spark.executor.memory', '2g'). \
config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
enableHiveSupport(). \
master('yarn'). \
getOrCreate()
```



Note : The disadvantage of static resource allocation is that the resources can go idle when there are not many tasks to be executed and therefore underutilised.

To overcome this limitation of Static allocation, “spark-submit” can be used to execute the jobs on the cluster.

spark-submit

Is a utility to package/bundle the code and deploy on the cluster.

- Once the logic is developed in the development environment using notebooks.
- The functional code should then be packaged/ bundled and deployed on the cluster for regular scheduled job executions.


Steps for executing a code file using spark-submit

1. Develop the code in an IDE or any code editor.
2. Upload the code file to the cluster
3. Execute the following spark-submit command on the terminal.

```
spark-submit \  
--master yarn \  
--num-executors 1 \  
--executor-cores 1 \  
--executor-memory 1G \  
--conf spark.dynamicAllocation.enabled=false \  
prog1.py
```

4. After executing spark-submit, in the Spark UI, you can see History Server that shows the details of job execution.
5. The above code was executed in **Client Mode** and therefore the results were visible in the same terminal.
6. To execute the code in **Cluster Mode**, do the following changes to the code.


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



In this case the results will be present in one of the worker nodes and not on the local gateway node terminal.

spark-submit Configuration precedence

1. Configs set in the code while creating **Spark Session**. (Priority 1)
2. Configs set in **spark-submit** (Priority 2)
3. Configs set in **default Configuration File** (Priority 3)

Initial number of partitions in a Dataframe

-Initial partitions are different from Shuffle partitions (Shuffle partitions are set using the configuration property - "spark.sql.shuffle.partitions" which is set to 200 by default)

-Number of Partitions play an important role in determining the level of parallelism that can be achieved.

-Initial number of partitions is determined by Spark based on the following configurations :

- a) Number of CPU Cores in the cluster - Default Parallelism
- b) Default Partition Size

Challenges of large partition-size = Could lead to Out-of-Memory error / Data doesn't fit into the execution memory of the machine...

-Recommended/Default partition-size = 128MB

-No. of initial partitions is determined by the Partition Size

How to change the default partition size? (Not Recommended)

By modifying the configuration property "**spark.sql.maxPartitionBytes**"

Formula to calculate the Partition Size

Partition Size = min(maxPartitionBytes, file-size/defaultParallelism)

Partition-Size should be ideally $\leq 128\text{MB}$ (Partition-size should be set in a manner that the cluster resources are utilized efficiently without any resource wastage.

Example Scenarios of handling a 1.1GB file

2 Executors with 1GB Memory and 1 CPU Core	8 Executors with 2GB Memory and 2 CPU Cores
<ul style="list-style-type: none"> • Total CPU Cores = 2 • No. of Tasks that can be executed in parallel = 2 • Default parallelism = 2 	<ul style="list-style-type: none"> • Total CPU Cores = 8 executors * 2 CPU Cores = 16 • No. of Tasks that can be executed in parallel = 16 • Default parallelism = 16
Partition Size = min(128MB , Data-file-size/ No. of CPU Cores) = min(128MB , (1.1GB/2)) = min(128MB , 550MB) = 128MB No. of Partitions = File-Size / Partition-Size = 1.1GB / 128MB = 9	Partition Size = min(128MB , Data-file-size/ No. of CPU Cores) = min(128MB , (1.1GB/16)) = min(128MB , 75MB) = 75MB No. of Partitions = File-Size / Partition-Size = 1.1GB / 75MB = 16

Initial Number of Partitions for a Single Non-Splittable File :

```
1 from pyspark.sql import SparkSession
2
3 spark = SparkSession. \
4     builder. \
5     config('spark.shuffle.useOldFetchProtocol', 'true'). \
6     config("spark.sql.warehouse.dir", "/user/{username}/warehouse"). \
7     enableHiveSupport(). \
8     master('yarn'). \
9     getOrCreate()
10
11 orders_schema = "order_id long, order_date string, cust_id long, order_status string"
12
13 orders_df = spark.read \
14     .format("csv") \
15     .schema(orders_schema) \
16     .load("/public/trendytech/orders/orders_1gb.csv")
17
18 print(orders_df.rdd.getNumPartitions())
19
20 spark.stop()
```

```
[itv006753@g01 ~]$ spark3-submit \
> --master yarn \
> --num-executors 3 \
> --executor-memory 2G \
> --executor-cores 4 \
> --conf spark.dynamicAllocation.enabled=false \
> InitialPartitionsDemo.py
SPARK_MAJOR_VERSION is set to 3, using Spark3
/opt/spark3-client/python/lib/pyspark.zip/pyspark/context.py:2
e also the plan for dropping Python 2 support at https://spa
DeprecationWarning)
12
```

- Total CPU Cores = 12 (num-executors * executor-cores)
- Default parallelism = 12
- Partition Size = min(128MB , 1.1GB/12)

```
[itv006753@g01 ~]$ spark3-submit \
> --master yarn \
> --num-executors 3 \
> --executor-memory 2G \
> --executor-cores 3 \
> --conf spark.dynamicAllocation.enabled=false \
> InitialPartitionsDemo.py
SPARK_MAJOR_VERSION is set to 3, using Spark3
/opt/spark3-client/python/lib/pyspark.zip/pyspark/context.py:2
e also the plan for dropping Python 2 support at https://spa
DeprecationWarning)
9
```

- Total CPU Cores = 9 (num-executors * executor-cores)
- Default parallelism = 9
- Partition Size = min(128MB , 1.1GB/9)

- In case of a file that is not splittable, then only one partition will be created.
- Usage of compression techniques like Snappy / Gzip with file formats like CSV would lead to non-splittable files (such files will get corrupted if split)
- However, compression techniques when applied on Parquet file format, It is splittable.

Initial Number of Partitions for Multiple File :

- In case of multiple files, if the partition size is noticeably less, 2 or more files will be combined into one partition to approximately make up to the default partition size of 128 MB.

Ex - If there are 20 multiple files of 53MB each, then 2 files will be combined together to reach a default partition size of 128MB. Therefore, there would be 10 partitions created in this case.

Important Configurations

1. `spark.conf.get("spark.sql.files.maxPartitionBytes")`
2. `spark.sparkContext.defaultParallelism`
3. `spark.conf.get("spark.sql.files.openCostInBytes")`

(openCostInBytes is the amount of resources that would be utilized to open a file. By Default it is 4MB, i.e., spark would have read 4MB of data in the time it would have taken to open a file for reading.)

