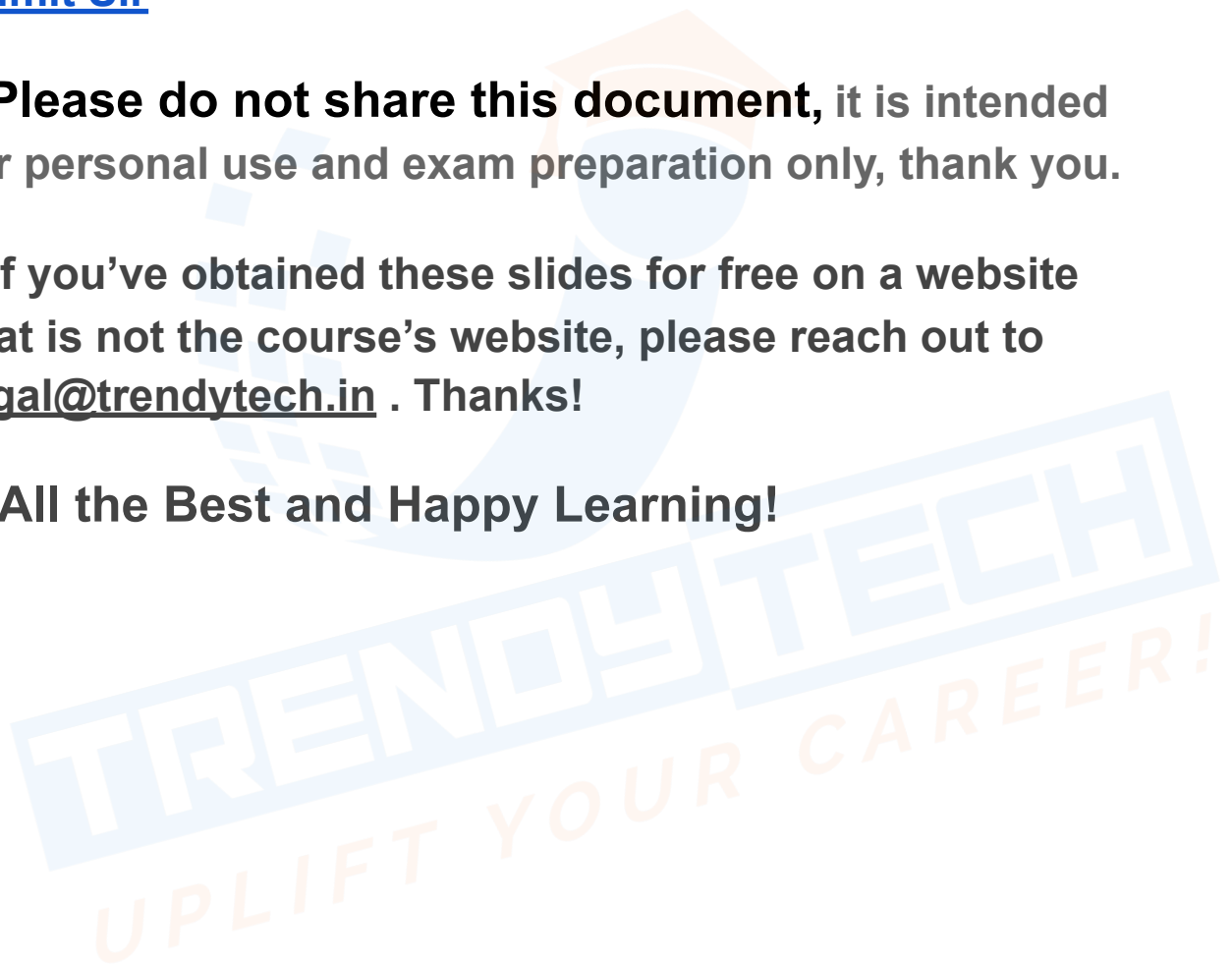


Disclaimer: These slides are copyrighted and strictly for personal use only

- 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!

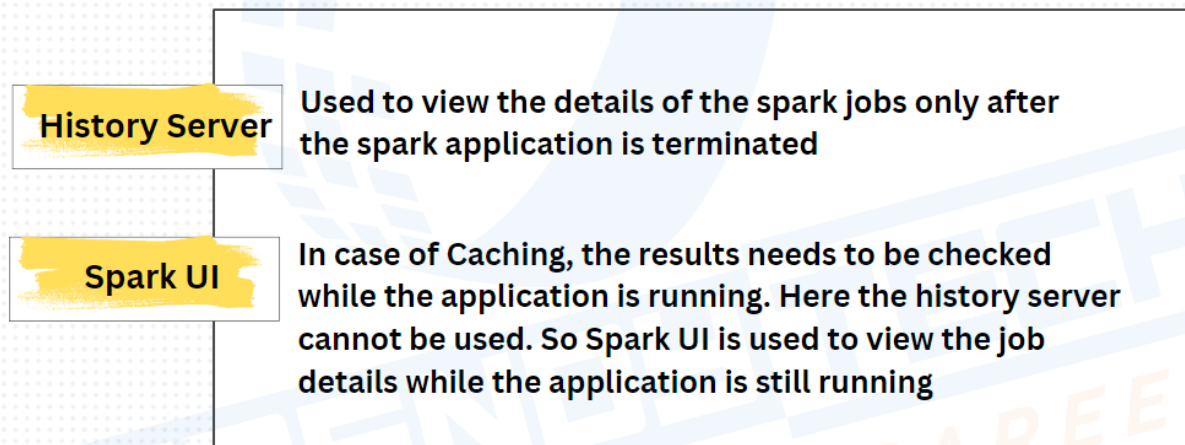


Cache

How to access Spark UI and Resource Manager (YARN)

- Whenever a spark application terminates, the detailed information regarding all the spark jobs are made available in the History Server.
- The application that is running at any given point in time will not show up on the history server until it is terminated using the `spark.stop()` command.

What is the need for Spark UI when we have a History Server?



Proxy way of accessing Resource Manager and Spark UI

1. On Windows - Install Cygwin to connect to the server through SSH
2. On MAC - Use Terminal to connect to the server through SSH
3. Install Foxy Proxy Google extension

Note : A detailed document with the steps to install the required software for accessing the SparkUI is provided under the Downloadable section

What is the need for Cache, When we are working with RDDs and Dataframes that are already in-memory?

- Caching is required to read and save the frequently used data in-memory, without having to hit the disk for subsequent data loading to create the RDD / Dataframe.
- With caching, data needs to be loaded only once initially and can be used for further transformations without having to read from the disk and create RDDs/ Dataframes multiple times.
- The results of transformations which could potentially be used multiple times for further processing can also be cached to save the processing time.

Note :

- Dataframes which are reused multiple times need to be cached for performance benefits.
- Never cache large dataframes that could consume the majority of available memory. Cache medium sized dataframes that will be reused.
- Cache is Lazy

Caching RDDs, Dataframes and Spark Tables

RDDs - By default caches to memory

Dataframes & Highlevel Constructs - By default caches to memory, if there is not enough memory available, then caches to disk.

Dataframes Caching -

Note: If the data is already on Disk, then why cache to disk again?

There are two parts to the Disk space of every Worker Node- HDFS & Local Disk. Initially data is present in HDFS, caching brings this data to the local disk storage with which faster data access can be achieved.

Persist - Is more flexible than cache in terms of the Storage level options. In case of cache, data can be cached only in-memory or on disk. Default Storage levels for persist are memory and disk but this can be changed by setting an optional parameter.

Need for Caching -

- Helps in improving the performance and saving computational cost by avoiding redoing the already performed computations.

Points to note

- Caching can be performed on RDDs, Dataframes and Spark Tables.
- **What kind of Dataframes are best suited for Caching?** Dataframes that are not too large and are reused frequently.

Spark UI is where we can view the currently executed job details of a spark application. It also displays other details like the no.of executors that are allocated.

In case of cached data, the caching details are available under the Storage tab on the Spark UI

Example Dataset considered to explain the concepts - orders_1gb.csv

```
from pyspark.sql import SparkSession
import getpass
username = getpass.getuser()
spark = SparkSession. \
    builder. \
    config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
    enableHiveSupport(). \
    master('yarn'). \
    getOrCreate()
```

```
orders_schema = 'order_id long, order_date date, customer_id long, order_status string'
```

```
orders_df = spark.read \
    .format("csv") \
    .schema(orders_schema) \
    .load("/public/trendytech/orders/orders_1gb.csv")
```

```
orders_df.show()
```

Once the Spark Session is created an entry of this particular spark application will be present in the SparkUI as shown below

hadoop All Application

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used
50301	0	13	50288	39	65 GB

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes
3	0	0	1

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[memory-mb (unit=M), vcores]	<memory:1024, vCores:1>

Tools

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	UI
application_1675889795986_50798	itv006753	pyspark-shell	SPARK		default	0	Tue May 30 16:03:03 +0550 2023	Tue May 30 16:03:03 +0550 2023	N/A	RUNNING	U
application_1675889795986_50798	itv006686	pyspark-shell	SPARK		default	0	Tue May 30 16:01:17 +0550 2023	Tue May 30 16:01:18 +0550 2023	N/A	RUNNING	U
application_1675889795986_50797	itv002708	pyspark-shell	SPARK		default	0	Tue May 30 15:56:28	Tue May 30 15:56:28 +0550	N/A	RUNNING	U

spark 3.0.1 Jobs Stages Storage Environment Executors SQL pyspark-shell application UI

Spark Jobs (?)

User: itv006437
Total Uptime: 42 min
Scheduling Mode: FIFO
Completed Jobs: 4

Initially when the Job Starts,
1 driver and 2 executors are allocated as per the default config

Event Timeline

Executors

- Added
- Removed

Jobs

- Succeeded
- Failed
- Running

Completed Jobs (4)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/05/30 20:32:07	45 ms	1/1	1/1
2	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/05/30 20:18:25	0.1 s	1/1	1/1

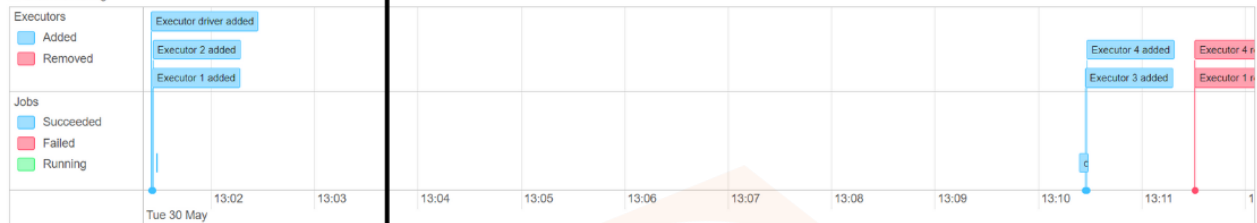
On invoking an action on the Dataframe : Ex - orders_df.count()

- Multiple executors will be dynamically added as required to complete the job execution much faster.

Spark Jobs (7)

User: liv006753
Total Uptime: 11 min
Scheduling Mode: FIFO
Completed Jobs: 2

▼ Event Timeline
☐ Enable zooming



On executing, orders_df.count()
a job is created where,
no.of tasks = no.of partitions

▼ Completed Jobs (2)

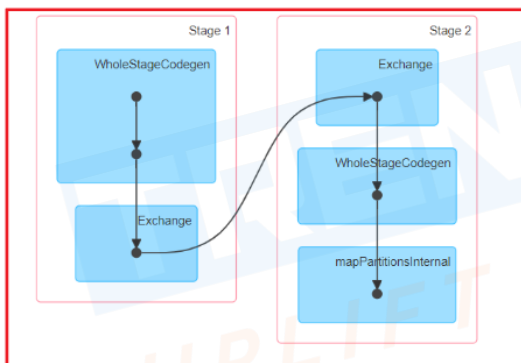
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2023/05/30 13:10:23	5 s	2/2	10/10
0	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/05/30 13:01:27	0.9 s	1/1	1/1



Details for Job 1

Status: SUCCEEDED
Completed Stages: 2

► Event Timeline
▼ DAG Visualization



2 Stages are created as it involves shuffling of data

▼ Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
2	count at NativeMethodAccessorImpl.java:0	+details 2023/05/30 21:30:24	0.5 s	1/1	
1	count at NativeMethodAccessorImpl.java:0	+details 2023/05/30 21:30:19	5 s	9/9	1073.4 MB

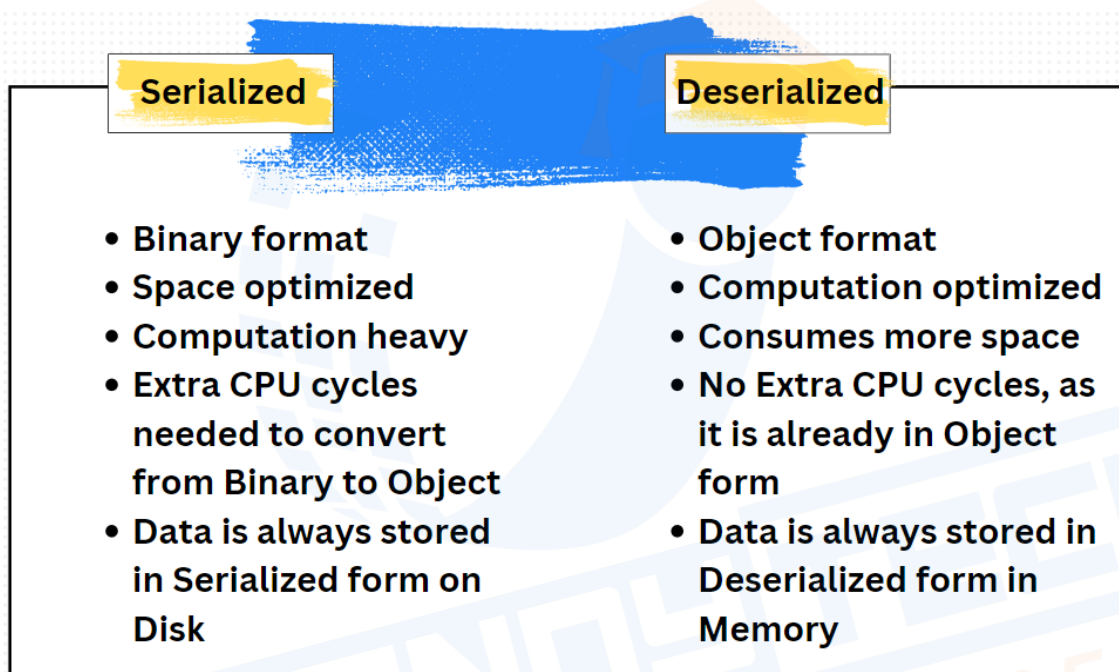
Key points :

1. Spark being smart, caches only the required data.

Ex : Say you are executing an action - head() on a dataset of 1.1GB (No.of partitions created for 1.1GB data = 9)

Spark knows that it needs only the first partition to give the results and thereby caches only the first partition of the entire data.

2. Memory Deserialized 1x Replicated

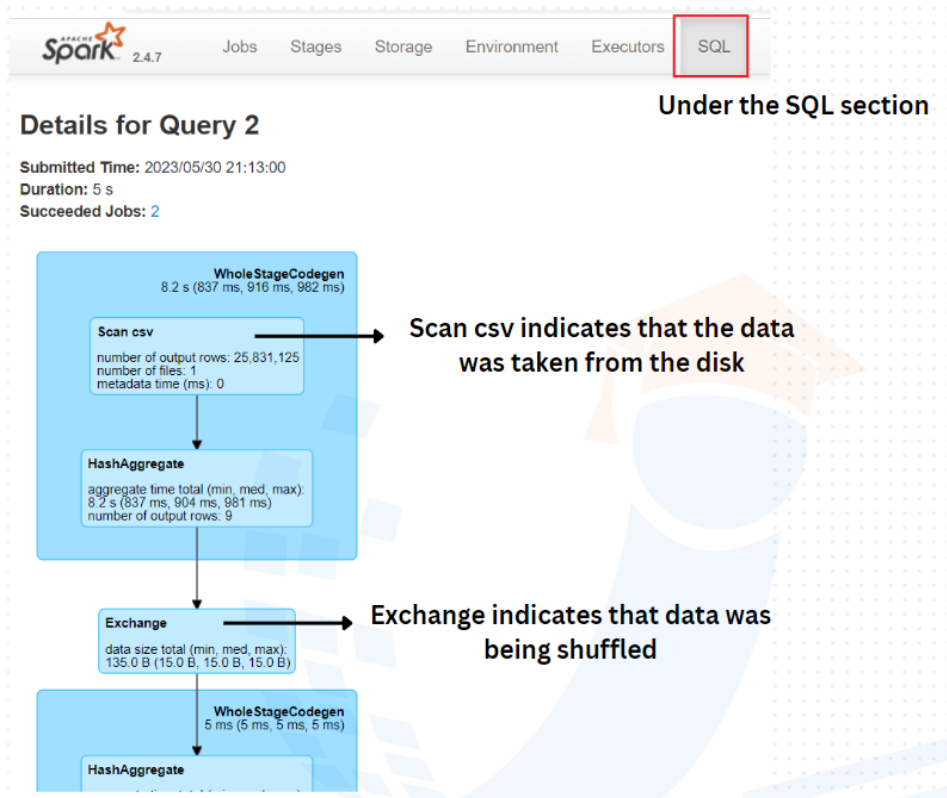


In case of caching :

- a) In Memory - Serialized or Deserialized form
- b) On Disk - Serialized form

3. Initially for the first execution, Cache is Lazy.

Caching would require additional processing time to cache the data for the first time. However, subsequent executions will be much faster as the data is accessed directly from the cache without reaching to the disk.



- On **orders_df.cache()** command, the storage tab is populated with the details of cached data, as shown below

Storage

Cached Data

▼ RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
12	'(1) FileScan csv [order_id#98L,order_date#99,customer_id#100L,order_status#101] Batched: false, Format: CSV, Location: InMemoryFileIndex[hdfs://m01.itversity.com:8000/public/trendytech/orders/orders_1gb.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<order_id:bigint,order_date:date,customer_id:bigint,order_status:string>	Memory Deserialized 1x Replicated	1	11%	33.0 MB	0.0 B

APACHE
Spark[™] 2.4.7

JobsStagesStorageEnvironmentExecutorsSQL

Details for Query 4

Submitted Time: 2023/05/30 21:13:15
Duration: 29 ms
Succeeded Jobs: 4

InMemoryTableScan
number of output rows: 10,000
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)

WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)

Project

CollectLimit

InMemory TableScan indicates
that the data was taken from
cache

4. Locality Level - NODE LOCAL Vs PROCESS LOCAL

NODE_LOCAL works on the principle of data locality. Every worker node has a part/block of the entire data in HDFS saved locally. This portion of data saved locally, will be processed by the respective worker nodes only

PROCESS LOCAL means that the data is present in-memory.

Details for Stage 4 (Attempt 0)

Total Time Across All Tasks: 20 ms
 Locality Level Summary: Process local: 1
 Input Size / Records: 33.0 MB / 1

[DAG Visualization](#)
[Show Additional Metrics](#)
[Event Timeline](#)

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile
Duration	20 ms	20 ms
GC Time	0 ms	0 ms
Input Size / Records	33.0 MB / 1	33.0 MB / 1

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	
5	stdout stderr	w03.itversity.com:37933	30 ms	1	0

Tasks (1)

Index	ID	Attempt	Status	Locality Level	Executor ID	Host
0	12	0	SUCCESS	PROCESS_LOCAL	5	w03.itversity.com

Details for Stage 3 (Attempt 0)

Total Time Across All Tasks: 8 s
 Locality Level Summary: Node local: 1
 Input Size / Records: 128.1 MB / 3081900

[DAG Visualization](#)
[Show Additional Metrics](#)
[Event Timeline](#)

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile
Duration	8 s	8 s
GC Time	83 ms	83 ms
Input Size / Records	128.1 MB / 3081900	128.1 MB / 3081900

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	
5	stdout stderr	w03.itversity.com:37933	8 s	1

Tasks (1)

Index	ID	Attempt	Status	Locality Level	Executor ID	Host
0	11	0	SUCCESS	NODE_LOCAL	5	w03.itversity.com

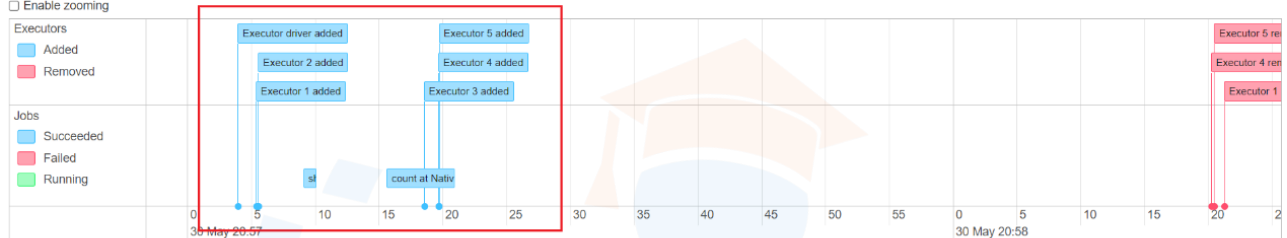
5. Dynamic allocation of executors is beneficial in terms of using the resources efficiently.

Initially a spark application starts with default no.of executors as set in the configurations. The executors will then be dynamically added or retrieved based on the requirement.

Spark Jobs (?)

User: itv006753
Total Uptime: 10 min
Scheduling Mode: FIFO
Completed Jobs: 2

▼ Event Timeline
☐ Enable zooming



▼ Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2023/05/30 20:57:15	5 s	2/2	10/10
0	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/05/30 20:57:09	1 s	1/1	1/1

Jobs are executed faster when multiple executors are running in parallel

The more the number of executors allocated for a job, lesser the task completion time and thereby faster job execution.

6. Unpersist is used to de-cache the data

Unpersist is used to clear the cached data. The cache details under the storage section will be removed on running the unpersist() command.

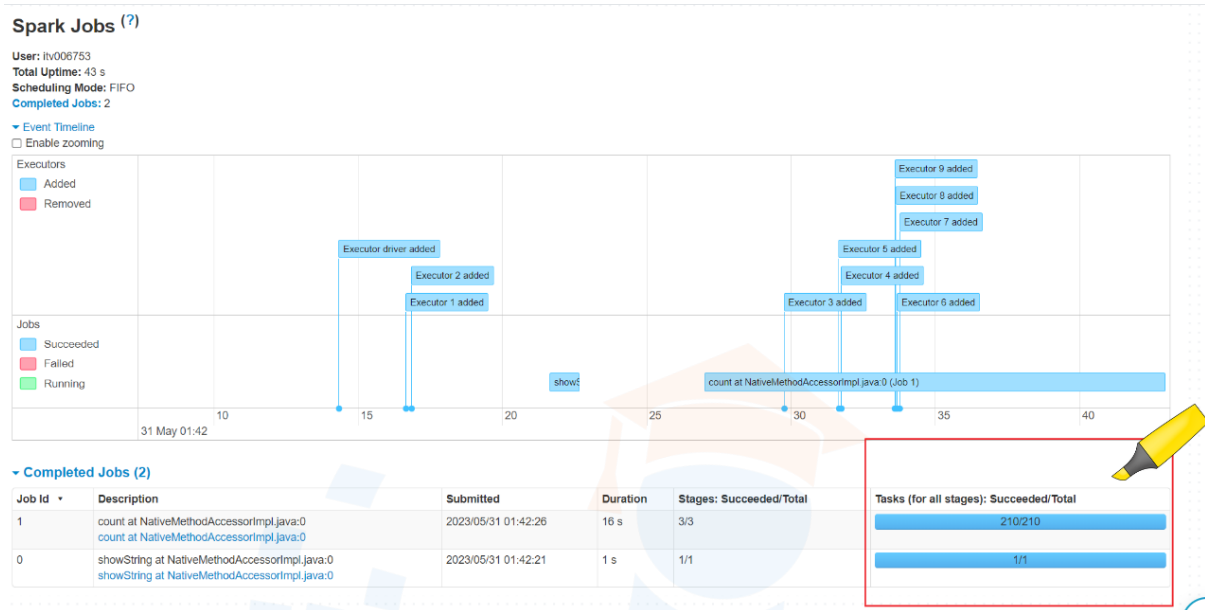
Example Use-Case

Consider an example where we are applying a transformation `distinct()` followed by an action `count()` on a dataframe.

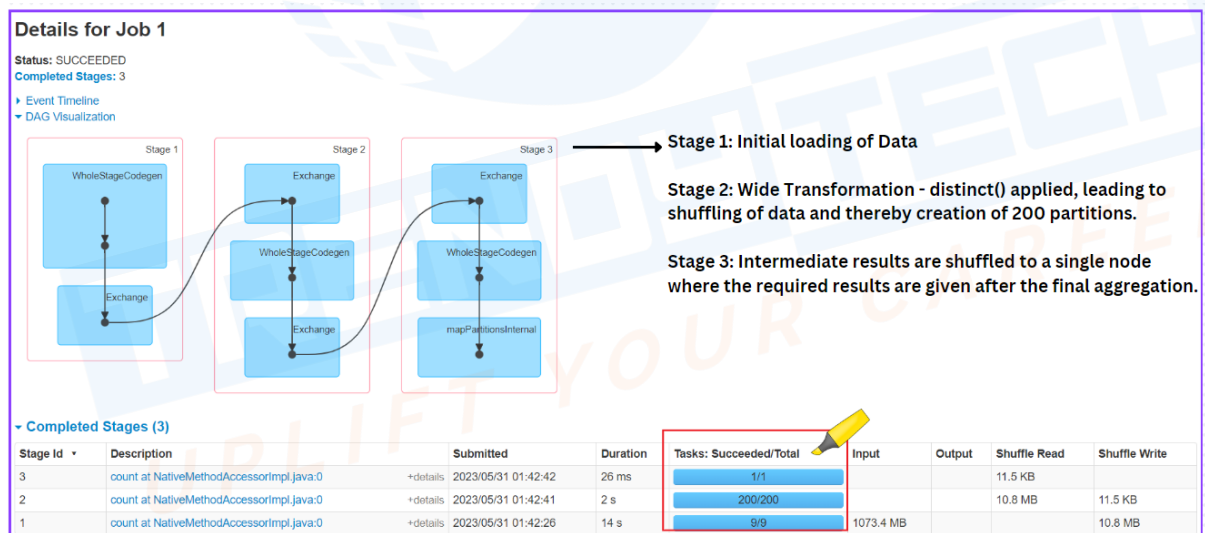
`df.distinct().count()`

- Each executor will perform local distinct and then these distinct intermediate results from all the executors will be shuffled to another machine.

(When a wide transformation is invoked, 200 partitions are created by default. Therefore, the local distinct results from each of the executors of Stage 1 will be spanned across 200 partitions.)



- Task where the final aggregation takes place to give the distinct values.



Why store the cached data in another Dataframe, when it can be used directly?

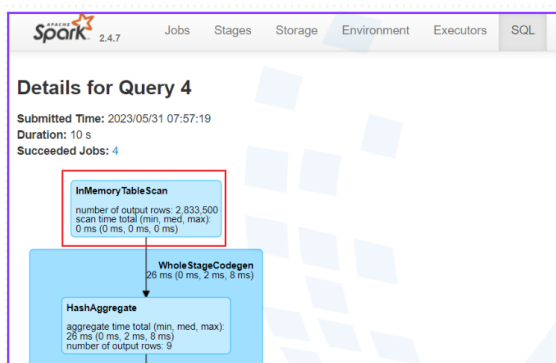
Consider the following :

```
df.select("<only-required-columns-selected>").filter("<only-required-rows>").count()
```

(Since we are applying filters, only the required data / subset of the entire data is cached)

Ex -

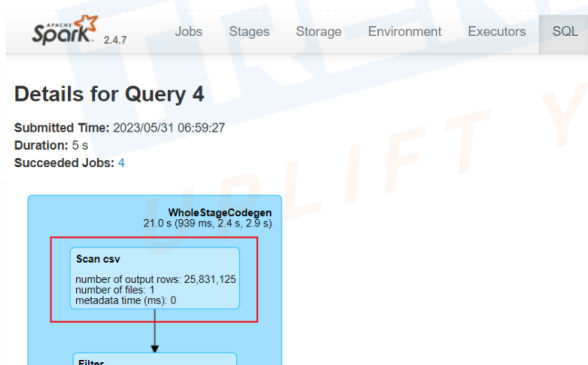
1. `orders_df.select("order_id","order_status").filter("order_status == 'CLOSED').cache()`



Tasks (9)

Index	ID	Attempt	Status	Locality Level	Executor ID	Host	stdout	stderr
0	33	0	SUCCESS	PROCESS_LOCAL	1	w01.itversity.com	stdout	stderr
1	32	0	SUCCESS	PROCESS_LOCAL	2	w03.itversity.com	stdout	stderr
2	34	0	SUCCESS	PROCESS_LOCAL	1	w01.itversity.com	stdout	stderr
3	36	0	SUCCESS	PROCESS_LOCAL	1	w01.itversity.com	stdout	stderr
4	35	0	SUCCESS	PROCESS_LOCAL	2	w03.itversity.com	stdout	stderr

2. `orders_df.filter("order_status == 'CLOSED').select("order_id","order_status").count()`



Tasks (9)

Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	stdout	stderr
0	2	0	SUCCESS	NODE_LOCAL	1	w01.itversity.com	2023/05/31 07:55:24	stdout	stderr
1	3	0	SUCCESS	NODE_LOCAL	1	w01.itversity.com	2023/05/31 07:55:27	stdout	stderr
2	4	0	SUCCESS	NODE_LOCAL	1	w01.itversity.com	2023/05/31 07:55:29	stdout	stderr
3	5	0	SUCCESS	NODE_LOCAL	1	w01.itversity.com	2023/05/31 07:55:32	stdout	stderr

Note : As the Analysed logical plan is different for the above 2 queries, spark wont go ahead with the cached table for the second query where filter

Predicate Push Down (Optimization)- Moving filters ahead, so that spark will be processing as few records as possible.

Data will be taken from the Cache
(InMemory table) for execution if the
Analysed Logical Plan matches

Caching of Spark Tables

spark.write is used to write the results of dataframe as a table to the disk by loading the Data captured in the dataframe as a table in a Database.

```
spark.sql("create database caching_demo_db") // Creating Database
```

```
orders_df.write.format("csv").saveAsTable("caching_demo_db.orders1")
```

// Storing the Dataframe results into a Managed Spark table on Disk. Data is residing in the path - /user/{username}/warehouse and Metadata is present in a Database

```
spark.sql("select count(*) from caching_demo_db.orders1").show()
```

//Invoking count(*) to check the no.of records. Since the data is not cached here, it is scanned from disk

```
spark.sql("cache table caching_demo_db.orders1")
```

//Caching the data. In case of Spark SQL caching is not lazy unlike as in RDDs and Dataframes.

```
spark.sql("uncache table caching_demo_db.orders1")
```

//Un-Caching the data. On uncache, the data will no longer be in memory and it has to be scanned from disk. The data locality will change to NODE_LOCAL from PROCESS_LOCAL

```
spark.sql("cache lazy table caching_demo_db.orders1")
```

//Cache is eager in case of Spark SQL. In order to make it Lazy, add a lazy keyword

Note : Whenever the data changes (Ex- New record inserted), spark is smart to refresh the cache and thereby display the right results

Different ways of uncaching -

1. `spark.sql("clear cache")` //clears all the cached objects of the session.
2. `spark.sql("uncache table <tableName>")` //clears only the specified table.
3. `spark.catalog.clearCache()`

Creating External Spark Table

```
spark.sql("create database caching_demo_db_ext") // Creating Database

spark.sql("create table caching_demo_db_ext.orders_ext(order_id long, order_date string, customer_id long, order_status string) using csv location '/user/itv006753/orders'") //Creating spark external table

spark.sql("describe extended caching_demo_db_ext.orders_ext").show() //query to check if the table is an external table

spark.sql("cache table caching_demo_db_ext.orders_ext") //Caching the data. In case of Spark SQL caching is not lazy rather eager caching

spark.sql("uncache table caching_demo_db.orders1") //Un-Caching the data. On uncache, the data will no longer be in memory and it has to be scanned from disk. The data locality will change to NODE_LOCAL from PROCESS_LOCAL

spark.sql("insert into caching_demo_db_ext.orders_ext values (111111, '2023-05-29', 222222, 'BOOKED')") //Inserting a record
```

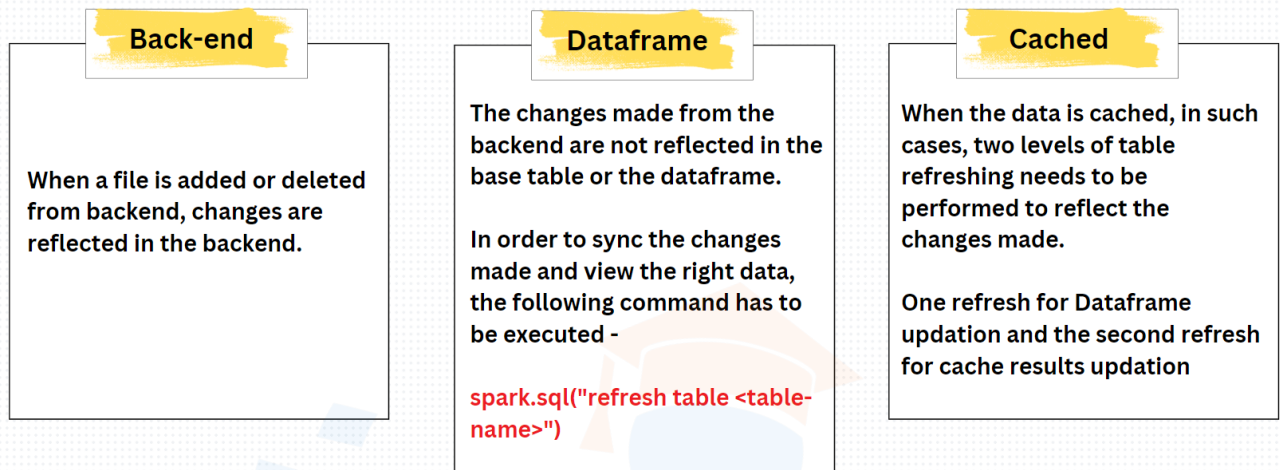
Points to note :

- When data is inserted using the insert command, then spark automatically keeps track of the changes made and it will refresh the invalidated cache for subsequent execution.
- However, when the files are manually added / removed from the backend, then spark cannot track these changes to refresh the cache.
- On inserting incremental data, it is updated at the backend but the cache does not automatically get refreshed. It needs to be manually refreshed to display the correct results using the following command

`spark.sql("refresh table <table-name>")`

- Cache gets invalidated when the data is changed at the back-end.

3 Layers of Data



2 kinds of file formats

Row-based : Data is stored in a row-by-row format. Ex - csv, avro

In case you need to read only a subset of data(only a few columns), with row-based format, the entire data has to be scanned.

Row-based style of storing data

```
1, 2013-07-25 00:00:00.0, 11599, CLOSED
2, 2013-07-25 00:00:00.0, 256, PENDING_PAYMENT
```

Column-based : Data is stored in a columnar fashion. Ex - parquet, orc

In this case, a subset of the entire data can be scanned. That is, only the required columns can be read.

Column-based style of storing data

```
1 2 | 2013-07-25 00:00:00.0 2013-07-25 00:00:00.0 | 11599 256 | CLOSED PENDING_PAYMENT
```


Parquet is the default format that is most compatible with Spark.

Parquet provides space optimization and that is the reason when any action is invoked on a dataframe of a parquet file, it takes noticeably less time in comparison to other formats like csv.

When can Caching lower the performance?

Parquet without caching



Metadata is used for executing an action, therefore it is much faster

Parquet with caching



Scans the entire data and thereby takes more processing time.

Persist

Persist works exactly the same way as cache but with persist, there is an additional flexibility of changing the default storage levels with an optional parameter.

Persist Storage Level arguments

1. **Disk** - Whether the data has to be persisted in Disk? True/False
2. **Memory** - Whether the data has to be persisted in Memory? True/False
3. **Off heap** - Whether the data has to be persisted Off heap? True/False
4. **Deserialized** - Whether the data is serialized? True/False
5. **Number of Cache Replicas**

```
from pyspark.storagelevel import StorageLevel
```


```
orders_df.persist(StorageLevel(True, True, False, True, 1))
```

If Memory full,
cache to Disk

Persist in Memory

Data
Deserialized

One Replica


2.4.7

[Jobs](#)
[Stages](#)
[Storage](#)
[Environment](#)
[Executors](#)
[SQL](#)

pyspark-shell application

Storage

▼ RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
38	*(1) FileScan csv [order_id#172L,order_date#173,customer_id#174L,order_status#175] Batched: false, Format: CSV, Location: InMemoryFileIndex[hdfs://m01.itversity.com:9000/public/trendytech/orders/orders_1gb.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<order_id.bigint,order_date.date,customer_id.bigint,order_status.string>	Disk Serialized 1x Replicated	1	11%	0.0 B	15.9 MB

Command to unpersist the data

```
orders_df.unpersist()
```

Other ways of executing Persist

```
=> orders_df.persist(StorageLevel(True,False,False,True,1))
```

|| Equal to ||

```
orders_df.persist(StorageLevel.DISK_ONLY)
```

```
=> orders_df.persist(StorageLevel(True, True, False, True, 1))
```

|| Equal to ||

```
orders df.persist(StorageLevel.MEMORY_AND_DISK)
```

```
=> orders_df.persist(StorageLevel(True,True,False,False,1))
```

|| Equal to ||

```
orders df.persist(StorageLevel.MEMORY AND DISK SER)
```