The Apache Spark User Interface

Objectives

After watching this video, you will be able to:

- Explain how to connect to the Apache Spark user interface web server
- Navigate basic elements of the Spark user interface

Access the Application UI

Start your application
Connect to the application UI using the following URL:

```
http://<driver-node>:4040
```

The SparkContext starts a web server for the application UI

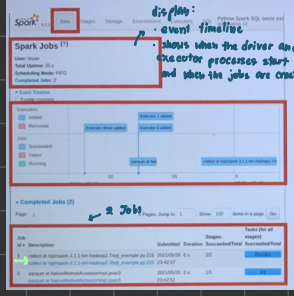
- The driver program is the host
- The port defaults to 4040
- The UI is available only while the application is active

Why use the Application UI

The Spark user interface shows information about the running application:

- Shows jobs, stages and tasks
- Storage of persisted RDDs and DataFrames
- Environment configuration and properties
- Executor summary
- SQL information (if SQL queries exist)

Jobs Information



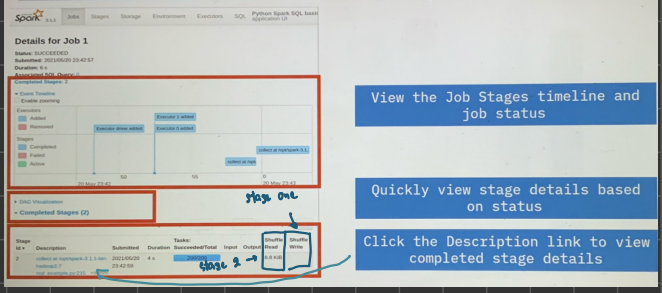
The Jobs tab

Jobs summary info

Jobs event timeline

Click the Description link to view completed job details

Job Details

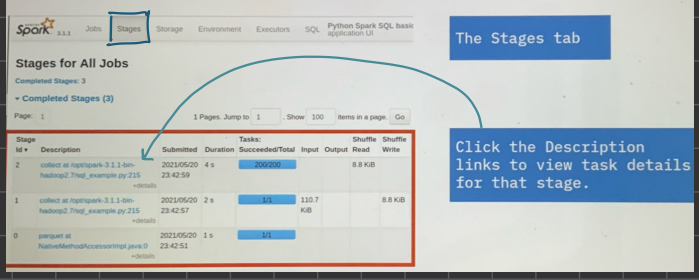


View the Job Stages timeline and job status

Quickly view stage details based on status

Click the Description link to view completed stage details

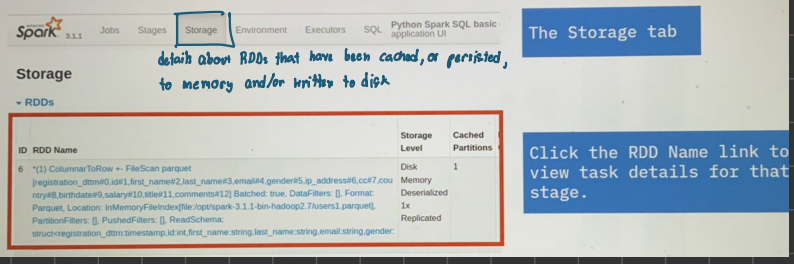
Stages Information



The Stages tab

Click the Description links to view task details for that stage.

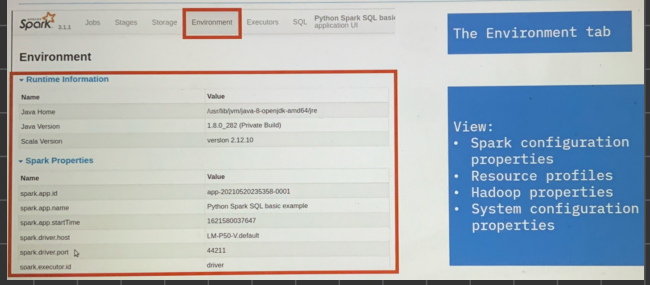
Storage Information



The Storage tab

Click the RDD Name link to view task details for that stage.

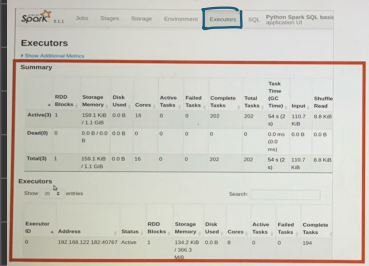
Environment Information



The Environment tab

View:
- Spark configuration properties
- Resource profiles
- Hadoop properties
- System configuration properties

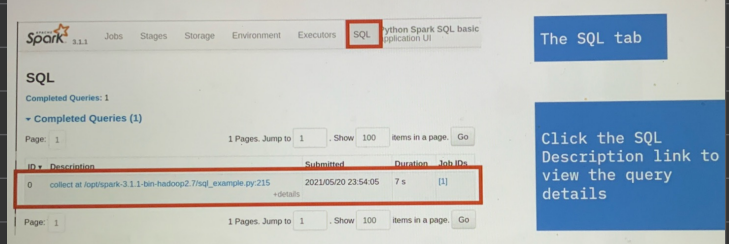
Executors Information



The Executors tab

Scroll the window to view
- Spark configuration properties
- Resource profiles
- Hadoop properties
- System configuration properties

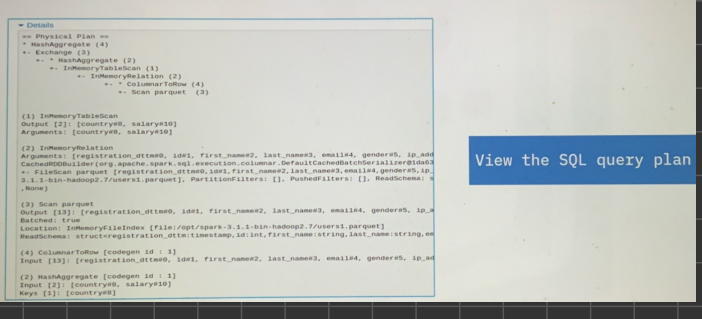
SQL Information



The SQL tab

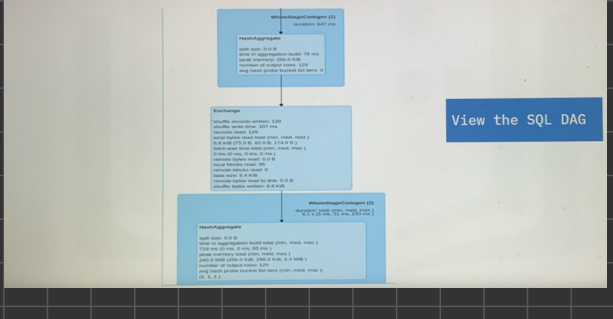
Click the SQL Description link to view the query details

SQL Query Details – query plan



View the SQL query plan

SQL Query Details - the DAG



View the SQL DAG

Monitoring Application Progress

Objectives

After watching this video, you will be able to:

- Define parts of the application workflow
- Utilize the UI webserver to monitor application progress

Why Monitor an Application

Monitoring an application using the Spark Application UI provides these benefits:

- Quickly identify failed jobs and tasks → and find the root cause of failure
- Fast access to locate inefficient operations
- Application workflow optimization

How Does the Application Flow

Multiple related jobs

- From different data sources
- One or more DataFrames
- Actions applied to the DataFrames

Workflows can include:

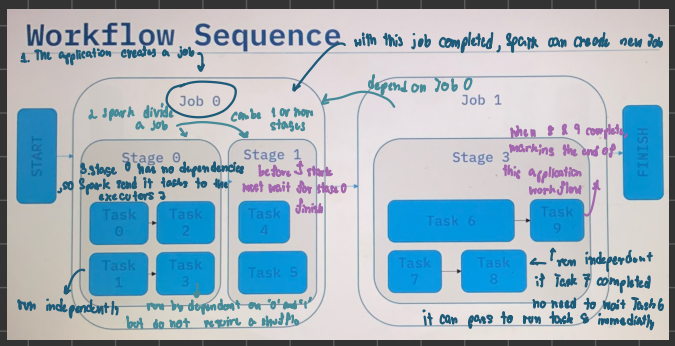
- Jobs created by the SparkContext in the driver program
- Jobs in progress running as tasks in the executors
- Completed jobs transferring results back to the driver or writing to disk

How Do Jobs Progress?

1. Spark jobs divide into stages, which connect as a directed acyclic graph (DAG)
2. Tasks for the current stage are scheduled on the cluster
3. As the stage completes all its tasks, the next dependent stage in the DAG begins
4. The job continues through the DAG until all stages complete

If any tasks within a stage fail, after several attempts, Spark marks the task, stage, and job as failed and stops the application

Workflow Sequence



Application Code Example

Monitoring a PySpark application

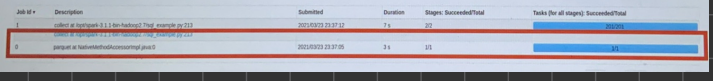
```
# Read a Parquet file from disk
df = spark.read.parquet("users.parquet")

# Select certain columns and cache to memory
df = df.select("country", "salary").cache()

# Group data by country and compute mean salary per country
mean_salaries = df.groupBy("country").agg({"salary": "mean"}).collect()
```

file format
result file loaded from disk to produce a DataFrame
using this, 2 columns are selected
triger a job creation and schedules the task
calculating the mean of salary

Application Jobs Example

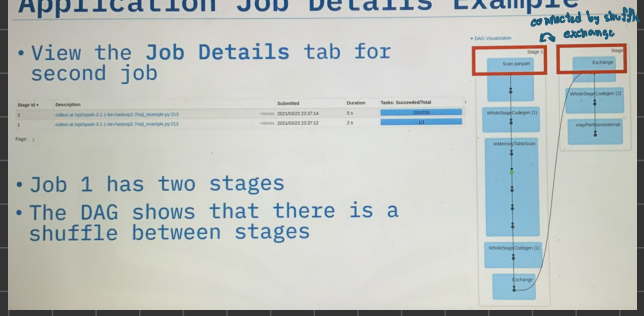
The Spark Application UI **Jobs** tab displays two jobs

- Job 1: Reads the Parquet file
- Job 2: Performs the action to collect (lazily computed) the computations to send to the driver



Application Job Details Example

- View the **Job Details** tab for second job



- Job 1 has two stages
- The DAG shows that there is a shuffle between stages

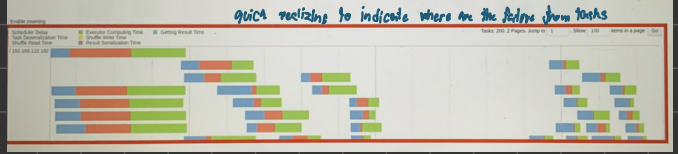
Application Stage Details Example

indicates each task's state using color coding

Select a stage to view the tasks

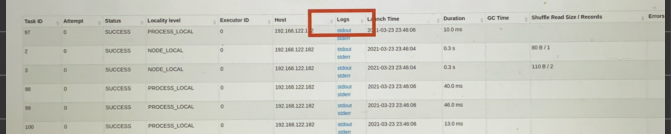
See which tasks are scheduled, their duration, and other task states

quick resizing to indicate where are the failure from tasks



Application Task Status Example

- View the task status, duration, and amount of data transferred
- Access the executor logs for that task



Application Completion Example

- The application jobs complete
- The SparkContext stops → can be manually or automatically
- The application UI is no longer available
- If event logging is enabled, view the application UI by starting the Spark History Server and connecting to it

means all events in the app. workflow are logged to a file, and the UI can be viewed History without losing it if it got some accident that shut down the server

Viewing UI with History Server

Before the application is started, verify that event logging is enabled

```
spark.eventLog.enabled true
spark.eventLog.dir <path-for-log-files>
```

View the application UI by connecting to Spark History Server

```
# Command to connect to the Spark application UI history server
http://<host-url>:18080
```

Start the History server

```
# Command to start the History Server
./sbin/start-history-server.sh
```

Debugging Apache Spark Application Issues

Objectives

After watching this video, you will be able to:

- Identify common Apache Spark application issues
- Debug issues through the application UI
- Describe application log file locations and content

Common Application Issues

Common areas of Spark application issues include:

- User Code
- Configuration
- Application Dependencies
- Resource Allocation

What is User Code

User code is made up of:

Driver program: code that runs in the driver process and the functions and variables serialized that the executor runs in parallel

Serialized closures contain the code's necessary functions, classes, and variables

The serialized closures are distributed into the cluster to run in parallel by the executors

User Code Issues

- User code can error in the driver

Syntax	Serialization
Data validation	Errors located in other code locations

→ Reports task errors to the driver program

The application will terminate

the cause of failure, can be found in the driver event log

Application dependency issues

Application dependencies include:

- Application files:
  - Source files examples: Python script files, Java JAR files
  - Required data files
- Application libraries

Dependencies must be available for all nodes of the cluster

Application Resource Issues

- CPU and memory resources must be available for tasks to run
- Driver and executor processes require some amount of CPU and memory to run
- Any Worker with free resources can start processes
- Resources are acquired until a process completes
- If resources are not available, Spark retries until a worker is free
- Lack of resources results in task time-outs, also called task starvation!

Examining the Log Files

↳ provide detail that give more insight of failure causes

- Application logs are found in `work/` directory named as work/<application-id>/<stdout|error>

Spark installation directory

- Spark standalone writes master/worker logs to the `log/` directory

Understanding Memory Resources

Objectives

After watching this video, you will be able to:

- Describe memory parameters
- Describe Spark memory management
- Describe Spark data persistence
- Explain how to set executor memory on submit
- Explain how to set memory for Spark Standalone memory and cores

Configuring Spark Process Memory

Spark applications allow configuration of driver and executor memory

Upper limit on useable memory enables apps to run without using all available cluster memory

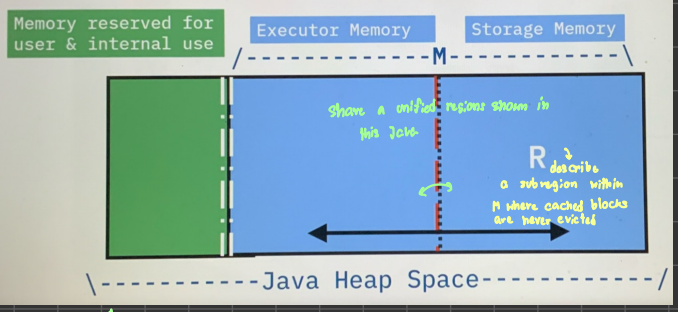
Exceeding memory requirements causes disk spill or out-of-memory errors

Memory Setting Considerations

Executor Memory:
- Processing
- Caching
- Excessive caching leads to issues

Driver Memory:
- Loads data, broadcasts variables
- Handles results, such as collections

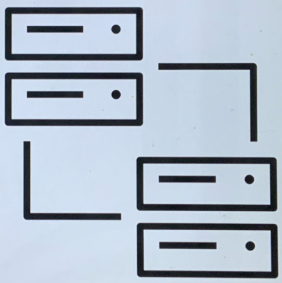
Spark Unified Memory



- This design ensures several preferable properties
  1. app. that do not use caching can use the entire space for executor memory
  2. app. that do not use caching can reserve a minimum storage space

Spark Data Persistence

Spark data persistence:
- Store intermediate calculations for reuse
- Persist to memory/disk (can be reuse the same data)
- Less computation
- Faster iterations over data when training a model



Data Persistence Example

Caching a DataFrame with a features column of random values

```
# Define DataFrame with feature column
df = spark.range(100).withColumn("features", rand()).cache()

# Features are generated then cached first time action is applied
print(df.filter(col("features") > 0.5).count())

# Cached DataFrame with features is reused in subsequent calls
print(df.filter(col("features") < 0.5).count())
```

Setting Memory on Submit

Set the executor memory for a Spark standalone cluster: (good practice to configure memory every time)

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://<spark-master-URL>:7077 \
  --executor-memory 10G \
  /path/to/examples.jar \
  1000
```

Set each executor to use up to 10GB of memory

Setting Worker Node Resources

Set the Spark Standalone worker memory and cores:

```
# Start standalone worker with MAX 10Gb memory, 8 cores
$ ./sbin/start-worker.sh \
    spark://<spark-master-URL> \
    --memory 10G --cores 8
```

Default
- All available memory minus 1GB
- All available cores

Understanding Processor Resources

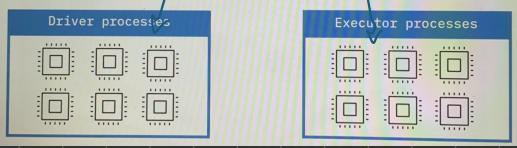
Objectives

After watching this video, you will be able to:

- Explain how Apache Spark uses processor cores
- Describe how to use Spark to configure cores for an application

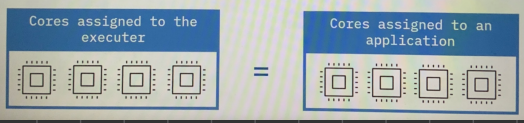
CPU Cores as a Spark Resource

- Spark assigns CPU cores to driver and executor processes



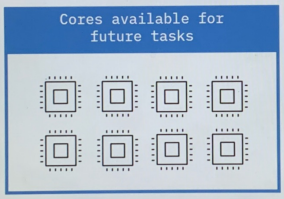
CPU Cores as a Spark Resource

- Parallelism is limited by the number of cores available
- Executors process tasks in parallel up to the number of cores assigned to the application



CPU Cores as a Spark Resource

- After processing, CPU cores become available for future tasks



CPU Cores as a Spark Resource

- Workers in the cluster contain a limited number of cores
- If no cores are available to an application, the application must wait for currently running tasks to finish

Default CPU Core Usage

- Spark queues tasks and waits for available executors and cores for maximized parallel processing
- Parallel processing tasks mainly depend on the number of data partitions and operations
- Application settings will override default behavior

Setting Executor Cores on Submit

- Specifies the number of executor cores for a Spark standalone cluster *per executor process*

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://<spark-master-URL>:7077 \
  --executor-cores 8 \
  /path/to/examples.jar \
  1000
```

Setting Executor Cores on Submit

- Specifies the executor cores for a Spark standalone cluster *for the application*

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://<spark-master-URL>:7077 \
  --total-executor-cores 50 \
  /path/to/examples.jar \
  1000
```

Setting Worker Node Resources

```
# Start standalone worker with MAX 10Gb memory, 8 cores
$ ./sbin/start-worker.sh \
    spark://<spark-master-URL> \
    --memory 10G --cores 8
```

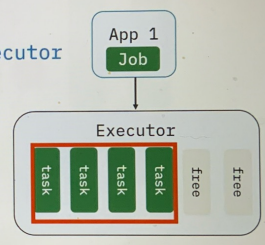
- By default, Spark uses all available memory minus 1GB and all available cores

Core Utilization Example

A Spark standalone cluster with one worker node and six cores

- Submit App 1
- The application requests 4 cores per executor

```
$ ./bin/spark-submit \
  --master spark://<spark-master-URL>:7077 \
  --executor-cores 4 \
  examples/src/main/python/pi.py \
  1000
```

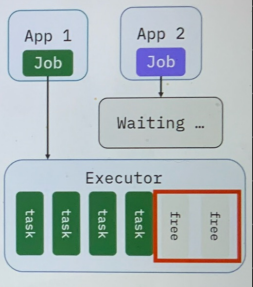


- App 1 occupies four cores

Core Utilization Example

- Submit App 2 using the same cluster
- The application requests 4 cores per executor

```
$ ./bin/spark-submit \
  --master spark://<spark-master-URL>:7077 \
  --executor-cores 4 \
  examples/src/main/python/pi.py \
  1000
```

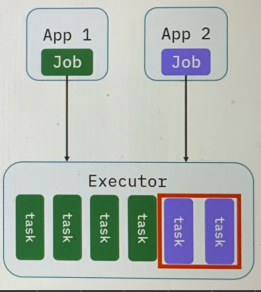


- Two cores are available
- App 2 must wait until two additional cores are available

Core Utilization Example

- Start App 2 and request two cores

```
$ ./bin/spark-submit \
  --master spark://<spark-master-URL>:7077 \
  --executor-cores 2 \
  examples/src/main/python/pi.py \
  1000
```



- App 1 and App 2 now run simultaneously