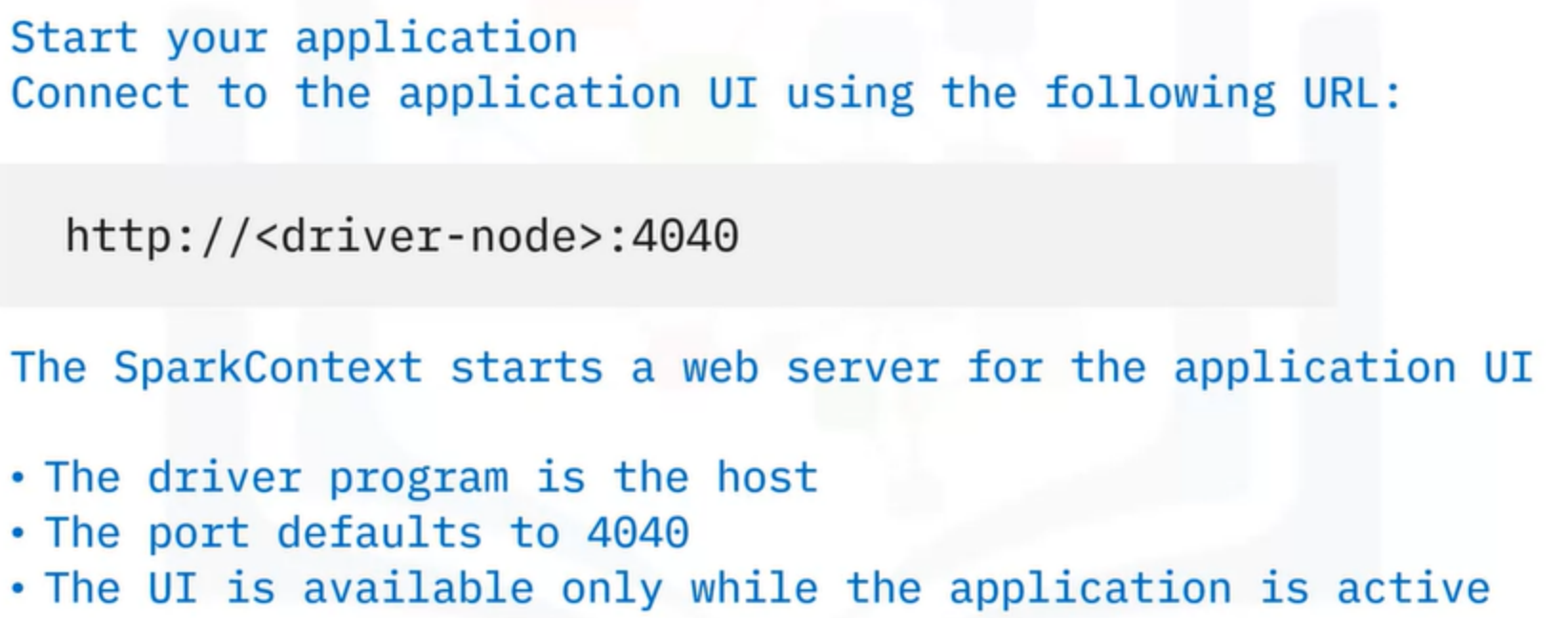
Week 6 - Monitoring and Tuning

The Apache Spark User Interface

Acceess the Application UI

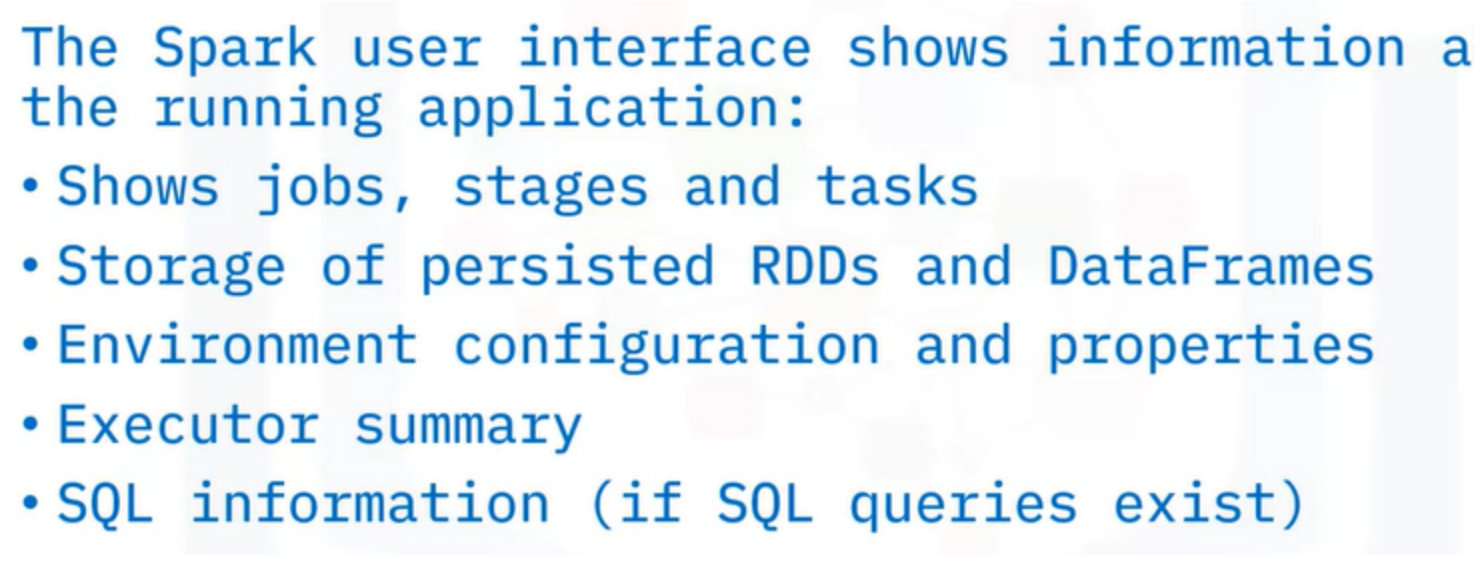


When a Spark application is being run, as the driver program creates a SparkContext, Spark starts a web server that serves as the application user interface.

You can connect to the UI webserver by entering the hostname of the driver followed by port 4040 in a browser once that application is running.

The web server runs for the duration of the Spark application, so once the SparkContext stops, the server shuts down and the application UI is no longer accessible.

Why use the Application UI



The Spark user interface provides valuable insights, organized on multiple tabs, about the running application.

The `Jobs tab` displays the application's jobs, including job status.

The `Stages tab` reports the state of tasks within a stage.

The `Storage tab` shows the size of any RDDs or DataFrames that persisted to memory or disk.

The `Environment tab` information includes any environment variables and system properties for Spark or the JVM.

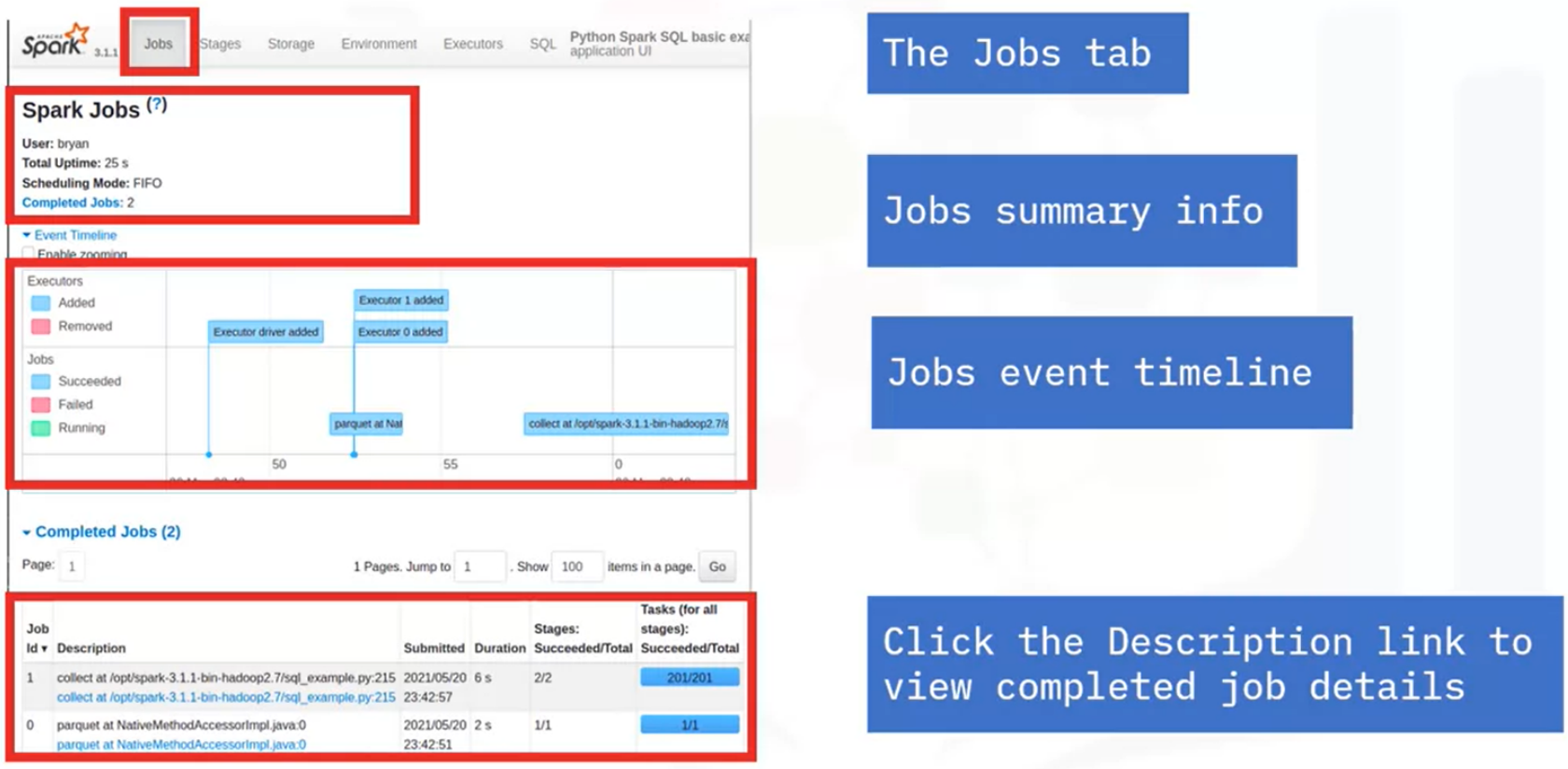
The `Executor tab` displays a summary that shows memory and disk usage for any executors in use for the application.

Additional tabs display based on the type of application in use.

For example, if the application performs SQL queries, the SQL tab displays and shows metrics for each query.

Job Information

The top of the window displays the event timeline and shows when the driver and executor processes start and when the jobs are created.



In this example, the application has completed two jobs. These two jobs appear in the list at the bottom which also has additional job-related information such as duration, number of stages, and number of total tasks for all stages. The job list also displays Description hyperlinks to view the job details.

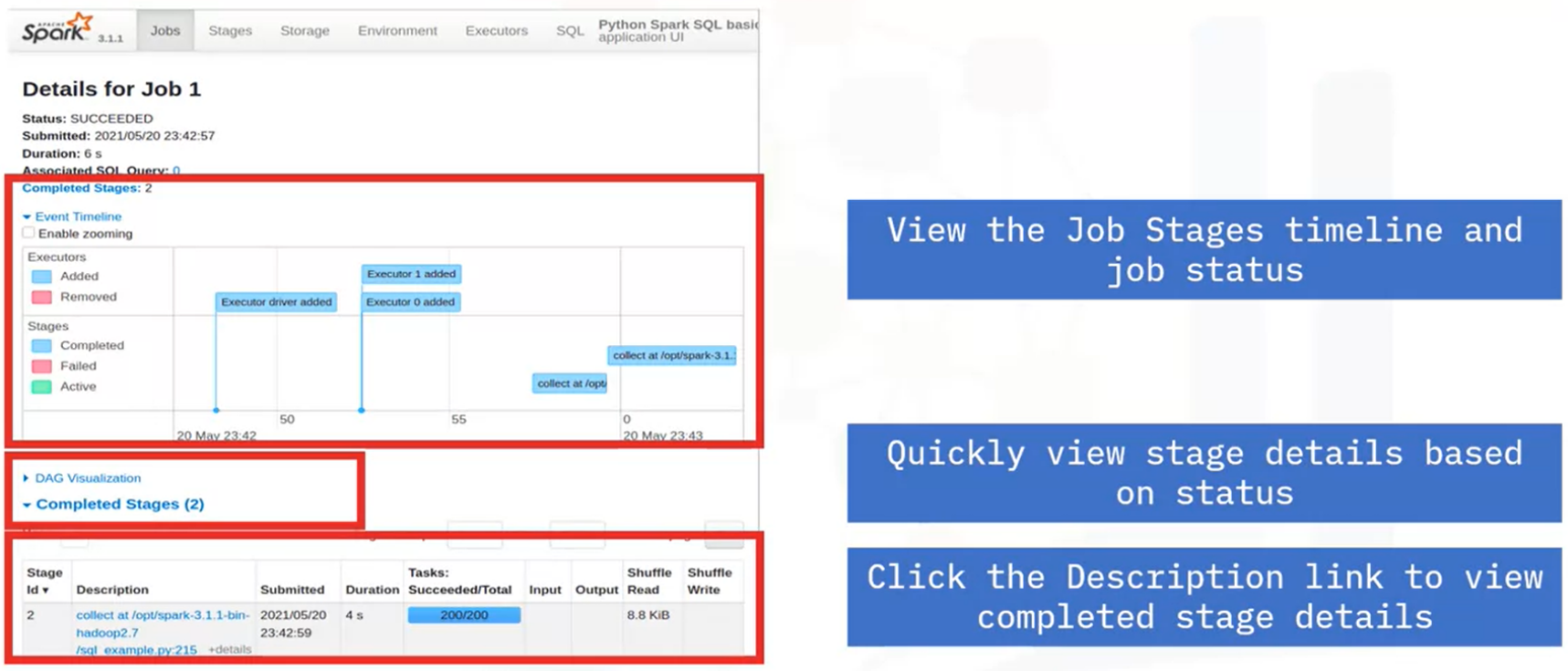
Job Details

Job Details for information about the different stages of a specific job.

The timeline displays each stage, where you can quickly see the job's timing and duration.

Below the timeline display you'll see Completed Stages. In the parentheses beside the heading, you'll see a quick view that displays the number of completed stages.

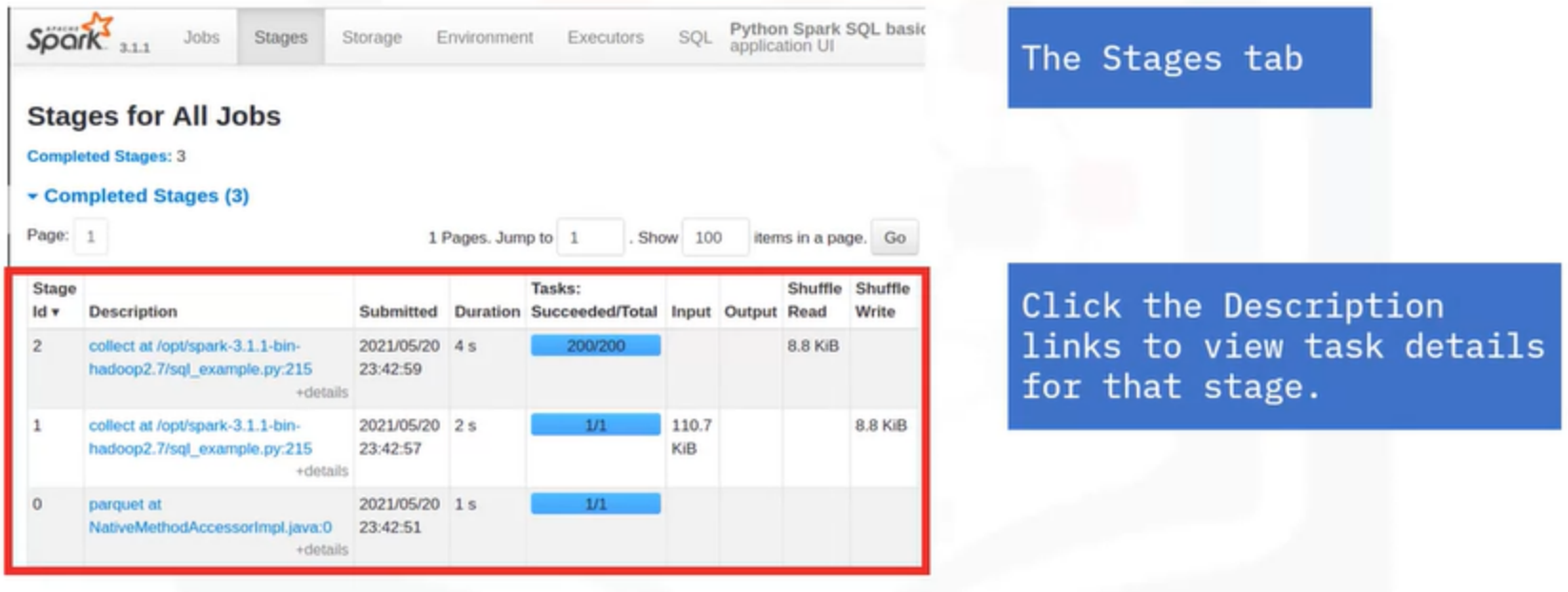
Then, view the list of the stages within the job and job metrics including when the job was submitted, input/output sizes, the number of attempted tasks, the number of succeeded tasks, and how much data was read or written because of a shuffle.



This example shows two stages separated by a shuffle. So, stage one wrote shuffle data and stage two read that data. To view more details about specific stage within a job, click the Stage ID Description hyperlinks.

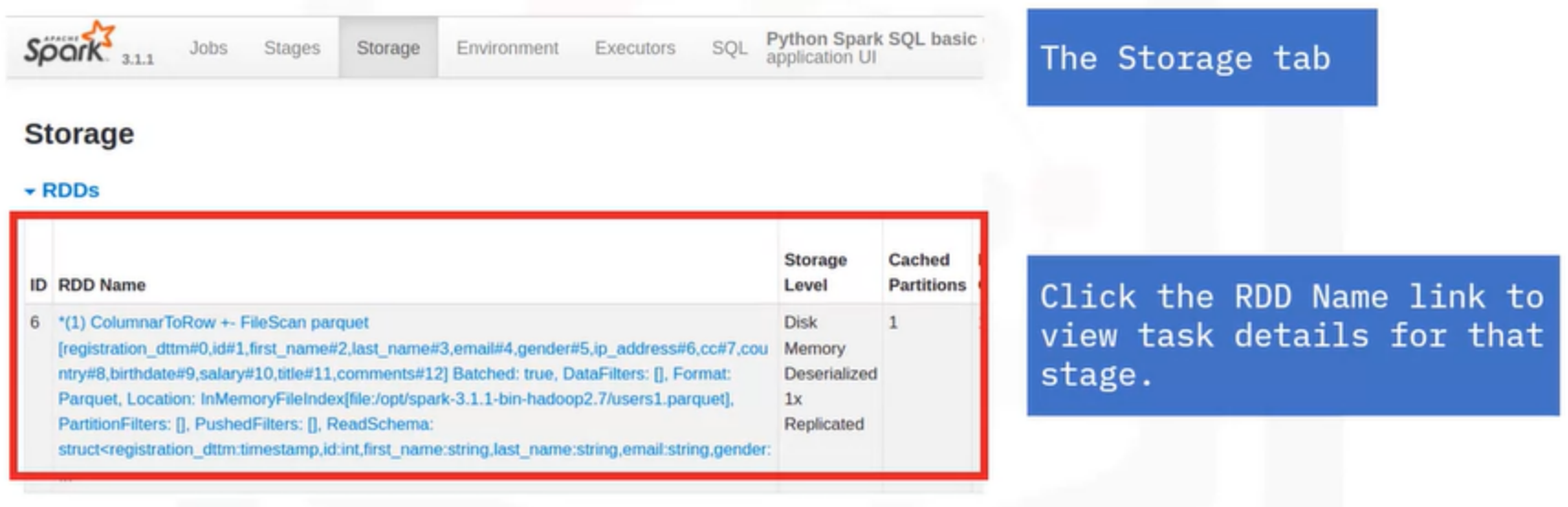
Stages Information

The Stages tab shows a list of all stages in the application, grouped by the current state of either completed, active, or pending. This example displays three completed stages. Click the Stage ID Description hyperlinks to view task details for that stage.



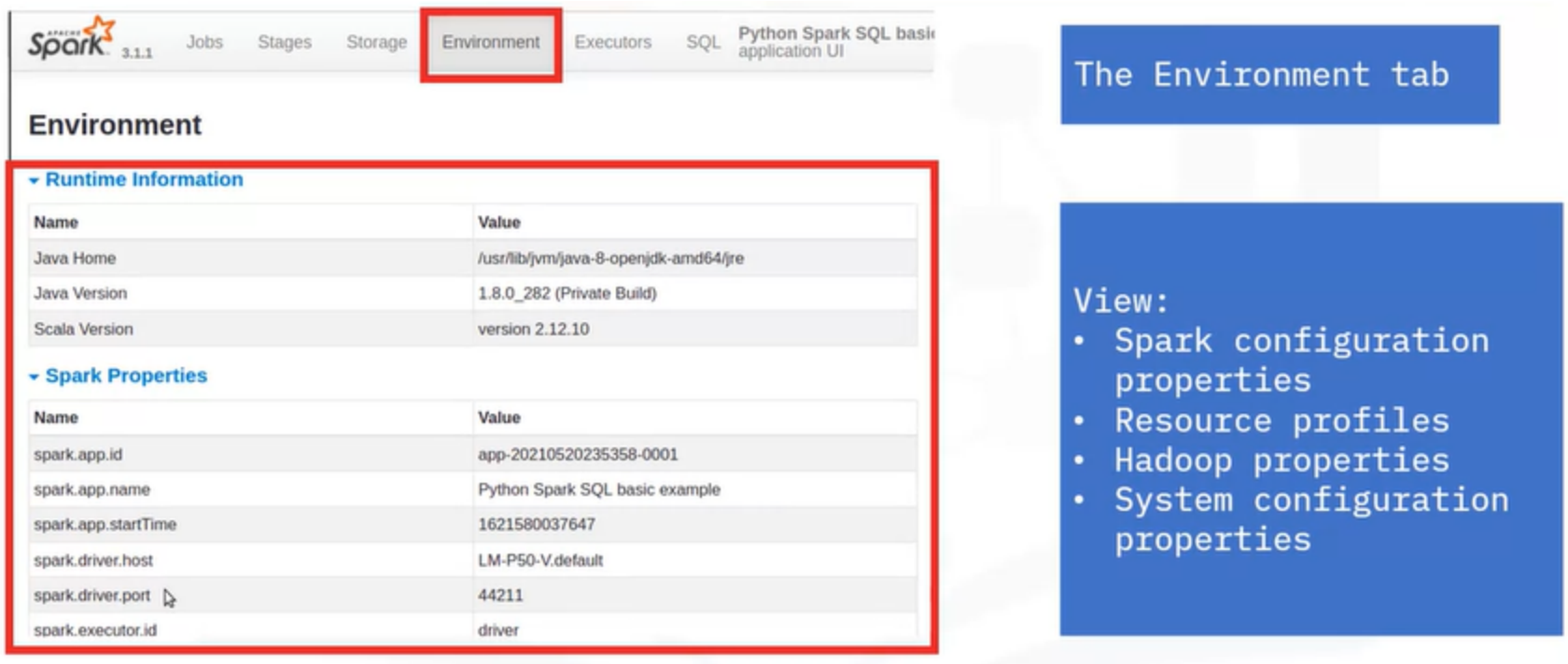
Storage Information

The Storage tab displays details about RDDs that have been cached, or persisted, to memory and/or written to disk. This example image shows one RDD that's been cached to memory only.



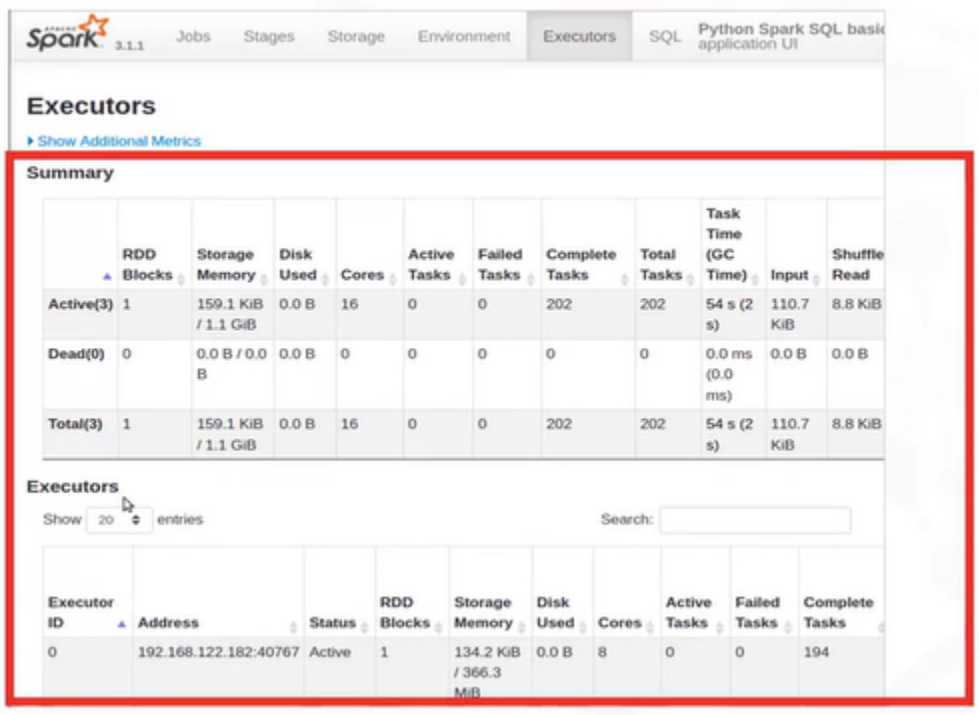
Environment Information

The Environment tab has several lists to describe the environment of the running application. These lists include the Spark configuration properties and resource profiles, as well as properties for Hadoop and the current system properties.



Executor Information

The Executors tab displays a summary table at the top that shows metrics on all active or dead executors, along with a full range of metrics on the task numbers, data I/O, disk, and memory usage. At the bottom is a list of all executors, including the driver, that have been in use during the application, again with a full range of metrics. This list also provides links to the stdout and stderror log messages of the executor process.

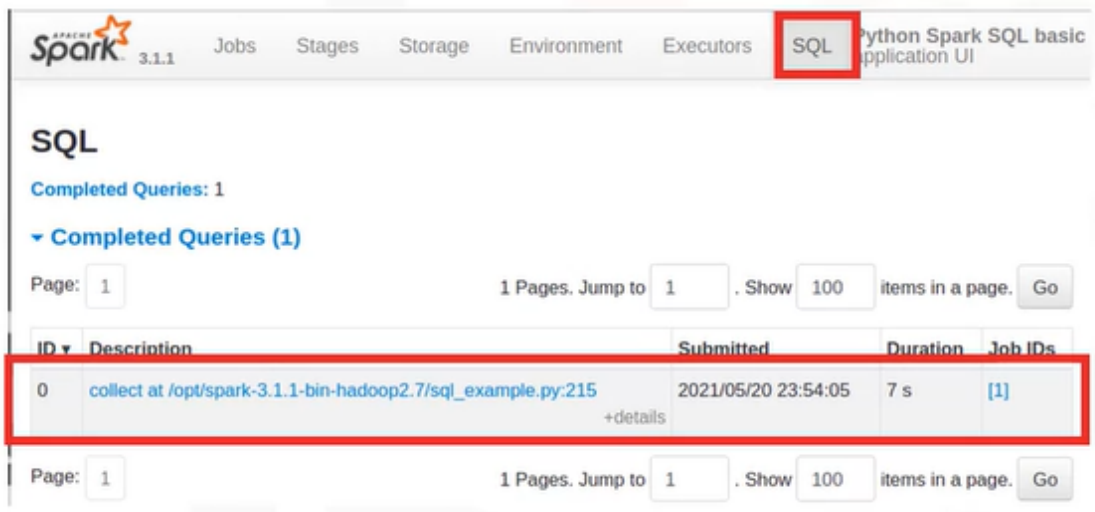


The Executors tab

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

SQL Information

If the application runs SQL queries, the UI displays and populates the SQL tab. Click the SQL Description hyperlink to display the query's details.



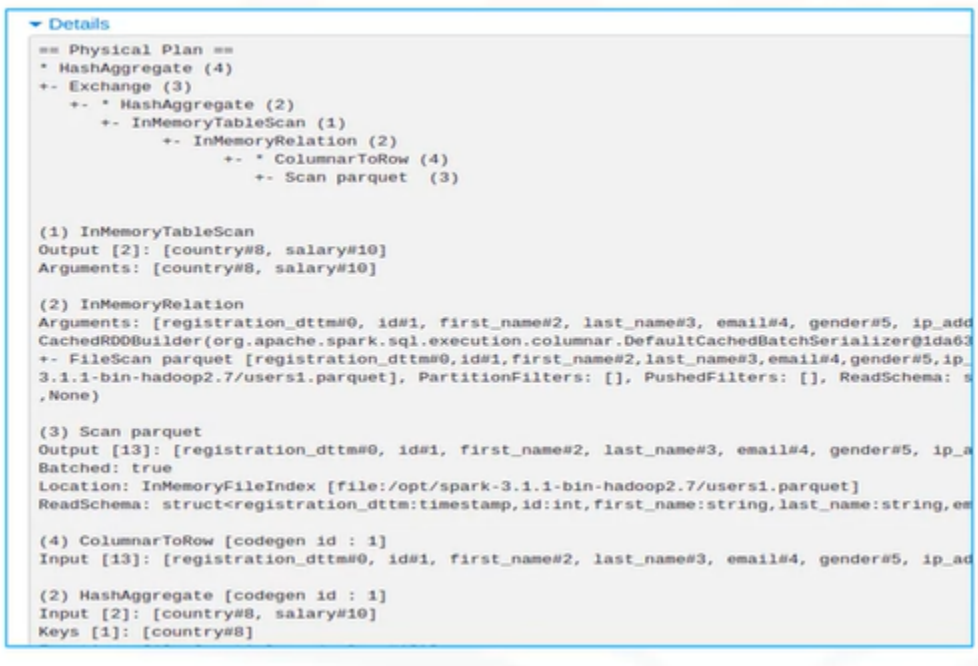
The SQL tab

Click the SQL Description link to view the query details

SQL Query Detail

Query plan

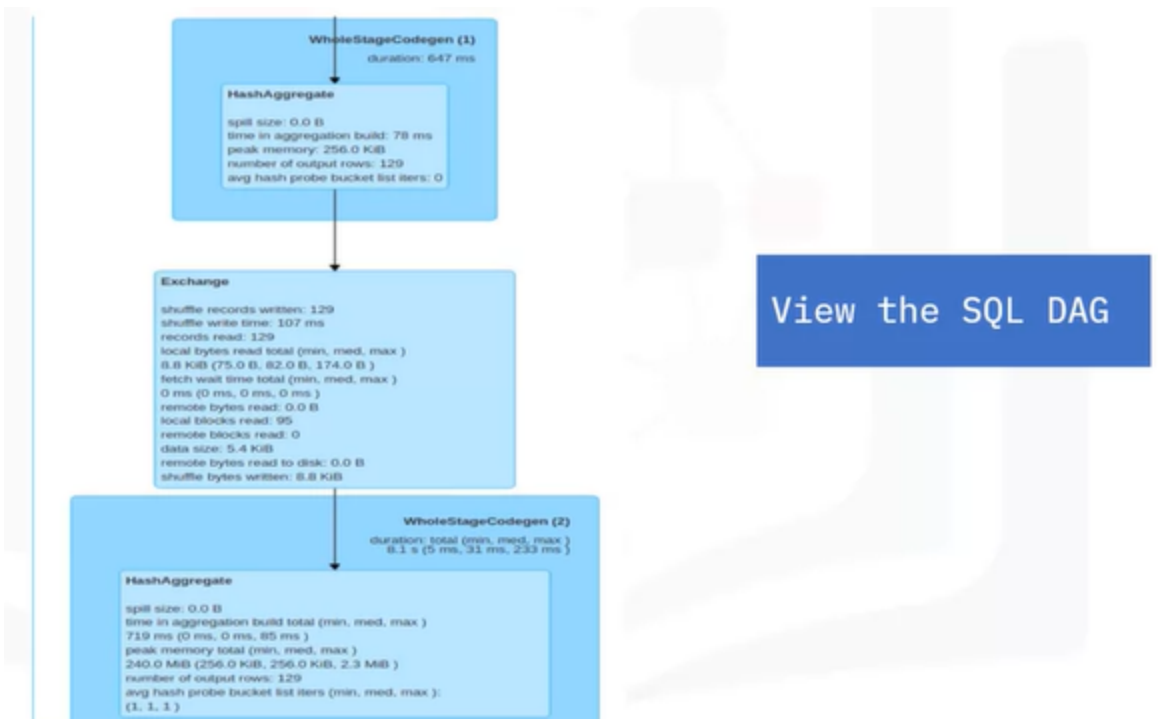
You can view the SQL query plan when you click the Details expander arrow.



View the SQL query plan

Query's DAG

You can also view the query's DAG including the logical physical plans as determined by the optimizer.



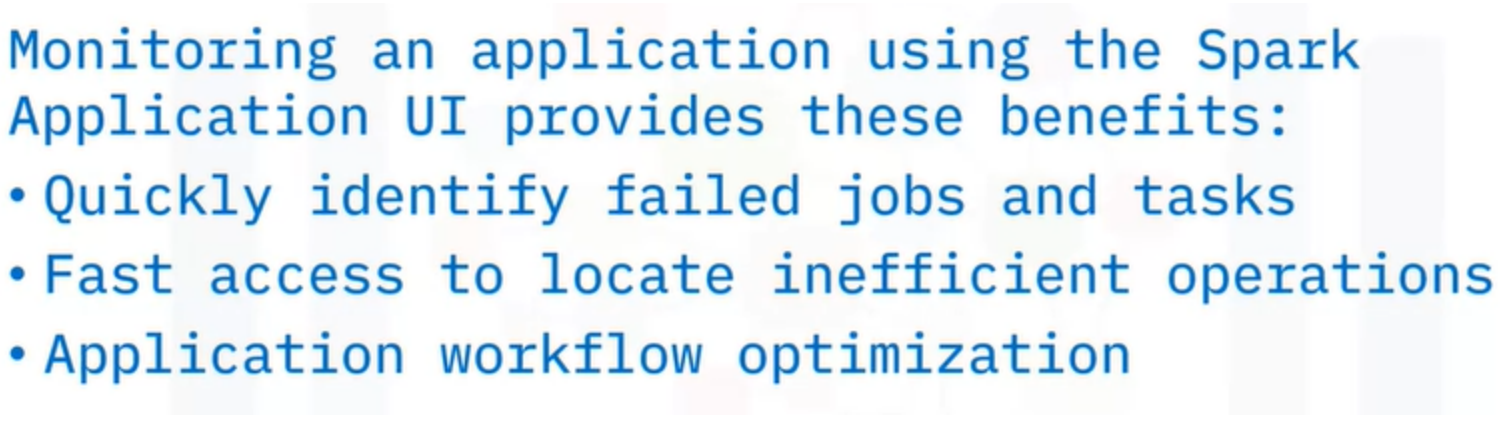
Conclusion

- The Jobs tab displays the application's jobs, including job status
- The Stages tab reports the state of tasks within a stage.
- The Storage tab shows the size of RDDs or DataFrames that persisted to memory or disk.
- The Environment tab information includes any environment variables and system properties for Spark or the JVM.
- The Executors tab displays a summary that shows memory and disk usage for any executors in use
- If the application runs SQL queries, select the SQL tab and the Description hyperlink to display the query's details.

Monitoring Application Progress

Why monitor an application

Running a Spark application can sometimes take a long time and have many possible points of failure.



When an issue arises, say a faulty worker node is causing the occasional task to fail, it is essential to find and address the problem quickly, so that cluster resources do not stay idle.

The Spark Application UI is a great way to monitor a running application.

- Centralizes critical information, including status information, and organizes information logically, resulting in convenient and fast access.
- Can quickly identify failures, then drill down to the lowest levels of the application to find out the root causes of failures.
- The UI can also help you quickly locate and analyze application processing bottlenecks.

How does the Application flow

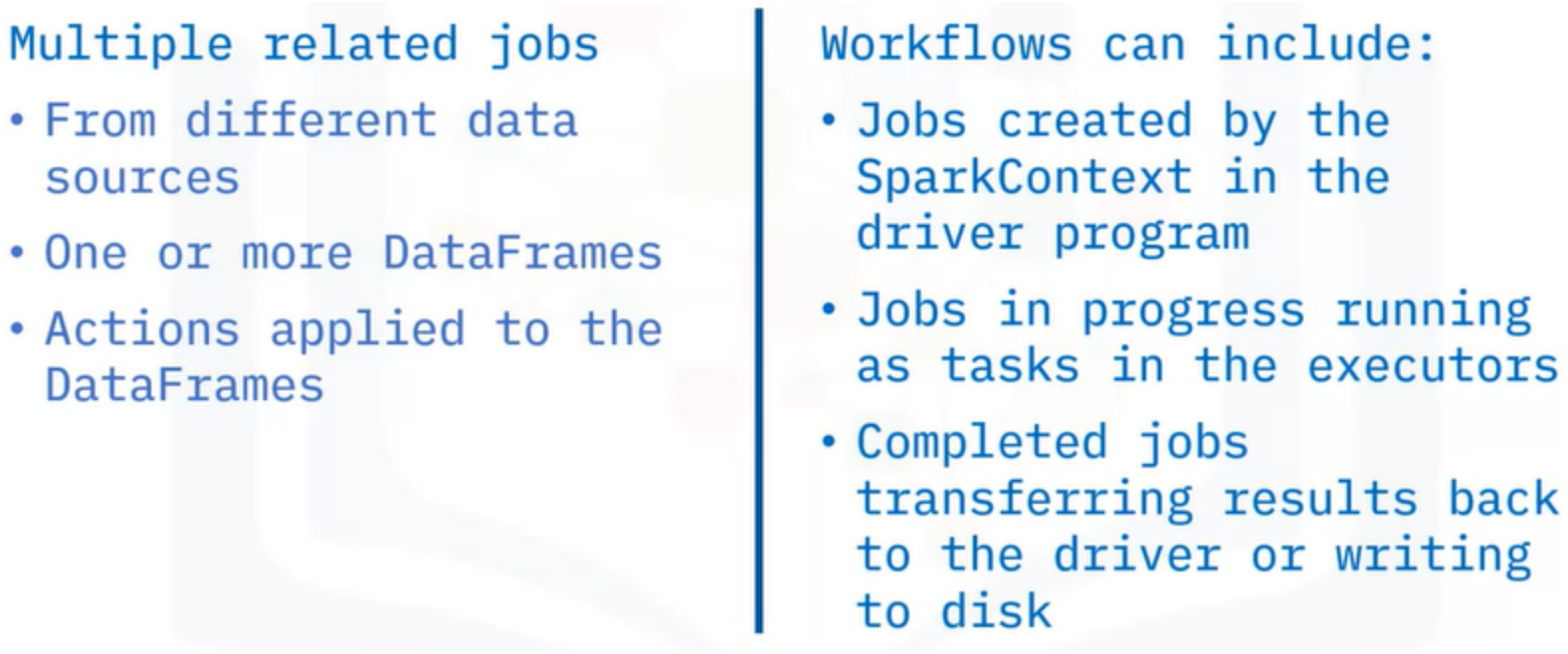
A Spark application can consist of many parallel, and often, related jobs, including multiple jobs resulting from:

- multiple data sources,

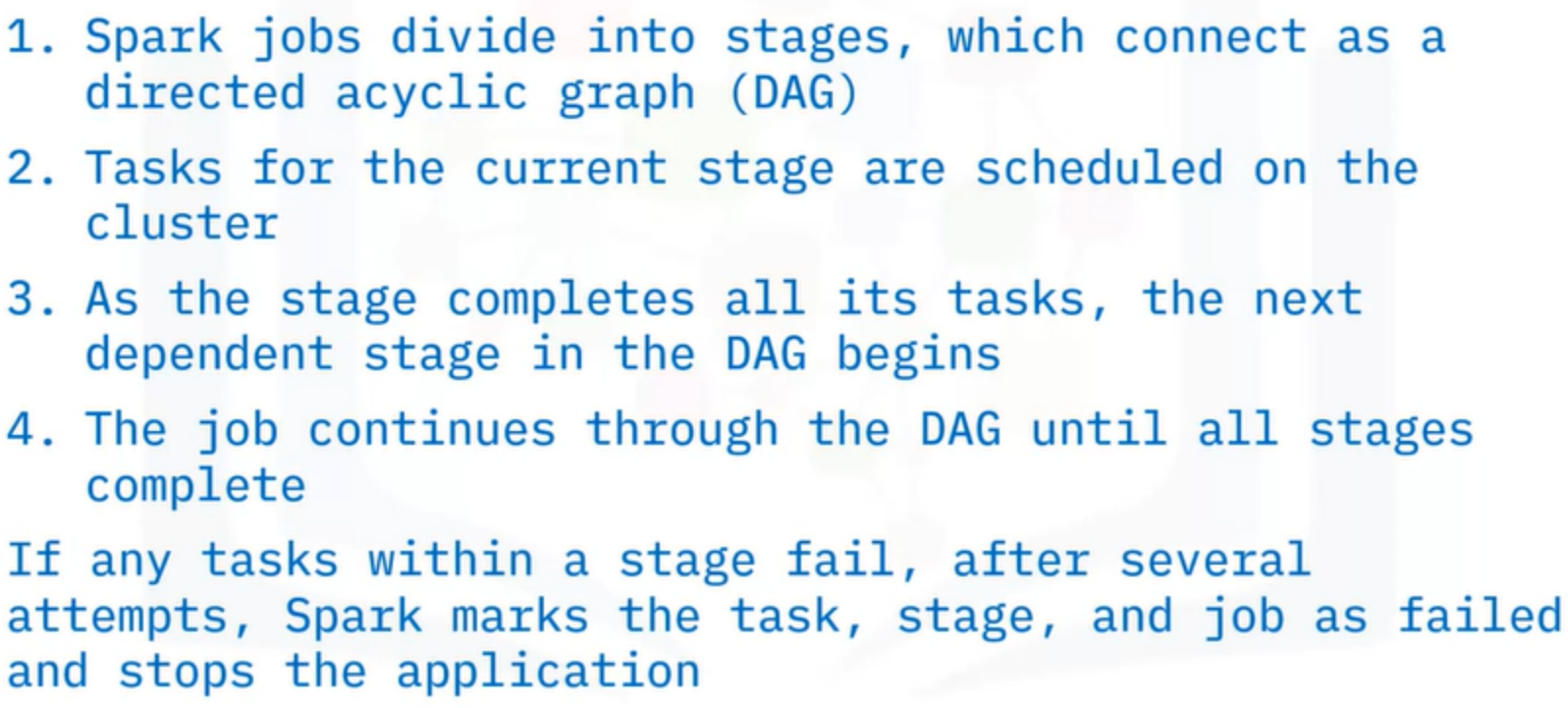
- multiple DataFrames,
- the actions applied to the DataFrames

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.



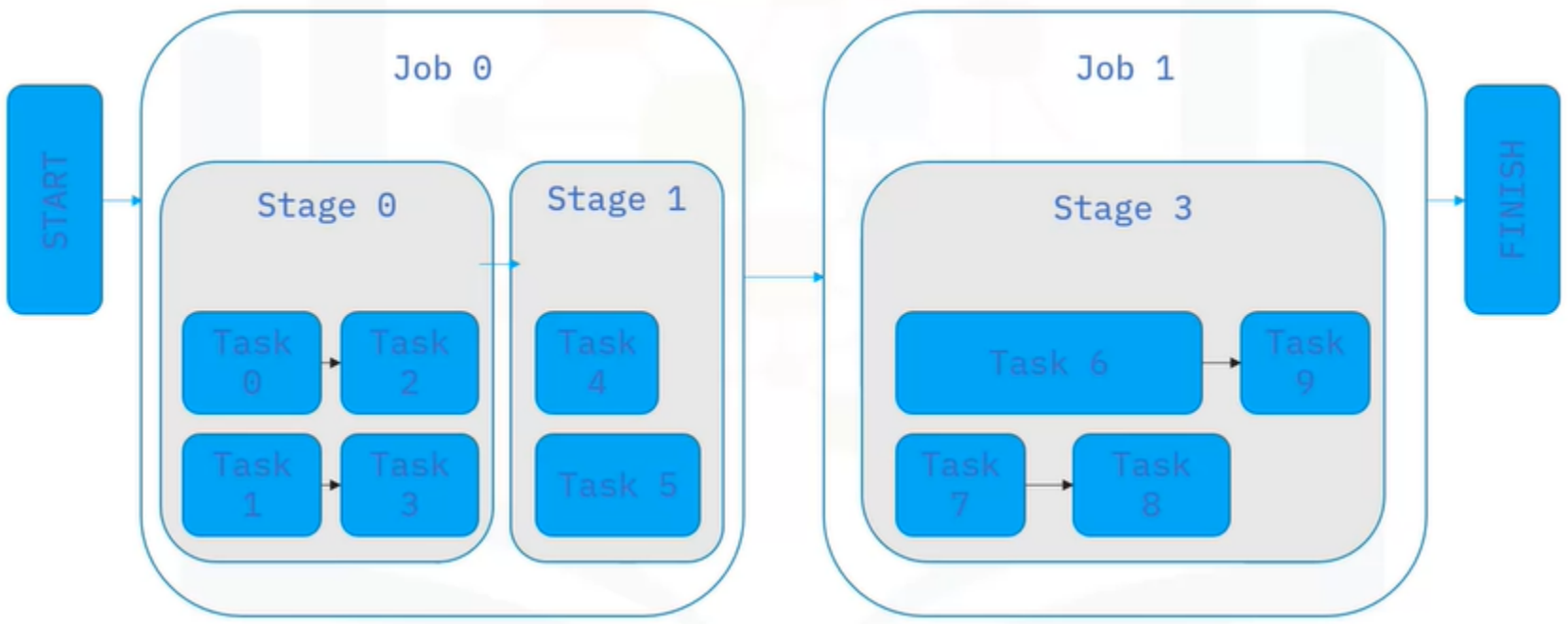
How does Jobs progesss?



1. Spark jobs divide into stages, which connect as a Directed Acyclic Graph, or DAG.
2. Tasks for the current stage are scheduled on the cluster.
3. When the stage completes all of its tasks, the next dependent stage in the DAG begins.
4. The job progresses through the DAG all stages are completed.

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

Workflow sequence



The application first creates a job.

Next, Spark divides the job into one or more stages.

The first stage, Stage "0," has no dependencies, so Spark sends its tasks to the executors.

Stage "0" now has two tasks started.

The width of the task indicates the elapsed run time.

Two more tasks that are dependent on “0” and “1” run, but do not require a shuffle, such as map operations, for example, so they remain part of Stage “0” and run independently.

The end of Stage "0" now marks the beginning of the next Stage "1."

This boundary exists because a shuffle was required, which means that all tasks for Stage “1” must wait for all tasks in Stage “0” to finish before starting.

Here you can see that tasks 4 and 5 have different run times. Their run times differ because each task is running independently and on different data partitions. However, the stage is not complete until all tasks have finished.

With this job completed, Spark can begin a new job.

In this example, Job 1 depends on the data from Job 0.

The next job consists of one stage and starts Tasks 6 and 7. Since tasks within a stage can run independently, when Task 7 completes, the executor running Task 7 can immediately start Task 8 while Task 6 continues to run.

When the application completes Tasks 8 and 9, the stage and job are complete, marking the end of this application workflow.

Application code example

Next, view this example application to see how the code translates to a workflow you can monitor using the UI.

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()
```

This application’s single data source is a Parquet file loaded from disk to produce a DataFrame.

Using that same DataFrame, two columns are selected.

The caching action is specific to this example. The application groups the data by the "country" column and then aggregates the data by calculating the mean of the "salary" column.

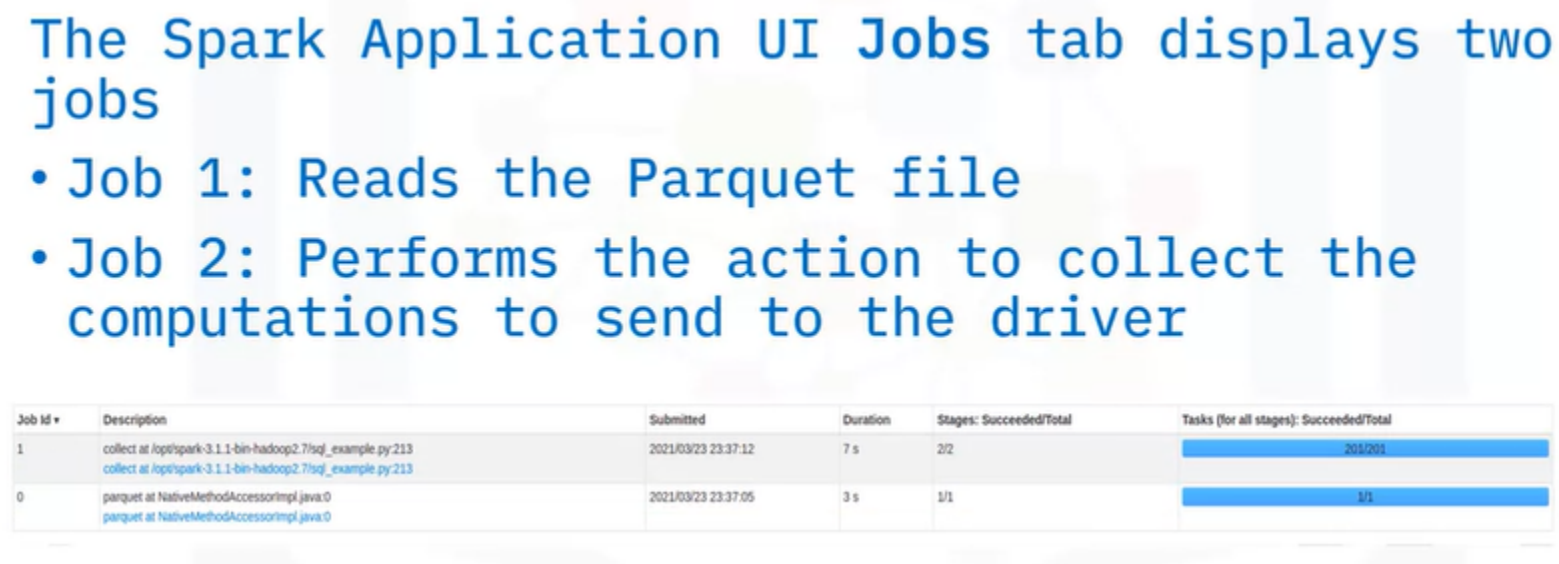
Next the “collect” action runs. This action triggers the job creation and schedules the tasks, as the previous operations are all lazily computed.

Application jobs example

After you submit the application, start the Spark Application UI and view the Jobs tab, which displays two jobs.

One job reads the Parquet file from disk.

The second job is the result of the action to collect the grouped aggregate computations to send to the driver.



Application Job details example

On the Jobs tab, click a specific job to display its Job Details page.

Here you can see the number of stages and the DAG that links the stages.

This example has two stages connected by a shuffle exchange, which is due to grouping the data by country in the application.

Application Stage details example

Select a stage to view the tasks.

The Stage Details timeline indicates each task's state using color coding.

View the timeline to see when each task was started and the task's duration.

Use this information to quickly locate failed tasks, see which tasks are taking a long time to run, and determine how well your application is parallelized.

Select a stage to view the tasks

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

Application task status example

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

Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
97	0	SUCCESS	PROCESS_LOCAL	0	192.168.122.182	stdout stderr	2021-03-23 23:46:06	10.0 ms			
2	0	SUCCESS	NODE_LOCAL	0	192.168.122.182	stdout stderr	2021-03-23 23:46:04	0.3 s		80 B / 1	
3	0	SUCCESS	NODE_LOCAL	0	192.168.122.182	stdout stderr	2021-03-23 23:46:04	0.3 s		110 B / 2	
98	0	SUCCESS	PROCESS_LOCAL	0	192.168.122.182	stdout stderr	2021-03-23 23:46:06	40.0 ms			
99	0	SUCCESS	PROCESS_LOCAL	0	192.168.122.182	stdout stderr	2021-03-23 23:46:06	46.0 ms			
100	0	SUCCESS	PROCESS_LOCAL	0	192.168.122.182	stdout stderr	2021-03-23 23:46:06	13.0 ms			

The task list provides even more metrics, including status, duration, and amount of data transferred as part of a shuffle.

Here, you see two tasks that read one and two records as part of a shuffle. You see these two tasks because, by default for a shuffle, Spark repartitions the data into a larger number of partitions.

You can also access a task's executor logs. The data used in this example is small. Therefore, many tasks only have a small number of records to process.

Application complete example

- The application jobs complete
- The SparkContext stops
- 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

When all application jobs are complete and the results are sent to the driver or written to disk:

- the SparkContext can be stopped either manually or automatically when the application exits.
- When the application UI server shuts down the UI is no longer available.
- To view the Application UI after the application stops, event logging must be enabled. This means that all events in the application workflow are logged to a file, and the UI can be viewed with the Spark History 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
```

To view the Application UI with the History Server, first verify that event logging is enabled.

Enter the event log path as seen using the properties displayed onscreen before submitting the application.

When the application completes, the Application UI populates the log files in the event log directory.

To start the history server, apply the command shown onscreen.

Once the history server is started, connect to the history server by typing the history server host URL followed by the default port number 18080.

You can see a list of completed applications and select one to view its application UI.

Conclusion

- The Spark application workflow includes:
  - 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.
- The Spark application UI centralizes critical information, including status information.
- You can quickly identify failures, then drill down to the lowest levels of the application to discover their root causes.

Debugging Apache Spark Application Issues

Common application issues

Common areas of Spark application issues

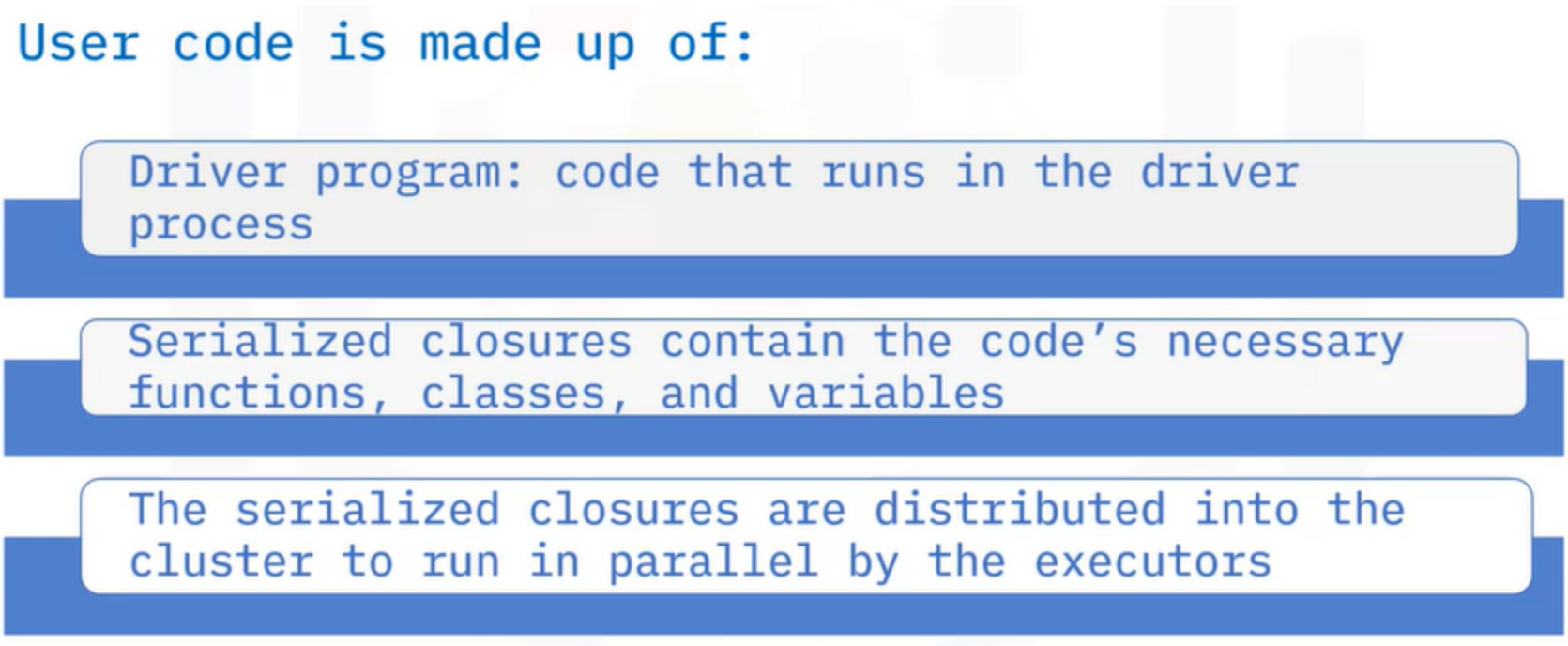
- User Code
- Configuration
- Application Dependencies
- Resource Allocation

Running a Spark application on a cluster is a complex process with many working parts and many ways in which an application can fail.

Common reasons for application failure on a cluster include:

- user code
- system and application configurations
- application dependencies that are missing or an incorrect version
- improper resource allocation, and network communication among cluster nodes.

What is user code?



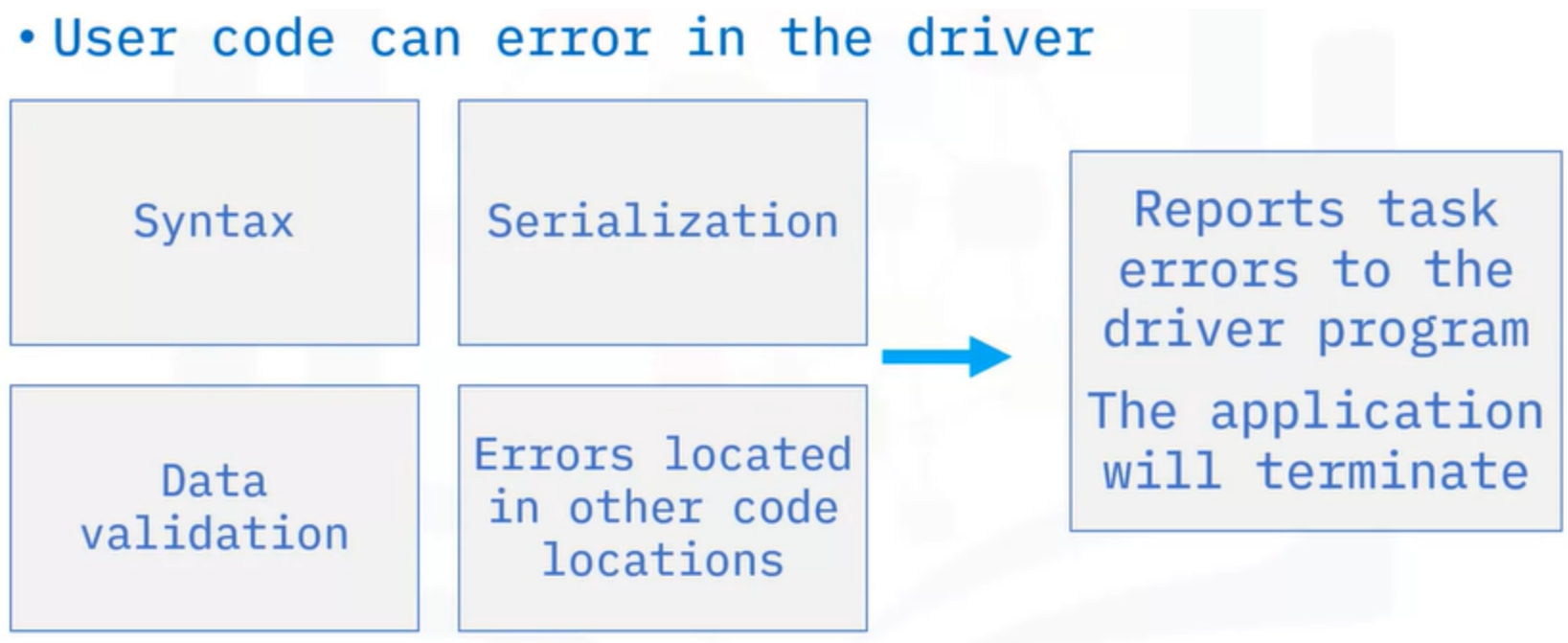
User code is made up of:

- The driver program, which runs in the driver process and the functions and variables serialized that the executor runs in parallel. Both the driver and executor processes run the application user code of an application passed to the spark-submit script. The user code in the driver creates the SparkContext and creates jobs based on operations to the DataFrames.
- These DataFrame operations become serialized closures sent throughout the cluster and run-on executor processes as tasks.
- The serialized closures contain the necessary functions, classes, and variables needed to run each task.

User code issues

Spark usually immediately terminates when syntax, serialization, data validation, and other user errors occur.

Spark reports task errors to the driver and immediately cancels the related executor tasks, terminating the application.



For example, closures are cleaned before being serialized, flushing out most issues right away and ensuring that closures can be serialized and executed remotely.

User code serialized as a closure might not error until after the user code runs on another executor process. These errors could be due to runtime calculations, network communication issues, or unexpected data issues.

If a task fails due to an error, Spark can attempt to rerun the task for a set number of retries.

If all attempts to run the task fail, Spark reports an error to the driver and the application is terminated.

The cause of an application failure can usually be found in the driver event log.

Application dependency issues

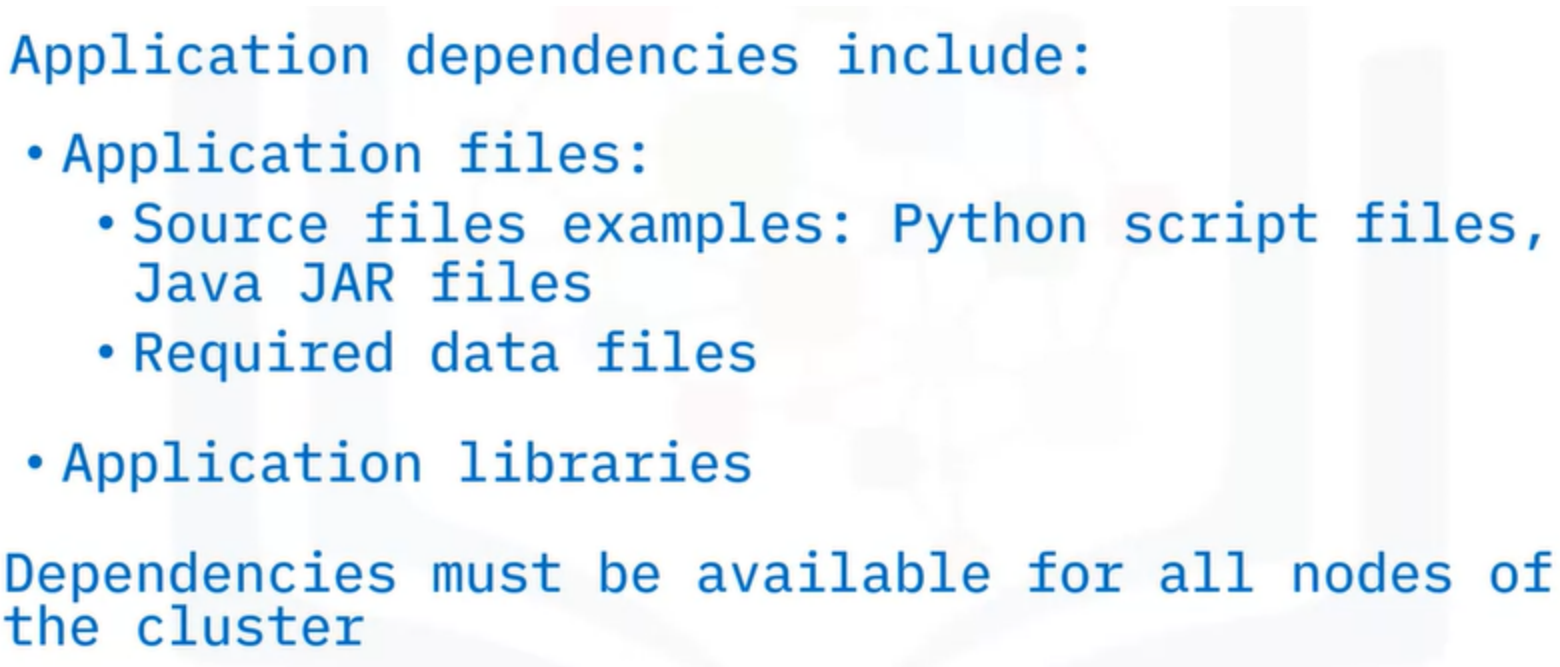
A Spark application can have many dependencies including:

- Application files such as:
  - Python script files, Java JAR files
  - Required data files.
- The libraries used and their dependencies.

Dependencies must be made available on all nodes of the cluster, either:

- by pre-installation
- by including the dependencies in the spark-submit script bundled with the application, or as additional arguments.

> 💡 *For example, a task will error if a Python library is not installed in the Python environment of the executor process.*
>
> *An even more subtle error can occur if a library is installed with different versions on executors that might have different APIs or produce unexpected results.*
>
> *The best way to identify this type of issue is by examining the event log for stack trace errors that identify which libraries the application loaded.*



Application resource issues

Application resources, such as CPU cores and memory, can become an issue if a task is in the scheduling queue and the available workers do not have enough available resources to run the tasks.

As a worker finishes a task, the CPU and memory in use are freed, allowing the scheduling of another task.

However, if the application asks for more resources that can ever become available, the tasks might never be run and eventually time out.

Similarly, suppose that the executors are running long tasks that never finish. In that case, their resources never become available, which also causes future tasks never to run, resulting in a timeout error.

You can readily access these errors when you view the UI or event logs.

- 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!

Examine the Log files

Viewing the log files provides details that give more insights into possible application failure causes.

Find the application log files in the Spark installation directory:

- under `work/<application-id>/`, where you will find:
  - one log file for `stdout`
  - one log file for `stderr` output.

The `application-id` is a unique ID that Spark assigns to each application.

These log files appear for each executor and driver process that the application runs.

Additionally, if you are running a Spark standalone cluster, the master and workers both output log files to the `log/` directory under the Spark installation directory from where they run.

- Application logs are found in `work/` directory named as work/<application-id>/<stdout|error>
- Spark standalone writes master/worker logs to the `log/` directory

Conclusion

- Common reasons for application failure on a cluster include user code, system and application configurations, missing dependencies, improper resource allocation, and network communications.
- Application log files will often show the complete details of a failure, which are located in the Spark installation directory.

Understanding Memory Resources

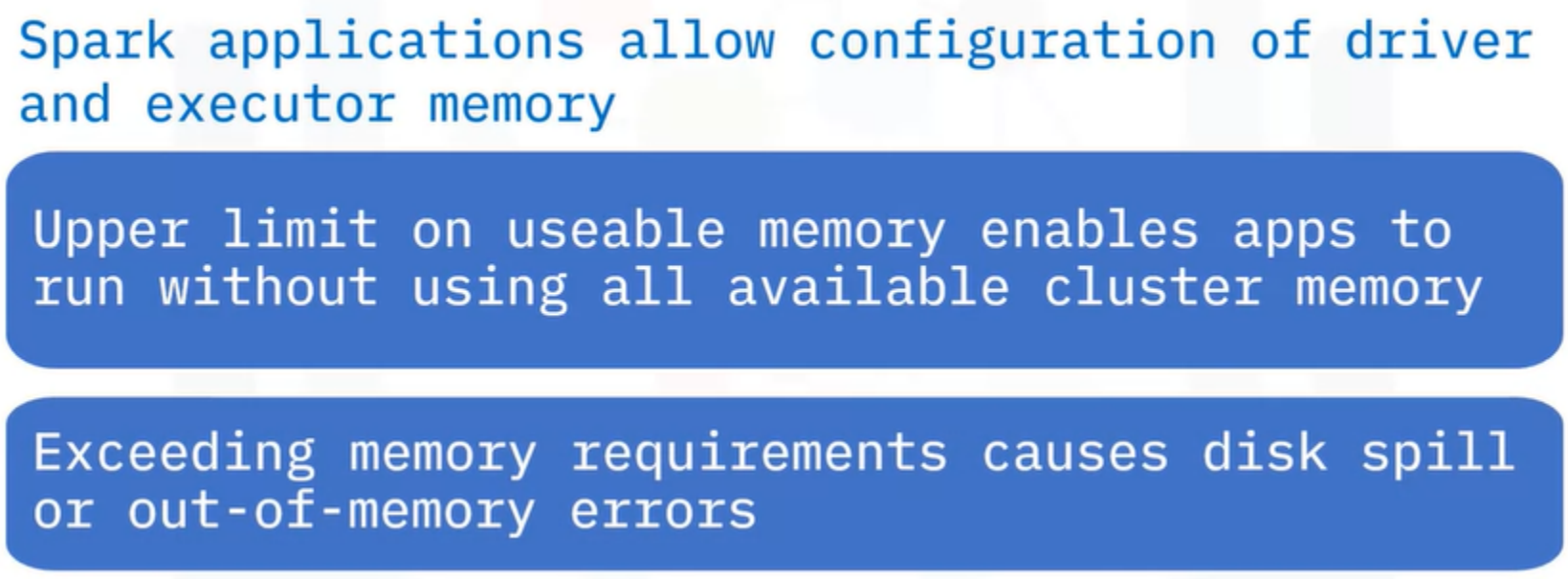
Configuring Spark process memory

When running a Spark application, the driver and executor processes launch with an upper memory limit.

The upper memory limit enables a Spark application to run without using all the available cluster memory, but this limit also requires that the memory limits are set high enough to perform necessary tasks.

An application runs best when processes complete within the requested memory.

If the driver and executor processes exceed the memory requirements, this situation can result in poor performance with data spilling to disk or even out-of-memory errors.



Memory setting considerations

Executors Memory

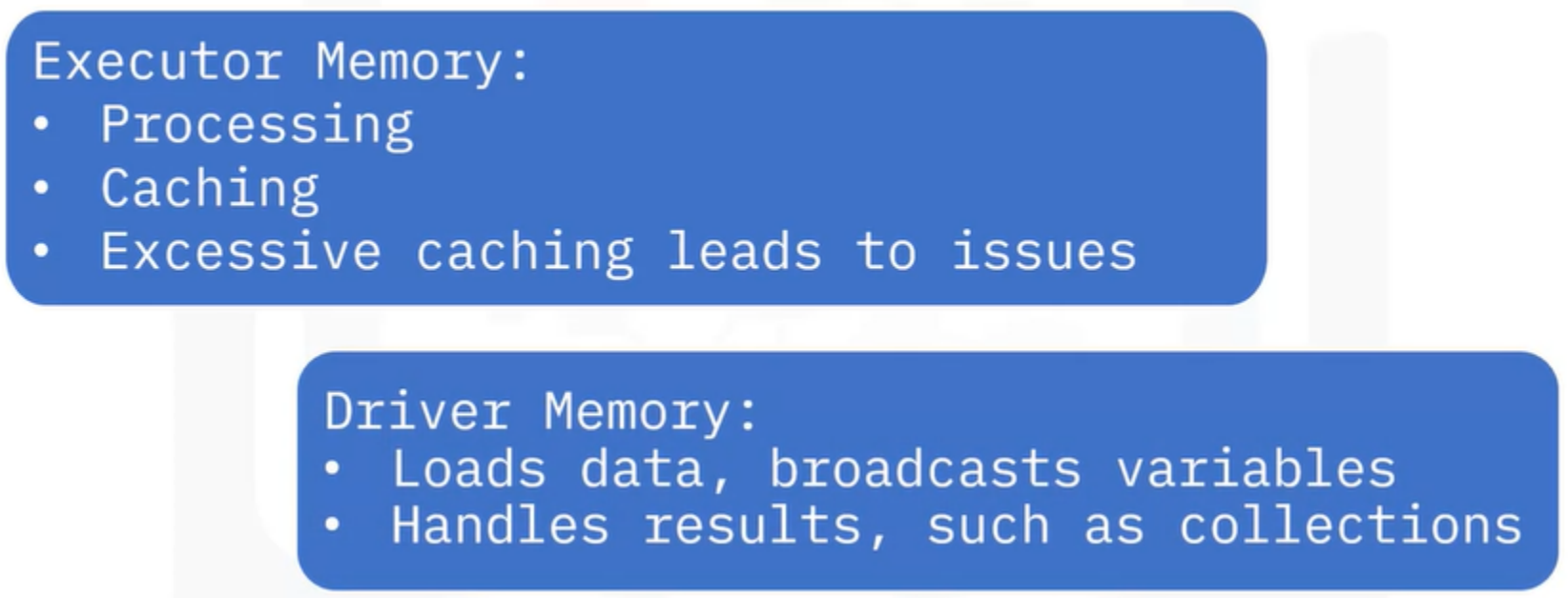
- Use memory for processing and additional memory if caching is enabled.
- However, excessive caching can lead to out-of-memory errors.

Collecting data as a result of operations will be done in the driver.

Driver memory

- Loads the data, broadcasts variables,
- Handles results, such as collections.

Because large data sets can easily exceed the driver's memory capacity, if collecting to the driver, filter the data and use a subset of the data.



Spark unified memory

In Spark, executor memory and storage memory share a unified regions shown in this Java Heap Space in the space labeled M.

When no executor memory is used, storage can acquire all the available memory and vice versa.

Executor memory can evict storage memory if necessary, but only until total storage memory usage falls under a certain threshold.

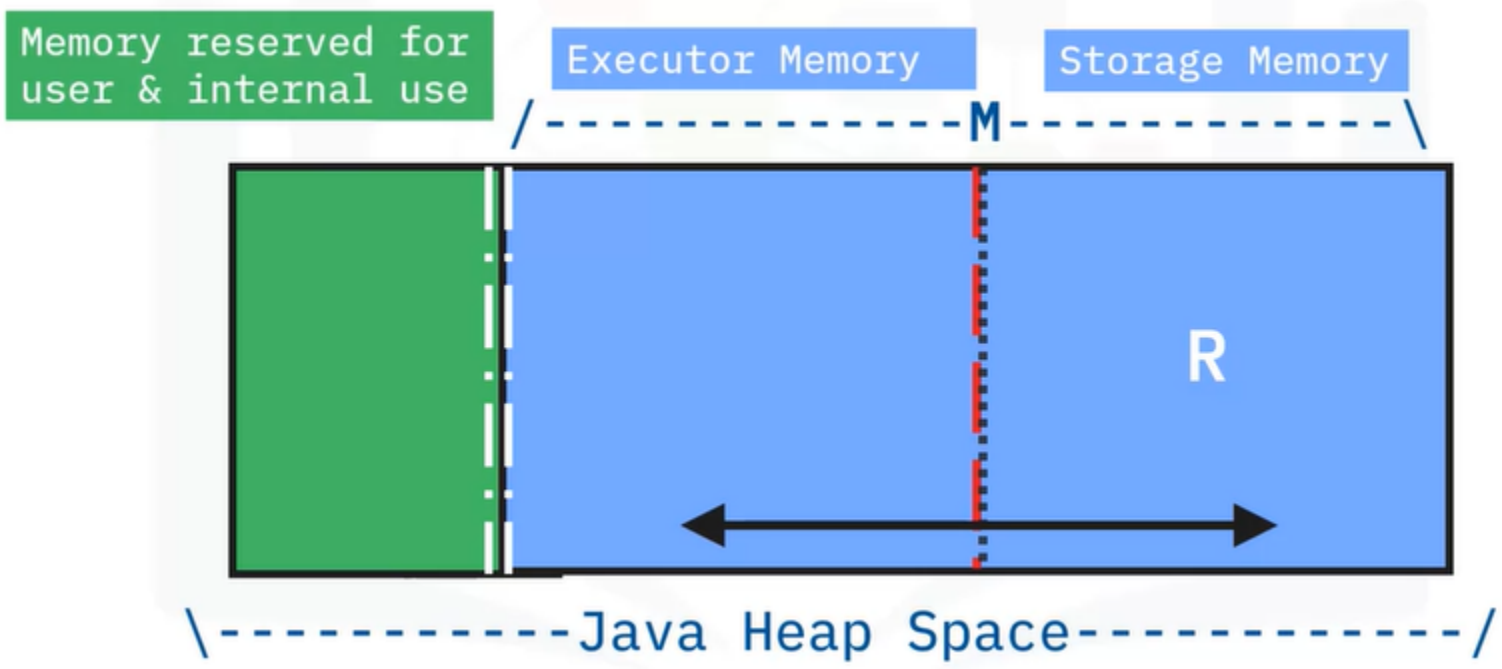
In other words, R describes a subregion within M where cached blocks are never evicted.

Storage is not allowed to evict executor memory due to complexities in implementation.

This design ensures several preferable properties.

- First, applications that do not use caching can use the entire space for executor memory, obviating unnecessary disk spills.
- Second, applications that do use caching can reserve a minimum storage space, the area labeled R, where their data blocks are immune to being evicted.

- Lastly, this approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise of how memory is divided internally.



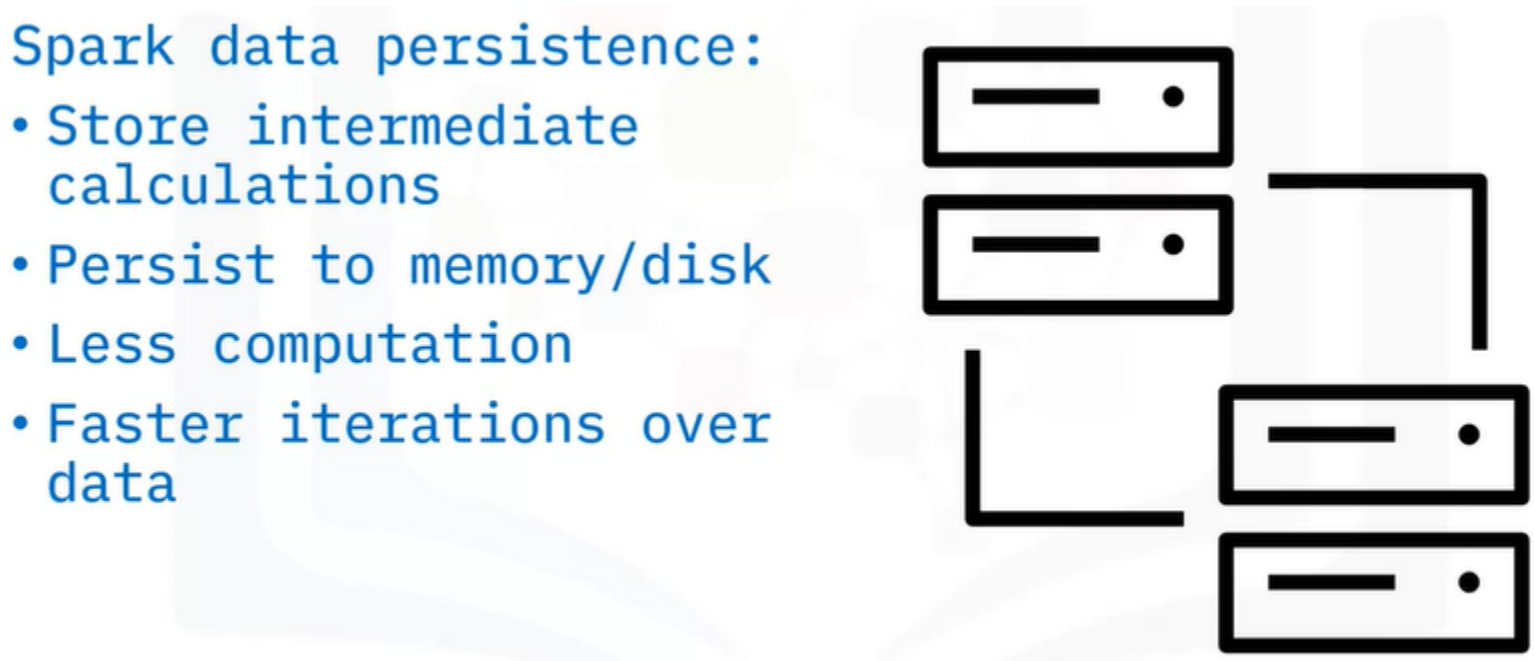
Spark data persistence

`Data persistence`, or caching data, in Spark means being able to **store intermediate calculations** for reuse.

Setting persistence in either or both memory and disk.

After the intermediate data is calculated to produce a new DataFrame, and if memory is cached, then any other operations on that DataFrame can reuse the same data, rather than re-loading data from the source and re-calculating all prior operations.

This capability is essential to speed machine learning workloads that often require many iterations over the same data set when training a model.



Data persistence example

This sample PySpark code creates a DataFrame with column features consisting of random values.

After creating the DataFrame, the `cache()` method is called to mark the random values to cache in memory.

Caching the DataFrame here means that the application only needs to generate random features once. At this point, the random values are not yet generated; caching is done lazily after computing the values.

In this example, the `count()` action is invoked on the DataFrame, which generates the values that are then stored in memory.

Subsequent DataFrame calls can use the cached values, which can save a great deal of computational time if the cost to recreate a DataFrame is high.

If this DataFrame is not cached, then different random features would be generated with each action on the DataFrame, because the function `rand()` is called each time.

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

Ways to set the memory for executors in a cluster:

- Setting a value in the properties file.
- Specify a memory configuration when submitting the application to the cluster.
- Tailor memory so that each application has enough memory to run effectively but does not use all available memory in the executor.

This example configures the application to run on a Spark standalone cluster and reserves ten gibibytes per executor when running tasks.

Set the executor memory for a Spark standalone cluster:

```
$ ./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

If using the Spark Standalone cluster to manage and start a worker manually, you can specify the total memory and CPU cores that the application can use.

These specifications determine the resources available when workers start the executors.

Avoid assigning more resources than are available on the physical machine.

> 💡 *For instance, if the machine has a CPU with 8 cores and the worker starts with 16 cores, too many threads might run simultaneously and cause performance degradation.*
>
> *You can tailor memory so that each application has enough memory to run effectively but does not use all available memory in the executor.*

The default configuration:

- Use all available memory minus 1 gigabyte
- All available cores.

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

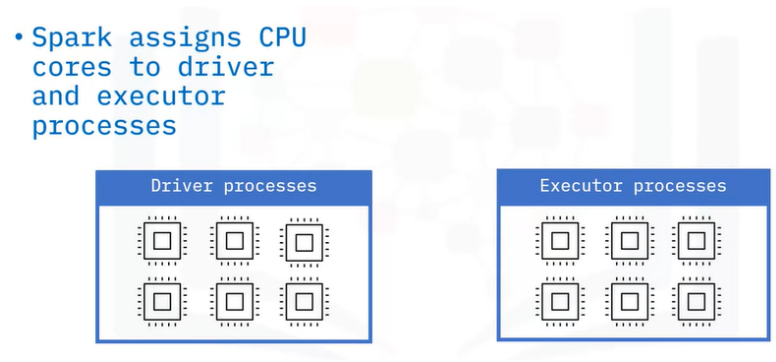
Conclusion

- Spark has configurable memory for executor and driver processes.
- Executor memory and Storage memory share a region that can be tuned as needed.
- Caching data can help improve application performance.

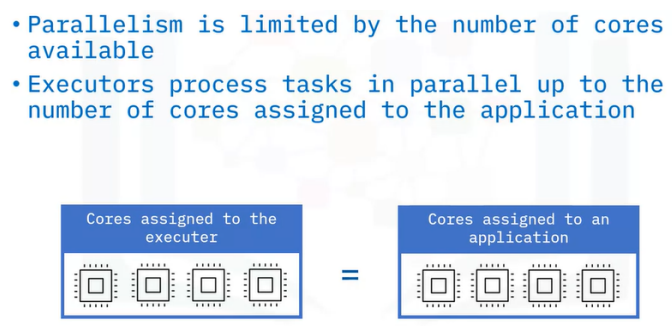
Understanding Processor Resources

CPU cores as Spark resources

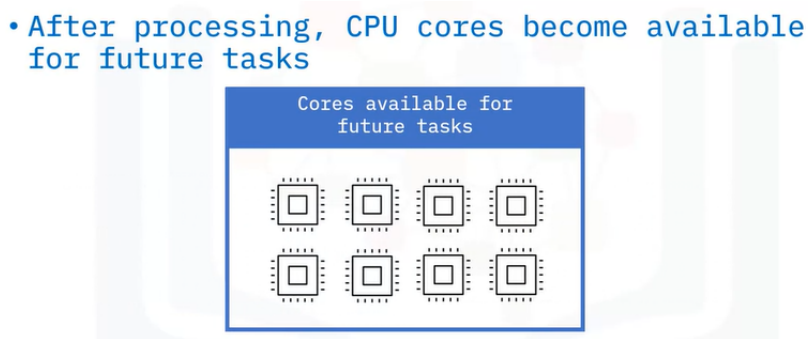
Just as with memory, CPU cores are a resource assigned to both the driver and executor processes.



An executor can process tasks in parallel only up to the number of cores assigned to the application.



Once the application tasks finish processing, the cores no longer in use return to the available pool.



The 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

When Spark launches an application and creates tasks, Spark places those tasks in a scheduling queue.

Spark assigns tasks to any available executor.

The default behavior is for each task that is ready, to be scheduled to an executor with available cores to maximize parallelism.

Any remaining tasks will have to wait in the queue until more cores become available.

If you provided specific settings for the number of cores, such as configuring how many cores the executor uses, the application overrides the default behavior.

Setting Executor Cores on submit

Per executor process

Configure executor cores on application submission:

- use the argument `- executor-cores`
- followed by the specific number of cores required for the application tasks to run on an executor.

> 💡 As always, if the executor does not have the required number of cores available, tasks do not start until the specified number of cores are free or another executor becomes available.

- 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
```

For the application

Alternatively, the argument `-total-executor-cores` followed by a number is the total amount of cores throughout the cluster to use for the application, not per executor process.

Tasks will be scheduled to executors whenever they have a free core, until the total number of cores reaches the specified value.

- 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

When starting a worker manually in a Spark standalone cluster, specify the number of cores the application uses by using the argument `- cores` followed by the number of cores.

Spark's default behavior is to use *all available cores.*

A common practice is to start one worker per node, which under default behavior, allows the worker to create one executor process with exactly the same number of threads as cores available to run tasks.

If another process is running on the same machine as the worker, such as the master, then a good practice is to apply these settings to reduce the number of cores available to the worker.

Specifying the total number of CPU cores can reserve processing time available for the master process so essential operations are not blocked and time out.

```
# 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

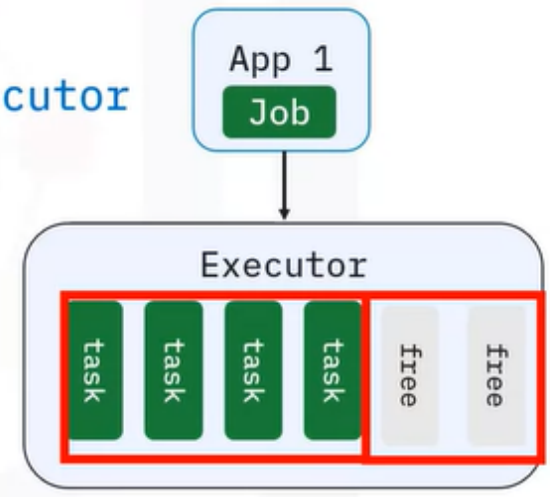
Core utilization example

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
- Two cores remain free



Let’s look at an example that uses a small standalone cluster with 1 worker node and 6 cores.

This example uses two identical applications submitted to the same cluster in overlapping time, one right after the other.

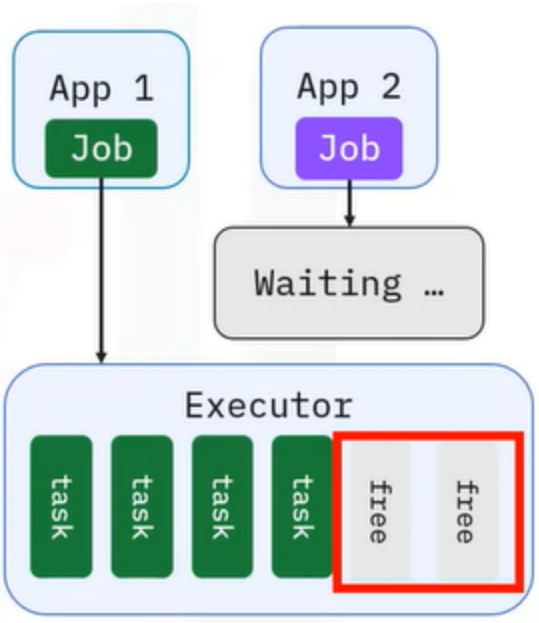
The first application is submitted to the cluster and requests 4 cores per executor.

The executor is currently idle with 6 cores available.
The first 4 tasks of the application are then scheduled to the executor and begin to run.
2 cores of the executor remain available.

- 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



While the first application is running, the second application is submitted.

The second application also requests 4 cores from the executor.

Because tasks that belong to the first application are still running and are using 4 cores, only 2 cores are available.

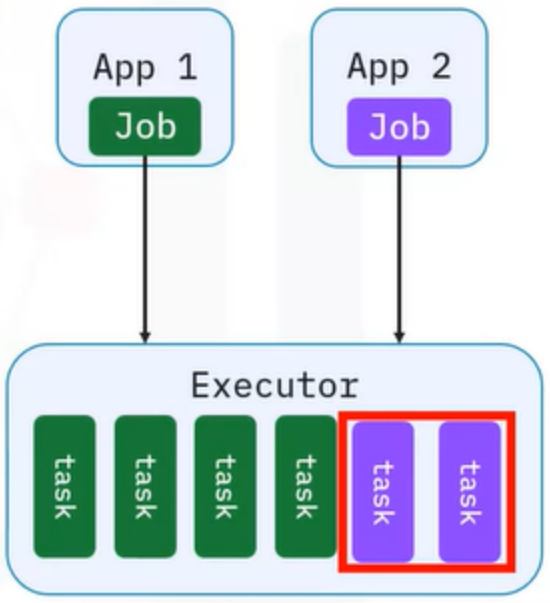
Therefore, the second application must wait until the executor has at least four cores available.

As soon as any two tasks from the first application complete and four cores become available, the second application can begin.

- 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



What if you start the second application and request only two cores to run two tasks?

Using this scenario, you can submit a second application, and two tasks immediately start.

However, the second application might take longer to finish because of running only two tasks at a time.

💡 Maximizing Spark application performance can be a tricky balance between configuration of application workloads and cluster resources and requires careful tuning of both the application and the clusters.

Conclusion

- Spark assigns CPU cores to driver and executor processes during application processing,
- Executors process tasks in parallel according to the number of cores available or assigned by the application
- If using the Spark Standalone cluster manager, you can specify the total memory and CPU cores workers can use.

Labs

Hands-on Lab: Monitoring and Performance Tuning: https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-BD0225EN-SkillsNetwork/labs/SparkMonitoringAndDebugging.md.html

---

Summary & Highlights

- To connect to the Apache Spark user interface web server, start your application and connect to the application UI using the following URL: **http://<driver-node>:4040**
- The Spark application UI centralizes critical information, including status information into the **Jobs**, **Stages**, **Storage**, **Environment** and **Executors** tabbed regions. You can quickly identify failures, then drill down to the lowest levels of the application to discover their root causes. If the application runs SQL queries, select the **SQL** tab and the **Description** hyperlink to display the query's details.
- The Spark application workflow includes jobs created by the Spark Context in the driver program, jobs in progress running as tasks in the executors, and completed jobs transferring results back to the driver or writing to disk.
- Common reasons for application failure on a cluster include user code, system and application configurations, missing dependencies, improper resource allocation, and network communications. Application log files, located in the Spark installation directory, will often show the complete details of a failure.
- User code specific errors include syntax, serialization, data validation. Related errors can happen outside the code If a task fails due to an error, Spark can attempt to rerun tasks for a set number of retries. If all attempts to run a task fail, Spark reports an error to the driver and terminates the application. The cause of an application failure can usually be found in the driver event log.
- Spark enables configurable memory for executor and driver processes. Executor memory and Storage memory share a region that can be tuned.

- Setting data persistence by caching data is one technique used to improve application performance.
- The following code example illustrates configuration of executor memory on submit for a Spark Standalone cluster:

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

- The following code example illustrates setting Spark Standalone worker memory and core parameters:

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

- Spark assigns processor cores to driver and executor processes during application processing. Executors process tasks in parallel according to the number of cores available or as assigned by the application.
- You can apply the argument **‘--executor-cores 8 \’** to set executor cores on submit *per executor*. This example specifies eight cores.
- You can specify the executor cores for a Spark standalone cluster *for the application* using the argument **“--total-executor-cores 50’** followed by the number of cores for the application. This example specifies 50 cores.
- When starting a worker manually in a Spark standalone cluster, you can specify the number of cores the application uses by using the argument **‘--cores‘** followed by the number of cores. Spark’s default behavior is to use all available cores.

Quiz

Practice Quiz: Introduction to Monitoring & Tuning

1.Question 1

Select the option that includes the available tabs within the Apache Spark User Interface.

- Jobs, Stages, Storage, Environment, and SQL
- Jobs, Stages, Storage, Executor, and SQL
- Jobs, Storage, Environment, Executor, and SQL
- **Jobs, Stages, Storage, Environment, Executor, and SQL**

2.Question 2

Which action triggers job creation and schedules the tasks?

- The jobs() action
- **The collect() action**
- The schedule() action
- The create() action

Yes! This answer is correct! This action triggers job creation and schedules the tasks, as the previous operations are all lazily computed.

3.Question 3

Select the Workflow options you can monitor using the Spark Application UI

1 / 1 point

- [ ] Jobs assigned to other applications.
- [x] ~~**Jobs created by the SparkContext in the driver program**~~

- ☑ ~~Completed jobs transferring results back to the driver or writing to disk.~~
- ☑ ~~Jobs in progress running as tasks in the executors~~

4.Question 4

Identify common areas where Spark application issues can happen
- User Code
- Configuration
- Application Dependencies
- Resource Allocation
- Network Communication

- User code, Configuration, Application Dependencies, Resource allocation, External logins
- User code, Configuration, Application Dependencies, Resource allocation, Network security measures
- User code, Configuration, Application Dependencies, and Cloud providers
- **User code, Configuration, Application Dependencies, Resource allocation, Network Communication**

Yes! User code, Configuration, Application Dependencies, Resource allocation, and Network Communication are common areas where Spark application issues can happen.

5.Question 5

Select an option to fill-in-the-blank. If a DataFrame is not cached, then different random features would be generated with each action on the DataFrame, because the function _____ is called each time.

- `regernate()`
- **‘rand()’**
- `cache()`
- `random()`

Yes! If a DataFrame is not cached, then different random features would be generated with each action on the DataFrame, because the function ‘rand()’is called each time.

Graded Quiz: Introduction to Monitoring & Tuning

1.Question 1

Select the option that includes the available tabs within the Apache Spark User Interface.

- Jobs, Stages, Storage, Executor, and SQL
- Jobs, Storage, Environment, Executor, and SQL
- **Jobs, Stages, Storage, Environment, Executor, and SQL**
- Jobs, Stages, Storage, Environment, and SQL

Yes! This answer is correct! All these tabbed options are available within the Spark Application UI. However, the SQL tab is optional, and is displayed based on the application.

2.Question 2

Which action triggers job creation and schedules the tasks?

- The create() action
- The schedule() action
- **The collect() action**
- The jobs() action

Feedback: Yes! This answer is correct! This action triggers the job creation and schedules the tasks, as the previous operations are all lazily computed.

3.Question 3

Syntax, serialization, data validation, and other user errors can occur when running Spark applications. View the numbered list and select the option that places this numbered list in the order of how Spark handles application errors.

1. View the driver event log to locate the cause of an application failure.
2. If all attempts to run the task fail, Spark reports an error to the driver and the application is terminated.
3. If a task fails due to an error, Spark can attempt to rerun the task for a set number of retries.

- 3,1,2
- **3,2,1**
- 1,2,3
- 2,1,3

Yes! This answer is correct. Spark can rerun tasks. If all attempts to run the task fail, Spark reports an error to the driver and the application is terminated. Then, view the driver event log to locate the cause of an application failure.

4.Question 4

Select an option to fill in the blank. If a DataFrame is not cached, then different random features would be generated with each action on the DataFrame, because the function _____ is called each time.

- `cache()`
- `regernate()`
- `random()`
- **‘rand()’**

Yes! If a DataFrame is not cached, then different random features would be generated with each action on the DataFrame, because the function ‘rand()’is called each time.

5.Question 5

Which command specifies the number of executor cores for a Spark standalone cluster *per executor process*?

- Use the command ‘--executor-process-cores’ followed by the number of cores
- Use the command ‘--process--executor--cores’ followed by the number of cores
- Use the command ‘--per--executor—cores’ followed by the number of cores.
- **Use the command ‘--executor-cores’ followed by the number of cores.**