# Tool: Spark UI Simulator - Exploring the Spark UI

#### **Notes**: The scenario [Exploring the Spark UI](https://www.databricks.training/spark-ui-simulator/exploring-the-spark-ui/v002) includes the following features to support demoing the Spark UI and Cluster UI while providing consistency in the delivery of this lesson.

#### **Notebook**: a pre-ran version of a notebook with semi-arbitrary code blocks

#### **Notebook Source**: a copy of the notebook the instructor and/or students can import into Databricks for a slightly different classroom experience

#### **Spark UI**: recreates a significant amount of the Spark UI

#### **Cluster**: recreates a significant amount of the Cluster’s configuration

#### **Lab**: Includes 80+ questions, broken down into 9 different categories

# 

# Lesson #1.1: Spark UI Demo

#### **Duration:** 10 minutes

#### **Objective:** Identify key pieces of information related to a spark application such as cluster configuration and metrics for jobs, stages, executors, caches, and SQL queries.

#### **Notes**: The first exercise of this lesson is a tour of the Databricks environment with a specific emphasis on the Spark UI. Resources have been provided to do the demo with the [Spark UI Simulator / Exploring the Spark UI](https://www.databricks.training/spark-ui-simulator/exploring-the-spark-ui/v002) or live with the source notebook bundled with the simulator. **WARNING**: There is a direct correlation between the amount of time spent on this activity and the amount of time required to complete the following lab.

## 

## Step-By-Step Instructions

### Introduce the Spark UI Simulator

1. Open the [Spark UI Simulator / Exploring the Spark UI](https://www.databricks.training/spark-ui-simulator/exploring-the-spark-ui/v002)  
   <https://www.databricks.training/spark-ui-simulator/exploring-the-spark-ui/v002>
2. Introduce it as an alternative to the live notebook which can provide indeterminate results in the following lab and having to execute long-running code.

### 

### Notebook - Review the notebook that this simulation is based on

1. **Cmd 1** declares local variables and displays files in our source directory
2. **Cmd 2** simply reads in the data, caches it, and then materializes the cache.  
   The use of the **foreach** transformation has no real meaning other than to do nothing to each and every record.
3. **Cmd 3** has no real meaning to its steps other than to execute a series of wide and narrow transformations.
4. **Cmd 4** returns to the driver every single record. Logically the same as collect but the Catalyst optimizer will result in a significantly different physical plan because it is told how many records to return.
5. **Cmd 5** uses the **initialDF** to build a **DataFrame** that is large enough to exhaust the resources of this cluster.  
   The real goal is to induce a spill before one of the **SortMergeJoin**s.
6. Call out the **Jobs**, **Stages**, and **Tasks** associated with each command.
7. Note that both the **Cluster UI** and **Spark UI** can be accessed by clicking on the cluster’s name but in the simulator, they are accessible from the navigation bar.

### 

### Cluster UI / Configuration

1. Open the Cluster UI by clicking on the Cluster button in the navigation bar
2. Call out the specific configuration of this cluster:
   1. **Cluster Mode**
   2. **Pool** - Explain what a pool is
   3. **Runtime** - Spark & Scala version, Explain DBR
   4. **Autopilot Options**
      * Enable autoscaling
      * Enable autoscaling local storage
      * Terminate after N minutes of inactivity
   5. Worker Types
      * VM type
      * RAM
      * Cores
      * DBUs
   6. Driver Type
   7. JSON configuration
   8. Advanced Options are not enabled - using defaults
      * Instances (AWS Only)
        1. Availability zone
        2. Max Spot Price
        3. Instance Profile
      * Spark (AWS & MSA)
        1. Spark Config
        2. Environment Variables
      * Tags - (AWS & MSA)
      * SSH configuration (AWS Only)
        1. SSH Public Key
      * Logging - (AWS & MSA)
        1. Cluster Log Path
      * Init Scripts - (AWS & MSA)
      * Permissions (AWS Only)

### 

### Cluster UI / Notebooks

1. Which notebook is attached
2. The status of that notebook (e.g. is it OK to terminate this cluster)
3. When the last command was running - identifies a stale cluster, still in use, by whom?

### 

### Cluster UI / Libraries

1. Install and remove notebooks
2. Identify which libraries are currently attached.  
   Best Practice: Always use a specific library version to avoid future library conflicts

### 

### Cluster UI / Event Log

1. Answers questions like why my cluster was started (error, restarted by a user, inactivity)

### 

### Cluster UI / Driver Logs

1. Standard output
2. Standard error
3. Log4j

### 

### Cluster UI / Metrics

1. This is where we find the Ganglia UI
2. Includes snapshots of Ganglia over time

### 

### Cluster UI / Spark Cluster UI - Master and Workers

1. A high-level overview of information readily available in the Spark UI
2. Quick access to the executor’s logs (see Worker 0 and Worker 1) for **stdout** and **stderr**
3. Note that Databricks always runs one-to-one between worker and executor

### 

### Spark UI / Jobs

1. **Event Timeline**
   1. Gives an overview of application behavior
   2. When the driver starts and stops
   3. When each executor starts and stops - helpful for understanding behaviors on an auto-scaling cluster
   4. When a job starts and stops
      * Gaps between jobs can give a visual clue to driver-side performance problems such as a long file scan.
      * In Databricks, the text of the cell is the label in the UI
      * HINT: Start each cell with a descriptive comment for better context here
2. **Completed Jobs**
   1. Jobs start with 0
   2. Jobs listed with the most recent on top
   3. Columns
      * **Job ID (Job Group)**
      * **Description**
        1. Navigate to the job-specific details via the link in the description
        2. Call out that the description provides information as to what operation (usually an action) was executed
        3. Call out that it includes the first line (comment) of the corresponding cell
      * **Submitted** - when the job was submitted to the cluster
      * **Duration** - the duration of the job
      * **Stages: Succeeded/Total**
      * **Tasks (for all stages): Succeeded/Total**

### Spark UI / Jobs / Job

1. Select **Job #4**
2. **Event Timeline**
   1. Gives an overview of this one job’s behavior as opposed to the application.
   2. Visualizing stages in time help to understand how Spark 3 is processing your query
   3. Stages can now be run in parallel which can be seen here! (not demoed here)
3. **DAG Visualization**:
   1. Call out the green dot as the caching event
   2. Generally not deemed to be too helpful, see the SQL diagrams instead - but does help to visualize the various stages/shuffles
   3. Complex DAGs are your first clue of a potential performance problem
   4. More specific information is provided in the Stage Details.
4. **Completed Stages:**
   1. Lists each stage that this job is composed of.
   2. Columns - what is shown depends on the execution (e.g. if last stage, there is no shuffle write, or shuffle read on the first stage)
      * **Stage Id**
      * **Pool Name** - ignore/skip this
      * **Description:**
        1. Navigate to the stage-specific details via the link in the description
        2. Call out that the description provides information as to what operation (usually an action) was executed
      * **Submitted** - [*already covered*]
      * **Duration** - [*already covered*]
      * **Tasks: Succeeded/Total**
      * **Input -** Bytes read in from storage (aka S3)
      * **Output -** Bytes written to storage (aka S3)
      * **Shuffle Read -** Bytes and records read from a shuffle in a previous stage
      * **Shuffle Write** - Bytes and records written for a shuffle in a future stage

### 

### Spark UI / Stages

1. **Fair Scheduler Pools**
   1. Databricks generally only has one scheduling pool
   2. Note the scheduling mode: FIFO
2. **Completed Stages:**
   1. Starts with 0, most recent on top
   2. Columns
      * **Stage Id**
      * **Pool Name** - [*already covered*]
      * **Description**
        + Call out that the description provides information as to what operation (usually an action) was executed.
        + Note that some descriptions include the same action multiple times because those stages belong to the same job.
        + Navigate to the job-specific details via the link in the description
      * **Submitted -** [*already covered*]
      * **Duration** - The duration of the stage
      * **Tasks: Succeeded/Total**
      * **Input:** Bytes read in from storage (aka S3)
      * **Output:** Bytes written to storage (aka S3)
      * **Shuffle Read:** Bytes and records read from a shuffle in a previous stage
      * **Shuffle Write:** Bytes and records written for a shuffle in a future stage

### 

### Spark UI / Stages / Stage

1. Select Stage #13
2. **Details for Stage #**
   1. **Total Time Across All Tasks**
   2. **Locality Level Summary**: Explain each locality level later
   3. **Input Size / Records:** The size and number of records read in [*not shown here*]
   4. **Output Size / Records:** The size and number of records written out [*not shown here*]
   5. **Shuffle Write Size / Records:** The size and number of records written out
   6. **Shuffle Read Size / Records:** The size and number of shuffle records read in
   7. **Spill (Memory)**: The size of the spilled data while in memory
   8. **Spill (Disk)**: The size of the spilled data while on disk
   9. **Associated Job Ids**
3. **DAG Visualization**:
   1. Updated in Spark 3 - be more useful (note the **foreach at command**)
   2. Some of the Physical Plan is visible here
   3. Generally not deemed to be too helpful, see the SQL diagrams instead
4. **Event Timeline**:
   1. Shows how much time each task is spending in the various stages (scheduling vs serialization vs execution vs etc)
   2. Nearly all green is good! We want to spend as much time in execution as possible. This is covered more during the lab - save contrasts until then
   3. You will not see every task represented on this map

### 

### Spark UI / Stages / Stage

1. **Summary Metrics**:
   1. Recap meaning of **Min**, **25th percentile**, **Median**, **75th percentile** and **Max**
   2. Columns:
      * **Duration:** Time that the executor spent processing a task
      * **GC Time:** Time that the executor spent paused for Java garbage collection while the task was running. Compare to **Duration** to evaluate for performance. While subjective, zero or 1% of **Duration** is great, whereas equal to or greater than **Duration** is really bad.
      * **Shuffle spill (memory):** Total bytes of the deserialized form of the data in memory at the time the data was spilled.
      * **Shuffle spill (disk):** Total bytes of the serialized form of the data on disk after it was spilled. Compression from serializing generally results in a smaller number of bytes compared to the deserialized form.
      * **Shuffle Read Size / Records:** Total shuffle bytes and records read (includes both data and read locally and data read from remote executors)
      * **Shuffle Write Size / Records:** Bytes and records written to disk in order to be read by a shuffle in a future stage.
      * **Scheduler Delay:** Scheduler delay includes time to ship the task from the scheduler to the executor, and time to send the task result from the executor to the scheduler. If scheduler delay is large, consider decreasing the size of tasks or decreasing the size of task results.
      * **Task Deserialization Time:** Time spent deserializing the task closure on the executor, including the time to read the broadcasted task.
      * **Shuffle Read Blocked Time:** Time that the task spent blocked waiting for shuffle data to be read from remote machines
      * **Shuffle Remote Reads:** Total shuffle bytes read from remote executors. This is a subset of the shuffle read bytes; the remaining shuffle data is read locally.
      * **Result Serialization Time:** Time spent serializing the task result on the executor before sending it back to the driver.
      * **Getting Result Time:** Time that the driver spends fetching task results from workers. If this is large, consider decreasing the amount of data returned from each task.
      * **Peak Execution Memory:** Execution memory refers to the memory used by internal data structures created during shuffles, aggregations, and joins when Tungsten is enabled. The value of this accumulator should be approximately the sum of the peak sizes across all such data structures created in this task. For SQL jobs, this only tracks all unsafe operators broadcast joins and external sort.
      * **Input Size / Records:** Bytes and records read in from storage (aka S3) [*not shown here*]
      * **Output Size / Records:** Bytes written to storage (aka S3) [*not shown here*]
   3. Great for diagnosing different types of problems
      * Any evidence of a spill should be addressed
      * Look for high GC Time meaning we are spending too much time in garbage collection
      * Look for **Input Size / Records** that have a **Max** value well above the **Median**. This is a pretty good indication that you have a significant skew in your data because one partition (and thus task) has significantly more data than any other
      * Variance in the **Peak Execution Memory** can help diagnose GC causes, OOMs, or general poor performance.

### Spark UI / Stages / Stage

1. **Aggregated Metrics by Executor**:
   1. Columns
      * **Executor ID**
      * **Logs** - quick link to the standard error and out log files per executor
      * **Address**
      * **Task Time** - the aggregated sum of all task durations per executor
      * **Total Tasks** - total number of tasks executed per executor
      * **Failed Tasks** - the number of tasks that failed per executor
      * **Killed Tasks** - the number of tasks that were killed per executor
      * **Succeeded Tasks** - the number of successfully completed tasks per executor
      * **Blacklisted** - if the cluster was banned from use after many errors
      * **Shuffle Read Size / Records, Shuffle Write Size / Records** - [*already covered*]
      * **Input Size / Records, Output Size / Records:** [*not shown here*] [*already covered*]
   2. General Notes
      * Break down by executor - generally not too useful unless there are problems at the machine/VM level with a specific executor

### 

### Spark UI / Stages / Stage

1. **Tasks**:
   1. Columns:
      * **Index**
      * **Task ID**
      * **Attempt** - Indicates how many times spark attempted to execute a given task
      * **Status** - The status of a task, usually SUCCESS
      * **Locality Level -** On Databricks, this is almost always **PROCESS\_LOCAL** and not worth going into at this time (Delta IO will occasionally show **NODE\_LOCAL**)
        1. **PROCESS\_LOCAL** data is in the same JVM as the running code. This is the best locality possible
        2. **NODE\_LOCAL** data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS\_LOCAL because the data has to travel between processes
        3. **NO\_PREF** data is accessed equally quickly from anywhere and has no locality preference
        4. **RACK\_LOCAL** data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch
        5. **ANY** data is elsewhere on the network and not in the same rack
      * **Executor ID**
      * **HOST**
      * **Logs** [*already covered*]
      * **Launch Time** - when the task was started
      * **Duration, GC Time, Scheduler Delay, Task Deserialization Time, Shuffle Read Blocked Time**, **Shuffle Remote Reads, Result Serialization Time, Getting Result Time, Peak Execution Memory** [*already covered*]
      * **Input Size / Records, Output Size / Records:** [*already covered, not shown here*]
      * **Write Time:** **TODO ???**
      * **Shuffle Write Size / Records, Shuffle Read Size / Records:** [*already covered*]
      * **Errors** - If a specific task threw an exception during processing or had some other type of error
   2. What would be the concern if we saw scores if not hundreds of tasks with zero records - underutilization, delays from scheduling overhead
   3. Columns can be sorted so we can see which task read in the most data (help narrow down skew)
   4. Columns can be sorted so we can see which task used the most memory, or too much memory, helping to identify GC performance issues or potential OOM errors before they occur.

### 

### Spark UI / Storage

1. The **Parquet IO Cache**, aka **Delta Cache**, shows the amount of disk space used and available for caching on a worker
2. Each host (aka worker) is identified only by IP address
3. **Delta Cache** is only available on specific VM types and files types (as of 5/1/20, Parquet and Delta only)
4. The **RDD** cache is the storage space RDD caching (aka DataFrame.cache or DataFrame.persist, SparkSession.catalog.cacheTable)
5. Name is ugly - computed dynamically from the DataFrame.  
   Best Practice: Use **SparkSession.catalog.cacheTable** to control the name
6. General Metrics: storage level, number of partitions cached, percentage of the total dataset that is cached, size in memory (RAM) and size on disk (work’s HDD)
7. Navigate to the RDD’s cache by clicking on the link in the **RDD Name** column.

### 

### Spark UI / Storage / RDD

1. Select RDD #5
2. Header recaps information on the previous screen
3. The **Data Distribution** section shows how much memory is used and remains for each worker
4. Easily see if data is cached to disk or memory
5. Default is memory and disk - here you can see how much spilled from memory to disk
6. Easy way to see how data is balanced across all partitions - promotes caching which is actually an anti-pattern

### 

### Spark UI / Environment

1. A lot of random values - but helpful when you need to know specifics like which version of Java are you running or speculative execution would help with a job - is it actually turned on?
2. A user claims they configured the cluster for auto termination but **spark.databricks.clusterUsageTAgs.autoTerminationMinutes** is set to zero
3. When I attached library X to the cluster, what other libraries were transitively attached to the cluster? It helps to diagnose library conflicts.

### 

### Spark UI / Executors

1. A lot of redundant information
2. Things of interest include:
   1. Failed Tasks
   2. GC Time
   3. If an executor was blacklisted (not common)
   4. Number of cores (but you should already know that)
3. The most valuable usage of this screen is access to the Logs, Thread Dumps, and Heap Histograms for the driver and each executor.
   1. Updates to the Spark UI make the logs accessible in other contexts such as a stage or task

### 

### Spark UI / SQL

1. The key concept here is the correlation between Queries and Jobs.
2. Not shown here, but not uncommon for one query to be supported by 2+ jobs.
3. The description is not as helpful here as it is in the Jobs and Stages tab unless you annotate your cells with a meaningful comment
4. Expanding **+details** will show initiating code/action seen in the jobs and stages details
5. Navigate to a query’s details by clicking on the link in the description column

### 

### Spark UI / SQL / Details (Query 91)

1. Probably the most useful of all the detailed screens!
2. Documents actual metrics for each stage of a query such as
   1. Did any of the shuffle spill? [shown here with **0.0 B**]
   2. Might a join have skew [not shown here]
   3. If I’m having OOM errors, I can see what stage/operation is peaking on memory usage (e.g. right after an **explode** operation)
   4. Estimations give you the opportunity to employ specific optimizations. For example, if the number of records is estimated to be **2,345,943 records**, but it is actually **549 records**, then you might explicitly add a broadcast hint during join operations (not demoed here)
3. The **Details** at the bottom itemize the
   1. **Parsed Logical Plan**
   2. **Analyzed Logical Plan**
   3. **Optimized Logical Plan**
   4. **Physical Plan**
4. While the plans are hard to understand at first, you **MUST** invest in learning this.
5. Look at the **Physical Plan**, last line, **File Scan**: Note that the **PartitionFilters** is empty - if I saw this on a partitioned dataset that was filtering on a partitioned column, this will tell me that the predicate is not being pushed down

### 

### Spark UI / JDBC

1. Nothing to show here (yet), but simply call out more information is available here for those types of jobs.

### 

### Spark UI / Structured Streaming

Columns:

1. **Name** - The name of the batch as represented by **StreamingQuery.queryName(name)**
2. **Status** - The current status of the stream
3. **ID** - The ID of a specific stream
4. **Run ID** - The ID of the **run** of a specific stream
5. **Start Time** - When the stream was started
6. **Duration** - How the the stream has been (or was) running for
7. **Avg Input /sec** - The average number of records input to the stream per second
8. **Avg Process /sec** - The average number of records processed by the stream per second
9. **Latest Batch** - The latest batch ID to have been executed
10. **Error** - Links to the errors, if any

### Spark UI / Structured Streaming

While not possible with the simulator, each of the line graph and bar charts provide additional information via tool-tips

Graphs:

1. **Input Rate** - The aggregate (across all sources) rate of data arriving
2. **Process Rate** - The aggregate (across all sources) rate at which Spark is processing data
3. **Input Rows** - The aggregate (across all sources) number of records processed in a trigger
4. **Batch Duration** - The process duration of each batch
5. **Operation Duration** - The amount of time taken to perform the various operations expressed in milliseconds
   * walCommit
   * queryPlanning
   * latestOffset
   * getBatch
   * addBatch