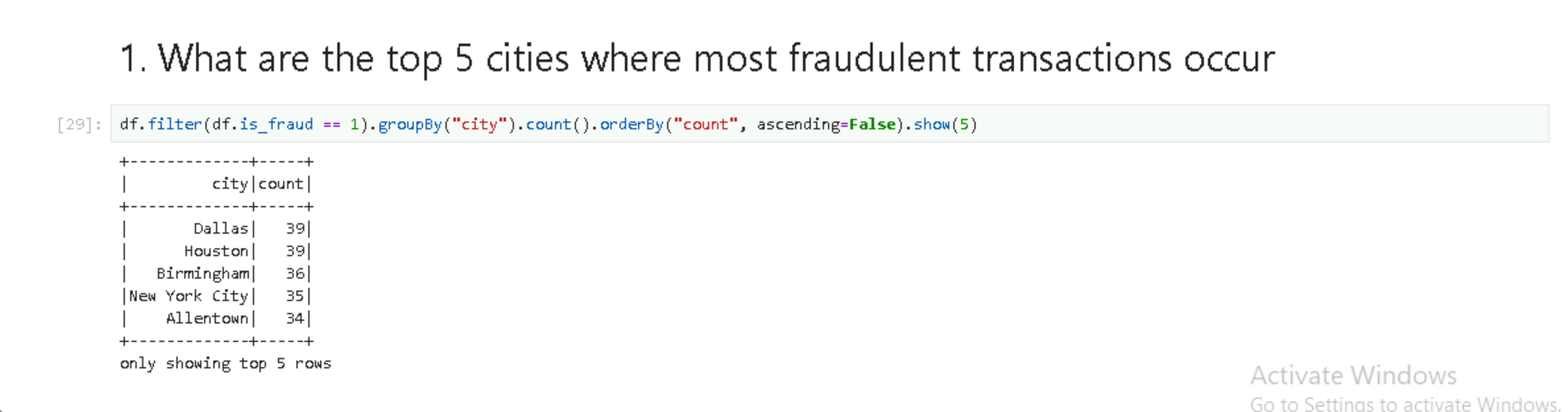
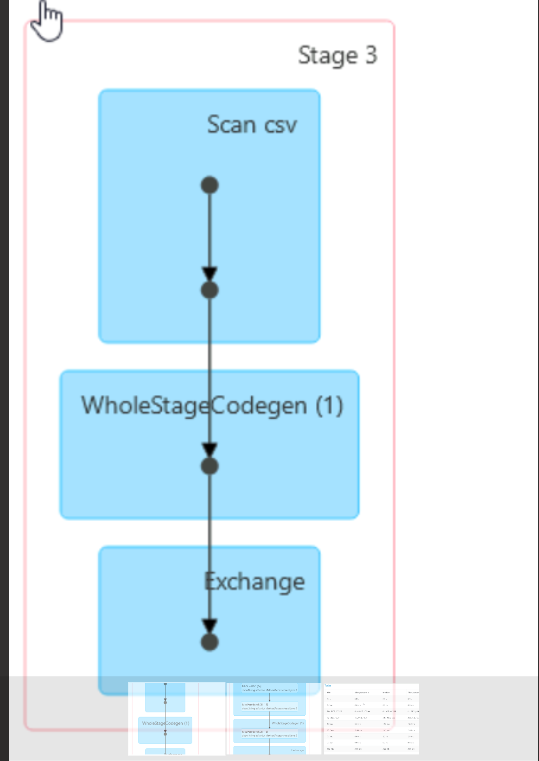
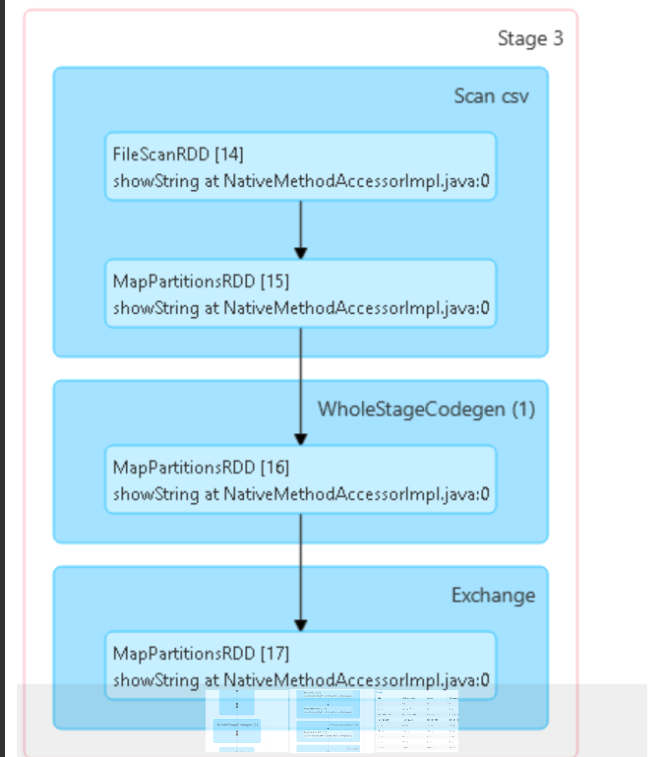
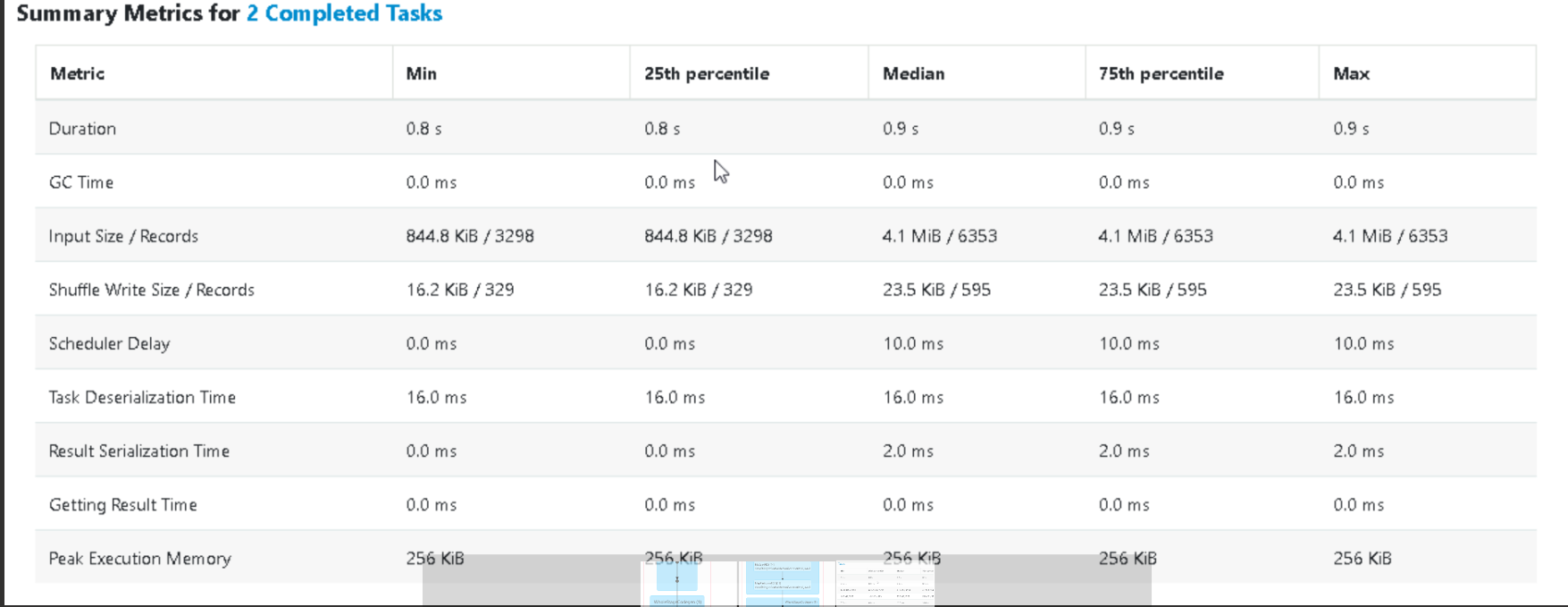
**GROUP E**

**Q1)** What are the top 5 cities where most fraudulent transactions occur?







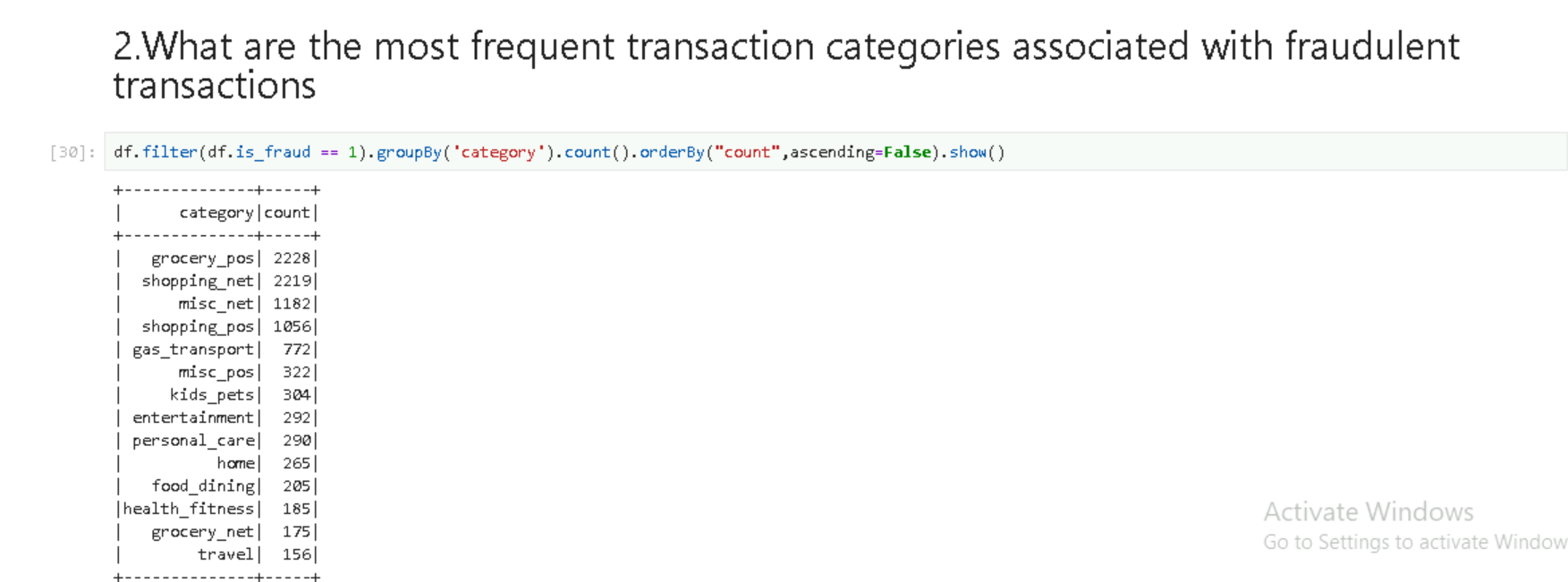


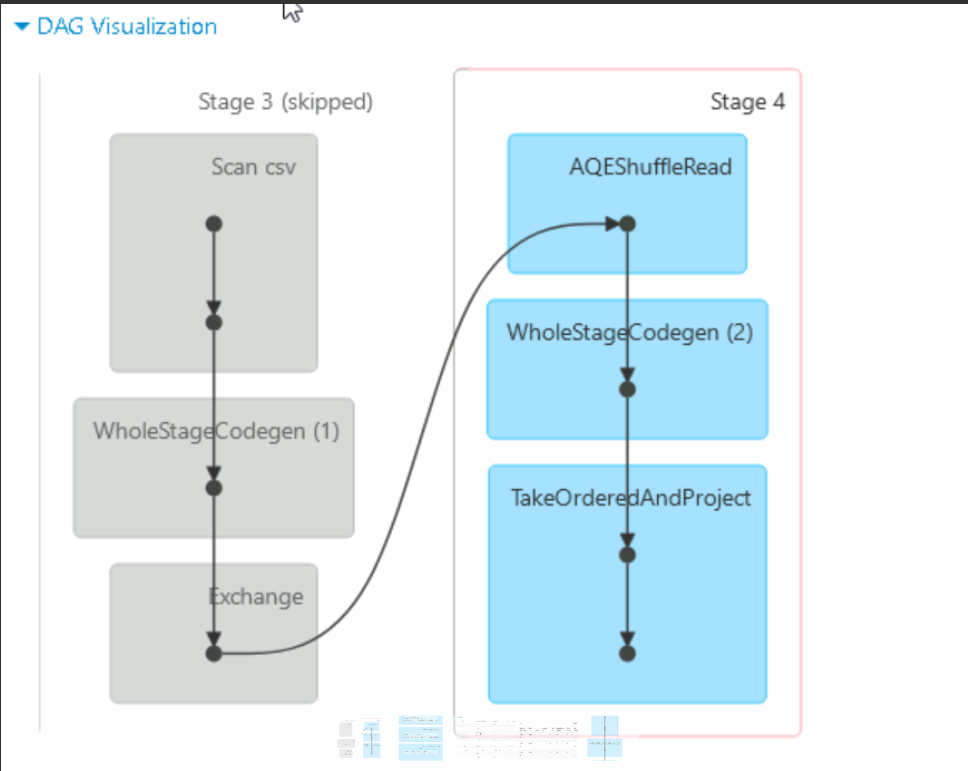
GC Time: Time spent on garbage collection during task execution.

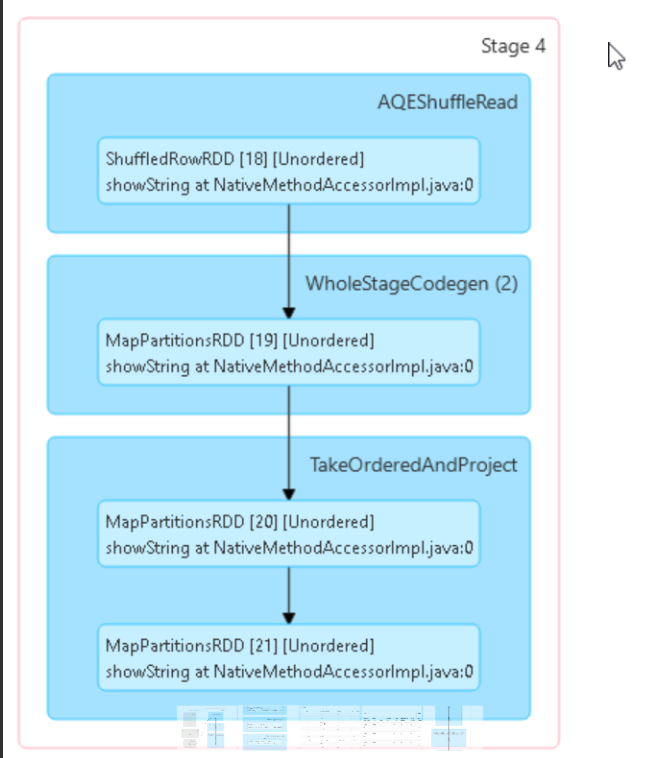
Result Serialization Time: Time taken to serialize the task result before sending it back to the driver.

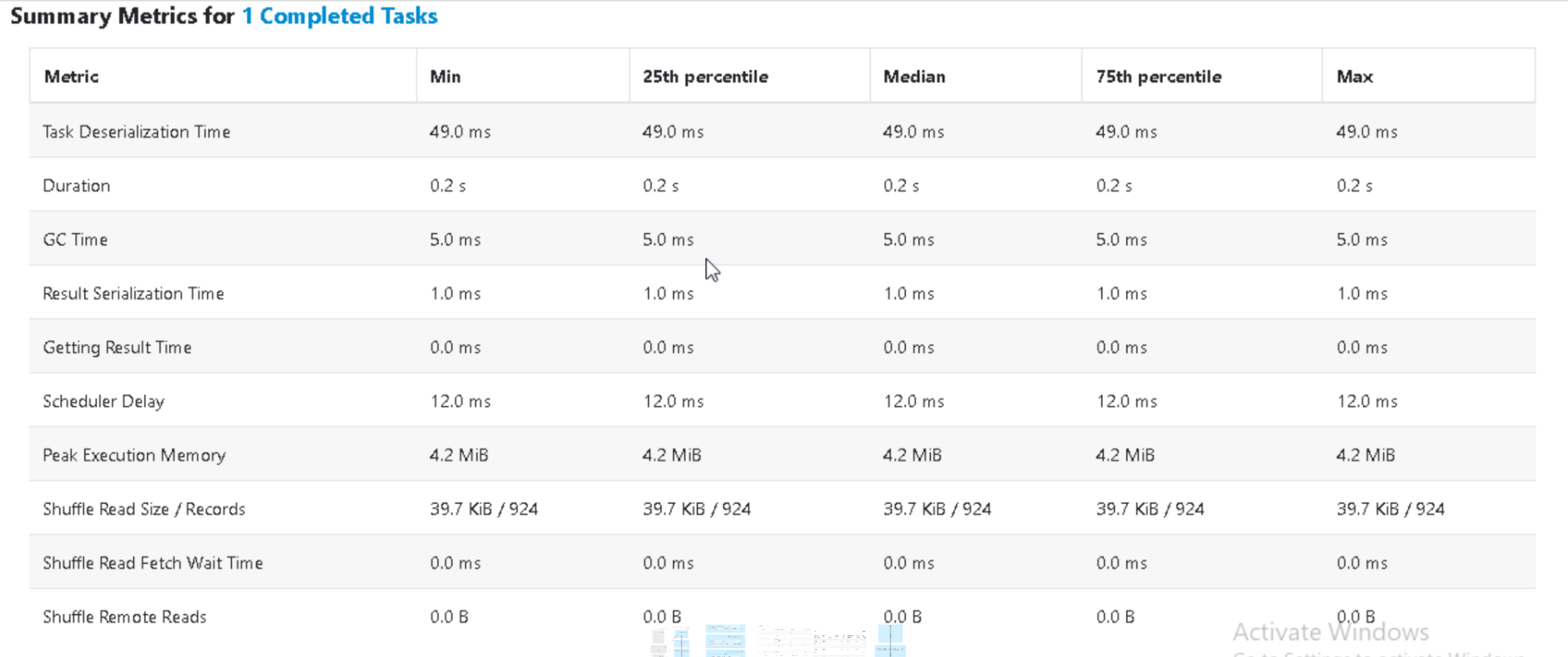
Task Deserialization Time: Time taken to deserialize a task before execution.

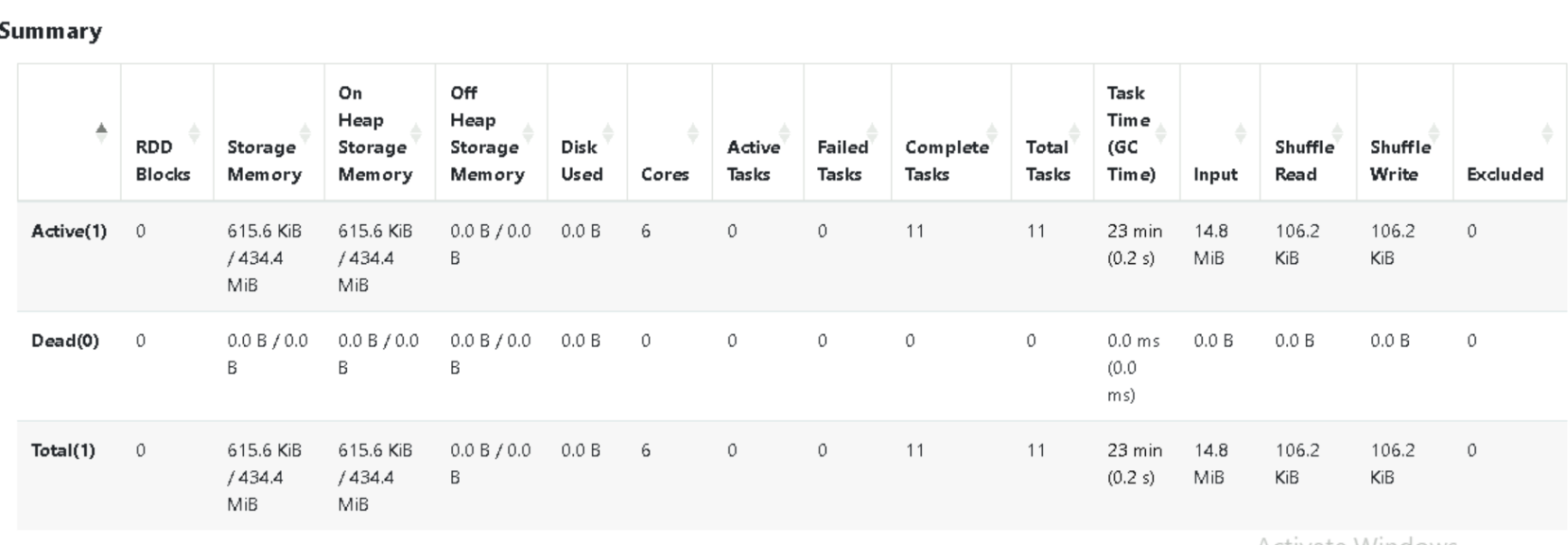
**Q2)** What are the most frequent transaction categories associated with fraudulent transactions?





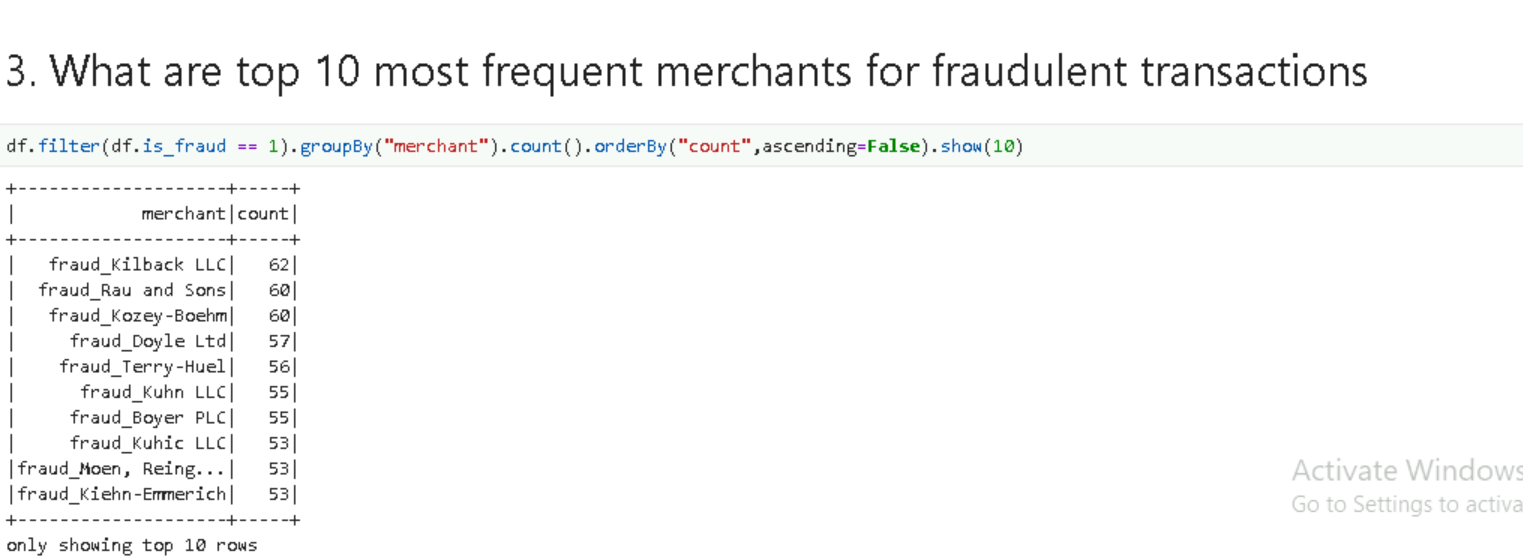


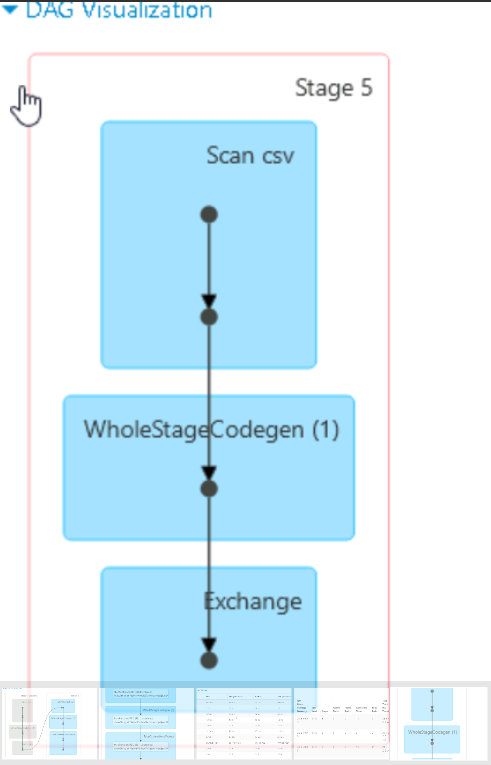




In Spark UI, a skipped stage refers to a stage that Spark does not execute because its output is already available.

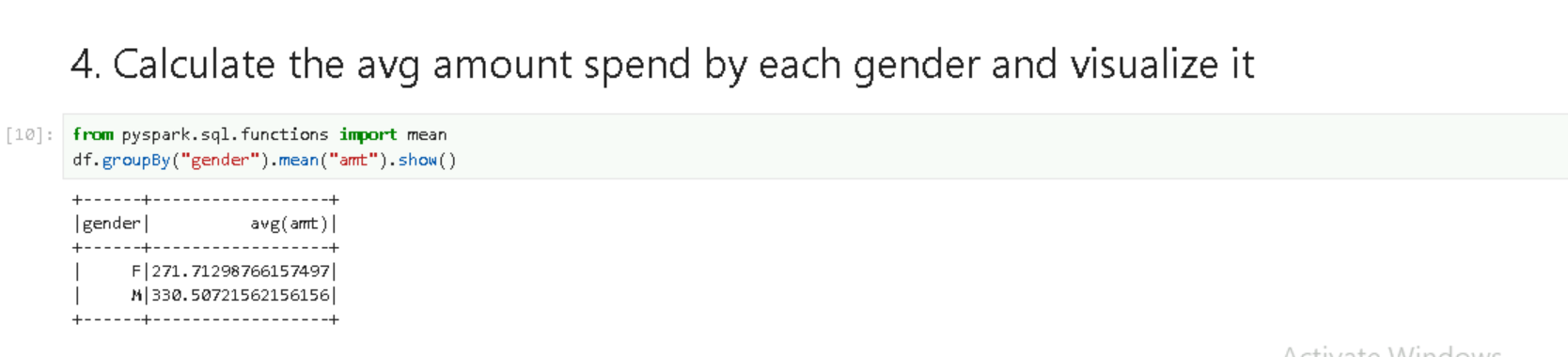
**Q3**) What are top 10 most frequent merchants for fraudulent transactions?

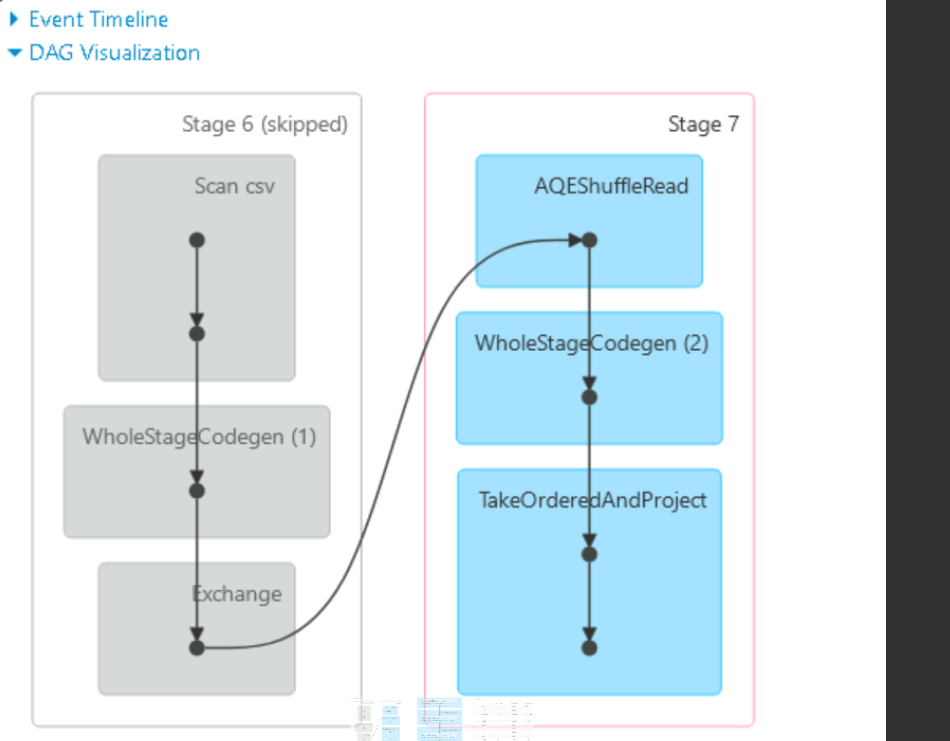


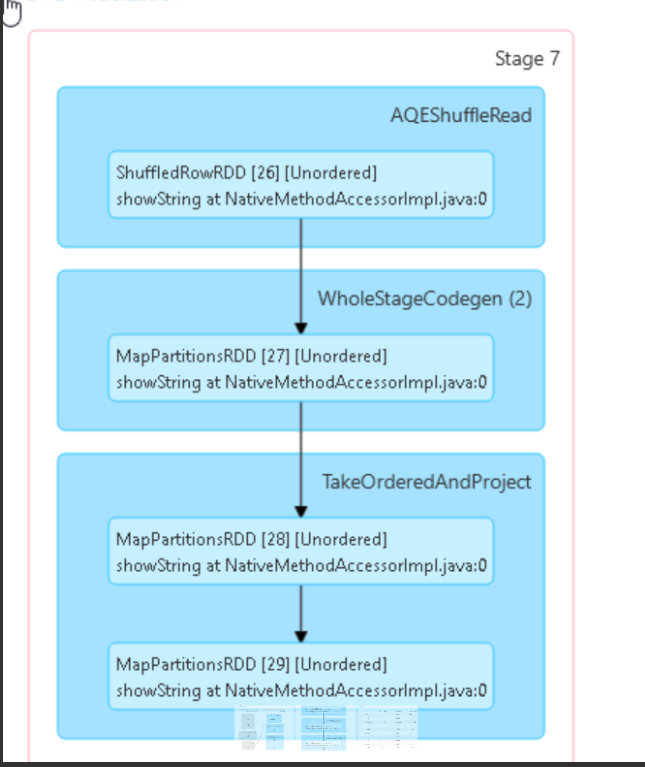


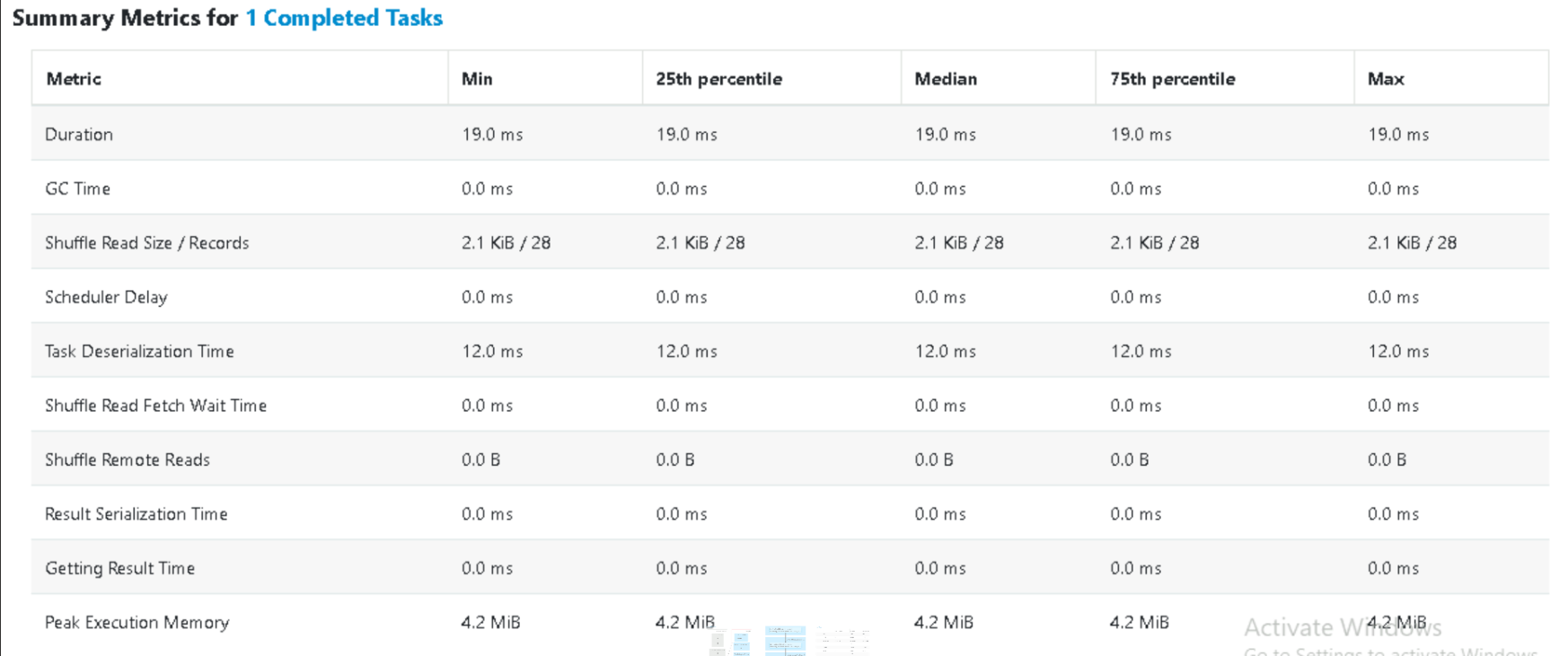


**Q4)** Calculate the avg amount by each gender and visualize it

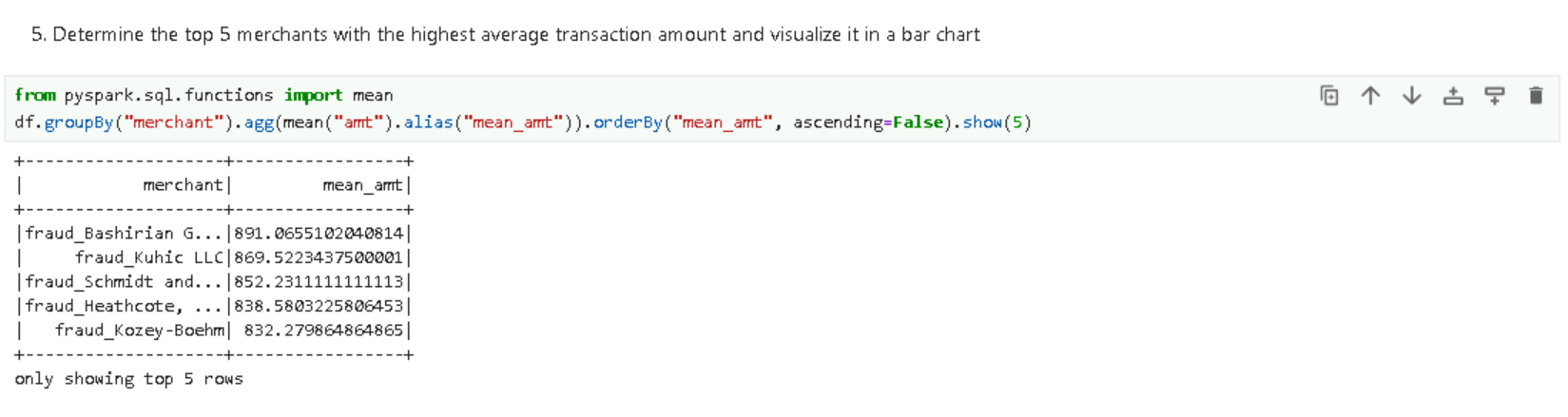


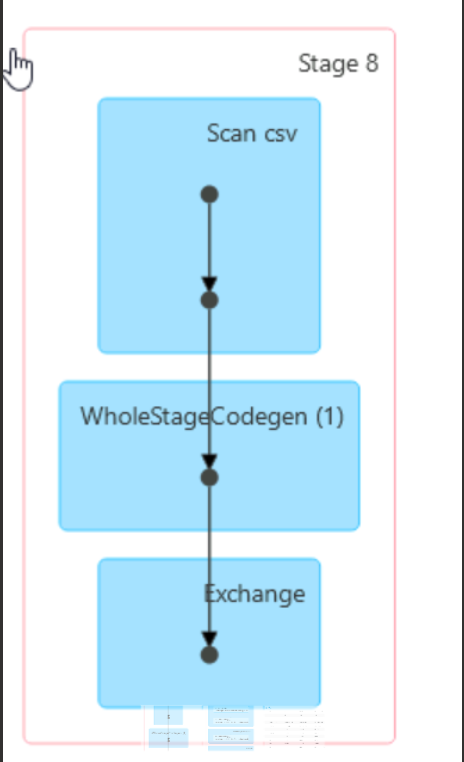


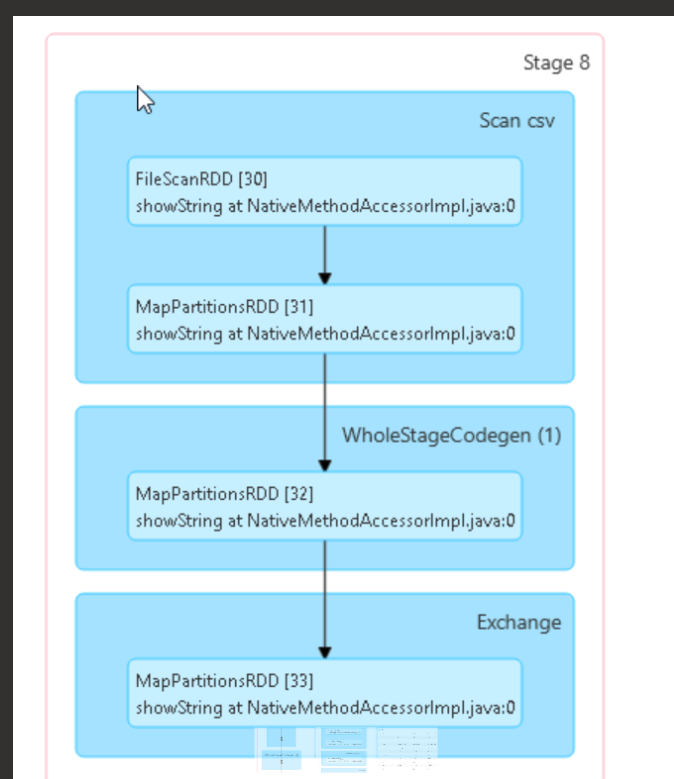


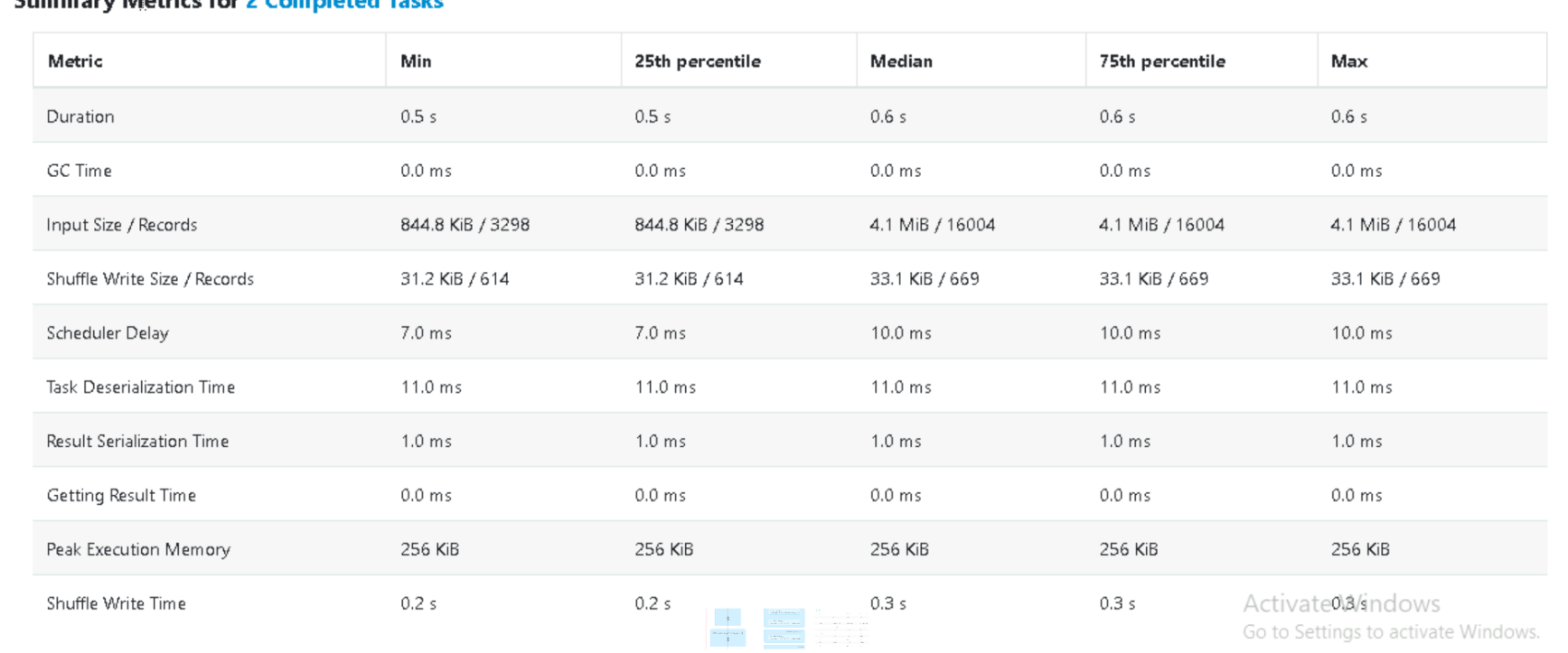


**Q5)** Determine the top 5 merchants with the highest average transaction amount and visualize it in a bar chart.

****

****

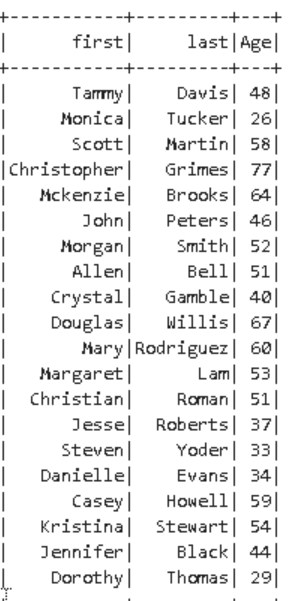
****

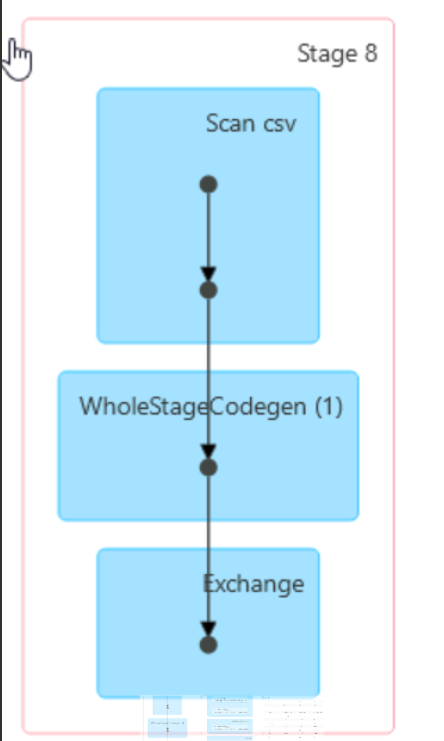
****

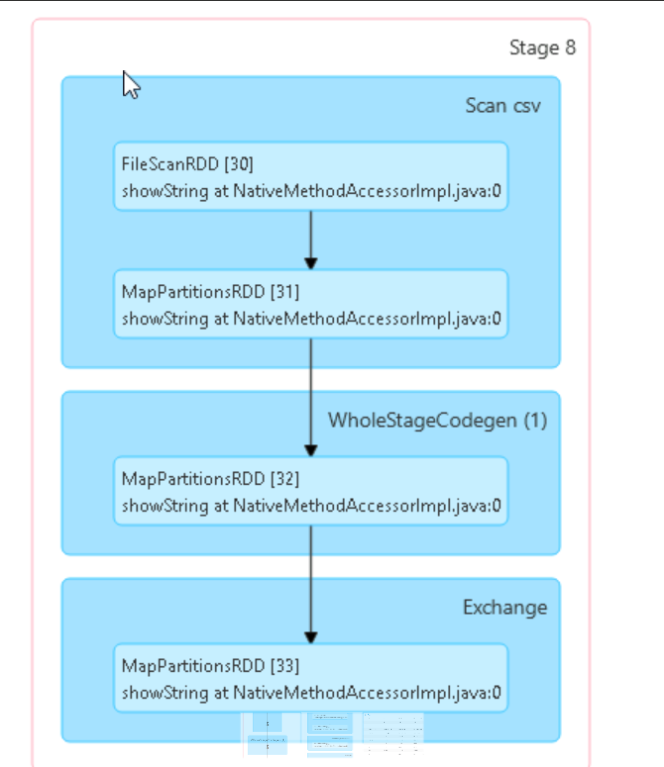
**Q6)** Calculate the age of each customer and create a histogram of ages for fraudulent transactions.

****

**OUTPUT**

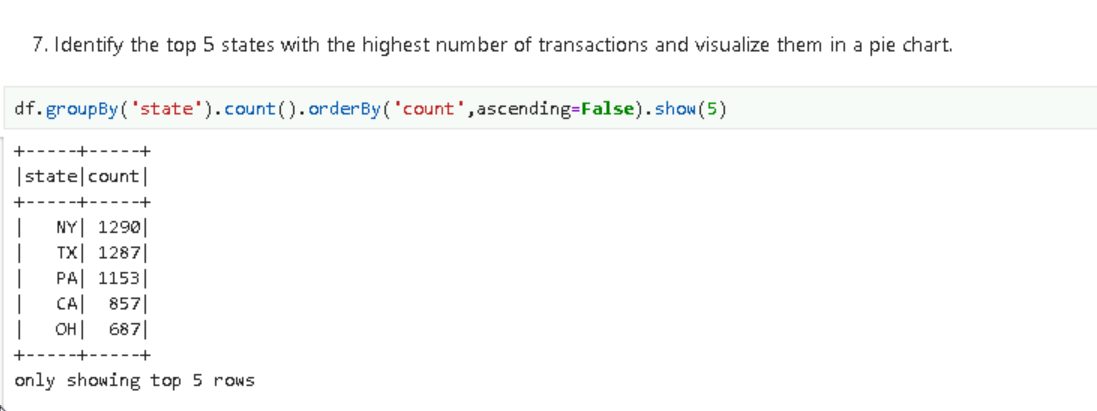
****

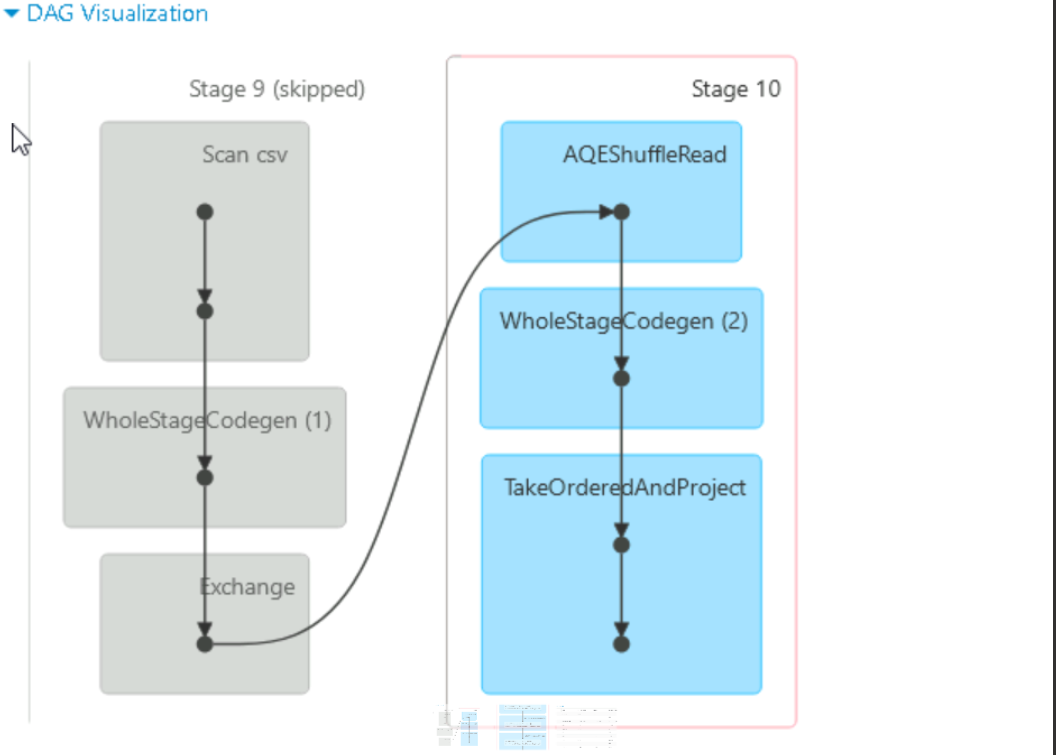
****

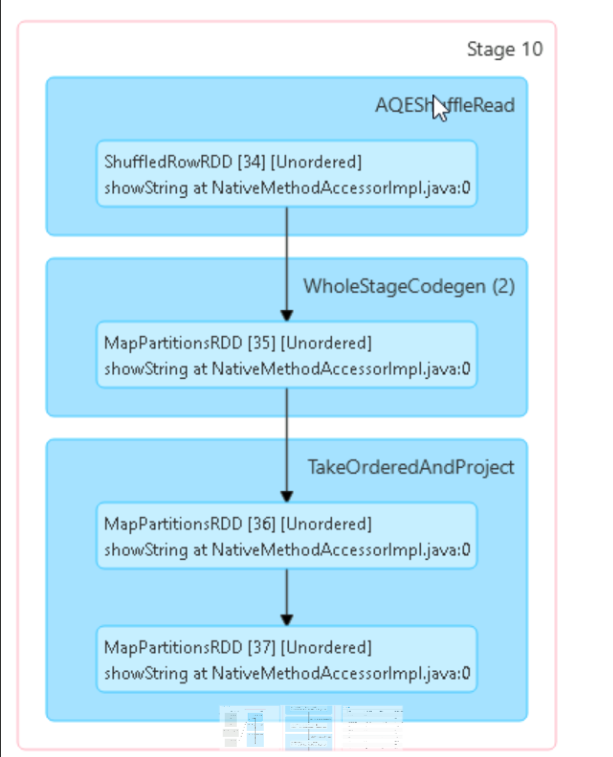
****

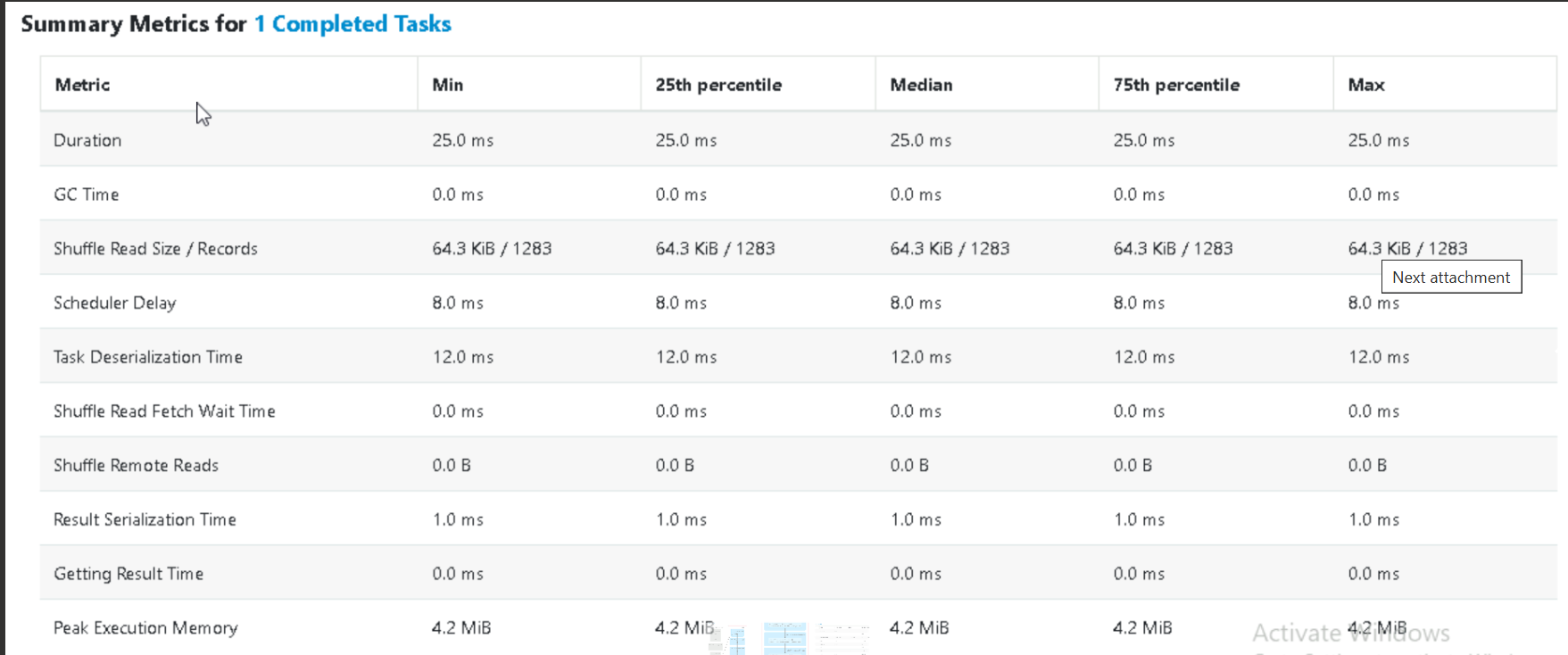
****

**Q7)** Identify the top 5 states with the highest number of transactions and visualize them in a pie chart

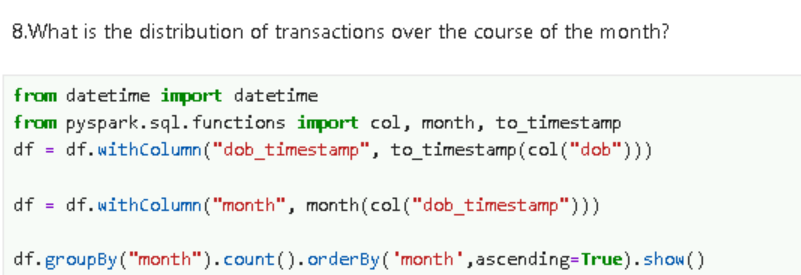
****

****

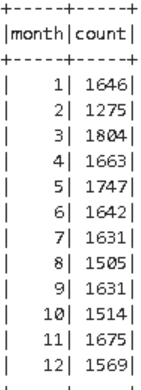
****

****

**Q8)** What is the distribution of transactions over the course of the month?

****

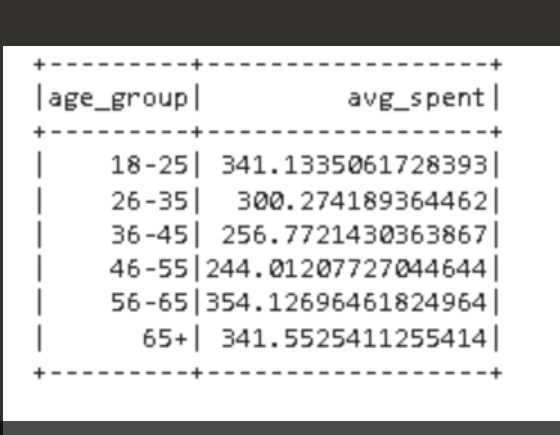
OUTPUT



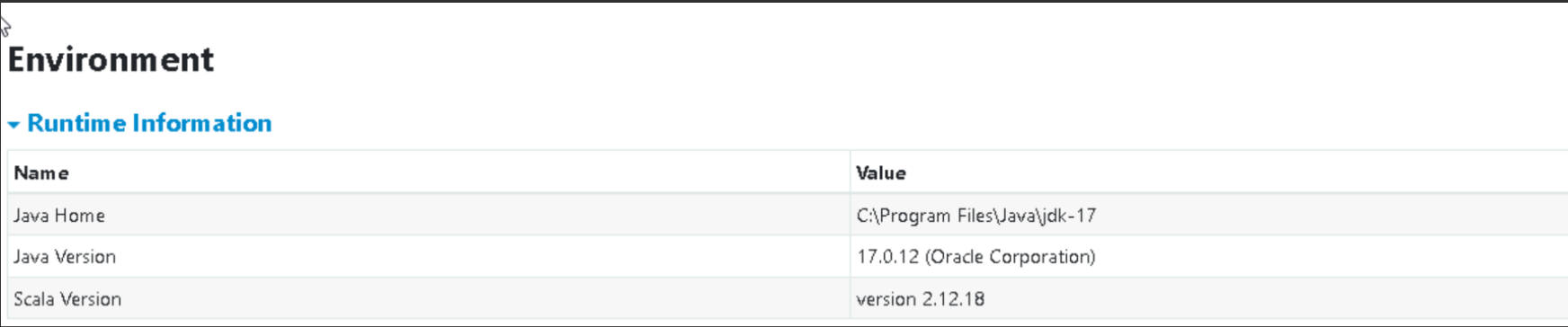
**Q10**) Analyse spending pattern based on age groups and avg spending amount for each group

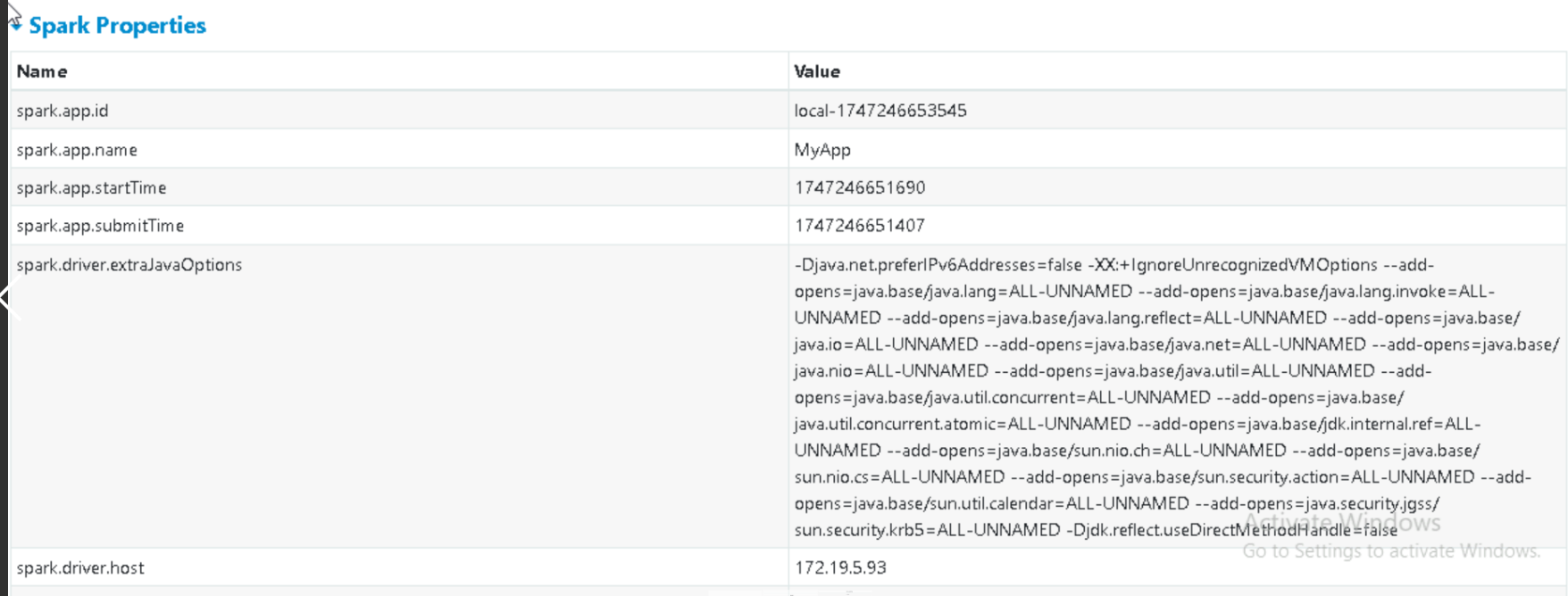


OUTPUT



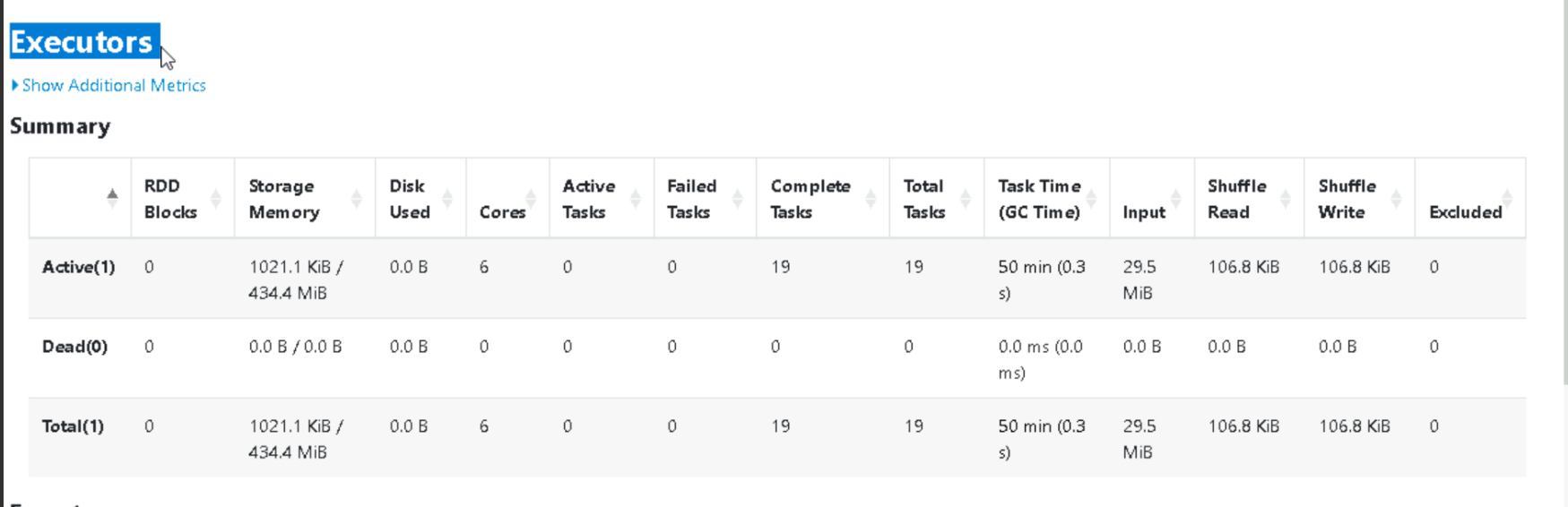
EXPLORATIONS

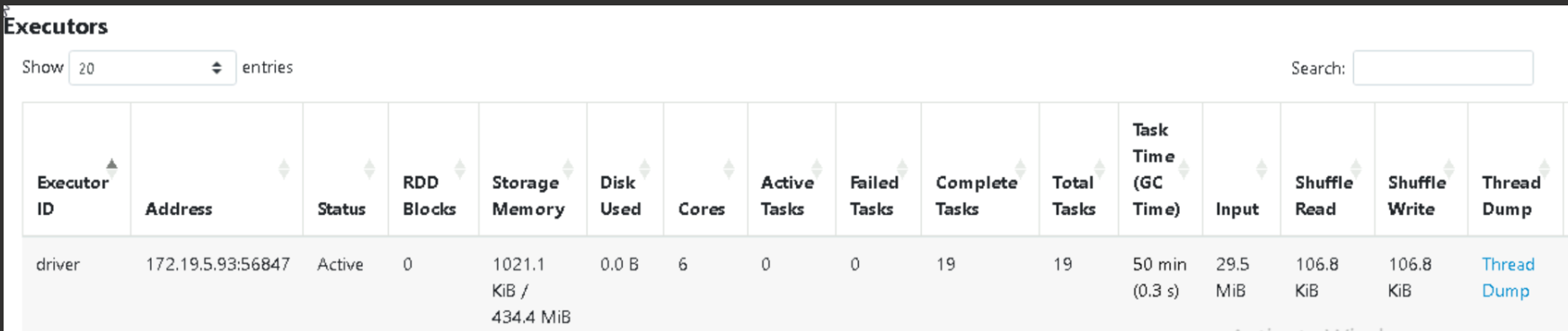




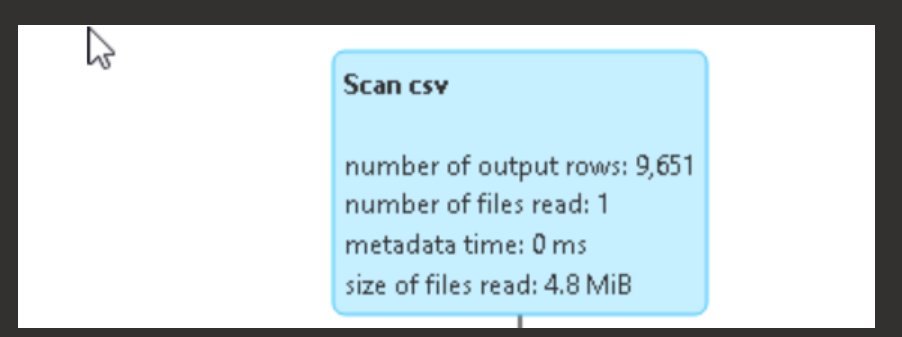


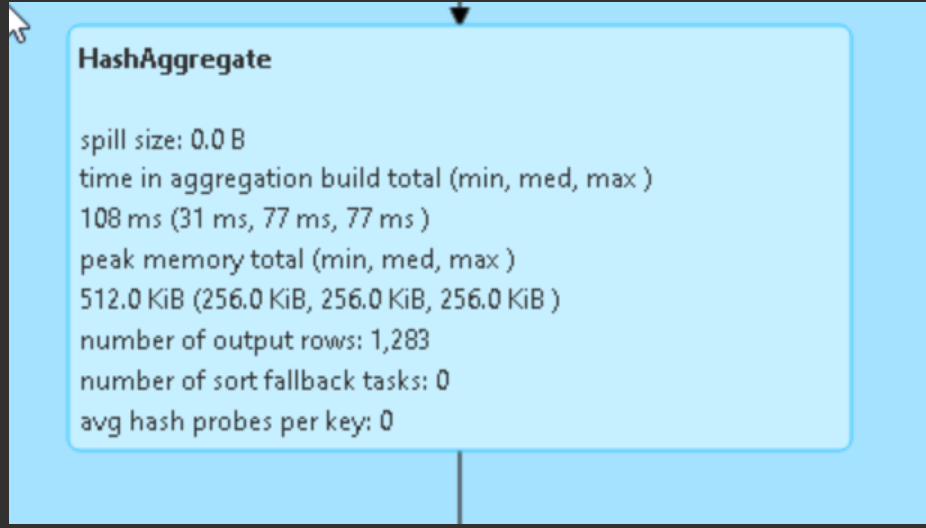
Executors





HASH AGGREGATE AND CSV





Scan csv scans the csv files processing of reading and analysing csv data using the sparks built in capabilities

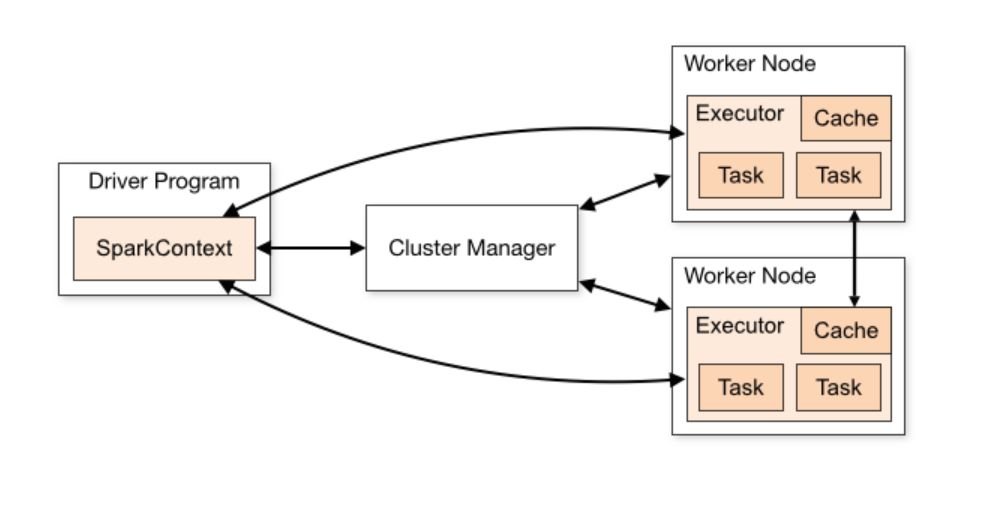
Hash aggregation is physical execution strategy used for performing aggregation operations efficiently. Commonly seen in query plans when spark processes grouping and aggregation function like GROUP BY, COUNT, SUM, etc.

SPARK – TYPES OF MEMORY AND MANAGEMENT

The Spark application includes two JVM processes: driver and executor.

* The driver is the main control process, which is responsible for creating the SparkSession/SparkContext, submitting the job, converting the job to a task, and coordinating the task execution between executors.
* The executor is mainly responsible for performing specific calculation tasks and returning the results to the driver.

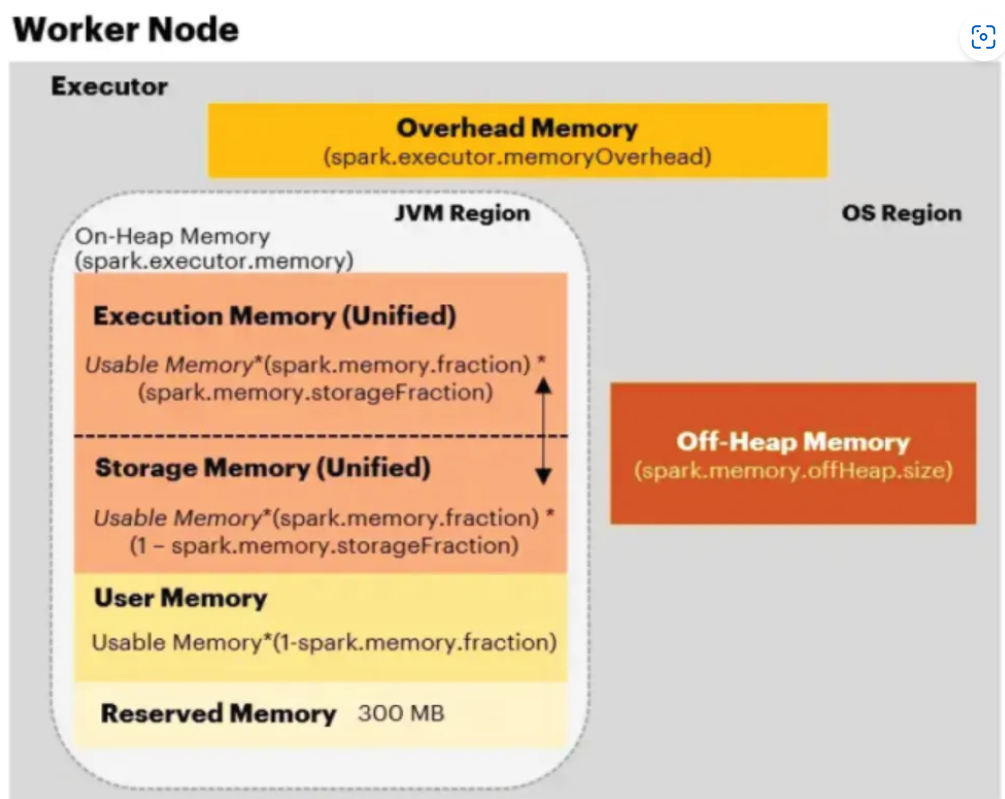
Driver's memory management is relatively simple; Spark does not make specific plans



The executor acts as a JVM process launched on a worker node. So, it is important to understand JVM memory management.

JVM memory management is categorized into two types:

1. **On-Heap Memory Management (In-Heap Memory):** Objects are allocated on the JVM Heap and bound by GC.
2. **Off-Heap Memory Management (External Memory):** Objects are allocated in memory outside the JVM by serialization, managed by the application, and are not bound by GC.
3. **Overhead memory:** It is used for the various internal Spark operations. Normally, it is set to about 10% of the total spark.executor.memory, and the default value is also 10%



On-Heap Memory:

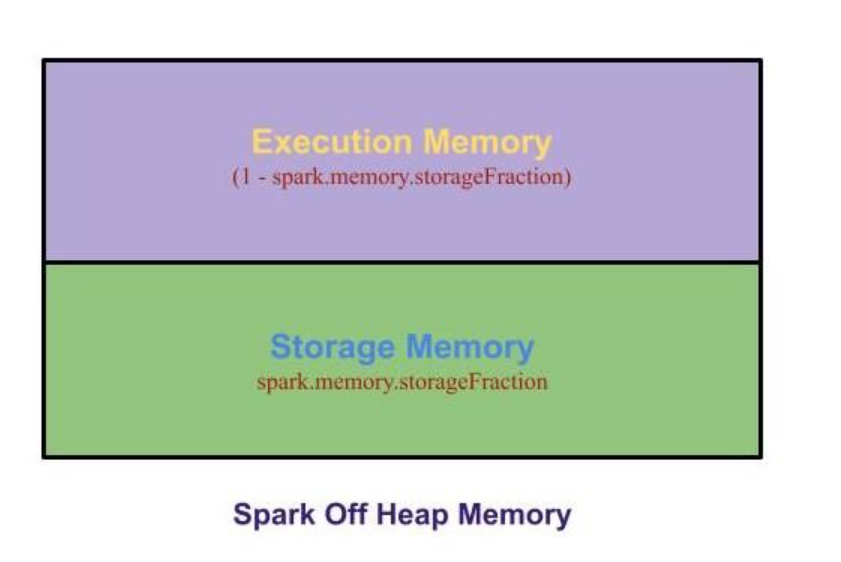
Most of Spark’s operations run on On-Heap memory, which is managed by the JVM and used to store Resilient Distributed Datasets (RDDs). When data is first loaded into PySpark, it is stored in On-Heap memory as an RDD.

The On-Heap memory is divided into four main sections:

* **Execution memory** is used for operations such as shuffles, joins, sorts, and aggregations. It is typically shorter-lived than storage memory, as it is only required for the duration of each task. Once an operation is completed, the memory is immediately freed up and made available for the next set of tasks, ensuring efficient use of resources.
* **Storage Memory** is used for caching and broadcasting data.
* **User Memory** is the memory used to store user-defined data structures, Spark internal metadata, any UDFs created by the user, and the data needed for RDD conversion operations, such as RDD dependency information, etc.
* **Reserved Memory** is to store internal objects. It guarantees to reserve sufficient memory for the system, it is hardcoded and equal to 300 MB

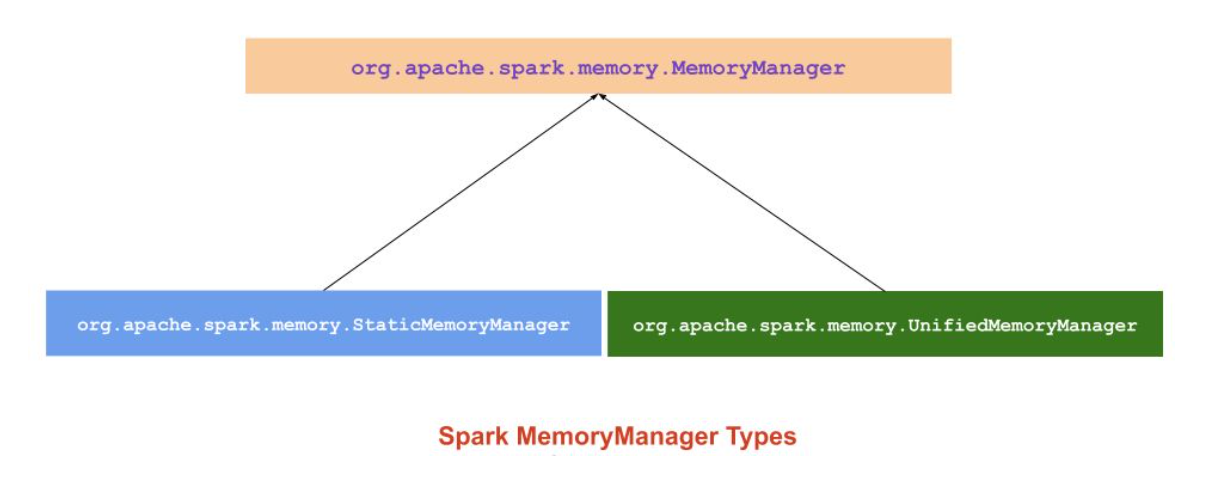
**Off-Heap memory:**

* Off-heap memory exists separately from the JVM area and is located within the Spark container (Executor).
* Off-heap refers to objects (serialized to byte array) that are managed by the operating system.
* Accessing this data is slightly slower than accessing the on-heap storage but still faster than reading/writing from a disk.
* **It can be used only when spark.memory.offHeap.enabled=true** is set. (default = false)



**Memory Allocation:**

Spark provides an interface for memory management via MemoryManager. It implements the policies for dividing the available memory across tasks and for allocating memory between storage and execution. Execution and Storage memory together is called as **Unified Memory.**



Since Spark 1.6.0, Unified Memory Manager has been set as the default memory manager for Spark.

Static Memory Manager has been deprecated because of its lack of flexibility.

In both memory managers, a portion of Java Heap is located for processing Spark applications, while the rest of the memory is reserved for Java class references and metadata usage.

 Static Memory Manager (SMM)

Static Memory Manager (SMM) is the traditional model and simple scheme for memory management.

It divides memory into two fixed partitions statically.

The size of storage memory, execution memory, and other memory is fixed during application processing, but users can configure it before the application starts.

Note: The Memory allocation method has been eliminated in Spark 3.0

Advantage:

* The static Manager mechanism is simple to implement

Disadvantage:

* Even though space is available with storage memory, we can’t use it, and there is a disk spill since executor memory is full. (vice versa).

Unified Memory Manager (UMM)

* Since Spark 1.6.0, a new memory manager has been adopted to replace the static memory manager and provide Spark with dynamic memory allocation.
* It allocates a region of memory as a unified memory container that is shared by storage and execution.
* When execution memory is not used, the storage memory can acquire all the available memory, and vice versa.
* If any of the storage or execution memory needs more space, a function called acquireMemory() will expand one of the memory pools and shrink another one.
* Borrowed storage memory can be evicted at any given time. Borrowed execution memory, however, will not be evicted in the first design due to complexities in implementation.

Advantages:

1. The boundary between storage memory and execution memory is not static, and in cases of memory pressure, the boundary would be moved, i.e., one region would grow by borrowing space from another one.
2. When the application has no cache and is propagating, execution uses all the memory to avoid unnecessary disk overflow.
3. When the application has a cache, it will reserve the minimum storage memory so that the data block is not affected.
4. This approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise in how memory is divided internally.