Surveillance et débogage d'Apache Spark

Temps estimé nécessaire : 30 minutes

Ce laboratoire vous expliquera comment surveiller et déboguer une application Spark via l'interface utilisateur Web.

Objectifs

Après avoir terminé ce laboratoire, vous serez capable de :

1. Démarrez un cluster autonome Spark et connectez-vous au shell PySpark.
2. Créez un DataFrame et ouvrez l'interface utilisateur Web de l'application.
3. Déboguez une erreur d'exécution en localisant la tâche ayant échoué dans l'interface utilisateur Web.
4. Exécutez une requête SQL pour surveiller, puis augmentez la capacité en ajoutant un autre worker au cluster.

Exercice 1 : démarrer un cluster Spark autonome

Dans cet exercice, vous allez initialiser un cluster autonome Spark avec un maître et un travailleur.
Ensuite, vous allez démarrer un shell PySpark qui se connecte au cluster et ouvrir l'
interface utilisateur Web de l'application Spark pour le surveiller. Nous utiliserons le terminal Theia pour exécuter des commandes et des conteneurs basés sur Docker pour lancer les processus Spark.

Tâche A : Télécharger des exemples de données pour Spark

1. Ouvrez un terminal Theia en cliquant sur l'élément de menu `Terminal -> New Terminal`.

2. Utilisez la commande suivante pour télécharger l'ensemble de données que nous utiliserons dans ce laboratoire dans le conteneur exécutant Spark.

   ```
   wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-BD0225EN-SkillsNetwork/labs/data/cars.csv
   ```

Tâche B : Initialiser le cluster

1. Arrêtez tous les conteneurs précédemment en cours d'exécution avec la commande :

   ```
   for i in `docker ps | awk '{print $1}' | grep -v CONTAINER`; do docker kill $i; done
   ```

2. Retirez tous les conteneurs précédemment utilisés :

   ignorez les erreurs indiquant « Aucun conteneur de ce type »

   ```
   docker rm spark-master spark-worker-1 spark-worker-2
   ```

3. Démarrez le serveur Spark Master :

   ```
   docker run \
       --name spark-master \
       -h spark-master \
       -e ENABLE_INIT_DAEMON=false \
       -p 4040:4040 \
       -p 8080:8080 \
       -v `pwd`:/home/root \
       -d bde2020/spark-master:3.1.1-hadoop3.2
   ```

4. Démarrez un Spark Worker qui se connectera au Master :

   ```
   docker run \
       --name spark-worker-1 \
       --link spark-master:spark-master \
       -e ENABLE_INIT_DAEMON=false \
       -p 8081:8081 \
       -v `pwd`:/home/root \
       -d bde2020/spark-worker:3.1.1-hadoop3.2
   ```

Tâche C : connecter un shell PySpark au cluster et ouvrir l'interface utilisateur

1. Lancez un shell PySpark dans le conteneur Spark Master en cours d'exécution :

   ```
   docker exec \
       -it `docker ps | grep spark-master | awk '{print $1}'` \
       /spark/bin/pyspark \
       --master spark://spark-master:7077
   ```

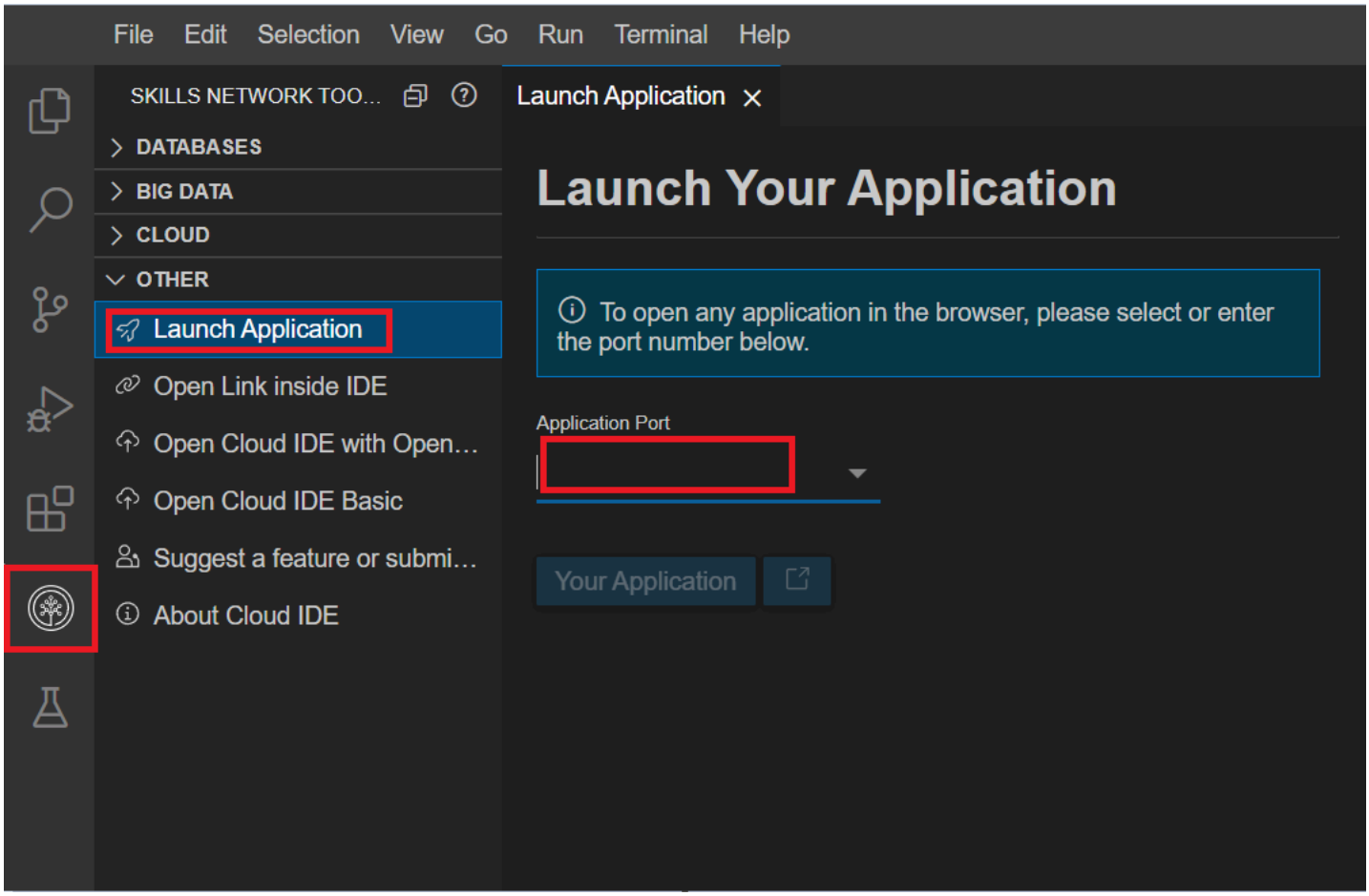
2. Créez un DataFrame dans le shell avec :

   REMARQUE : appuyez deux fois sur Entrée pour continuer après avoir exécuté la commande dans le shell

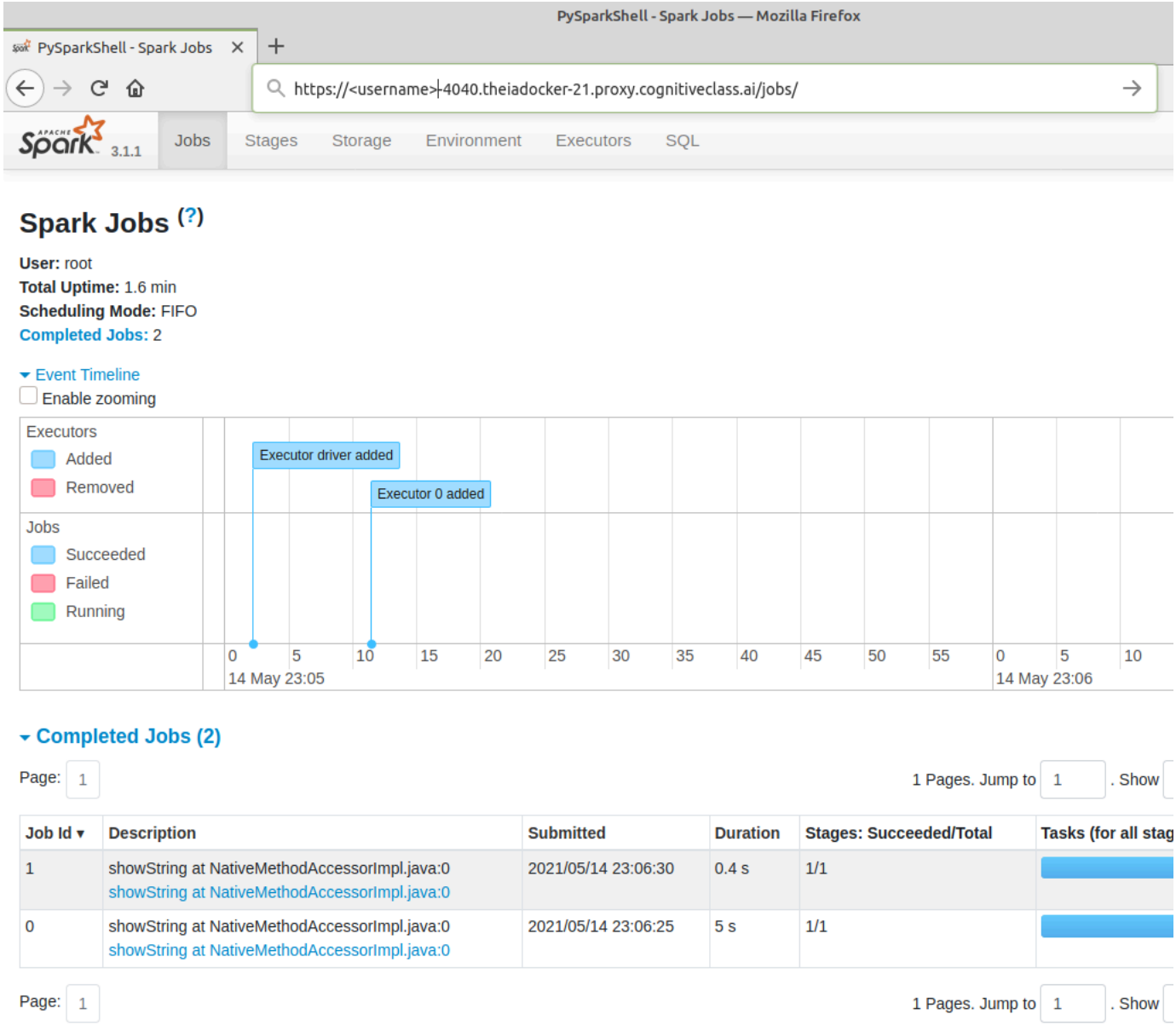
   ```
   df = spark.read.csv("/home/root/cars.csv", header=True, inferSchema=True) \
   ```

```
    .repartition(32) \
    .cache()
df.show()
```

3. Cliquez sur le bouton Skills Network à gauche, cela ouvrira la « Skills Network Toolbox ». Cliquez ensuite `OTHER`sur `Launch Application`. À partir de là, vous devriez pouvoir saisir le numéro de port `4040`et lancer l'interface utilisateur de l'application Spark dans votre navigateur.



4. Vérifiez que vous pouvez voir la page des offres d'emploi qui devrait ressembler à ce qui suit, bien que pas nécessairement exactement la même :



Exercice 2 : Exécuter une requête SQL et déboguer dans l'interface utilisateur de l'application

Dans cet exercice, vous allez définir une fonction définie par l'utilisateur (UDF) et exécuter une requête qui génère une erreur. Nous localiserons cette erreur dans l'interface utilisateur de l'application et en trouverons la cause première. Enfin, nous corrigerons l'erreur et réexécuterons la requête.

Tâche A : Exécuter une requête SQL

1. Définissez un UDF pour afficher le type de moteur. Copiez et collez le code et cliquez sur `Enter`.

```
from pyspark.sql.functions import udf
import time
@udf("string")
def engine(cylinders):
    time.sleep(0.2)  # Intentionally delay task
    eng = {6: "V6", 8: "V8"}
    return eng[cylinders]
```

2. Ajoutez l'UDF en tant que colonne dans le DataFrame

```
df = df.withColumn("engine", engine("cylinders"))
```

3. Regrouper le DataFrame par « cylindres » et agréger les autres colonnes

```
dfg = df.groupby("cylinders")
```

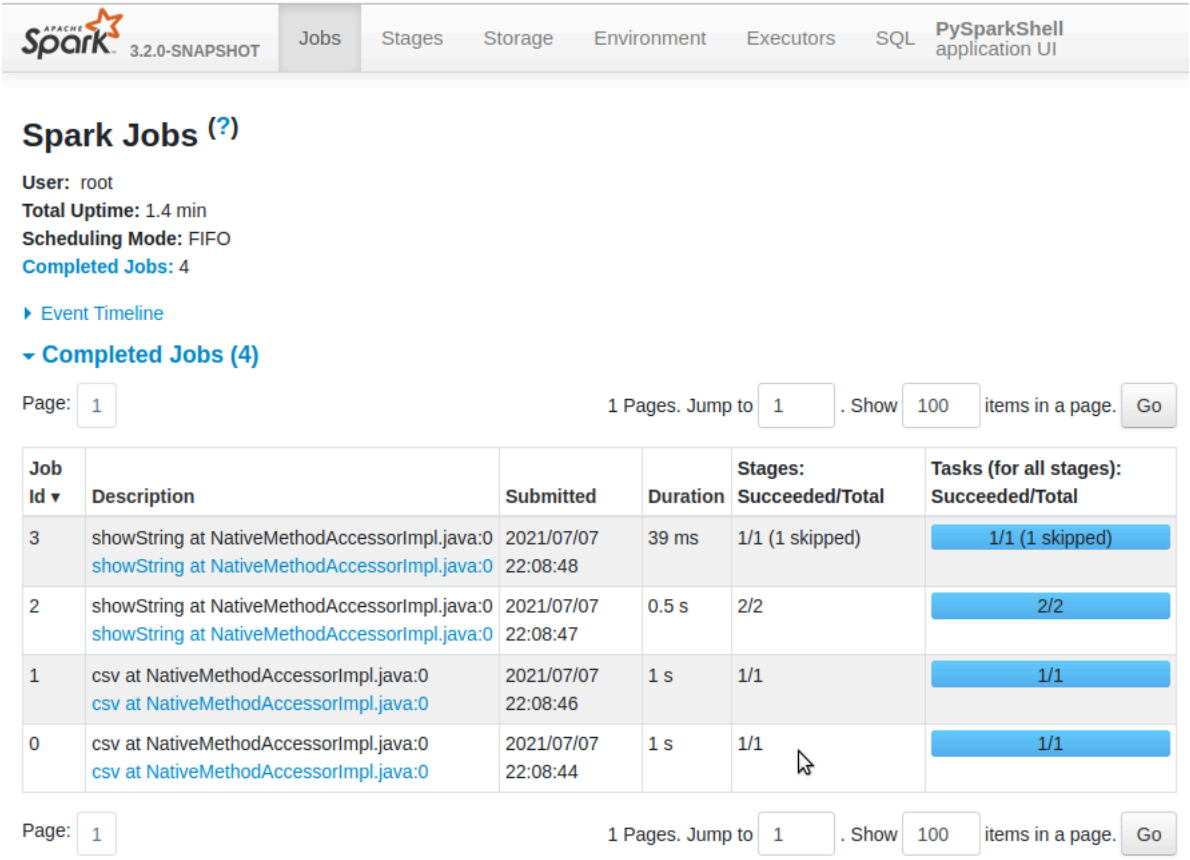
```
dfa = dfg.agg({"mpg": "avg", "engine": "first"})
```

```
dfa.show()
```

4. La requête a échoué et vous devriez voir de nombreux messages et résultats dans la console.
La tâche suivante consiste à localiser l'erreur dans l'interface utilisateur de l'application et à déterminer la cause première.

Tâche B : Déboguer l'erreur dans l'interface utilisateur de l'application

1. Recherchez l'erreur dans l'interface utilisateur de l'application.
   Ouvrez l'interface utilisateur des tâches, regardez la liste des tâches ayant échoué, cliquez sur la première tâche.



2. Cela affichera les
   détails du travail avec une liste des étapes pour ce travail. Dans la liste des étapes ayant échoué, cliquez sur la première étape ayant échoué pour afficher les détails de l'étape avec une liste des tâches pour cette étape.

	+details							
2	showString at NativeMethodAccessorImpl.java:0 +details	2021/07/07 22:08:47	0.2 s	1/1	20.6 KiB			23.4 KiB
1	csv at NativeMethodAccessorImpl.java:0 +details	2021/07/07 22:08:46	1 s	1/1	20.6 KiB			
0	csv at NativeMethodAccessorImpl.java:0 +details	2021/07/07 22:08:44	1 s	1/1	20.6 KiB			

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Skipped Stages (2)

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	showString at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/1				
4	showString at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/1				

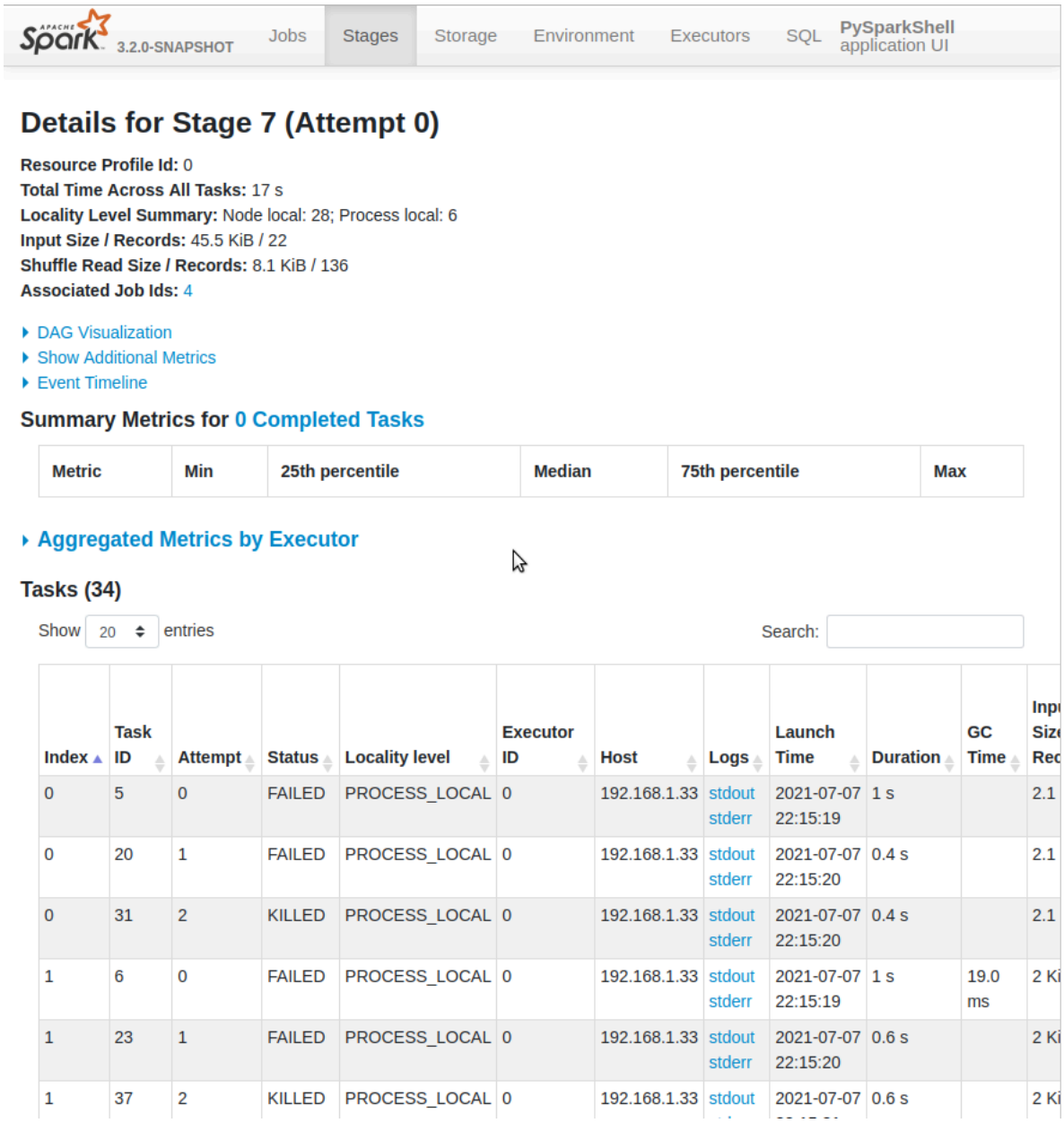
Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Failed Stages (1)

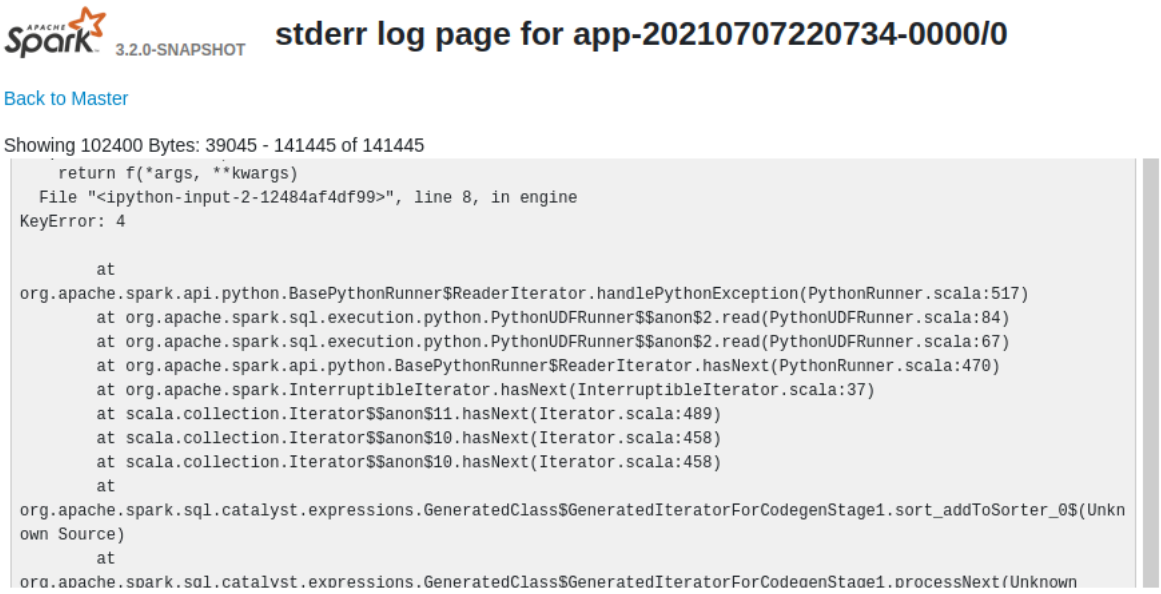
Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
7	showString at NativeMethodAccessorImpl.java:0 +details	2021/07/07 22:15:19	2 s	0/32 (28 failed) (6	45.5 KiB		8.1 KiB		Job aborted due to stage 7.0 failed 4 t Lost task 3.3 in sta (192.168.1.33 exe org.apache.spark.a Traceback (most r

3. Ici, nous voyons de nombreuses
tâches échouées. En regardant la première, la colonne la plus à droite montre les détails de l'échec.



Cliquez pour développer les détails.



Faites défiler un peu vers le bas jusqu'à ce que vous puissiez voir la dernière partie de l'erreur Python qui montre la cause. Vous devriez pouvoir voir que cela a été causé par une KeyError dans notre UDF `engine()`.

Vous pouvez également visualiser ces erreurs en regardant la colonne qui contient des liens vers les journaux et en cliquant sur « std err » pour afficher le journal des erreurs standard.

Fermez l'onglet du navigateur PySpark.

4. Dans le terminal Theia, corrigez l'UDF en ajoutant une entrée au dictionnaire des types de moteurs et fournissez une valeur par défaut pour tous les autres types. Copiez et collez ce code et cliquez sur `Enter`.

```
@udf("string")
def engine(cylinders):
    time.sleep(0.2)  # Intentionally delay task
    eng = {4: "inline-four", 6: "V6", 8: "V8"}
    return eng.get(cylinders, "other")
```

5. Relancez la requête. Vous devrez ajouter à nouveau la colonne « moteur » et saisir la requête puisque nous avons modifié l'UDF.

```
df = df.withColumn("engine", engine("cylinders"))

dfg = df.groupby("cylinders")

dfa = dfg.agg({"mpg": "avg", "engine": "first"})

dfa.show()
```

Une fois la requête terminée sans erreur, vous devriez voir un résultat similaire à celui-ci.

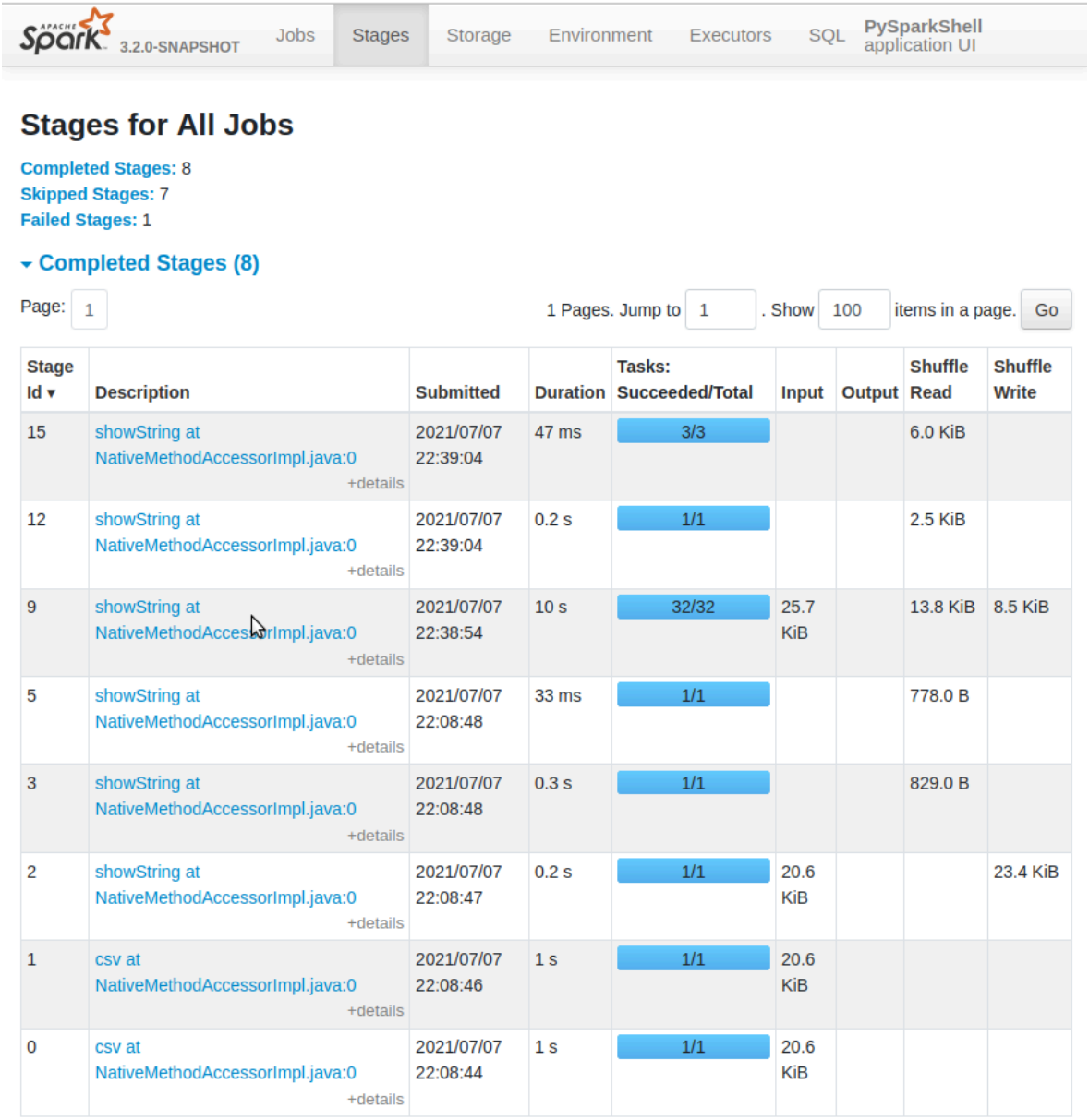
```
+---------+-----------------+-------------+
|cylinders|          avg(mpg)|first(engine)|
+---------+-----------------+-------------+
|        6|19.985714285714288|           V6|
|        3|            20.55|        other|
|        5|27.366666666666664|        other|
|        4|29.286764705882348|  inline-four|
|        8|14.963106796116506|           V8|
+---------+-----------------+-------------+
```

Exercice 3 : Surveiller les performances des applications avec l'interface utilisateur

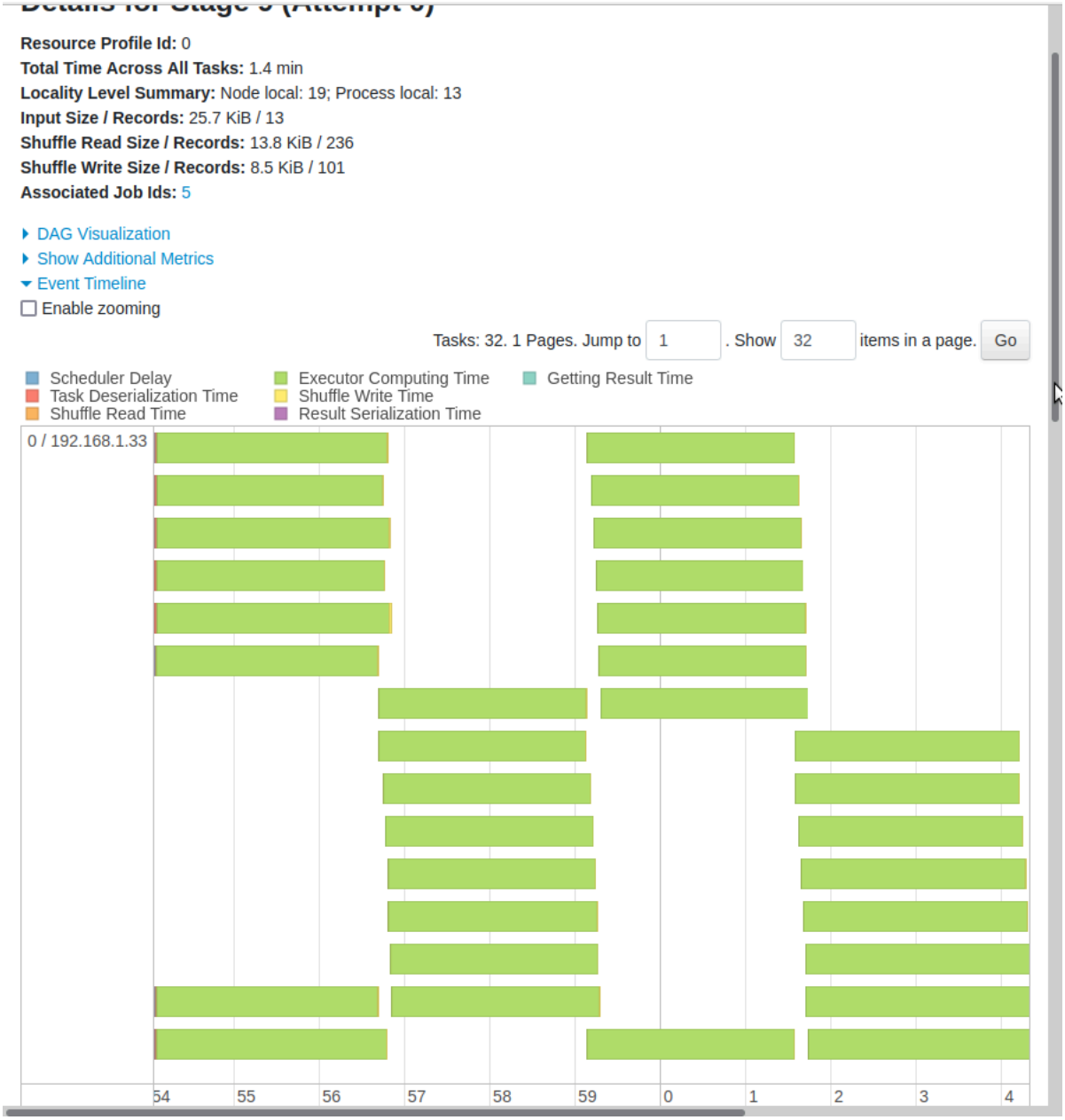
Maintenant que nous avons exécuté notre requête avec succès, nous allons faire évoluer notre application en ajoutant un worker au cluster. Cela permettra au cluster d'exécuter davantage de tâches en parallèle et d'améliorer les performances globales.

Tâche A : Ajouter un travailleur au cluster

1. Affichez l'onglet Étapes, puis cliquez sur l'étape comportant 32 tâches. À ce stade, notre UDF est appliqué à chaque partition du DataFrame.



En regardant la chronologie, vous pouvez voir qu'il existe un seul travailleur avec un identifiant `0 / <ip-address>`qui peut exécuter jusqu'à un certain nombre de tâches en parallèle à la fois. L'ajout d'un autre travailleur permettra d'exécuter des tâches supplémentaires en parallèle.



2. Ouvrez un nouveau terminal Theia en cliquant sur l'élément de menu `Terminal -> New Terminal`.

3. Ajoutez un deuxième travailleur au cluster avec la commande dans le nouveau terminal :

```
docker run \
    --name spark-worker-2 \
    --link spark-master:spark-master \
    -e ENABLE_INIT_DAEMON=false \
    -p 8082:8082 \
    -d bde2020/spark-worker:3.1.1-hadoop3.2
```

4. Si la commande réussit, il y aura une sortie unique indiquant l'ID du conteneur :

```
theia@theiadocker-user:/home/project$ docker run \
>     --name spark-worker-2 \
>     --link spark-master:spark-master \
>     -e ENABLE_INIT_DAEMON=false \
>     -p 8082:8082 \
>     -d bde2020/spark-worker:3.1.1-hadoop3.2
1935a71827668ae3476e6a16f0bebcd4c2a342a21271dc22be487aa1b1731708
theia@theiadocker-user:/home/project$
```

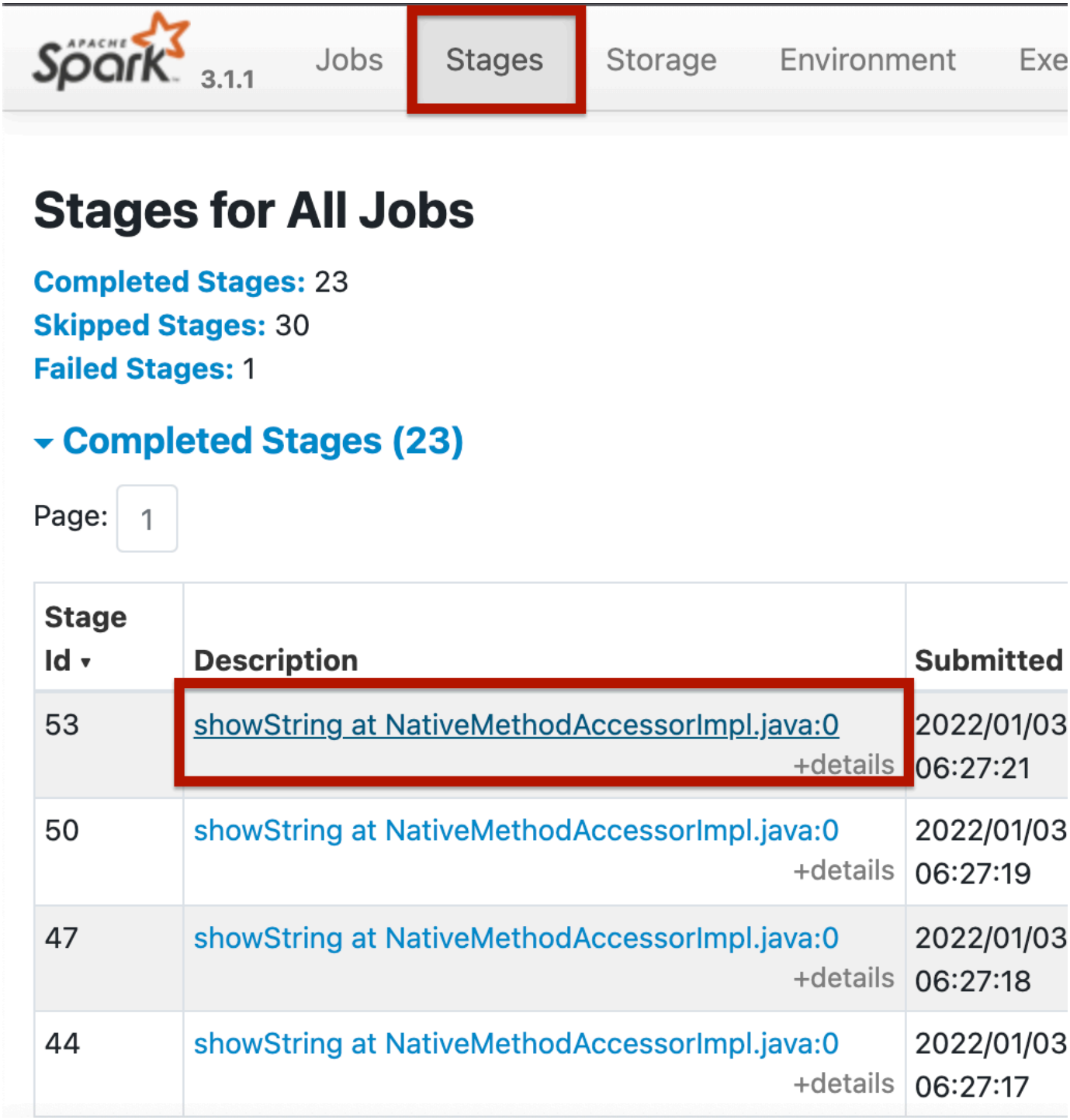
5. Cliquez sur le premier terminal sur lequel le shell PySpark est ouvert pour continuer.

Tâche B : Réexécuter la requête et vérifier les performances

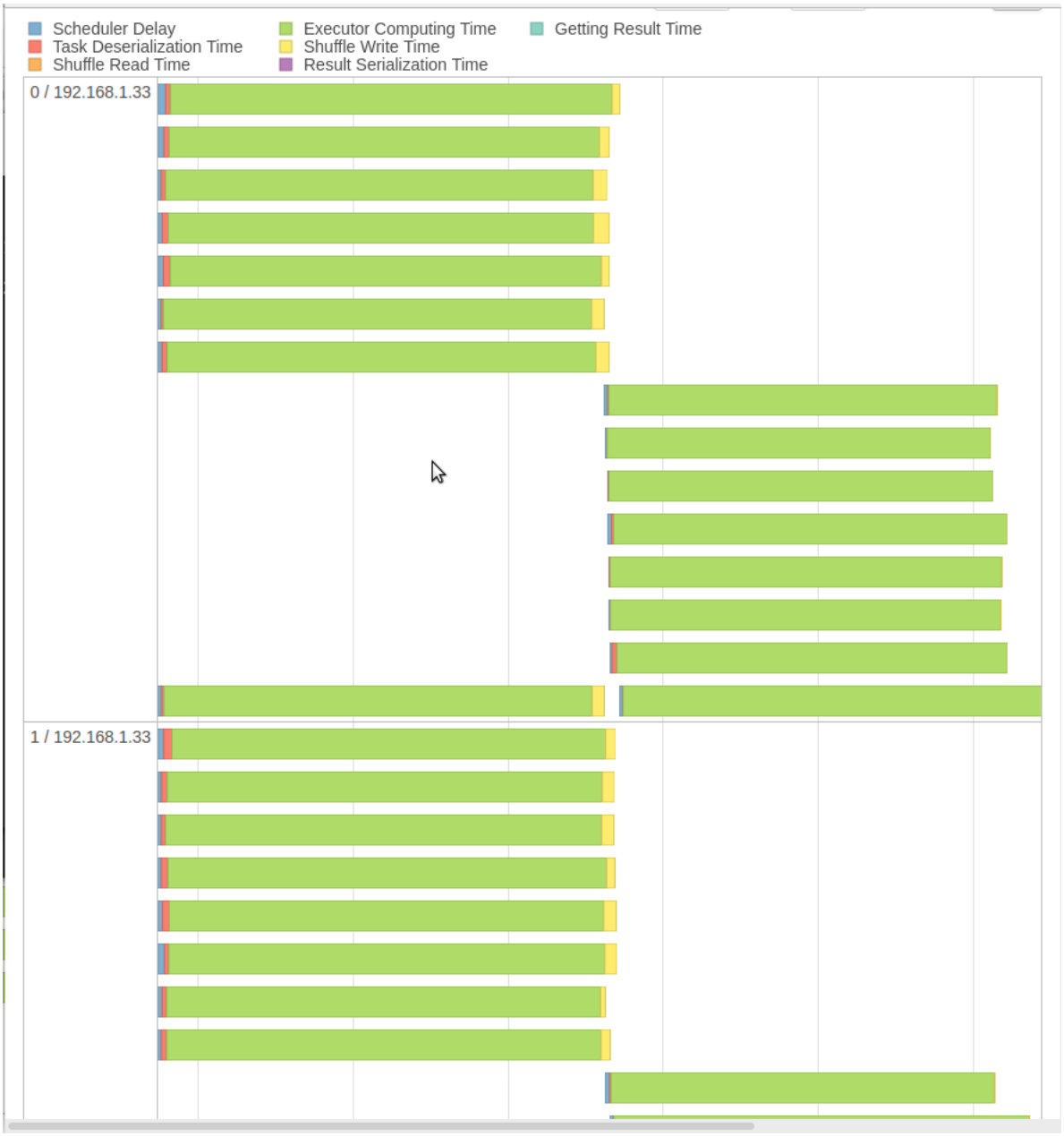
1. Réexécutez la requête, cette fois nous pouvons simplement appeler `show()`à nouveau :

```
dfa.show()
```

2. Lancez l'application sur le numéro de port `4040`en suivant le même processus que ci-dessus, pour ouvrir le navigateur PySpark. Accédez à l' onglet **Étapes** et consultez l'ID de l'étape la plus récente.



3. Vous verrez que le travailleur supplémentaire avec l'ID `1 / <ip-address>`est répertorié et permet désormais d'exécuter davantage de tâches en parallèle. La chronologie des tâches doit ressembler à ce qui suit.



Auteur(s)

Aije

Autre(s) contributeur(s)

Lavande

© IBM Corporation 2021. Tous droits réservés.