OpenRules and Apache Spark Integration

OpenRules, Inc.

www.openrules.com

Table of Contents

INTRODUCTION

This manual explains how to integrate OpenRules and Apache Spark to efficiently execute hundreds of millions of complex rules-based transactions.

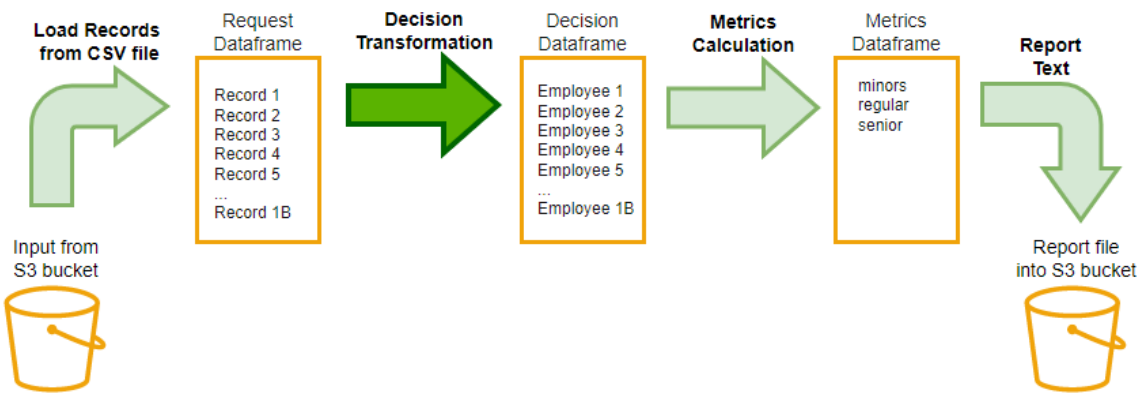
- **OpenRules® Decision Manager** helps enterprises develop and maintain operational decision services that can be deployed on-premise or on the cloud and invoked from any decision-making business applications. OpenRules helps subject matter experts to produce superfast decision services with very complex business logic.
- **Apache® Spark™** is an open-source, distributed processing system used for big data workloads. open-source cluster computing framework with in-memory processing to speed up analytic applications. Today Apache Spark is the most popular engine for scalable computing used by thousands of companies, including 80% of the Fortune 500.

OpenRules-based decision services can be integrated with an Apache Spark application that provides high scalability using cluster deployment and highly efficient parallel execution.

In this manual, we will demonstrate OpenRules-Spark integration using a simple decision service "VacationDays" that calculates employees' vacation days. While this service is very fast (takes less than a millisecond per employee), but when there are too many employees (say 1 billion) the sequential execution of this service could still take hours. However, if we put this service inside a Spark application, we will show that the total execution time for **1 billion of employees is under 3 minutes averaging 1.6 million decisions per second**!

DESIGN STEPS AND PERFORMANCE RESULTS

We designed an OpenRules-Spark Integration demo within the Amazon EMR Serverless runtime environment. The following diagram shows how our demo decision service "VacationDays" works in the Spark application:



Here are the main design steps for preparing and running our demo decision service:

1. **Prepare Data and Request Dataframe**
    a. Generate 1B records (employees) in a CSV file and load it into the AWS S3 bucket
    b. Prepare Request Dataframe
2. **Decision Service Transformation**
    a. Apply OpenRules-based decision service "VacationDays" for each of 1B records
3. **Metrics Calculation**
    a. Apply Spark analytics to all 1B employees with already calculated vacations days: we categorize all employees into 3 groups (minor, regular, senior) and calculate average, minimal, maximal, and total vacation days for each category
4. **Report Results**

The total execution time for all steps was 7 min 21 seconds. The decision service transformation took less than 3 minutes producing **6M decisions per second**! Here are the produced metrics:

```
Decision Task elapsed time : 165,800 ms
Total processed records  : 1,000,000,000
Decisions per second     : ~6,031,363

Vacation Days statistics by age groups.
+--------+-----------------+------------------+--------------+--------------+
|category|totalVacationDays|averageVacationDays|minVacationDays|maxVacationDays|
+--------+-----------------+------------------+--------------+--------------+
|minor   |1519564785       |25.33             |22            |27            |
|regular |16488653513      |22.9              |22            |27            |
|senior  |5858096754       |26.63             |24            |30            |
+--------+-----------------+------------------+--------------+--------------+
```

In the following sections, we will describe all steps in detail allowing OpenRules users to reproduce the same demo included in the standard OpenRulesSamples workspace using their own Apache Spark environment.

Creating OpenRules-based Decision Service

The standard installation workspace "OpenRules Decision Manager" comes with a sample decision model "VacationDays" that calculates vacation days for an employee using various business rules. You may build, test, and deploy this decision model as a regular jar file using the standard "package.bat" (>mvn install). You may invoke this model using a simple Java API with one employee:

```
public static void main(String[] args) {
    DecisionModel model = new DecisionModelVacationDays();
    Goal goal = model.createGoal();

    Employee employee = new Employee ();
    employee.setId("Mary Grant");
    employee.setAge(46);
    employee.setService(18);
    goal.use("Employee", employee);
    goal.execute();
    System.out.println("Vacation Days = " + employee.getVacationDays());
}
```

Creating a Large Dataset

While this decision service takes less than a millisecond to process 1 employee, it still will take a lot of time if we want to apply it to a large number of employees (say 1 billion) and use just a sequential execution. That's why we decided to put this service inside a Spark application and rely on its clusters with massive parallel execution. But first, we need to generate the proper large dataset. So, we created a simple Java program that uses a random generator to generate any number of Employees and save them in a CSV file. You can see its code here. We used this generator to create two test datasets, one with 100M of employees and another with 1B of employees. We uploaded these datasets to the AWS S3 Input bucket "openrules-spark-demo".

CREATING SPARK APPLICATION

Now we need to put our decision service in a Spark application. While Spark is a multi-language platform, we decided to write our Spark application in Java. You may find its complete source code at CalculateVacationDays.java. We will explain the key pieces of this application.

First, we need to set up a Spark context:

```
final SparkSession spark = SparkSession.builder()
        .appName("VacationDays")
        .getOrCreate();
```

Next, we need to read the data into a dataset:

```
final Dataset<Row> dfCsv = spark.read()
        .format("csv")
        .option("header", "true")
        .load(inputFile);
```

The decision service "VacationDays" expects a Java object of class Employee as its input. When our Spark application receives a collection of employees from the CSV file, it will need to convert each dataset records onto an Employee object. This will be done by our first Spark transformation function "**convert**":

```
// convert Row to Employee
final MapFunction<Row,Employee> convert = r -> {
    Employee e = new Employee();
    e.setId(r.getAs("Id"));
    e.setAge(Integer.parseInt(r.getAs("Age")));
    e.setService(Integer.parseInt(r.getAs("Service")));
    return e;
};
```

Then we will need to invoke the decision model to calculate vacation days for each employee. It can be done similarly to the above Java API. This can be done by our second Spark transformation function "**decisionTask**".

```
// decision model runner
final MapFunction<Employee,Employee> decisionTask = e -> {
    final Goal goal = new VacationDaysModel().createGoal();
    goal.use("Employee", e);
    goal.execute();
    return (Employee) goal.take("Employee");
};
```

Now we can apply these two transformations to the dataset. The following code will calculate vacation days for each employee and capture how much time it took to do the calculations.

```
        // run decisions
        long start = System.currentTimeMillis();

        Dataset<Employee> dfEmp = dfCsv
                .map( convert , Encoders.bean(Employee.class) )
                .map( decisionTask, Encoders.bean(Employee.class));

        long count = dfEmp.count();

        long elapsed = System.currentTimeMillis() - start;
```

Then we can apply any analytics supported by Spark to the results. In this demo, we decided to group the dataset into 3 age categories (minor, regular, and senior) and calculate metrics such as average, minimal, maximal, and total vacation days for each category:

```
// analytics
Dataset<Row> aggregated = dfEmp
        .withColumn("category",
                expr("case when age <= 18 then 'minor' when age >= 55 then 'senior' else 'regular' end"))
        .groupBy("category")
        .agg(
                sum("vacationDays").as("totalVacationDays"),
                round(avg("vacationDays"), 2).as("averageVacationDays"),
                min("vacationDays").as("minVacationDays"),
                max("vacationDays").as("maxVacationDays"))
        .orderBy("category");
String result = aggregated.showString(3, 0, false);
```

And finally, we may create a simple report and store it in a text file:

```
// statistics
long tps = (count * 1000) / elapsed;

reportFile.ifPresent(report -> {
    StringBuilder sb = new StringBuilder();
    sb.append(String.format(" Decision Task elapsed time : %,d ms\n", elapsed));
    sb.append(String.format(" Total processed records  : %,d\n", count));
    sb.append(String.format(" Decisions per second     : ~%,d\n", tps));
    sb.append("\n");
    sb.append(String.format(" Vacation Days statistics by age groups.\n"));
    sb.append(result);
    spark.createDataset(Arrays.asList(sb.toString()), Encoders.STRING())
            .write()
            .mode(SaveMode.Overwrite)
            .text(report);
});
```

Configuring and Packaging

We use Maven to configure and build our Spark application. The project's "pom.xml" file is a standard Maven project file, but there are some technical details worth mentioning. To deploy

our application to a cluster as a Spark job, we need to create a single jar that contains our code and all required dependencies, except Spark libraries, as they are provided by the cluster environment. We mark Spark dependency as "**provided**"

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_${scala.version}</artifactId>
    <version>${spark.version}</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
        </exclusion>
    </exclusions>
    <scope>provided</scope>
</dependency>
```

Also, we exclude not used class packages from the final jar to reduce the size of the jar.

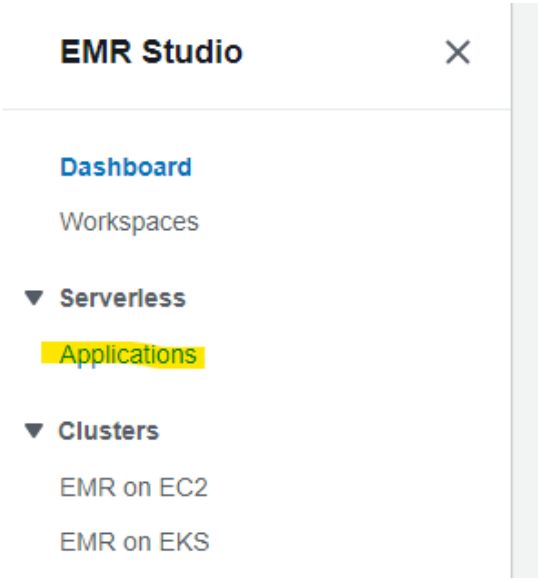
```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>3.4.1</version>
    <executions>
      <execution>
        <phase>package</phase>
        <goals>
          <goal>shade</goal>
        </goals>
        <configuration>
          <minimizeJAR>true</minimizeJAR>
          <artifactSet>
            <excludes>
              <exclude>com.fasterxml.jackson.core:*</exclude>
              <exclude>org.apache.logging.log4j:*</exclude>
              <exclude>org.fusesource.jansi:*</exclude>
            </excludes>
          </artifactSet>
          <transformers>
            <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
              <manifestEntries>
                <Main-Class>vacation.days.spark.CalculateVacationDays</Main-Class>
                <Version>${project.version}</Version>
              </manifestEntries>
            </transformer>
          </transformers>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
```

When we run the command ">**mvn package"**, it should build a JAR file **VacationDaysSpark-1.0.0.jar** in the folder "target"**.** After that, we are ready to deploy our application to a Spark cluster and perform a simple benchmarking to measure the performance of the OpenRules

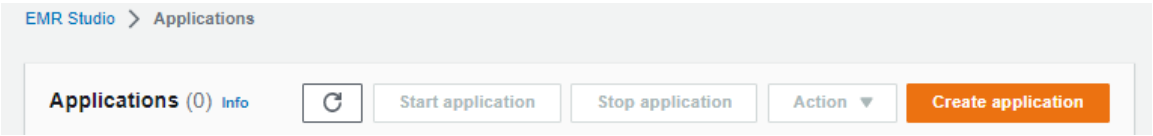
decision model running as a Spark transformation step. To make the created jar file VacationDaysSpark-1.0.0.jar available for a Spark cluster. For this reason, we upload it into the same bucket "**openrules-spark-demo**" where we placed our large CSV file with test employees.

Creating Spark Cluster and Running Application

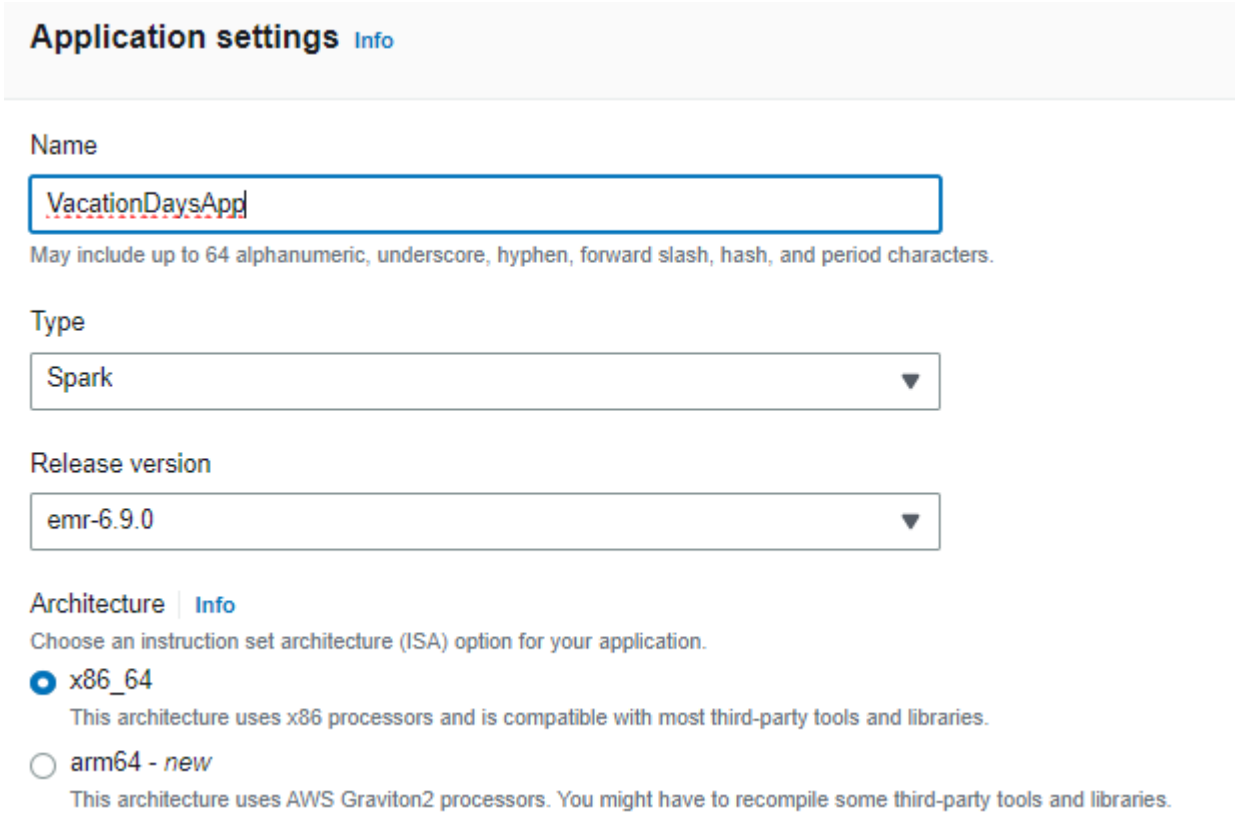
The simplest way to configure and run a Spark cluster is to use Amazon EMR (Elastic Map Reduce) service. We used it to create an EMR Serverless application and submit an EMR job with our jar file. First, we created EMR Studio and then selected the link "**Applications":**



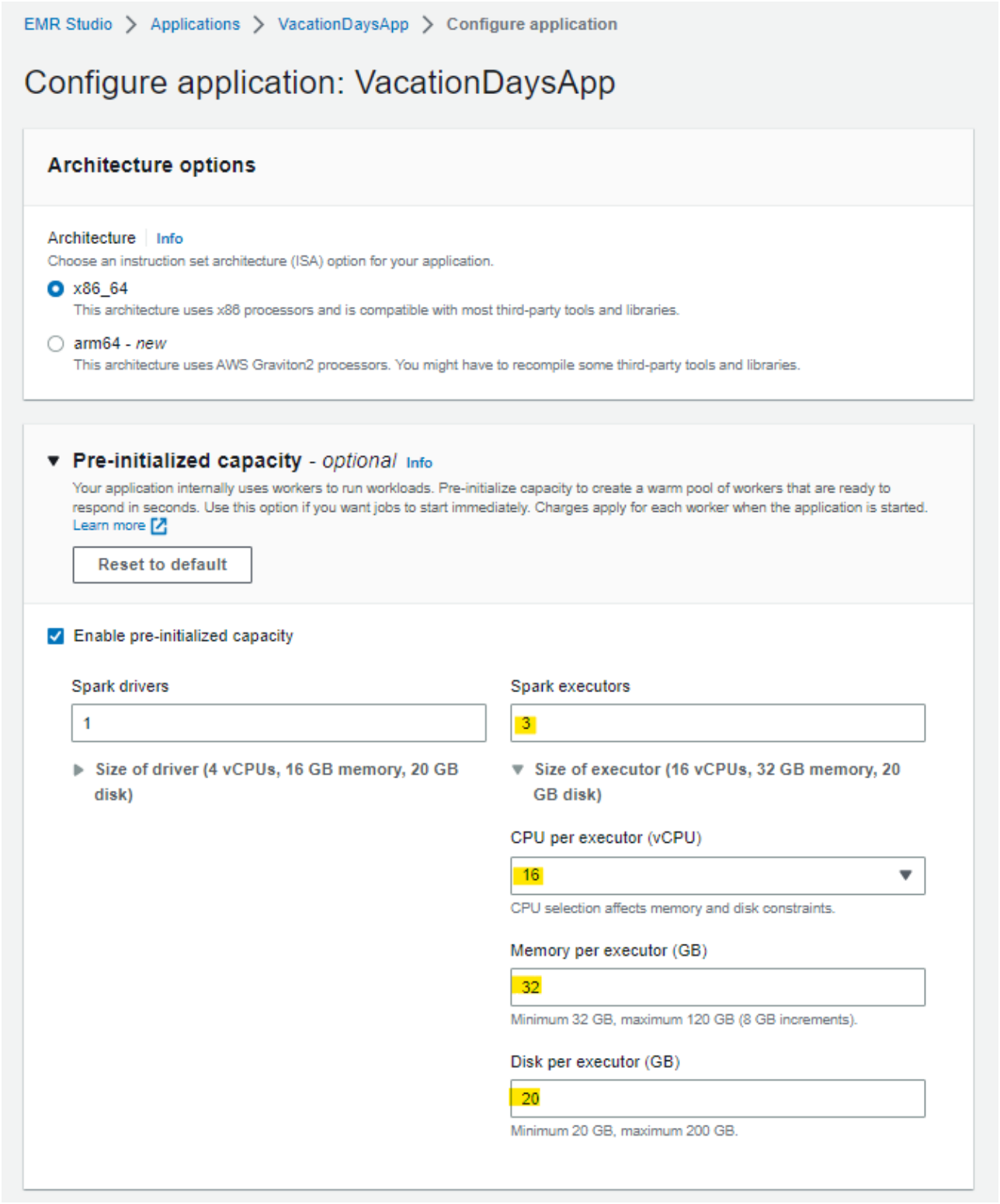
After a click on the button "**Create application**" we received:



Then we filled in an application name, software version, and hardware type:

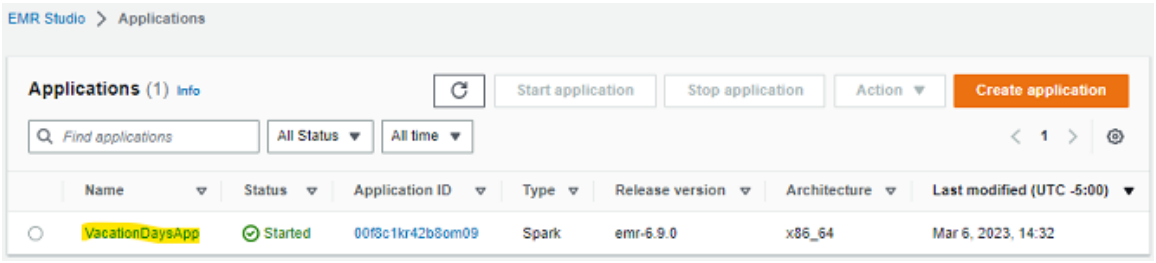


To process a large number of records (100M or even 1B) our cluster should have enough power. So, we selected "**Custom settings**" and configured our cluster with 3 "warm" executors, where each executor has 16 vCPU and 32G of memory.



We left all other settings unchanged and clicked on the button: **Create application**

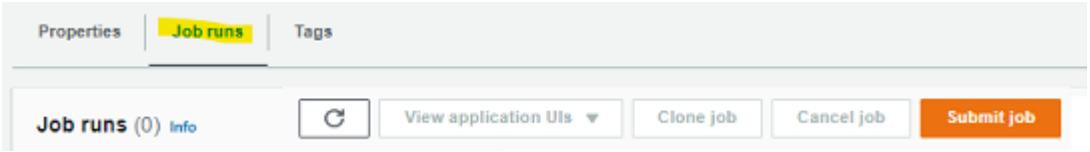
The created applications in the studio dashboard:



The application is created but not yet started. We clicked on the application name (highlighted above) and then hit the button “Start application”:

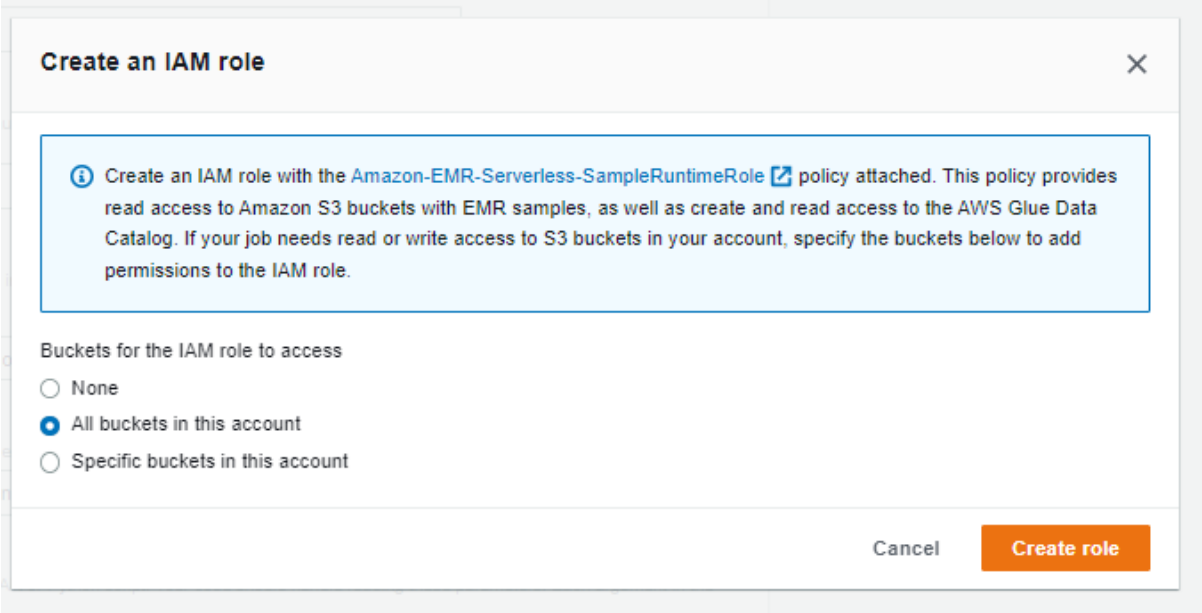
EMR Service took about 2-3 min to provision and start our Spark cluster.

While the application was starting, we configured our benchmarking job by selecting the “Job runs” tab and then clicking on the button “**Create job**”:



Then we filled in all details about the job “**CalculateVacationDays”.**

The next step was a creation of a new execution role:



Then we specified the location of our jar file (already uploaded to our S3 bucket) and the main class name. Our code accepts 3 arguments:

- **input_file**

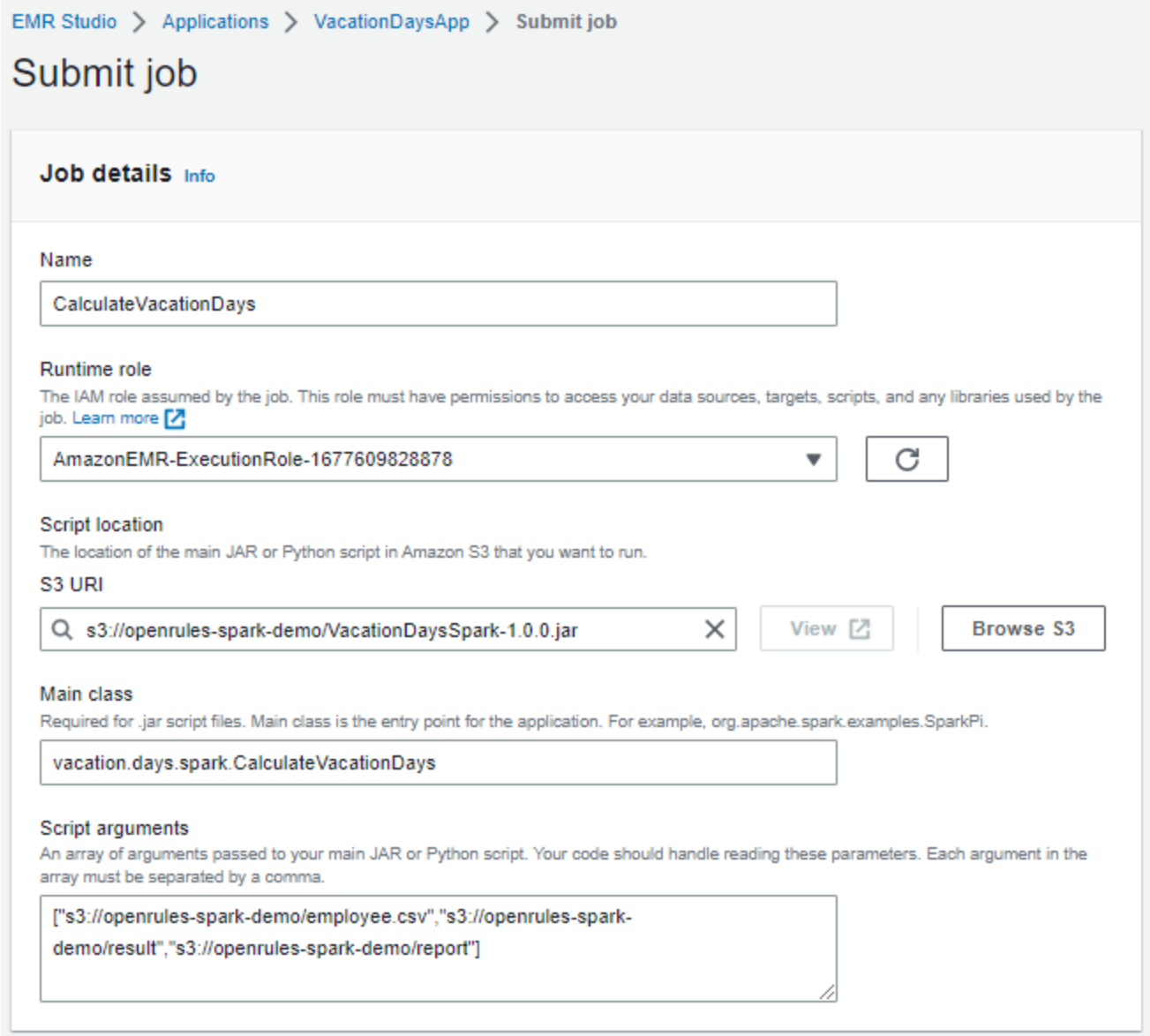
- output_folder

- report_folder.

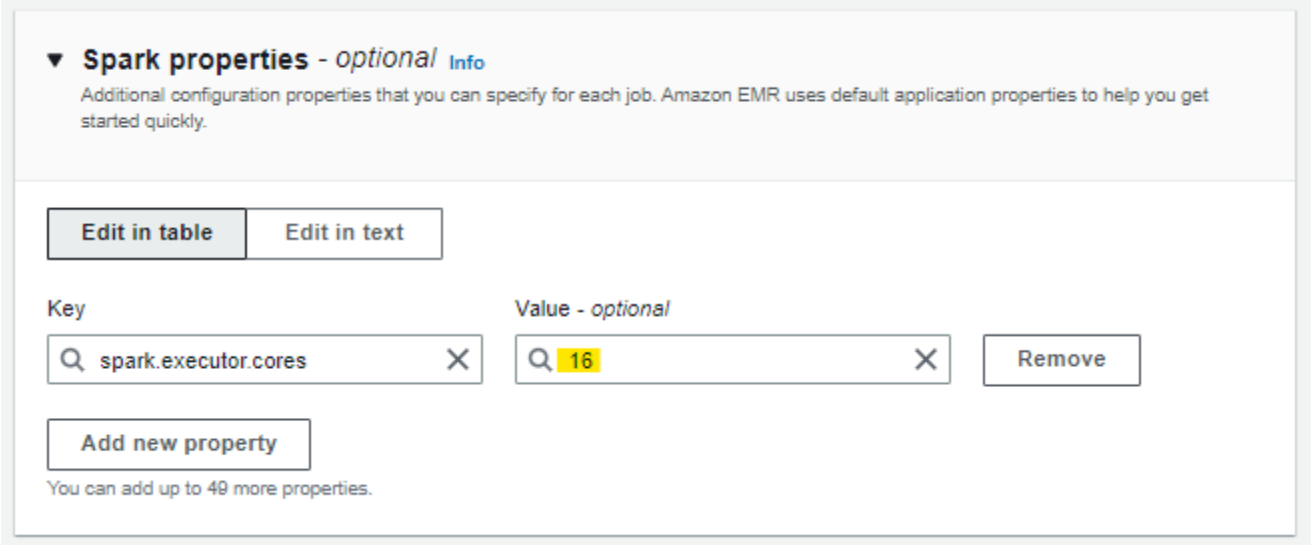
Script location: **s3://openrules-spark-demo/VacationDaysSpark-1.0.0.jar**

Main class: **vacation.days.spark.CalculateVacationDays**

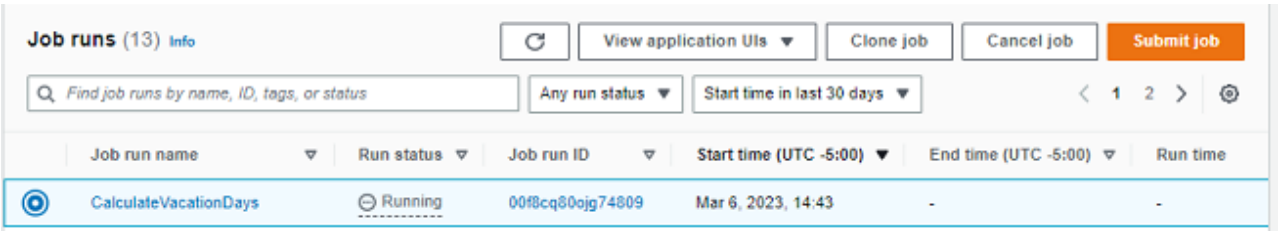
Script arguments: **["s3://openrules-spark-demo/employee.csv","s3://openrules-spark-demo/result","s3://openrules-spark-demo/report"]**



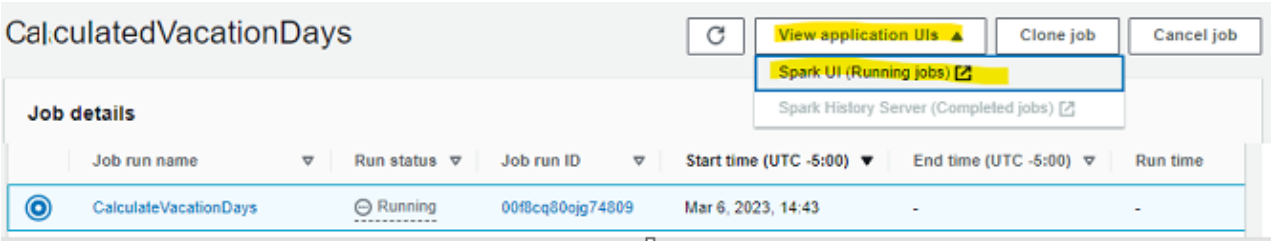
Previously we configured our Spark cluster to use 3 executors with 16 vCPUs. Now we needed to tell the framework to use all CPUs for this job. To do this, we expanded the “Spark properties” section and set the property “**spark.executor.cores**” to 16, and then hit the “Submit job” button at the bottom of the page.

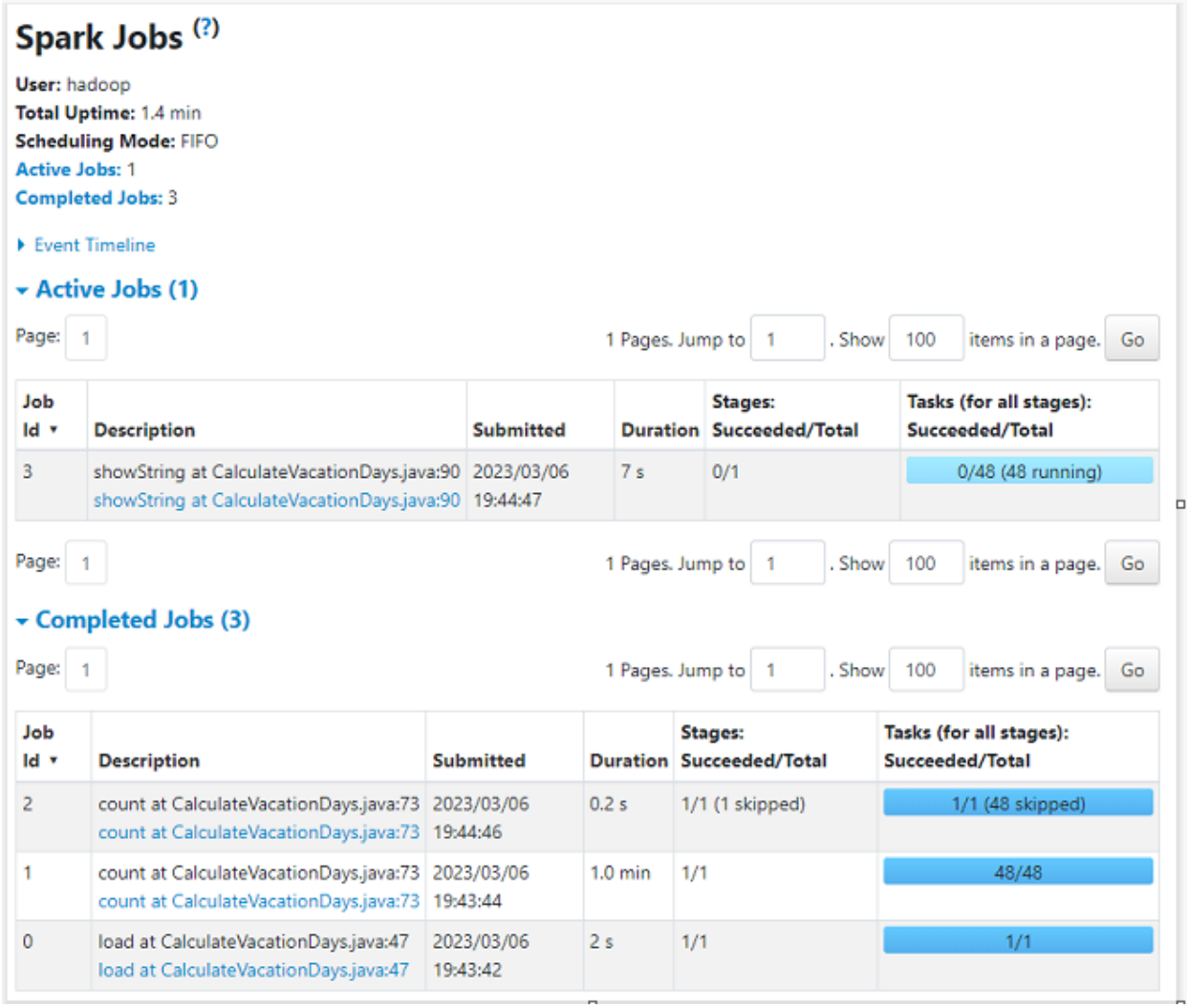


After the job is submitted, we can monitor the job execution by clicking on the job's name



Then we clicked on "View application UI" and selected "Spark UI (Running jobs)":





Spark Jobs (?)

User: hadoop
Total Uptime: 1.4 min
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 3

Event Timeline

Active Jobs (1)

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

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	showString at CalculateVacationDays.java:90 showString at CalculateVacationDays.java:90	2023/03/06 19:44:47	7 s	0/1	0/48 (48 running)

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

Completed Jobs (3)

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

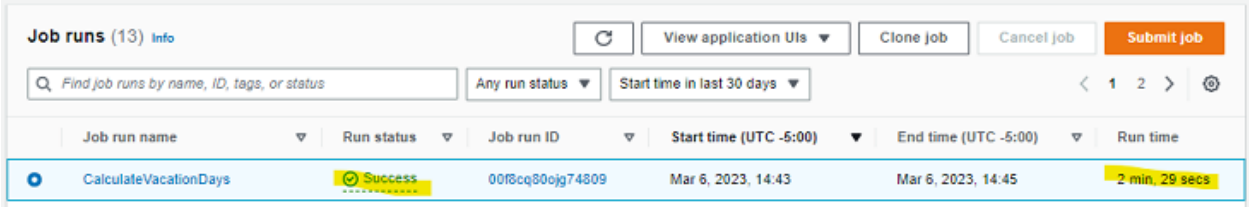
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at CalculateVacationDays.java:73 count at CalculateVacationDays.java:73	2023/03/06 19:44:46	0.2 s	1/1 (1 skipped)	1/1 (48 skipped)
1	count at CalculateVacationDays.java:73 count at CalculateVacationDays.java:73	2023/03/06 19:43:44	1.0 min	1/1	48/48
0	load at CalculateVacationDays.java:47 load at CalculateVacationDays.java:47	2023/03/06 19:43:42	2 s	1/1	1/1

Completed Stages (1)

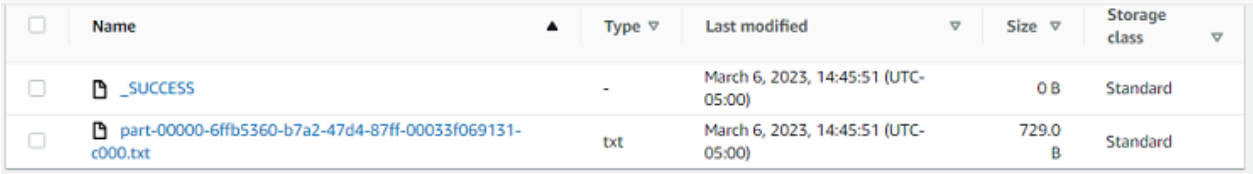
Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at CalculateVacationDays.java:73 +details	2023/03/06 19:43:44	1.0 min	48/48	2.7 GiB			2.3 KiB

We refreshed the job list and saw that the job was completed and “Run status” was “Success”.



Then we opened the **openrules-spark-demo** S3 bucket and saw that there was a folder named '**report'** which contained the file **_SUCCESS** and text file **part-00000-6ffb5360-b7a2-47d4-87ff-00033f069131-c000.txt** that contains the jobs report data.



We downloaded this text file with the execution results.

Performance Metrics

Performance Results for 100M Records

Here is what we saw in the downloaded text file:

```
Decision Task elapsed time : 62,481 ms
Total processed records  : 100,000,000
Decisions per second     : ~1,600,486

Vacation Days statistics by age groups.
+--------+-----------------+-------------------+---------------+---------------+
|category|totalVacationDays|averageVacationDays|minVacationDays|maxVacationDays|
+--------+-----------------+-------------------+---------------+---------------+
|minor   |152015650        |25.33              |22             |27             |
|regular |1648943535       |22.9               |22             |27             |
|senior  |585711171        |26.63              |24             |30             |
+--------+-----------------+-------------------+---------------+---------------+
```

These results look very good:

1. **100,000,000** records from a CSV file were loaded into the Spark cluster.
2. Our VacationDays decision service was executed for all 100M records within **62,481 milliseconds** (a bit more than **1 minute**).
3. The Spark cluster's workers, running in parallel, executed the OpenRules rule engine **averaging 1,6M decisions per second.**
4. The results were grouped by age categories and for each category, we calculated various metrics.
5. The entire execution cycle took **2 min 29 secs.**

Performance Results for 1B Records

Then we uploaded another sample CSV file with 1B records, its size is about **28Gb**.

The cluster completed the entire job processing 1B records in 7 min 21 secs.

Here are the execution results:

```
Decision Task elapsed time : 165,800 ms
Total processed records  : 1,000,000,000
Decisions per second     : ~6,031,363

Vacation Days statistics by age groups.
+--------+-----------------+-------------------+---------------+---------------+
|category|totalVacationDays|averageVacationDays|minVacationDays|maxVacationDays|
+--------+-----------------+-------------------+---------------+---------------+
|minor   |1519564785       |25.33              |22             |27             |
|regular |16488653513      |22.9               |22             |27             |
|senior  |5858096754       |26.63              |24             |30             |
+--------+-----------------+-------------------+---------------+---------------+
```

1. **1,000,000,000** records from a CSV file were loaded into the Spark cluster.
2. Our VacationDays decision service was executed for all 1B records within **165,800 milliseconds** (**less than 3 minutes**).
3. The Spark cluster's workers, running in parallel, executed the OpenRules rule engine **averaging 6M decisions per second.**
4. The results were grouped by age categories and for each category, we calculated various metrics.
5. The entire execution cycle took **7 min 21 secs.**

CONCLUSION

This tutorial provides step-by-step instructions for how to integrate OpenRules-based Decision Service in the Apache Spark application to achieve maximal performance for huge datasets. OpenRules-Spark integration provides high scalability using cluster deployment and highly efficient parallel execution.

We used a simple "VacationDays" that calculates employees' vacation days as a sample. We put this service inside a Spark application without any changes, and it was able to handle **1 billion records in under 3 minutes averaging 1.6 million decisions per second**! Any OpenRules-based decision service can be similarly deployed as a Spark application with superfast performance and minimal configuration efforts.

If you have any questions about OpenRules-Spark integration, direct all your technical questions to support@openrules.com.