..

Migrating Spark to CDP Private Cloud

Date published: 2022-07-29
Date modified: 2023-01-25

CLOUDERA

Legal Notice

Contents

Migrating Spark workloads to CDP

Migrating Spark workloads from CDH or HDP to CDP involves learning the Spark semantic changes in your source cluster and the CDP target cluster. You get details about how to handle these changes.

Spark 1.6 to Spark 2.4 Refactoring

Because Spark 1.6 is not supported on CDP, you need to refactor Spark workloads from Spark 1.6 on CDH or HDP to Spark 2.4 on CDP.

This document helps in accelerating the migration process, provides guidance to refactor Spark workloads and lists migration. Use this document when the platform is migrated from CDH or HDP to CDP.

Handling prerequisites

You must perform a number of tasks before refactoring workloads.

About this task
Assuming all workloads are in working condition, you perform this task to meet refactoring prerequistes.

Procedure

1. Identify all the workloads in the cluster (CDH/HDP) which are running on Spark 1.6 - 2.3.
2. Classify the workloads.

   Classification of workloads will help in clean-up of the unwanted workloads, plan resources and efforts for workload migration and post upgrade testing.

   Example workload classifications:

   - Spark Core (scala)
   - Java-based Spark jobs
   - SQL, Datasets, and DataFrame
   - Structured Streaming
   - MLlib (Machine Learning)
   - PySpark (Python on Spark)
   - Batch Jobs
   - Scheduled Jobs
   - Ad-Hoc Jobs
   - Critical/Priority Jobs
   - Huge data Processing Jobs
   - Time taking jobs
   - Resource Consuming Jobs etc.
   - Failed Jobs

Identify configuration changes

3. Check the current Spark jobs configuration.

   - Spark 1.6 - 2.3 workload configurations which have dependencies on job properties like scheduler, old python packages, classpath jars and might not be compatible post migration.
   - In CDP, Capacity Scheduler is the default and recommended scheduler. Follow Fair Scheduler to Capacity Scheduler transition guide to have all the required queues configured in the CDP cluster post upgrade. If any configuration changes are required, modify the code as per the new capacity scheduler configurations.
   - For workload configurations, see the Spark History server UI http://spark_history_server:18088/history/<application_number>/environment/.

4. Identify and capture workloads having data storage locations (local and HDFS) to refactor the workloads post migration.

5. Refer to unsupported Apache Spark features, and plan refactoring accordingly.

Spark 1.6 to Spark 2.4 changes

A description of the change, the type of change, and the required refactoring provide the information you need for migrating from Spark 1.6 to Spark 2.4.

New Spark entry point SparkSession

There is a new Spark API entry point: SparkSession.

Type of change

Syntactic/Spark core

Spark 1.6

Hive Context and SQLContext, such as import SparkContext, HiveContext are supported.

Spark 2.4

SparkSession is now the entry point.

Action Required

Replace the old SQLContext and HiveContext with SparkSession. For example:

```
import org.apache.spark.sql.SparkSession
    val spark = SparkSession
    .builder()
    .appName("Spark SQL basic example")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
```

.

Dataframe API registerTempTable deprecated

The Dataframe API registerTempTable has been deprecated in Spark 2.4.

Type of change:

Syntactic/Spark core change

Spark 1.6

registerTempTable is used to create a temporary table on a Spark dataframe. For example, df.registerTempTable('tmpTable").

Spark 2.4

registerTempTable is deprecated.

Action Required

Replace registerTempTable using createOrReplaceTempView. df.createOrReplaceTempView('tmpTable').

union replaces unionAll

The dataset and DataFrame API unionAll has been deprecated and replaced by union.

Type of change: Syntactic/Spark core change

Spark 1.6

unionAll is supported.

Spark 2.4

unionAll is deprecated and replaced by union.

Action Required

Replace unionAll with union. For example:val df3 = df.unionAll(df2) with val df3 = df.union(df2)

Empty schema not supported

Writing a dataframe with an empty or nested empty schema using any file format, such as parquet, orc, json, text, or csv is not allowed.

Type of change: Syntactic/Spark core

Spark 1.6 - 2.3

Writing a dataframe with an empty or nested empty schema using any file format is allowed and will not throw an exception.

Spark 2.4

An exception is thrown when you attempt to write dataframes with empty schema. For example, if there are statements such as df.write.format("parquet").mode("overwrite").save(somePath), the following error occurs: org. apache.spark.sql.AnalysisException: Parquet data source does not support null data type.

Action Required

Make sure that DataFrame is not empty. Check whether DataFrame is empty or not as follows:

```
if (!df.isEmpty) df.write.format("parquet").mode("overwrite").save("somePath
")
```

Referencing a corrupt JSON/CSV record

In Spark 2.4, queries from raw JSON/CSV files are disallowed when the referenced columns only include the internal corrupt record column.

Type of change: Syntactic/Spark core

Spark 1.6

A query can reference a _corrupt_record column in raw JSON/CSV files.

Spark 2.4

An exception is thrown if the query is referencing _corrupt_record column in these files. For example, the following query is not allowed: spark.read.schema(schema).json(file).filter($"_corrupt_record".isNotNull).count()

Action Required

Cache or save the parsed results, and then resend the query.

```
val df = spark.read.schema(schema).json(file).cache()
    df.filter($"_corrupt_record".isNotNull).count()
```

Dataset and DataFrame API explode deprecated

Dataset and DataFrame API explode has been deprecated.

Type of change: Syntactic/Spark SQL change

Spark 1.6

Dataset and DataFrame API explode are supported.

Spark 2.4

Dataset and DataFrame API explode have been deprecated. If explode is used, for example dataframe.explode(), the following warning is thrown:

```
warning: method explode in class Dataset is deprecated: use flatMap() or sel
ect() with functions.explode() instead
```

Action Required

Use functions.explode() or flatMap (import org.apache.spark.sql.functions.explode).

CSV header and schema match

Column names of csv headers must match the schema.

Type of change: Configuration/Spark core changes

Spark 1.6 - 2.3

Column names of headers in CSV files are not checked against the against the schema of CSV data.

Spark 2.4

If columns in the CSV header and the schema have different ordering, the following exception is thrown:java.lang.Il
legalArgumentException: CSV file header does not contain the expected fields.

Action Required

Make the schema and header order match or set enforceSchema to false to prevent getting an exception. For example, read a file or directory of files in CSV format into Spark DataFrame as follows: df3 = spark.read.option("delimiter", ";").option("header", True).option("enforeSchema", False).csv(path)

The default "header" option is true and enforceSchema is False.

If enforceSchema is set to true, the specified or inferred schema will be forcibly applied to datasource files, and headers in CSV files are ignored. If enforceSchema is set to false, the schema is validated against all headers in CSV files when the header option is set to true. Field names in the schema and column names in CSV headers are checked by their positions taking into account spark.sql.caseSensitive. Although the default value is true,you should disable the enforceSchema option to prevent incorrect results.

Table properties support

Table properties are taken into consideration while creating the table.

Type of change: Configuration/Spark Core Changes

Spark 1.6 - 2.3

Parquet and ORC Hive tables are converted to Parquet or ORC by default, but table properties are ignored. For example, the compression table property is ignored:

```
CREATE TABLE t(id int) STORED AS PARQUET TBLPROPERTIES (parquet.compression
'NONE')
```

This command generates Snappy Parquet files.

Spark 2.4

Table properties are supported. For example, if no compression is required, set the TBLPROPERTIES as follows: (parquet.compression 'NONE').

This command generates uncompressed Parquet files.

Action Required

Check and set the desired TBLPROPERTIES.

Managed table location

Creating a managed table with nonempty location is not allowed.

Type of change: Property/Spark core changes

Spark 1.6 - 2.3

You can create a managed table having a nonempty location.

Spark 2.4

Creating a managed table with nonempty location is not allowed. In Spark 2.4, an error occurs when there is a write operation, such as df.write.mode(SaveMode.Overwrite).saveAsTable("testdb.testtable"). The error side-effects are the cluster is terminated while the write is in progress, a temporary network issue occurs, or the job is interrupted.

Action Required

Set spark.sql.legacy.allowCreatingManagedTableUsingNonemptyLocation to true at runtime as follows:

```
spark.conf.set("spark.sql.legacy.allowCreatingManagedTableUsingNonemptyLocation","true")
```

Write to Hive bucketed tables

Type of change: Property/Spark SQL changes

Spark 1.6

By default, you can write to Hive bucketed tables.

Spark 2.4

By default, you cannot write to Hive bucketed tables.

For example, the following code snippet writes the data into a bucketed Hive table:

```
newPartitionsDF.write.mode(SaveMode.Append).format("hive").insertInto(hive_test_db.test_bucketing)
```

The code above will throw the following error:

```
org.apache.spark.sql.AnalysisException: Output Hive table `hive_test_db`.`test_bucketing` is bucketed but Spark currently does NOT populate bucketed output which is compatible with Hive.
```

Action Required

To write to a Hive bucketed table, you must use hive.enforce.bucketing=false and hive.enforce.sorting=false to forego bucketing guarantees.

Rounding in arithmetic operations

Arithmetic operations between decimals return a rounded value, instead of NULL, if an exact representation is not possible.

Type of change: Property/Spark SQL changes

Spark 1.6

Arithmetic operations between decimals return a NULL value if an exact representation is not possible.

Spark 2.4

The following changes have been made:

• Updated rules determine the result precision and scale according to the SQL ANSI 2011.
• Rounding of the results occur when the result cannot be exactly represented with the specified precision and scale instead of returning NULL.

- A new config spark.sql.decimalOperations.allowPrecisionLoss which default to true (the new behavior) to allow users to switch back to the old behavior. For example, if your code includes import statements that resemble those below, plus arithmetic operations, such as multiplication and addition, operations are performed using dataframes.

```
from pyspark.sql.types import DecimalType
from decimal import Decimal
```

Action Required

If precision and scale are important, and your code can accept a NULL value (if exact representation is not possible due to overflow), then set the following property to false. spark.sql.decimalOperations.allowPrecisionLoss = false

Precedence of set operations

Set operations are executed by priority instead having equal precedence.

Type of change: Property/Spark SQL changes

Spark 1.6 - 2.3

If the order is not specified by parentheses, equal precedence is given to all set operations.

Spark 2.4

If the order is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION, EXCEPT or MINUS operations.

For example, if your code includes set operations, such as INTERSECT , UNION, EXCEPT or MINUS, consider refactoring.

Action Required

Change the logic according to following rule:

If the order of set operations is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION, EXCEPT or MINUS operations.

If you want the previous behavior of equal precedence then, set spark.sql.legacy.setopsPrecedence.enabled=true.

HAVING without GROUP BY

HAVING without GROUP BY is treated as a global aggregate.

Type of change: Property/Spark SQL changes

Spark 1.6 - 2,3

HAVING without GROUP BY is treated as WHERE. For example, SELECT 1 FROM range(10) HAVING true is executed as SELECT 1 FROM range(10) WHERE true, and and returns 10 rows.

Spark 2.4

HAVING without GROUP BY is treated as a global aggregate. For example, SELECT 1 FROM range(10) HAVING true returns one row, instead of 10, as in the previous version.

Action Required

Check the logic where having and group by is used. To restore previous behavior, set spark.sql.legacy.parser.havingWithoutGroupByAsWhere=true.

CSV bad record handling

How Spark treats malformations in CSV files has changed.

Type of change: Property/Spark SQL changes

Spark 1.6 - 2.3

CSV rows are considered malformed if at least one column value in the row is malformed. The CSV parser drops malformed rows in the DROPMALFORMED mode or outputs an error in the FAILFAST mode.

Spark 2.4

A CSV row is considered malformed only when it contains malformed column values requested from CSV datasource, other values are ignored.

Action Required

To restore the Spark 1.6 behavior, set spark.sql.csv.parser.columnPruning.enabled to false.

Spark 2.4 CSV example
A CSV example illustrates the CSV-handling change in Spark 2.4.

In the following CSV file, the first two records describe the file. These records are not considered during processing and need to be removed from the file. The actual data to be considered for processing has three columns (jersey, name, position).

```
These are extra line1
These are extra line2
10,Messi,CF
7,Ronaldo,LW
9,Benzema,CF
```

The following schema definition for the DataFrame reader uses the option DROPMALFORMED. You see only the required data; all the description and error records are removed.

```
schema=Structtype([Structfield("jersy",StringType()),Structfield("name",Stri
ngType()),Structfi
eld("position",StringType())])
df1=spark.read\
.option("mode","DROPMALFORMED")\
.option("delimiter",",")\
.schema(schema)\
.csv("inputfile")
df1.select("*").show()
```

Output is:

jersy	name	position
10	Messi	CF
7	Ronaldo	LW
9	Benzema	CF

Select two columns from the dataframe and invoke show():

```
df1.select("jersy","name").show(truncate=False)
```

jersy	name
These are extra line1	null
These are extra line2	null
10	Messi
7	Ronaldo
9	Benzema

Malformed records are not dropped and pushed to the first column and the remaining columns will be replaced with null.This is due to the CSV parser column pruning which is set to true by default in Spark 2.4.

Set the following conf, and run the same code, selecting two fields.

```
spark.conf.set("spark.sql.csv.parser.columnPruning.enabled",False)
```

```
df2=spark.read\
    .option("mode","DROPMALFORMED")\
    .option("delimiter",",")\
    .schema(schema)\
    .csv("inputfile")
    df2.select("jersy","name").show(truncate=False)
```

jersy	name
10	Messi
7	Ronaldo
9	Benzema

Conclusion: If working on selective columns, to handle bad records in CSV files, set spark.sql.csv.parser.columnPruning.enabled to false; otherwise, the error record is pushed to the first column, and all the remaining columns are treated as nulls.

Configuring storage locations

To execute the workloads in CDP, you must modify the references to storage locations. In CDP, references must be changed from HDFS to a cloud object store such as S3.

About this task

The following sample query shows a Spark 2.4 HDFS data location.

```
scala> spark.sql("CREATE TABLE IF NOT EXISTS default.sales_spark_2(Region st
ring, Country string,Item_Type string,Sales_Channel string,Order_Priority st
ring,Order_Date date,Order_ID int,Ship_Date date,Units_sold string,Unit_Pric
e string,Unit_cost string,Total_revenue string,Total_Cost string,Total_Profi
t string) row format delimited fields terminated by ','")
scala> spark.sql("load data local inpath '/tmp/sales.csv' into table defau
lt.sales_spark_2")
scala> spark.sql("select count(*) from default.sales_spark_2").show()
```

The following sample query shows a Spark 2.4 S3 data location.

```
scala> spark.sql("CREATE TABLE IF NOT EXISTS default.sales_spark_2(Region st
ring, Country string,Item_Type string,Sales_Channel string,Order_Priority st
ring,Order_Date date,Order_ID int,Ship_Date date,Units_sold string,Unit_Pric
e string,Unit_cost string,Total_revenue string,Total_Cost string,Total_Profi
t string) row format delimited fields terminated by ','")
scala> spark.sql("load data inpath 's3://<bucket>/sales.csv' into table de
fault.sales_spark_2")
scala> spark.sql("select count(*) from default.sales_spark_2").show()
```

Querying Hive managed tables from Spark

Hive-on-Spark is not supported on CDP. You need to use the Hive Warehouse Connector (HWC) to query Apache Hive managed tables from Apache Spark.

To read Hive external tables from Spark, you do not need HWC. Spark uses native Spark to read external tables. For more information, see the Hive Warehouse Connector documentation.

The following example shows how to query a Hive table from Spark using HWC:

```
spark-shell --jars /opt/cloudera/parcels/CDH/jars/hive-warehouse-connector-a
ssembly-1.0.0.7.1.4.0-203.jar --conf spark.sql.hive.hiveserver2.jdbc.url=jdb
c:hive2://cdhhdp02.uddeepta-bandyopadhyay-s-account.cloud:10000/default --co
nf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/cdhhdp02.uddeepta-band
yopadhyay-s-account.cloud@Uddeepta-bandyopadhyay-s-Account.CLOUD
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark).b
uild()
scala> hive.executeUpdate("UPDATE hive_acid_demo set value=25 where key=4")
scala> val result=hive.execute("select * from default.hive_acid_demo")
scala> result.show()
```

Compiling and running Spark workloads

After modifying the workloads, compile and run (or dry run) the refactored workloads on Spark 2.4.

You can write Spark applications using Java, Scala, Python, SparkR, and others. You build jars from these scripts using one of the following compilers.

- Java (with Maven/Java IDE),
- Scala (with sbt),
- Python (pip).
- SparkR (RStudio)

Compiling and running a Java-based job

You see by example how to compile a Java-based Spark job using Maven.

About this task

In this task, you see how to compile the following example Spark program written in Java:

```
/* SimpleApp.java */
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on
 your system
    SparkSession spark = SparkSession.builder().appName("Simple Applicatio
n").getOrCreate();
    Dataset<String> logData = spark.read().textFile(logFile).cache();

    long numAs = logData.filter(s -> s.contains("a")).count();
    long numBs = logData.filter(s -> s.contains("b")).count();

   System.out.println("Lines with a: " + numAs + ", lines with b: " + num
Bs);

    spark.stop();
  }
}
```

You also need to create a Maven Project Object Model (POM) file, as shown in the following example:

```
<project>
  <groupId>edu.berkeley</groupId>
  <artifactId>simple-project</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Simple Project</name>
  <packaging>jar</packaging>
```

```
  <version>1.0</version>
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
<dependencies>
    <dependency> <!-- Spark dependency -->
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.12</artifactId>
    <version>2.4.0</version>
    <scope>provided</scope>
    </dependency>
</dependencies>
</project>
```

Before you begin

- Install Apache Spark 2.4.x, JDK 8.x, and maven
- Write a Java Spark program .java file.
- Write a pom.xml file. This is where your Scala code resides.
- If the cluster is Kerberized, ensure the required security token is authorized to compile and execute the workload.

Procedure

1. Lay out these files according to the canonical Maven directory structure.
   For example:

```
$ find .
./pom.xml
./src
./src/main
./src/main/java
./src/main/java/SimpleApp.java
```

2. Package the application using maven package command.
   For example:

```
# Package a JAR containing your application
$ mvn package
...
[INFO] Building jar: {..}/{..}/target/simple-project-1.0.jar
```

   After compilation, several new files are created under new directories named project and target. Among these new files, is the jar file under the target directory to run the code. For example, the file is named simple-project-1.0.jar.

3. Execute and test the workload jar using the spark submit command.
   For example:

```
# Use spark-submit to run your application
spark-submit \
--class "SimpleApp" \
--master yarn \
target/simple-project-1.0.jar
```

Compiling and running a Scala-based job

You see by example how to use sbt software to compile a Scala-based Spark job.

About this task

In this task, you see how to use the following .sbt file that specifies the build configuration:

```
cat build.sbt
name := "Simple Project"
version := "1.0"
scalaVersion := "2.12.15"
libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.4.0"
```

You also need to create a compile the following example Spark program written in Scala:

```
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession

object SimpleApp {
  def main(args: Array[String]) {
 val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your
system
 val spark = SparkSession.builder.appName("Simple Application").getOrCrea
te()
 val logData = spark.read.textFile(logFile).cache()
 val numAs = logData.filter(line => line.contains("a")).count()
 val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
 spark.stop()
  }
```

Before you begin

- Install Apache Spark 2.4.x.
- Install JDK 8.x.
- Install Scala 2.12.
- Install Sbt 0.13.17.
- Wrtie an .sbt file for configuration specifications, similar to a C include file.
- Write a Scala-based Spark program (a .scala file).
- If the cluster is Kerberized, ensure the required security token is authorized to compile and execute the workload.

Procedure

1. Compile the code using sbt package command from the directory where the build.sbt file exists.
   For example:

```
# Your directory layout should look like this
$ find .
.
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.12/simple-project_2.12-1.0.jar
```

Several new files are created under new directories named project and target, including the jar file named simple-project_2.12-1.0.jar after the project name, Scala version, and code version.

---

2. Execute and test the workload jar using spark submit.
   For example:

```
# Use spark-submit to run your application
spark-submit \
  --class "SimpleApp" \
  --master yarn \
  target/scala-2.12/simple-project_2.12-1.0.jar
```

Running a Python-based job

You can run a Python script to execute a spark-submit or pyspark command.

About this task

In this task, you execute the following Python script that creates a table and runs a few queries:

```
/* spark-demo.py */
from pyspark import SparkContext
sc = SparkContext("local", "first app")
from pyspark.sql import HiveContext
hive_context = HiveContext(sc)
hive_context.sql("drop table default.sales_spark_2_copy")
hive_context.sql("CREATE TABLE IF NOT EXISTS default.sales_spark_2_copy as
 select * from default.sales_spark_2")
hive_context.sql("show tables").show()
hive_context.sql("select * from default.sales_spark_2_copy limit 10").show()
hive_context.sql("select count(*) from default.sales_spark_2_copy").show()
```

Before you begin

Install Python 2.7 or Python 3.5 or higher.

Procedure

1. Log into a Spark gateway node.
2. Ensure the required security token is authorized to compile and execute the workload (if your cluster is Kerberized).
3. Execute the script using the spark-submit command.

```
spark-submit spark-demo.py --num-executors 3 --driver-memory 512m --exec
utor-memory 512m --executor-cores 1
```

4. Go to the Spark History server web UI at http://<spark_history_server>:18088, and check the status and performance of the workload.

Using pyspark

About this task

Run your application with the pyspark or the Python interpreter.

Before you begin

Install PySpark using pip.

Procedure

1. Log into a Spark gateway node.
2. Ensure the required security token is authorized to compile and execute the workload (if your cluster is Kerberized).
3. Ensure the user has access to the workload script (python or shell script).

4. Execute the script using pyspark.

```
pyspark spark-demo.py --num-executors 3 --driver-memory 512m --executor-
memory 512m --executor-cores 1
```

5. Execute the script using the Python interpreter.

```
python spark-demo.py
```

6. Go to the Spark History server web UI at http://<spark_history_server>:18088, and check the status and performance of the workload.

Running a job interactively

About this task

Procedure

1. Log into a Spark gateway node.
2. Ensure the required security token is authorized to compile and execute the workload (if your cluster is Kerberized).
3. Launch the "spark-shell".
   For example:

```
 spark-shell --jars target/mylibrary-1.0-SNAPSHOT-jar-with-dependencies.
jar
```

4. Create a Spark context and run workload scripts.

```
cala> import org.apache.spark.sql.hive.HiveContext
scala> val sqlContext = new HiveContext(sc)
scala> sqlContext.sql("CREATE TABLE IF NOT EXISTS default.sales_spark_1(R
egion string, Country string,Item_Type string,Sales_Channel string,Order
_Priority string,Order_Date date,Order_ID int,Ship_Date date,Units_sold
string,Unit_Price string,Unit_cost string,Total_revenue string,Total_Cos
t string,Total_Profit string) row format delimited fields terminated by
','")
scala> sqlContext.sql("load data local inpath '/tmp/sales.csv' into table
default.sales_spark_1")
scala> sqlContext.sql("show tables")
scala> sqlContext.sql("select * from default.sales_spark_1 limit 10").s
how()
scala> sqlContext.sql ("select count(*) from default.sales_spark_1").show(
)
```

5. Go to the Spark History server web UI at http://<spark_history_server>:18088, and check the status and performance of the workload.

Post-migration tasks

After the workloads are executed on Spark 2.4, validate the output, and compare the performance of the jobs with CDH/HDP cluster executions.

After the workloads are executed on Spark 2.4, validate the output, and compare the performance of the jobs with CDH/HDP cluster executions. After you perform the post migration configurations, do benchmark testing on Spark 2.4.

Troubleshoot the failed/slow performing workloads by analyzing the application event logs/driver logs and fine tune the workloads for better performance.

For more information, see the following documents:

- https://spark.apache.org/docs/2.4.4/sql-migration-guide-upgrade.html

  https://spark.apache.org/releases/spark-release-2-4-0.html

  https://spark.apache.org/releases/spark-release-2-2-0.html

  https://spark.apache.org/releases/spark-release-2-3-0.html

  https://spark.apache.org/releases/spark-release-2-1-0.html

  https://spark.apache.org/releases/spark-release-2-0-0.html

  For additional information about known issues please also refer:

  Known Issues in Cloudera Manager 7.4.4 | CDP Private Cloud

Spark 2.3 to Spark 2.4 Refactoring

Because Spark 2.3 is not supported on CDP, you need to refactor Spark workloads from Spark 2.3 on CDH or HDP to Spark 2.4 on CDP.

This document helps in accelerating the migration process, provides guidance to refactor Spark workloads and lists migration. Use this document when the platform is migrated from CDH or HDP to CDP.

Handling prerequisites

You must perform a number of tasks before refactoring workloads.

About this task

Assuming all workloads are in working condition, you perform this task to meet refactoring prerequistes.

Procedure

1. Identify all the workloads in the cluster (CDH/HDP) which are running on Spark 1.6 - 2.3.
2. Classify the workloads.

   Classification of workloads will help in clean-up of the unwanted workloads, plan resources and efforts for workload migration and post upgrade testing.

   Example workload classifications:

   - Spark Core (scala)
   - Java-based Spark jobs
   - SQL, Datasets, and DataFrame
   - Structured Streaming
   - MLlib (Machine Learning)
   - PySpark (Python on Spark)
   - Batch Jobs
   - Scheduled Jobs
   - Ad-Hoc Jobs
   - Critical/Priority Jobs
   - Huge data Processing Jobs
   - Time taking jobs
   - Resource Consuming Jobs etc.
   - Failed Jobs

Identify configuration changes

3. Check the current Spark jobs configuration.

- Spark 1.6 - 2.3 workload configurations which have dependencies on job properties like scheduler, old python packages, classpath jars and might not be compatible post migration.
- In CDP, Capacity Scheduler is the default and recommended scheduler. Follow Fair Scheduler to Capacity Scheduler transition guide to have all the required queues configured in the CDP cluster post upgrade. If any configuration changes are required, modify the code as per the new capacity scheduler configurations.
- For workload configurations, see the Spark History server UI http://spark_history_server:18088/history/<application_number>/environment/.

4. Identify and capture workloads having data storage locations (local and HDFS) to refactor the workloads post migration.

5. Refer to unsupported Apache Spark features, and plan refactoring accordingly.

Spark 2.3 to Spark 2.4 changes

A description of the change, the type of change, and the required refactoring provide the information you need for migrating from Spark 2.3 to Spark 2.4.

Empty schema not supported

Writing a dataframe with an empty or nested empty schema using any file format, such as parquet, orc, json, text, or csv is not allowed.

Type of change: Syntactic/Spark core

Spark 1.6 - 2.3

Writing a dataframe with an empty or nested empty schema using any file format is allowed and will not throw an exception.

Spark 2.4

An exception is thrown when you attempt to write dataframes with empty schema. For example, if there are statements such as df.write.format("parquet").mode("overwrite").save(somePath), the following error occurs: org.apache.spark.sql.AnalysisException: Parquet data source does not support null data type.

Action Required

Make sure that DataFrame is not empty. Check whether DataFrame is empty or not as follows:

```
if (!df.isEmpty) df.write.format("parquet").mode("overwrite").save("somePath
")
```

CSV header and schema match

Column names of csv headers must match the schema.

Type of change: Configuration/Spark core changes

Spark 1.6 - 2.3

Column names of headers in CSV files are not checked against the against the schema of CSV data.

Spark 2.4

If columns in the CSV header and the schema have different ordering, the following exception is thrown:java.lang.IllegalArgumentException: CSV file header does not contain the expected fields.

Action Required

Make the schema and header order match or set enforceSchema to false to prevent getting an exception. For example, read a file or directory of files in CSV format into Spark DataFrame as follows: df3 = spark.read.option("delimiter", ";").option("header", True).option("enforeSchema", False).csv(path)

The default "header" option is true and enforceSchema is False.

If enforceSchema is set to true, the specified or inferred schema will be forcibly applied to datasource files, and headers in CSV files are ignored. If enforceSchema is set to false, the schema is validated against all headers in CSV files when the header option is set to true. Field names in the schema and column names in CSV headers are checked by their positions taking into account spark.sql.caseSensitive. Although the default value is true,you should disable the enforceSchema option to prevent incorrect results.

Table properties support

Table properties are taken into consideration while creating the table.

Type of change: Configuration/Spark Core Changes

Spark 1.6 - 2.3

Parquet and ORC Hive tables are converted to Parquet or ORC by default, but table properties are ignored. For example, the compression table property is ignored:

```
CREATE TABLE t(id int) STORED AS PARQUET TBLPROPERTIES (parquet.compression
'NONE')
```

This command generates Snappy Parquet files.

Spark 2.4

Table properties are supported. For example, if no compression is required, set the TBLPROPERTIES as follows: (parquet.compression 'NONE').

This command generates uncompressed Parquet files.

Action Required

Check and set the desired TBLPROPERTIES.

Managed table location

Creating a managed table with nonempty location is not allowed.

Type of change: Property/Spark core changes

Spark 1.6 - 2.3

You can create a managed table having a nonempty location.

Spark 2.4

Creating a managed table with nonempty location is not allowed. In Spark 2.4, an error occurs when there is a write operation, such as df.write.mode(SaveMode.Overwrite).saveAsTable("testdb.testtable"). The error side-effects are the cluster is terminated while the write is in progress, a temporary network issue occurs, or the job is interrupted.

Action Required

Set spark.sql.legacy.allowCreatingManagedTableUsingNonemptyLocation to true at runtime as follows:

```
spark.conf.set("spark.sql.legacy.allowCreatingManagedTableUsingNonemptyLocat
ion","true")
```

Precedence of set operations

Set operations are executed by priority instead having equal precedence.

Type of change: Property/Spark SQL changes

Spark 1.6 - 2.3

If the order is not specified by parentheses, equal precedence is given to all set operations.

Spark 2.4

If the order is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION, EXCEPT or MINUS operations.

For example, if your code includes set operations, such as INTERSECT , UNION, EXCEPT or MINUS, consider refactoring.

Action Required

Change the logic according to following rule:

If the order of set operations is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION, EXCEPT or MINUS operations.

If you want the previous behavior of equal precedence then, set spark.sql.legacy.setopsPrecedence.enabled=true.

HAVING without GROUP BY
HAVING without GROUP BY is treated as a global aggregate.

Type of change: Property/Spark SQL changes

Spark 1.6 - 2,3

HAVING without GROUP BY is treated as WHERE. For example, SELECT 1 FROM range(10) HAVING true is executed as SELECT 1 FROM range(10) WHERE true, and and returns 10 rows.

Spark 2.4

HAVING without GROUP BY is treated as a global aggregate. For example, SELECT 1 FROM range(10) HAVING true returns one row, instead of 10, as in the previous version.

Action Required

Check the logic where having and group by is used. To restore previous behavior, set spark.sql.legacy.parser.havingWithoutGroupByAsWhere=true.

CSV bad record handling
How Spark treats malformations in CSV files has changed.

Type of change: Property/Spark SQL changes

Spark 1.6 - 2.3

CSV rows are considered malformed if at least one column value in the row is malformed. The CSV parser drops malformed rows in the DROPMALFORMED mode or outputs an error in the FAILFAST mode.

Spark 2.4

A CSV row is considered malformed only when it contains malformed column values requested from CSV datasource, other values are ignored.

Action Required

To restore the Spark 1.6 behavior, set spark.sql.csv.parser.columnPruning.enabled to false.

Spark 2.4 CSV example
A CSV example illustrates the CSV-handling change in Spark 2.4.

In the following CSV file, the first two records describe the file. These records are not considered during processing and need to be removed from the file. The actual data to be considered for processing has three columns (jersey, name, position).

```
These are extra line1
These are extra line2
10,Messi,CF
7,Ronaldo,LW
9,Benzema,CF
```

The following schema definition for the DataFrame reader uses the option DROPMALFORMED. You see only the required data; all the description and error records are removed.

```
schema=Structtype([Structfield("jersy",StringType()),Structfield("name",Stri
ngType()),Structfi
eld("position",StringType())])
df1=spark.read\
.option("mode","DROPMALFORMED")\
.option("delimiter",",")\
.schema(schema)\
.csv("inputfile")
df1.select("*").show()
```

Output is:

jersy	name	position
10	Messi	CF
7	Ronaldo	LW
9	Benzema	CF

Select two columns from the dataframe and invoke show():

```
df1.select("jersy","name").show(truncate=False)
```

jersy	name
These are extra line1	null
These are extra line2	null
10	Messi
7	Ronaldo
9	Benzema

Malformed records are not dropped and pushed to the first column and the remaining columns will be replaced with null.This is due to the CSV parser column pruning which is set to true by default in Spark 2.4.

Set the following conf, and run the same code, selecting two fields.

```
spark.conf.set("spark.sql.csv.parser.columnPruning.enabled",False)
```

```
df2=spark.read\
    .option("mode","DROPMALFORMED")\
    .option("delimiter",",")\
    .schema(schema)\
    .csv("inputfile")
    df2.select("jersy","name").show(truncate=False)
```

jersy	name
10	Messi
7	Ronaldo
9	Benzema

Conclusion: If working on selective columns, to handle bad records in CSV files, set spark.sql.csv.parser.columnP runing.enabled to false; otherwise, the error record is pushed to the first column, and all the remaining columns are treated as nulls.

Configuring storage locations

To execute the workloads in CDP, you must modify the references to storage locations. In CDP, references must be changed from HDFS to a cloud object store such as S3.

About this task

The following sample query shows a Spark 2.4 HDFS data location.

```
scala> spark.sql("CREATE TABLE IF NOT EXISTS default.sales_spark_2(Region st
ring, Country string,Item_Type string,Sales_Channel string,Order_Priority st
ring,Order_Date date,Order_ID int,Ship_Date date,Units_sold string,Unit_Pric
e string,Unit_cost string,Total_revenue string,Total_Cost string,Total_Profi
t string) row format delimited fields terminated by ','")
scala> spark.sql("load data local inpath '/tmp/sales.csv' into table defau
lt.sales_spark_2")
scala> spark.sql("select count(*) from default.sales_spark_2").show()
```

The following sample query shows a Spark 2.4 S3 data location.

```
scala> spark.sql("CREATE TABLE IF NOT EXISTS default.sales_spark_2(Region st
ring, Country string,Item_Type string,Sales_Channel string,Order_Priority st
ring,Order_Date date,Order_ID int,Ship_Date date,Units_sold string,Unit_Pric
e string,Unit_cost string,Total_revenue string,Total_Cost string,Total_Profi
t string) row format delimited fields terminated by ','")
scala> spark.sql("load data inpath 's3://<bucket>/sales.csv' into table de
fault.sales_spark_2")
scala> spark.sql("select count(*) from default.sales_spark_2").show()
```

Querying Hive managed tables from Spark

Hive-on-Spark is not supported on CDP. You need to use the Hive Warehouse Connector (HWC) to query Apache Hive managed tables from Apache Spark.

To read Hive external tables from Spark, you do not need HWC. Spark uses native Spark to read external tables. For more information, see the Hive Warehouse Connector documentation.

The following example shows how to query a Hive table from Spark using HWC:

```
spark-shell --jars /opt/cloudera/parcels/CDH/jars/hive-warehouse-connector-a
ssembly-1.0.0.7.1.4.0-203.jar --conf spark.sql.hive.hiveserver2.jdbc.url=jdb
c:hive2://cdhhdp02.uddeepta-bandyopadhyay-s-account.cloud:10000/default --co
nf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/cdhhdp02.uddeepta-band
yopadhyay-s-account.cloud@Uddeepta-bandyopadhyay-s-Account.CLOUD
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark).b
uild()
scala> hive.executeUpdate("UPDATE hive_acid_demo set value=25 where key=4")
scala> val result=hive.execute("select * from default.hive_acid_demo")
scala> result.show()
```

Compiling and running Spark workloads

After modifying the workloads, compile and run (or dry run) the refactored workloads on Spark 2.4.

You can write Spark applications using Java, Scala, Python, SparkR, and others. You build jars from these scripts using one of the following compilers.

- Java (with Maven/Java IDE),
- Scala (with sbt),
- Python (pip).
- SparkR (RStudio)

Post-migration tasks

After the workloads are executed on Spark 2.4, validate the output, and compare the performance of the jobs with CDH/HDP cluster executions.

After the workloads are executed on Spark 2.4, validate the output, and compare the performance of the jobs with CDH/HDP cluster executions. After you perform the post migration configurations, do benchmark testing on Spark 2.4.

Troubleshoot the failed/slow performing workloads by analyzing the application event logs/driver logs and fine tune the workloads for better performance.

For more information, see the following documents:

- https://spark.apache.org/docs/2.4.4/sql-migration-guide-upgrade.html

  https://spark.apache.org/releases/spark-release-2-4-0.html

  https://spark.apache.org/releases/spark-release-2-2-0.html

  https://spark.apache.org/releases/spark-release-2-3-0.html

  https://spark.apache.org/releases/spark-release-2-1-0.html

  https://spark.apache.org/releases/spark-release-2-0-0.html

  For additional information about known issues please also refer:

  Known Issues in Cloudera Manager 7.4.4 | CDP Private Cloud

Spark 2.4 to Spark 3.2 Refactoring

When migrating from Spark 2.4 to Spark 3.x, there are significant changes to executing Dataset/ Dataframe APIs, DDL statements, and UDF functions.

See the *Apache Spark migration* documentation when migrating to Spark 3.x.

Related Information
Apache Spark migration documentation

Migrating Spark CDP to Cloudera Data Engineering

Cloudera Date Engineering Service (CDE) is designed as a fully managed service for Spark. Among many other features, CDE streamlines and provides better Spark jobs monitoring capabilities with an enhanced Job Analysis page, that builds upon the Spark UI and Apache Airflow for orchestrating Spark pipelines.

The CDE service currently supports Spark batch jobs only. Spark streaming is experimental and is not recommended for production. For information about guidelines and limitations, see *Experimental support for Spark Streaming and Spark Structured Streaming*.

CDE does not change Spark. It allows you to easily deploy managed Spark 2.4.0 or higher and Spark 3.0+ clusters in the cloud, so if you are already using Spark in your code, you can migrate the code to CDE as is.

The deployment mode changes from YARN to Kubernetes, but CDE automatically sets the required Kubernetes properties upon job creation, so you need not set them. However, you may have to convert some YARN related properties. Details of these YARN properties are discussed in the *Convert Spark Submits to CDE CLI Spark Submits* section.

Related Information
Experimental support for Spark Streaming and Spark Structured Streaming
Convert Spark Submits to CDE CLI Spark Submits

Cloudera Data Engineering Concepts

Cloudera Date Engineering Service (CDE) is designed as a fully managed service for Spark. Among many other features, CDE streamlines and provides better Spark jobs monitoring capabilities with an enhanced Job Analysis page, that builds upon the Spark UI and Apache Airflow for orchestrating Spark pipelines.

CDE Jobs

CDE introduces the concept of a CDE job which can be of two types: Spark and Airflow.

A Spark CDE job is a Spark submit. An Airflow CDE job is an Airflow Directed Acyclic Graph (DAG) that orchestrates any Spark CDE jobs and optionally Hive CDW queries, and more.

While using Airflow is recommended for complex pipelines it is not mandatory.

CDE Resources

CDE Resources allow you to store all files related to Spark and Airflow jobs along with their dependencies (JARs, ZIP, and text/config files) in the CDE Virtual Cluster.

Resources can also *manage Python Environments*. In other Spark environments, files may have been pre-populated, or Python packages installed through pip, Anaconda, or a similar installer. Resources replace these concepts in CDE and simplify the manageability of working with multiple environments.

Most importantly, Resources provide better Spark and Airflow job observability. Every past run can be mapped to a specific set of dependencies.

For examples of using both file-based and Python environment CDE Resources from a CDE Spark application, see *CDE CLI Demo*.

Other CDE Concepts

CDE Resources and Jobs are really all you need to deploy Spark pipelines in CDE. If you are new to CDE Cloudera recommends that you get familiar with the following concepts:

- Cloud Environment: A logical subset of your cloud provider account including a specific virtual network. For more information, see *Environments*.
- CDE service: The long-running Kubernetes cluster and services that manage the virtual clusters. The CDE service must be enabled in an environment before you can create any virtual clusters.
- CDE virtual cluster: An individual auto-scaling cluster with defined CPU and memory ranges. Virtual clusters in CDE can be created and deleted on demand. Jobs are associated with clusters.
- CDE job run: An individual run of a CDE job. All runs are easily accessible from the CDE UI or observable through the CDE CLI and API.

Learning about how to build Spark CDE Jobs

CDE jobs of type Spark correspond to a Spark Submit CLI and are easy to build. Just like Spark Submit commands, they use Spark Jar, Python, Java files and any other Spark Submit argument such as Class, number of executors, and so on.

There are three ways to build a Spark CDE job:

- Using the CDE web interface. For more information, see Running Jobs in Cloudera Data Engineering.
- Using the CDE CLI tool. For more information, see Using the Cloudera Data Engineering command line interface.
- Using CDE Rest API endpoints. For more information, see CDE API Jobs.

In addition, you can automate migrations from Oozie on CDP Public Cloud Data Hub, CDP Private Cloud Base, CDH and HDP to Spark and Airflow CDE jobs with the oozie2cde API. For information about using the API, see *Migrating Oozie to CDE with the oozie2cde API* .

The CDE CLI and API are equivalent in terms of functionality. The CLI requires downloading and installing a binary on your machine. On the other hand, the API requires submitting requests with a temporary token.

Generally, the CLI is more suitable for interactivity while the API is better for integrating CDE pipelines with external systems.

Related Information

Using Python virtual environments with Cloudera Data Engineering

CDE CLI Demo

Environments

Running Jobs in Cloudera Data Engineering

Using the Cloudera Data Engineering command line interface

CDE API Jobs

Migrating Oozie to CDE with the oozie2cde API

Convert Spark Submit commands to CDE CLI Spark Submit commands

The CDE CLI `cde spark submit` command is intended to closely match with Apache Spark's `spark-submit` command.

Usage

```
cde spark submit [flags]
```

Examples:

```
Local job file 'my-spark-app-0.1.0.jar' and Spark arguments '100' and '1000'
:
> cde spark submit my-spark-app-0.1.0.jar 100 1000 --class com.company.app
.spark.Main

Remote job file:

> cde spark submit s3a://my-bucket/my-spark-app-0.1.0.jar 100 1000 --class
com.company.app.spark.Main


Flags:
      --class string                      job main class
      --conf stringArray                  Spark configuration (format k
ey=value) (can be repeated)
      --driver-cores int                  number of driver cores
      --driver-memory string              driver memory
      --executor-cores int                number of cores per executor
      --executor-memory string            memory per executor
      --file stringArray                  additional file (can be repea
ted)
  -h, --help                              help for submit
      --hide-logs                         whether to hide the job run l
ogs from the output
      --initial-executors string          initial number of executors
      --jar stringArray                   additional jar (can be repeate
d)
      --job-name string                   name of the generated job
      --log-level string                  Spark log level (TRACE, DEBUG,
 INFO, WARN, ERROR, FATAL, OFF)
      --max-executors string              maximum number of executors
      --min-executors string              minimum number of executors
```

```
      --packages string                      additional dependencies as l
ist of Maven coordinates
      --py-file stringArray                  additional Python file (can be
 repeated)
      --pypi-mirror string                   PyPi mirror for --python-req
uirements
      --python-env-resource-name string      Python environment resource n
ame
      --python-requirements string           local path to Python requiremen
ts.txt
      --python-version string                Python version ("python3" or
 "python2")
      --repositories string                  additional repositories/resolve
rs for --packages dependencies
      --runtime-image-resource-name string   custom runtime image resource
 name
      --spark-name string                    Spark name
```

There are a few differences in terms of command syntax and functionality between `CDE/Spark-on-k8s` and `Spark-on-YARN` that you should be aware of. While not an exhaustive guide to converting your Spark-on-YARN (CDH/HDP/Datahub) application to CDE/Spark-on-k8s, the sections below cover some of the common configuration changes that is required.

Remove

The following options that you may have used with `spark-submit` should be removed when using CDE (for example, `cde spark submit`):

- drop `--master`: this is set internally by CDE.
- drop `--deploy-mode`: this is always cluster mode and internally set by CDE.
- drop `--spark.keytab`, `--spark.yarn.principal`, and so on: Kerberos authentication details handled internally by CDE, based on your CDP workload user's authentication.
- drop `shuffle.service.enabled=true`: external shuffle service is actively being developed by Cloudera for Spark-on-k8s (available in an upcoming release).

Update

spark-submit--files, --py-files,   –jars comma-separated syntax can be used:

--files f1.txt,f2.txt

> **Note:** By default, files included with job configuration, for example: --file some_file.txt, are available in the /app/mount/some_file.txt file and not in the Spark process working directory. Therefore, the application must refer to this full path to access the file instead of ./some_file.txt.
>
> If the application or entrypoint needs to be passed with additional arguments, these should be separated from the cde spark submit arguments using -- in front of them. This instructs the parser to treat the rest of the string literally, for example, :
>
> my_entrypoint.py -- -a 1 -b "twenty two"

- Rename `--app.name` to `--job-name`
- CDE defaults to Python3. If you intend to use legacy Python2, add `--python-version python2`. The Python version should always be set through CDE, for example, using the `--python-version` flag. Any previous references that is used to set the Python version, such as `spark.yarn.appMasterEnv.PYSPARK_PYTHON=python3.6`, should be removed.
- If you are migrating an application from an on-premise environment that uses HDFS for storage, you need to update hdfs://... paths in your configuration to the equivalent cloud storage URI of that data. For example, s3a://....

Review

- There are a number of *YARN-specific Spark configurations* that you must review and either remove or convert it to *Spark on Kubernetes specific configuration*. The links referred here are are specific to Spark 3.1.1. In some cases, such as `spark.yarn.executor.memoryOverhead`, Spark now provides more agnostic configuration like `spark.executor.memoryOverhead` that can be used.
- In other cases, you can use an equivalent kubernetes configuration. For example, setting environment variables for the Spark process `spark.yarn.appMasterEnv.TZ=America/Los_Angeles` becomes `spark.kubernetes.driverEnv.TZ=America/Los_Angeles`.
- Certain Spark-on-k8s configurations listed in the reference links above like configuration related to k8s namespaces, authentication, or volume mounts, may not apply or be compatible with CDE. Often CDE manages those, or their equivalents internally. Reach out to Cloudera support or your Cloudera account team if you have questions on this part of your migration.
- Configuration related to external or third-party vendor products should be reviewed and possibly removed. For example, Unravel Data configuration such as `spark.unravel.*` has to be reviewed and removed.

Related Information

YARN specific Spark configurations

Spark on Kubernetes specific configuration

Using the Cloudera Data Engineering CLI

If at any time you are having issues with the CDE CLI, you can view the CDE CLI options by adding the `--help` flag to any CLI commands:

```
cde spark --help

cde spark submit --help

cde airflow --help

cde resource --help
```

When new to the CDE CLI, a common approach is to start with the following steps:

1. Experimenting with CDE Spark Submit CLI
2. Creating a CDE Resource
3. Uploading all files to the CDE Resource
4. Creating CDE jobs with files uploaded to the Resource
5. Running CDE jobs

Step 1: Experimenting with CDE Spark Submits

This is the fastest way to launch a Spark Submit CLI in CDE. Notice however that the CDE job is not instantiated as a Spark CDE job and is therefore not reschedulable from the CDE UI.

```
cde spark submit pysparkjob.py
```
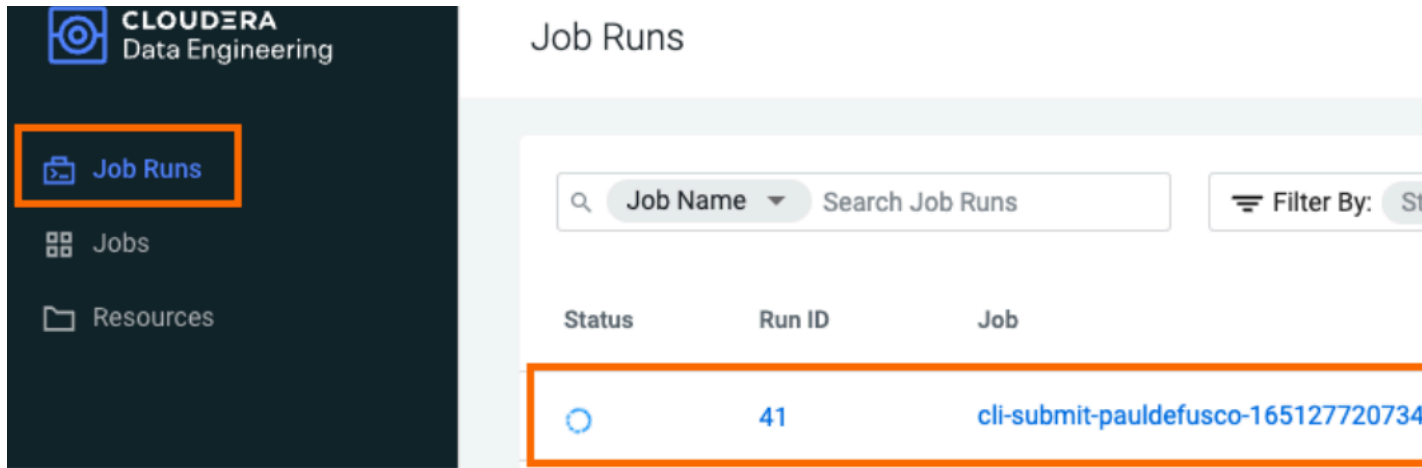
Step 2: Creating a CDE Resource

Cloudera recommends that you create one CDE Resource for every Spark Pipeline or Airflow DAG .

```
cde resource create --name cde_cli_resource
```

Step 3: Uploading all files to the CDE Resource

When uploading to a resource the two important inputs are the name of the target CDE Resource and the local path to the files being uploaded.

Cloudera recommends using the --help command to explore more options such as uploading files in bulk.

```
cde resource upload --name cde_cli_resource --local-path "pysparkjob.py" --r
esource-path "pysparkjob.py"
```

Step 4: Creating CDE jobs with files uploaded to the Resource

Once the files and dependencies have been uploaded you can easily instantiate a CDE job with the job create command.

For example, you can create a CDE job with the CDE Resource file and run it on a schedule.

```
cde job create --name "cde_cli_job" --type "spark"
                --application-file "pysparkjob.py"
                --cron-expression "0 */1 * * *" \
                --schedule-enabled "true"
                --schedule-start "2022-04-29"
                --schedule-end "2022-05-02"
                --mount-1-resource "cde_cli_resource"
```

Step 5: Running CDE jobs

Creating a resource and uploading dependencies is optional. Once that is done, you can trigger execution of the CDE jobs manually.

```
cde job run --name "cde_cli_job" --application-file "pysparkjob.py"
```

You have now completed a basic workflow to start experimenting with the CDE CLI. Below are some more useful examples:

More CDE CLI examples

- Search for CDE jobs based on attributes

  You can use attributes for your search. In this case, you can search by name.

  ```
  cde job list --filter 'name[like]%name_pattern%'
  ```

- List all CDE job runs

  ```
  cde run list
  ```

- Describe CDE job run

  Replace the integer with your job run id. For example, 47 is the ID referred in the below command.

  ```
  cde run describe --id 47
  ```

- Create a CDE job with Custom Spark Log Level

  A big advantage of using CDE is Spark observability. Logging level can be easily customized. Furthermore, every log is always available to the CDE user.

  Using the log-level parameter you can choose any of the following options: TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

  ```
  cde job create --name "cde_cli_job_custom_log_level" --type "spark"
                 --application-file "pysparkjob.py"
                 --log-level "DEBUG"
                 --schedule-enabled "false"
                 --mount-1-resource "cde_cli_resource"
  ```

- Collect CDE Job Run Logs

  You can download the Spark logs you have access to in CDE. Notice you have more options e.g. executor logs

  ```
  cde run logs --type "driver/stdout" --id 47
  ```

  You can modify the log type to any of the available tabs in the corresponding CDE job Run page. For example:

  - driver/stderr or Driver/stdout
  - executor id/stdout

Convert Spark Submits to CDE API Requests

You can execute each of the commands mentioned in the previous section, from the terminal, or in a third-party application such as GitLab or a Jupyter Notebook.

As mentioned in the introductory section, the API requires downloading a CDE Token, based on your user credentials and the CDE Virtual Cluster you want to connect to.

You need the following environment variables:

- CDP WorkloadUsername and WorkloadPassword. You can contact your CDP Administrator for the credentials.
- ACCESS_TOKEN. For information on how to get an access token, see *Getting an Access Token*.
- JOBS_API_URL. You can copy the URL from the CDE Virtual Cluster Service Details page as shown below.

Prior to proceeding with the examples, you must save the variables as environment variables in your local environment. For example:

```
export CDE_TOKEN=<access_token>
export JOBS_API_URL=<jobs_api_url>
```

Create a CDE Resource

```
# Create a CDE Resource with the API
curl -H "Authorization: Bearer $ACCESS_TOKEN" -X POST \
  "$JOBS_API_URL/resources" -H "Content-Type: application/json" \
  -d "{ \"name\": \"cde_api_resource\"}"
```

List CDE Resources at Virtual Cluster Level

```
# curl -H "Authorization: Bearer ${CDE_TOKEN}" -X GET \
# ${CDE_JOB_URL_AWS}/resources"
```

Create CDE job from Resource

```
curl -H "Authorization: Bearer $ACCESS_TOKEN" -X POST "$JOBS_API_URL/jobs" \
        -H "accept: application/json" \
        -H "Content-Type: application/json" \
        -d "{ \"name\": \"cde_cicd_job\", \"type\": \"spark\", \"retentio
nPolicy\":
        \"keep_indefinitely\", \"mounts\": [ { \"dirPrefix\": \"/\", \"resou
rceName\":
        \"cde_cicd_resource\" } ], \"spark\": { \"file\": \"pyspark.py\"},\"
schedule\": {
        \"enabled\": false} }"
```

Run CDE Job

```
curl -H "Authorization: Bearer $ACCESS_TOKEN" -H 'accept: application/json'
-H 'Content-Type:
application/json' -X POST "$CDE_VC_ENDPOINT/jobs/cml2cde_cicd_job/run" -d
 '{"overrides":
```

```
{"spark":{"driverCores": 2, "driverMemory": "4g", "executorCores": 4,
"executorMemory": "4g", "numExecutors": 4}}}'
```

Related Information

Getting an Access Token

Using Swagger Page

The CDE job options that you pass to the requests can get cumbersome. You can use the Swagger page to construct and test your requests.

Before you begin

You can access the Swagger page from the Virtual Cluster Service Details page by clicking API DOC as shown below.

An example procedure to construct and test request:

Procedure

1. Click the GET/jobs method in the jobs

   section.

2. Click Try it out.

3. Enter a few options in the provided fields. For example, ensure that the latestjob flag is set to true and enter name[eq]cde_api_job string in the first field.

4. Click Execute.

The Request URL field is populated now. You can use this to construct your next request, as shown below. Also a response preview is

provided.

5. Using the highlighted portion of the Request URL, construct a new request as shown below.

```
# Show the latest job whose name is cde_cli_job
curl -H "Authorization: Bearer $ACCESS_TOKEN" -X GET "$JOBS_API_URL/jobs
?latestjob=true&filter=name%5Beq%5Dcde_cli_job&limit=20&offset=0&orderby
=name&orderasc=true" \
          -H "accept: application/json" \
          -H "Content-Type: application/json"
```

Getting Started with CDE Airflow

Apache Airflow is a platform to author, schedule and execute Data Engineering pipelines. It is widely used to create dynamic and robust workflows for batch Data Engineering use cases because of its flexibility and ease of use.

Airflow is widely used because of its flexibility and ease of use. CDE embeds Apache Airflow at the CDE Virtual Cluster level. It is automatically deployed for the CDE user during CDE Virtual Cluster creation and requires no maintenance on the part of the CDE Admin.

Learning about Airflow CDE jobs

CDE jobs of type Airflow correspond to Airflow DAGs. Just like Spark CDE jobs there are three ways to build an Airflow CDE job:

- Using the CDE web interface. For more information, see *Running Jobs in Cloudera Data Engineering*.
- Using the CDE CLI tool. For more information, see *Using the Cloudera Data Engineering command line interface*.
- Using CDE Rest API endpoints. For more information, see *CDE API Jobs*.

In addition, you can automate migrations from Oozie on CDP Public Cloud Data Hub, CDP Private Cloud Base, CDH and HDP to Spark and Airflow CDE Jobs with the `oozie2cde` API. For information about using the API, see *Migrating Oozie to CDE with the oozie2cde API.*

Airflow Concepts

Airflow DAG

In Airflow, a DAG (Directed Acyclic Graph) is defined in a Python script that represents the DAGs structure (tasks and their dependencies) as code.

For example, for a simple DAG consisting of three tasks: A, B, and C. The DAG can specify that A has to run successfully before B can run, but C can run anytime. Also that task A times out after 5 minutes, and B can be restarted up to 5 times in case it fails. The DAG might also specify that the workflow runs every night at 10pm, but should not start until a certain date.

For more information about Airflow DAGs, see *Apache Airflow DAG* documentation. For an example DAG in CDE, see *CDE Airflow DAG documentation*.

Airflow UI

The Airflow UI makes it easy to monitor and troubleshoot your data pipelines. For a complete overview of the Airflow UI, see *Apache Airflow UI documentation*.

Related Information

Running Jobs in Cloudera Data Engineering

Using the Cloudera Data Engineering command line interface

CDE API Jobs

Migrating Oozie to CDE with the oozie2cde API

Apache Airflow documentation

CDE Airflow DAG documentation

Apache Airflow UI documentation

Using Airflow

CDE packages the open source version of Airflow. Airflow is maintained and upgraded at each CDE version update. For example, the version of CDE 1.16 includes Airflow 2.2.5.

CDE Airflow imposes no limitations on Operators, Plugins or other integrations with external platforms. Users are free to deploy Airflow DAGs in CDE as dictated by the use case. Cloudera contributed two operators to the Airflow Community:

- CDE Operator: used to orchestrate Spark CDE jobs. This has no requirements other than creating a Spark CDE job separately and then referencing it within the Airflow DAG syntax.
- CDW Operator: used to orchestrate CDW Hive or Impala queries. This requires a Cloudera Virtual Warehouse and setting up of an Airflow connection to it.

For an example DAG in CDE using the two operators, see *CDE Airflow DAG documentation*.

Accessing the CDE Airflow UI

- From your **Virtual Cluster Service Details** page, click Airflow UI to access the UI.

The **Airflow DAGs** page is displayed.

Using the CDE CLI

For Spark CDE jobs, the CDE CLI provides an intuitive solution to create and manage Airflow CDE jobs.

The CDE CLI commands to create a resource and upload files to it are identical as in the Spark CDE job section.

- Submit Airflow CDE job with local DAG file

```
cde airflow submit my-airflow-job.py
```

- Create Airflow CDE job

  Notice that the --application-file flag has been replaced with the --dag-file flag.

  ```
  cde job create --name myairflowjob --type airflow --dag-file path/to/air
  flow_dag.py
  ```

Using the CDE API

As with the CDE CLI, this section builds on the prior examples on the CDE API with Spark CDE jobs.

The payload now includes the argument airflow for the type parameter. For more details on building Airflow CDE requests, see the *Swagger API* page.

```
curl -H "Authorization: Bearer $ACCESS_TOKEN" -X 'POST' \
'$JOBS_API_URL/jobs' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{"type": "airflow", "airflow": {"dagFile": "my_dag.py"}, "identity":
{"disableRoleProxy": true},"mounts": [{"dirPrefix": "/","resourceName": "ooz
ie_migration"}],"name": "oozie2airflow_migration","retentionPolicy": "keep_i
ndefinitely"}'
```

Related Information
CDE Airflow DAG documentation

Using spark-submit drop-in migration tool for migrating Spark workloads to CDE

Cloudera Data Engineering (CDE) provides a command line tool cde-env to help migrate your CDP Spark workloads running on CDP Private Cloud Base (spark-on-YARN) to CDE without having to completely rewrite your existing spark-submit command-lines.

For information about using the cde-env tool, see Using spark-submit drop-in migration tool.