

..

# Migrating Spark to CDP Private Cloud

Date published: 2022-07-29

Date modified: 2022-07-29

# CLOUDERA

# Legal Notice

© Cloudera Inc. 2025. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

|   |          |
|---|----------|
| <b>Migrating Spark workloads to Cloudera.....</b> | <b>4</b> |
| Spark 1.6 to Spark 2.4 Refactoring.....           | 4        |
| Handling prerequisites.....                       | 4        |
| Spark 1.6 to Spark 2.4 changes.....               | 5        |
| Configuring storage locations.....                | 11       |
| Querying Hive managed tables from Spark.....      | 12       |
| Compiling and running Spark workloads.....        | 12       |
| Post-migration tasks.....                         | 17       |
| Spark 2.3 to Spark 2.4 Refactoring.....           | 17       |
| Handling prerequisites.....                       | 18       |
| Spark 2.3 to Spark 2.4 changes.....               | 18       |
| Configuring storage locations.....                | 22       |
| Querying Hive managed tables from Spark.....      | 23       |
| Compiling and running Spark workloads.....        | 23       |
| Post-migration tasks.....                         | 23       |

# Migrating Spark workloads to Cloudera

Migrating Spark workloads from CDH or HDP to Cloudera involves learning the Spark semantic changes in your source cluster and the Cloudera 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 Cloudera, you need to refactor Spark workloads from Spark 1.6 on CDH or HDP to Spark 2.4 on Cloudera.

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 Cloudera.

## 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 prerequisites.

### 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 (Cloudera AI)
- 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 Cloudera, 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 Cloudera 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/](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 select() 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 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("enforceSchema", 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.

#### CREATE OR REPLACE VIEW and ALTER VIEW not supported

ALTER VIEW and CREATE OR REPLACE VIEW AS commands are no longer supported.

Type of change: Configuration/Spark Core Changes

Spark 1.6

You can create views as follows:

```
CREATE OR REPLACE [ [ GLOBAL ] TEMPORARY ] VIEW [ IF NOT EXISTS ] view_name
  [ column_list ]
  [ COMMENT view_comment ]
  [ properties ]
  AS query

ALTER VIEW view_name
  { rename |
    set_properties |
    unset_properties |
    alter_body }
```

Spark 2.4

ALTER VIEW and CREATE OR REPLACE commands above are not supported.

Action Required

Recreate views using the following syntax:

```
CREATE [ [ GLOBAL ] TEMPORARY ] VIEW [ IF NOT EXISTS ] view_name
  [ column_list ]
  [ COMMENT view_comment ]
  [ properties ]
  AS query
```

### 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 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 1.4 - 2.3 CSV example

A Spark 1.4 - 2.3 CSV example illustrates the CSV-handling change in Spark 2.6.

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("jersey",StringType()),Structfield("name",StringType()),Structfield("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 Cloudera, you must modify the references to storage locations. In Cloudera, 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 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_Cost string,Total_Profit string) row format delimited fields terminated by ','")
scala> spark.sql("load data local inpath '/tmp/sales.csv' into table default.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 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_Cost string,Total_Profit string) row format delimited fields terminated by ','")
scala> spark.sql("load data inpath 's3://<bucket>/sales.csv' into table default.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 Cloudera. 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-assembly-1.0.0.7.1.4.0-203.jar --conf spark.sql.hive.hiveserver2.jdbc.url=jdbc:hive2://cdhhd02.uddeпта-bandyopadhyay-s-account.cloud:10000/default --conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/cdhhd02.uddeпта-bandyopadhyay-s-account.cloud@Uddeпта-bandyopadhyay-s-Account.CLOUD
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark).build()
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 Application")
        .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").getOrCreate()
    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.
- Write 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 --executor-memory 512m --executor-cores 1
```

4. Go to the Spark History server web UI at [http://<spark\\_history\\_server>:18088](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](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(Region 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_Cost 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").show()
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](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/latest/sql-migration-guide.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 to:

[Known Issues in Cloudera Manager 7.4.4 | Cloudera on premises](#)

## Spark 2.3 to Spark 2.4 Refactoring

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

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 Cloudera.

## 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 prerequisites.

### 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 (Cloudera AI)
- 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 Cloudera, 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 Cloudera 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/](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 1.6 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 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("enforceSchema", 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")
```

**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 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 1.4 - 2.3 CSV example

A Spark 1.4 - 2.3 CSV example illustrates the CSV-handling change in Spark 2.6.

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("jersey",StringType()),Structfield("name",StringType()),Structfield("position",StringType())])
df1=spark.read\
.option("mode","DROPMALFORMED")\
.option("delimiter",",")\
.schema(schema)\
.csv("inputfile")
df1.select("*").show()
```

Output is:

| jersey | name    | position |
|--------|---------|----------|
| 10     | Messi   | CF       |
| 7      | Ronaldo | LW       |
| 9      | Benzema | CF       |

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

```
df1.select("jersey","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 Cloudera, you must modify the references to storage locations. In Cloudera, 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 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_Cost string, Total_Profit string) row format delimited fields terminated by ', '")
scala> spark.sql("load data local inpath '/tmp/sales.csv' into table default.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 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_Cost string, Total_Profit string) row format delimited fields terminated by ', '")
```

```
scala> spark.sql("load data inpath 's3://<bucket>/sales.csv' into table default.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 Cloudera. 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-assembly-1.0.0.7.1.4.0-203.jar --conf spark.sql.hive.hiveserver2.jdbc.url=jdbc:hive2://cdhhd02.uddeпта-bandyopadhyay-s-account.cloud:10000/default --conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/cdhhd02.uddeпта-bandyopadhyay-s-account.cloud@Uddeпта-bandyopadhyay-s-Account.CLOUD
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark).build()
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/latest/sql-migration-guide.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 to:

[Known Issues in Cloudera Manager 7.4.4 | Cloudera on premises](#)