

CDP One

Migrating Workloads to CDP One

Date published: 2022-06-03

Date modified: 2022-08-15

CLOUDBERA

<https://docs.cloudera.com/>

Legal Notice

© Cloudera Inc. 2024. 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

Overview.....	4
Migrating Spark workloads to CDP.....	4
Spark 1.6 to Spark 2.4 Refactoring.....	4
Handling prerequisites.....	4
Spark 1.6 to Spark 2.4 changes.....	5
Configuring storage locations.....	12
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 Hive and Impala workloads to CDP One.....	24
Handling prerequisites.....	24
Hive 1 and 2 to Hive 3 changes.....	27
Reserved keywords.....	27
Spark-client JAR requires prefix.....	28
Hive warehouse directory.....	28
Replace Hive CLI with Beeline.....	28
PARTIALSCAN.....	29
Concatenation of an external table.....	29
INSERT OVERWRITE.....	30
Managed to external table.....	30
Property changes affecting ordered or sorted subqueries and views.....	31
Runtime configuration changes.....	32
Prepare Hive tables for migration.....	32
Impala changes from CDH to CDP.....	36
Impala configuration differences in CDH and CDP.....	36
Additional documentation.....	37

Overview

You can migrate workloads from Spark, Hive 1, and Hive 2 in CDH and HDP to Hive 3 in CDP. You learn about semantic changes in Spark and Hive versions that affect migration.

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 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 (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 One, 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 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 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.

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 2.4 CSV example

A CSV example illustrates the CSV-handling change in Spark 2.4. If you migrate to Spark 2.4 from Spark 1.6 - Spark 2.3, you need to understand this change.

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)
```

jersey	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("jersey","name").show(truncate=False)
```

jersey	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 One, 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 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 CDP One. 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.uddepta-bandyopadhyay-s-account.cloud:10000/default --conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/cdhhd02.uddepta-bandyopadhyay-s-account.cloud@Uddepta-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.Session;
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
        Session spark = Session.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").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, 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(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, 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 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 (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 One, 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.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 2.4 CSV example

A CSV example illustrates the CSV-handling change in Spark 2.4. If you migrate to Spark 2.4 from Spark 1.6 - Spark 2.3, you need to understand this change.

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())])
dfl=spark.read\
.option("mode","DROPMALFORMED")\
.option("delimiter",",")\
.schema(schema)\
.csv("inputfile")
dfl.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():

```
dfl.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 CDP One, 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 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 CDP One. 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/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](#)

Migrating Hive and Impala workloads to CDP One

You learn how to accelerate the migration process, to refactor Hive applications, and to handle semantic changes from Hive 1/2 to Hive 3. You get pointers to Impala documentation about workload migration and application refactoring.

Handling prerequisites

You must perform a number of tasks before refactoring workloads.

You need to handle the following prerequisites:

- Identify all the Hive 1/2 workloads in the cluster (CDH 5.x, CDH 6.x, HDP 2.x, HDP 3.x). (This document assumes all the identified workloads are in working condition).
- Prepare tables for migration.

To prepare the tables for migration, use the [Hive SRE Tool](#) which is a Cloudera Lab tool that scans your Hive Metastore and HDFS to identify common upgrade problems that can cause the CDP upgrade to fail. The tool provides guidance for fixing those problems before migrating the tables to CDP. This guidance is provided through reports that the administrator must take action on. The tool does not take corrective action itself.

Cloudera strongly recommends running the Hive SRE Tool to aid in this pre-upgrade HMS healthcheck. If you do not run it, you must manually investigate your HMS for the following types of problems.

Refer to [Hive SRE Tooling](#) for tool setup, tool execution & interpretation of output using Hive SRE Tool on Hive Metadata.

The general transition from Hive 1 and 2 to Hive 3 includes the following types of HMS operations. The Hive SRE Tool performs the equivalent types of checks in an automated fashion. Please review these from the Cloudera documentation site ([CDH 5](#), [CDH 6](#), [HDP 2](#), [HDP 3](#)).

The following table lists Hive table conditions, the impact of the conditions, and distributions affected. In this table, XXX stands for some value/count. This value is the number of affected paths you get from the Hive-SRE tool output reports. For each type of condition listed in the table, there is some number of paths/tables affected.

Condition	Impact	Distributions Affected
Bad Filenames	<p>Tables that would be converted from a Managed Non-Acid table to an ACID transactional table require the files to match a certain pattern. This process will scan the potential directories of these tables for bad filename patterns. When located, it will indicate which tables/partitions have file naming conventions that would prevent a successful conversion to ACID.</p> <p>The best and easiest way to correct these file names is to use HiveSQL to rewrite the contents of the table/partition with a simple 'INSERT OVERWRITE TABLE xxx SELECT * FROM xxx'.</p> <p>This type of statement will replace the current bad filenames with valid file names by rewriting the contents in HiveSQL.</p> <p>There are approximately XXX paths that may need remediation. The list of paths can be found in the output of the Hive assessment report in the u3/bad_filenames_for_orc_conversion.md.</p>	HDP 2
Missing Directories	<p>Missing Directories cause the upgrade conversion process to fail. This inconsistency is an indicator that data was removed from the file system, but the Hive MetaStore was not updated to reflect that operation. An example of this is deleting a partition in HDFS, without dropping the partition in Hive.</p> <p>There are XXX affected paths that need remediation. The list of directories can be found in the output of the Hive assessment report, in the file u3/loc_scan_missing_dirs.md.</p>	CDH5, CDH6, HDP2, HDP3
Managed Shadow Tables	<p>In Hive 3, Managed tables are 'ACID' tables. Sharing a location between two 'ACID' tables will cause compaction issues and data issues. These need to be resolved before the upgrade.</p> <p>There are XXX affected tables that need remediation. The list of paths can be found in the output of the Hive assessment report, in the file u3/hms_checks.md</p>	CDH5, CDH6, HDP2, HDP3
Managed Table Migrations	<p>This process will list tables that will and 'could' be migrated to "Managed ACID" tables during the upgrade process.</p> <p>Tables used directly by Spark or if data is managed by a separate process that interacts with the FileSystem, you may experience issues post-upgrade.</p> <p>Recommended: Consider converting these tables to external tables.</p> <p>There are XXX affected tables that may need remediation. The list of tables can be found in the output of the Hive assessment report in the file managed_upgrade_2_acid.sql.</p>	CDH5, CDH6, HDP2

Condition	Impact	Distributions Affected
Compaction Check	<p>In the upgrade to Hive 3, ACID tables must be compacted prior to initiating the upgrade.</p> <p>XXX tables were noted as requiring compaction. Because CDH does not support Hive ACID tables, this may be a leftover from the previous HDP to CDH migration that the customer implemented.</p> <p>This should be investigated further. The affected table is <code>ers_stage_tls.test_delete</code></p>	CDH5, CDH6, HDP2, HDP3
Unknown SerDe Jars	<p>A list tables using SerDe's that are not standard to the platform appears. Review the list of SerDes and verify they are still necessary and available for CDP.</p> <p>There are approximately XXX tables configured with 3rd party SerDes.</p>	CDH5, CDH6, HDP2
Remove <code>transactional=false</code> from Table Properties	<p>In CDH 5.x it is possible to create tables with the property <code>transactional=false</code> set. While this is a no-op setting, if any of your Hive tables explicitly set this, the upgrade process fails.</p> <p>You must remove <code>'transactional=false'</code> from any tables you want to upgrade from CDH 5.x to CDP.</p> <p>Alter the table as follows:</p> <pre>ALTER TABLE my_table UNSET TBLPROPERTIES ('transactional');</pre>	CDH5, CDH6
Make Tables SparkSQL Compatible	<p>Non-Acid, managed tables in ORC or in a Hive Native (but non-ORC) format that are owned by the POSIX user <code>hive</code> will not be SparkSQL compatible after the upgrade unless you perform manual conversions.</p> <p>If your table is a managed, non-ACID table, you can convert it to an external table using this procedure (recommended). After the upgrade, you can easily convert the external table to an ACID table, and then use the Hive Warehouse Connector to access the ACID table from Spark.</p> <p>Take one of the following actions.</p> <ul style="list-style-type: none"> Convert the tables to external Hive tables before the upgrade. <pre>ALTER TABLE ... SET TBLPROPERTIES('EXTERNAL'=TRUE,'external.table.purge'=true)</pre> Change the POSIX ownership to an owner other than <code>hive</code>. <p>You will need to convert managed, ACID v1 tables to external tables after the upgrade.</p>	CDH5, CDH6, HDP2
Legacy Kudu Serde Report	<p>Early versions of Hive/Impala tables using Kudu were built before Kudu became an Apache Project. After Kudu became an Apache Project, the base Kudu Storage Handler classname changed. This report locates and reports on tables using the legacy storage handler class.</p>	CDH5, CDH6, HDP2, HDP3

Condition	Impact	Distributions Affected
Legacy Decimal Scale and Precision Check	When the DECIMAL data type was first introduced in Hive 1, it did NOT include a Scale or Precision element. This causes issues in later integration with Hive and Spark. You need to identify and suggest corrective action for tables where this condition exists.	CDH5, CDH6, HDP2, HDP3
Database / Table and Partition Counts	Use this to understand the scope of what is in the metastore.	CDH5, CDH6, HDP2, HDP3
Small Files, Table Volumes, Empty Datasets	Identify and fix these details to clean up unwanted datasets in the cluster which would speed up the Hive upgrade process.	CDH5, CDH6, HDP2, HDP3
Merge Independent Hive and Spark Catalogs	In HDP 3.0 - 3.1.4, Spark and Hive use independent catalogs for accessing tables created using SparkSQL or Hive tables. A table created from Spark resides in the Spark catalog. A table created from Hive resides in the Hive catalog. Databases fall under the catalog namespace, similar to how tables belong to a database namespace. In HPD 3.1.5, Spark and Hive share a catalog in Hive metastore (HMS) instead of using separate catalogs. The Apache Hive schematool in HDP 3.1.5 and CDP releases supports the mergeCatalog task.	HDP3

Hive 1 and 2 to Hive 3 changes

A description of the change, the type of change, and the required refactoring provide the information you need for migrating from Hive 1 or 2 to Hive 3.

In addition to the topics in this section that describe Hive changes, see the following documentation about changes applicable to CDH and HDP to prepare the workloads.

- [Hive Configuration Changes Requiring Consent](#)
- [Unsupported Interfaces and Features](#)

Reserved keywords

Reserved words (also called keywords) have a predefined meaning and syntax in Hive. These keywords have to be used to develop programming instructions. Reserved words cannot be used as identifiers for other programming elements, such as the name of variable or function.

Hive 1 and 2

TIME, NUMERIC, SYNC are not reserved keywords.

Hive 3

TIME, NUMERIC, SYNC are reserved keywords.

Action Required

Reserved keywords are permitted as identifiers if you quote them. As an example, if in scripts there are identifiers like TIME, NUMERIC, SYNC use backtick `` to quote it like, `TIME`, `NUMERIC`, `SYNC`.

Distribution Affected

CDH5, CDH6, HDP2

Spark-client JAR requires prefix

The Spark-client JAR should be prefixed with hive- to make sure the jar name is consistent across all Hive jars.

Hive 1 and 2

You reference spark-client jar as spark-client.

Hive 3

You reference the spark-client jar as hive-spark-client.

Action Required

Change references as described above.

Hive warehouse directory

Hive stores table files by default in the Hive Warehouse directory.

Hive 1 and 2

The warehouse directory path is /user/hive/warehouse (CDH) or /apps/hive/warehouse (HDP)

Hive 3

The warehouse directory path for a managed table is /warehouse/tablespace/managed/hive and the default location for external table is /warehouse/tablespace/external/hive.

Action Required

As ACID tables reside by default in /warehouse/tablespace/managed/hive and only the Hive service can own and interact with files in this directory, applications accessing the managed tables directly need to change their behavior in Hive3.

As an example, consider the changes required for Spark application. As shown in the table below, if during upgrade Hive1/2 managed tables gets converted to an external table in Hive3, no refactoring is required for Spark application. If the Hive1/2 managed table gets converted to the managed table in Hive3, Spark application needs to refactor to use Hive Warehouse Connector (HWC) or the HWC Spark Direct Reader.

Hive Table before upgrade in Hive1/2	Hive Table after upgrade in Hive3	Spark Refactoring
Managed	External	None
Managed	Managed	Use HWC or HWC Spark Direct Reader
External	External	None

Distribution Affected

CDH5, CDH6, HDP2

Replace Hive CLI with Beeline

Beeline is the new default command line SQL interface for Hive3.

Hive 1 and 2

Hive CLI is deprecated.

Hive 3

The original hive CLI is removed and becomes a wrapper around beeline.

Action Required

Remove, replace and test all hive CLI calls with the beeline command line statements.

For more information, see [Converting Hive CLI scripts to Beeline](#).

Distribution Affected

CDH5, CDH6, HDP2, HDP3

PARTIALSCAN

You need to identify and apply configuration-level changes, including removing the PARTIALSCAN option

Hive 1 and 2

ANALYZE TABLE ... COMPUTE STATISTICS supports the PARTIALSCAN option.

Hive 3

ANALYZE TABLE ... COMPUTE STATISTICS does not support PARTIALSCAN, which is retired and throws error.

For example:

```
ANALYZE TABLE test_groupby COMPUTE STATISTICS PARTIALSCAN;
Error: Error while compiling statement: FAILED: ParseException line 1:46
extraneous input 'PARTIALSCAN' expecting EOF near '<EOF>' (state=42000,code=
40000)
```

Action Required

Remove statements containing ALTER TABLE ... COMPUTE STATISTICS PARTIALSCAN. For example:

```
ANALYZE TABLE test_groupby COMPUTE STATISTICS PARTIALSCAN;
Error: Error while compiling statement: FAILED: ParseException line 1:46
extraneous input 'PARTIALSCAN' expecting EOF near '<EOF>' (state=42000,code=
40000)
```

Distribution Affected

CDH5, CDH6, HDP2.

Concatenation of an external table

If the table or partition contains many small RCFiles or ORC files, then ALTER TABLE table_name [PARTITION (partition_key = 'partition_value' [, ...])] CONCATENATE will merge them into larger files. In the case of RCFile the merge happens at block level whereas for ORC files the merge happens at stripe level thereby avoiding the overhead of decompressing and decoding the data

Hive 1 and 2

You can concatenate an external table. For example:

```
ALTER TABLE table_name [PARTITION (partition_key = 'partition_value' [, ...]
)] CONCATENATE
```

Hive 3

Concatenation of an external table using CONCATENATE is not supported. For example, you get the following error:

```
alter table t6 concatenate;
Error: Error while compiling statement: FAILED: SemanticException [Error 300
34]: Concatenate/Merge can only be performed on managed tables (state=42000,
code=30034)
```

Action Required

CDH5, CDH6, HDP2

Property changes affecting ordered or sorted subqueries and views

Order by/sort by without limit in subqueries is not supported in Hive 3.

Hive 1 and 2

You can use order by/sort by functionality in subqueries.

For example, the optimizer orders a subquery of the following table:

```
+-----+-----+
| test_groupby.coll1 | test_groupby.coll2 |
+-----+-----+
| test                | 2                  |
| test                | 3                  |
| test1               | 5                  |
+-----+-----+
```

```
select * from (select * from test_groupby order by col2 desc) t2;
```

```
+-----+-----+
| t2.coll1 | t2.coll1 |
+-----+-----+
| test1    | 5        |
| test     | 3        |
| test     | 2        |
+-----+-----+
```

Hive 3

The optimizer removes order by/sort by without limit in subqueries and views. For example:

```
select * from (select * from test_groupby order by col2 desc) t2;
```

```
+-----+-----+
| t2.coll1 | t2.coll1 |
+-----+-----+
| test     | 2        |
| test     | 3        |
| test1    | 5        |
+-----+-----+
```

Action Required

If the outer query has to perform some functional logic based on order of a subquery, the following query in Hive 2 returns a different result from Hive 3:

```
select coll1 from (select * from test_groupby order by col2t desc ) t2 limit 1;
```

You must rewrite the query or set the following property to restore previous behavior:

```
set hive.remove.orderby.in.subquery=False
```

Distribution Affected

CDH5, CDH6

Runtime configuration changes

There are a number of runtime configurations that Hive 3 does not support.

Remove unsupported configurations if explicitly set in scripts.

The following configurations supported in Hive 1 or 2 have been removed from Hive 3 and are not supported:

- `hive.limit.query.max.table.partition`
- `hive.warehouse.subdir.inherit.perms`
- `hive.stats.fetch.partition.stats`

Development of an HBase metastore for Hive started in release 2.0.0, but the work has stopped and the code was removed from Hive 3.0.0. The following Hive Metastore HBase configurations have been removed and are not supported:

- `hive.metastore.hbase.cache.ttl`
- `hive.metastore.hbase.catalog.cache.size`
- `hive.metastore.hbase.aggregate.stats.cache.size`
- `hive.metastore.hbase.aggregate.stats.max.partitions`
- `hive.metastore.hbase.aggregate.stats.false.positive.probability`
- `hive.metastore.hbase.aggregate.stats.max.variance`
- `hive.metastore.hbase.cache.ttl`
- `hive.metastore.hbase.cache.max.writer.wait`
- `hive.metastore.hbase.cache.max.reader.wait`
- `hive.metastore.hbase.cache.max.full`
- `hive.metastore.hbase.catalog.cache.size`
- `hive.metastore.hbase.aggregate.stats.cache.size`
- `hive.metastore.hbase.aggregate.stats.max.partitions`
- `hive.metastore.hbase.aggregate.stats.false.positive.probability`
- `hive.metastore.hbase.aggregate.stats.max.variance`
- `hive.metastore.hbase.cache.ttl`
- `hive.metastore.hbase.cache.max.full`
- `hive.metastore.hbase.cache.clean.until`
- `hive.metastore.hbase.connection.class`
- `hive.metastore.hbase.aggr.stats.cache.entries`
- `hive.metastore.hbase.aggr.stats.memory.ttl`
- `hive.metastore.hbase.aggr.stats.invalidate.frequency`
- `hive.metastore.hbase.aggr.stats.hbase.ttl`

Prepare Hive tables for migration

To prepare the tables for migration, use [Hive SRE Tool](#) which is a Cloudera Lab tool that scans your Hive Metastore and HDFS to identify common upgrade problems that can cause the CDP upgrade to fail. The tool provides guidance for fixing those problems before migrating the tables to CDP. This guidance is provided through reports that the administrator must take action on. The tool does not take corrective action itself.

Cloudera strongly recommends running the Hive SRE Tool to aid in this pre-upgrade HMS healthcheck. If you do not run it, you must manually investigate your HMS for the following types of problems.

Please refer to [Hive SRE Tooling](#) for tool setup, tool execution & interpretation of output using Hive SRE Tool on Hive Metadata.

The general transition from Hive 1 and 2 to Hive 3 includes the following types of HMS operations. The Hive SRE Tool performs the equivalent types of checks in an automated fashion. Please review these from the Cloudera documentation site ([CDH 5](#), [CDH 6](#), [HDP 2](#), [HDP 3](#)).

Note for section below:XXX stands for some value/count, this is a value of the number of affected paths we will get from the Hive-SRE tool output reports. For each type of condition listed in the table , there will be some number of paths/tables affected.

Condition	Impact	Distribution Affected
Bad Filenames	<p>Tables that would be converted from a Managed Non-Acid table to an ACID transactional table require the files to match a certain pattern. This process will scan the potential directories of these tables for bad filename patterns. When located, it will indicate which tables/partitions have file naming conventions that would prevent a successful conversion to ACID.</p> <p>The best and easiest way to correct these file names is to use HiveSQL to rewrite the contents of the table/partition with a simple 'INSERT OVERWRITE TABLE xxx SELECT * FROM xxx'.</p> <p>This type of statement will replace the current bad filenames with valid file names by rewriting the contents in HiveSQL.</p> <p>There are approximately XXX paths that may need remediation. The list of paths can be found in the output of the Hive assessment report in the u3/bad_filenames_for_orc_conversion.md.</p>	
Missing Directories	<p>Missing Directories cause the upgrade conversion process to fail. This inconsistency is an indicator that data was removed from the file system, but the Hive MetaStore was not updated to reflect that operation. An example of this is deleting a partition in HDFS, without dropping the partition in Hive.</p> <p>There are XXX affected paths that need remediation. The list of directories can be found in the output of the Hive assessment report, in the file u3/loc_scan_missing_dirs.md.</p>	
Managed Shadow Tables	<p>In Hive 3, Managed tables are 'ACID' tables. Sharing a location between two 'ACID' tables will cause compaction issues and data issues. These need to be resolved before the upgrade.</p> <p>There are XXX affected tables that need remediation. The list of paths can be found in the output of the Hive assessment report, in the file u3/hms_checks.md</p>	

Condition	Impact	Distribution Affected
Managed Table Migrations	<p>This process will list tables that will and 'could' be migrated to "Managed ACID" tables during the upgrade process.</p> <p>Tables used directly by Spark or if data is managed by a separate process that interacts with the FileSystem, you may experience issues post-upgrade.</p> <p>Recommended: Consider converting these tables to external tables.</p> <p>There are XXX affected tables that may need remediation. The list of tables can be found in the output of the Hive assessment report in the file managed_upgrade_2_acid.sql.</p>	
Compaction Check	<p>In the upgrade to Hive 3, ACID tables must be compacted prior to initiating the upgrade.</p> <p>XXX tables were noted as requiring compaction. Because CDH does not support Hive ACID tables, this may be a leftover from the previous HDP to CDH migration that the customer implemented.</p> <p>This should be investigated further. The affected table is ers_stage_tls.test_delete</p>	
Unknown SerDe Jars	<p>Will list tables using SerDe's that are not standard to the platform. Review list of SerDes and verify they are still necessary and available for CDP.</p> <p>There are approximately XXX tables configured with 3rd party SerDes.</p>	
Remove transactional=false from Table Properties	<p>In CDH 5.x it is possible to create tables with the property transactional=false set. While this is a no-op setting, if any of your Hive tables explicitly set this, the upgrade process fails.</p> <p>You must remove 'transactional'=false from any tables you want to upgrade from CDH 5.x to CDP.</p> <p>Alter the table as follows:</p> <pre>ALTER TABLE my_table UNSET TBLPROPERTIES ('transactional');</pre>	

Condition	Impact	Distribution Affected
<p>Make Tables SparkSQL Compatible</p>	<p>Non-Acid, managed tables in ORC or in a Hive Native (but non-ORC) format that are owned by the POSIX user hive will not be SparkSQL compatible after the upgrade unless you perform manual conversions.</p> <p>If your table is a managed, non-ACID table, you can convert it to an external table using this procedure (recommended). After the upgrade, you can easily convert the external table to an ACID table, and then use the Hive Warehouse Connector to access the ACID table from Spark.</p> <p>Take one of the following actions.</p> <ul style="list-style-type: none"> Convert the tables to external Hive tables before the upgrade. <pre>ALTER TABLE ... SET TBLPROPERTIES('EXTERNAL' = 'TRUE', 'external.table.purge' = 'true')</pre> <ul style="list-style-type: none"> Change the POSIX ownership to an owner other than hive. <p>You will need to convert managed, ACID v1 tables to external tables after the upgrade.</p>	
<p>Legacy Kudu Serde Report</p>	<p>Early versions of Hive/Impala tables using Kudu were built before Kudu became an Apache Project. Once it became an Apache Project, the base Kudu Storage Handler classname changed. This report locates and reports on tables using the legacy storage handler class.</p>	
<p>Legacy Decimal Scale and Precision Check</p>	<p>When the DECIMAL data type was first introduced in Hive 1, it did NOT include a Scale or Precision element. This causes issues in later integration with Hive and Spark. We'll identify and suggest corrective action for tables where this condition exists.</p>	
<p>Database / Table and Partition Counts</p>	<p>Use this to understand the scope of what is in the metastore.</p>	
<p>Small Files, Table Volumes, Empty Datasets</p>	<p>Identify and fix these details to clean up unwanted datasets in the cluster which would speed up the Hive upgrade process.</p>	
<p>Merge Independent Hive and Spark Catalogs</p>	<p>In HDP 3.0 - 3.1.4, Spark and Hive use independent catalogs for accessing tables created using SparkSQL or Hive tables. A table created from Spark resides in the Spark catalog. A table created from Hive resides in the Hive catalog. Databases fall under the catalog namespace, similar to how tables belong to a database namespace. In HPD 3.1.5, Spark and Hive share a catalog in Hive metastore (HMS) instead of using separate catalogs.</p> <p>The Apache Hive schematool in HDP 3.1.5 and CDP releases supports the mergeCatalog task.</p>	

Impala changes from CDH to CDP

Cloudera documentation provides details about changes in Impala when migrating from CDH 5.13-5.16 or CDH 6.1 or later to CDP.

- [Impala changes in CDP](#): This document lists down the syntax, service, property and configuration changes that affect Impala after upgrade from CDH 5.13-5.16 or CDH 6.1 or later to CDP.
- [Change location of Datafiles](#): This document explains the impact on Hive warehouse directory location for Impala managed and external tables.
- [Set Storage Engine ACLs](#) : This document describes the steps to set ACLs for Impala to allow Impala to write to the Hive Warehouse Directory.
- [Automatic Invalidation/Refresh of Metadata](#): In CDP, HMS and file system metadata is automatically refreshed. This document provides the details of change in behavior for invalidation and refresh of metadata between CDH and CDP.
- [Metadata Improvements](#): In CDP, all catalog metadata improvements are enabled by default. This document lists down the metadata properties in CDP that can be customized for better performance and scalability.
- [Default Managed Tables](#): In CDP, managed tables are transactional tables with the insert_only property by default. This document describes modifying file systems on a managed table in CDP and the methods to switch to the old CDH behavior.
- [Automatic Refresh of Tables on Impala Clusters](#): In CDP tables or partitions are refreshed automatically on other impala clusters, this document describes the property that enables this behavior.
- [Interoperability between Hive and Impala](#): This document describes the changes made in CDP for the optimal interoperability between Hive and Impala.
- [ORC Support Disabled for Full-Transactional Tables](#): In CDP 7.1.0 and earlier versions, ORC table support is disabled for Impala queries. However, you have an option to switch to the CDH behavior by using the command line argument ENABLE_ORC_SCANNER.
- [Metadata Improvements](#): In CDP, all catalog metadata improvements are enabled by default. This document lists down the metadata properties in CDP that can be customized for better performance and scalability.
- [Default Managed Tables](#): In CDP, managed tables are transactional tables with the insert_only property by default. This document describes modifying file systems on a managed table in CDP and the methods to switch to the old CDH behavior.
- [Automatic Refresh of Tables on Impala Clusters](#): In CDP tables or partitions are refreshed automatically on other impala clusters, this document describes the property that enables this behavior.
- [Interoperability between Hive and Impala](#): This document describes the changes made in CDP for the optimal interoperability between Hive and Impala.
- [ORC Support Disabled for Full-Transactional Tables](#): In CDP 7.1.0 and earlier versions, ORC table support is disabled for Impala queries. However, you have an option to switch to the CDH behavior by using the command line argument ENABLE_ORC_SCANNER.
- [Authorization Provider for Impala](#): In CDP, Ranger is the authorization provider instead of Sentry. This document describes changes about Ranger enforcing a policy which may be different from using Sentry.
- [Data Governance Support by Atlas](#): As part of upgrading a CDH cluster to CDP, Atlas replaces Cloudera Navigator Data Management for your cluster. This document describes the difference between two environments.

Impala configuration differences in CDH and CDP

There are some configuration differences related to Impala in CDH and CDP. These differences are due to the changes made in CDP for the optimal interoperability between Hive and Impala and an improved user experience.

Review the changes before you migrate your Impala workload from CDH to CDP.

- [Config changes](#): This document describes the default value changes and new configurations between CDH and CDP.
- [Default File Formats](#): This document describes the change in default file format of the table in CDP and the steps to switch back to CDH behavior.

- [Reconnect to HS2 Session](#): This document describes the behavior change in CDP for client connection to Impala and HS2.
- [Automatic Row Count Estimation](#): This document describes the new default behavior in CDP for calculating statistics of the table and the property to switch back to CDH behavior.
- [Using Reserved Words in SQL Queries](#): CDP follows ANSI SQL compliance, in which reserved words in SQL queries are rejected. This document provides details about reserved keywords and the method to use them.
- [Other Miscellaneous Changes in Impala](#): This document describes other miscellaneous changes related to Impala Syntax and services that affect Impala during migration.

Additional documentation

Cloudera documentation can help you migrate your Hive workloads.

For more information about migrating Hive workload, see the following documentation:

- [Migrating tables to CDP](#)
- [Migrating Hive workloads from CDH](#)
- [Migrating Hive Workloads from HDP 2.6.5 after an in-place upgrade](#)
- [Replicating Hive data from HDP 3 to CDP](#)
- [Migrating Hive workloads to ACID](#)
- [Apache Hive Changes in CDP](#) capture additional changes that need to be considered while upgrading from Hive 1/2 to Hive 3. The following documents summarize these changes:
 - [Hive Configuration Property Changes](#)
 - [LOCATION and MANAGEDLOCATION clauses](#)
 - [Handling table reference syntax](#)
 - [Identifying semantic changes and workarounds](#)
 - [Unsupported Interfaces and Features](#)
 - [Changes to CDH Hive Tables](#)
 - [Changes to HDP Hive tables](#)
- [CDP Upgrade and Migrations Paths](#)
- [Migrating Hive 1-2 to Hive 3](#)
- [Migrating Hive Workloads from CDH](#)
- [SRE Tool](#)
- [Apache Hive Language Manual](#)
- [Release Notes](#)