# Oracle® Big Data Discovery

BDD Shell Guide

Version 1.4.0 • October 2016

# Copyright and disclaimer

# Table of Contents

# Preface

Oracle Big Data Discovery is a set of end-to-end visual analytic capabilities that leverage the power of Apache Spark to turn raw data into business insight in minutes, without the need to learn specialist big data tools or rely only on highly skilled resources. The visual user interface empowers business analysts to find, explore, transform, blend and analyze big data, and then easily share results.

## About this guide

This guide describes how to use a Python-based BDD Shell to explore and manipulate the internals of BDD, interact with Hadoop, and analyze data interactively.

## Audience

This guide is intended for data developers who need to programmatically explore BDD and its data.

## Conventions

The following conventions are used in this document.

### Typographic conventions

The following table describes the typographic conventions used in this document.

| Typeface | Meaning |
|---|---|
| **User Interface Elements** | This formatting is used for graphical user interface elements such as pages, dialog boxes, buttons, and fields. |
| `Code Sample` | This formatting is used for sample code segments within a paragraph. |
| *Variable* | This formatting is used for variable values.<br>For variables within a code sample, the formatting is ***`Variable`***. |
| `File Path` | This formatting is used for file names and paths. |

### Path variable conventions

This table describes the path variable conventions used in this document.

| Path variable | Meaning |
|---|---|
| `$ORACLE_HOME` | Indicates the absolute path to your Oracle Middleware home directory, where BDD and WebLogic Server are installed. |

| Path variable | Meaning |
|---|---|
| `$BDD_HOME` | Indicates the absolute path to your Oracle Big Data Discovery home directory, `$ORACLE_HOME/BDD-<version>`. |
| `$DOMAIN_HOME` | Indicates the absolute path to your WebLogic domain home directory. For example, if your domain is named `bdd-<version>_domain`, then `$DOMAIN_HOME` is `$ORACLE_HOME/user_projects/domains/bdd-<version>_domain`. |
| `$DGRAPH_HOME` | Indicates the absolute path to your Dgraph home directory, `$BDD_HOME/dgraph`. |

## Contacting Oracle Customer Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. This includes important information regarding Oracle software, implementation questions, product and solution help, as well as overall news and updates from Oracle.

You can contact Oracle Customer Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Chapter 1

# Introduction

This section provides a high-level introduction to the BDD Shell component of Big Data Discovery.

## BDD Shell overview

BDD Shell is a programming shell which provides a way to explore and manipulate the internals of BDD, interact with Hadoop, and analyze data interactively.

BDD Shell is an interactive tool designed to work with BDD without using Studio's front-end.

The BDD Shell core features are:

- **Python-based shell:** BDD Shell is based on Python, and supports both interactive and batch mode.
- **Exposes all BDD concepts:** Among these concepts are data sources (Hive tables), BDD data sets, and Views (Dgraph View and HDFS View).
- **Interacts with all BDD APIs:** such as Dgraph Gateway Web Services, DP Client, Studio.
- **Supports Spark:** Spark Python SDK is available to use in BDD Shell. `SparkContext(sc)` and `HiveContext(sqlContext)` are initialized when BDD Shell launches.
- **BDD Shell SDK:** Provides easy-to-use Python Wrappers for BDD APIs and Python Utilities for developers to use when they need to interact with BDD. It saves a lot of boilerplate code when developers want to try/test features of BDD in BDD Shell.
- **Extensibility:** BDD Shell SDK is designed with extensibility in mind, new APIs or features can be added easily.
- **Use of Third-party Libraries:** Third-party Python libraries can be used in BDD Shell as long as they are installed on the system and available on a standard Python shell, such as Pandas and NumPy.

When started, BDD Shell runs in the Python interpreter. Python documentation is available at the following sites:

- *Python Tutorial*: *https://docs.python.org/2/tutorial/index.html*
- Python documentation set: *https://docs.python.org/2/index.html*

## Starting BDD Shell

This topic describes how to start BDD Shell.

To start BDD Shell:

1. From the command prompt, change to the `$BDD_HOME/bdd-shell` directory.
2. Run the `bdd-shell.sh` script.

   ```
   ./bdd-shell.sh
   ```

3. When BDD Shell starts up, it displays a series of configuration and run-time messages. BDD Shell is ready for use when you see the Python interperter **>>>** prompt:

   ```
   Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
   16/09/11 10:30:57 INFO SparkContext: Running Spark version 1.5.0
   ...
   Welcome to
        ___   ___   ___       __   _     ____  _     _
       | |_) | | \ | | \     ( (` | |_| | |_  | |   | |
       |_|_) |_|_/ |_|_/     _)_) |_| | |_|__ |_|__ |_|__

   SparkContext available as sc, HiveContext available as sqlContext.
   BDD Context available as bc.

   >>>
   ```

Note that BDD Shell runs as a Spark application. Therefore, when it is running, it will appear on the YARN All Applications page with a State of "RUNNING" and a FinalStatus of "UNDEFINED". When you exit BDD Shell, the application will have a State of "FINISHED" and a FinalStatus of "SUCCEEDED".

To exit the Python interpreter, use the `quit()` command:

```
>>> quit()
16/09/11 17:11:30 INFO SparkUI: Stopped Spark web UI at http://10.152.105.219:4040
16/09/11 17:11:30 INFO DAGScheduler: Stopping DAGScheduler
...
16/09/11 17:11:31 INFO ShutdownHookManager: Shutdown hook called
16/09/11 17:11:31 INFO ShutdownHookManager: Deleting directory /tmp/spark-6cad369a-a80f-4105
```

## Getting BDD Shell help

You can get help on BDD Shell SDK API methods via Python's built-in help utility.

### Check what BDD Shell SDK packages and classes are available

Use:

```
>>> help('bdd_sdk')
```

The command output is:

```
Help on package bdd_sdk:

NAME
    bdd_sdk

FILE
    /scratch/localdisk/Oracle/Middleware/BDD-1.4.0.37.1202/bdd-shell/bdd_sdk/__init__.py
```

```
PACKAGE CONTENTS
    BddContext
    BddDataSource
    BddDataset
    BddDatasets
    BddViews
    DgraphView
    HdfsView
    utils (package)

(END)
```

Enter **Q** (in upper- or lower-case) to return to the Python prompt.

## Check for a specific class

For example, to check for the `bdd_sdk.BddDatasets` class, use:

```
>>> help('bdd_sdk.BddDatasets')
```

## Check for a specific method

For example, to check for the `bdd_sdk.BddDatasets.dataset()` method, use:

```
>>> help('bdd_sdk.BddDatasets.dataset')
```

Chapter 2

# Using BDD Shell

This section describes how to use BDD Shell from the Python interpreter command line.

## Working with data sets

This topic shows some of the operations you can use with BDD data sets.

Note that Dgraph Gateway must running before these commands can succeed. If Dgraph Gateway is not running, you will see a `Connection refused` message, as shown in this example:

```
>>> dss = bc.datasets()
[Errno 111] Connection refused
>>>
```

### Retrieving all data sets

To return all data sets:

```
>>> dss = bc.datasets()
```

### Finding the count

This command shows the number of returned data sets:

```
>>> dss.count
2
```

In the example, there are two data sets in BDD.

### Printing the data sets

You can use the Python `print` function to print the names and sources of each data set. A Python `for` loop will iterate over the data sets:

```
>>> dss = bc.datasets()
>>> for ds in dss:
...     print ds
...
```

```
WarrantyClaims  default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3
default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3   Hive   default.warrantyclaims

MassTowns       default_edp_da5ff7d5-521e-4851-a9c8-2755802f3053
default_edp_da5ff7d5-521e-4851-a9c8-2755802f3053   Hive   default.masstowns
>>>
```

In this example, there are two data sets:

- The data set with with a display name "WarrantyClaims" has a Dgraph database name "default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3" and its collection name is the same. The Hive source table is named "warrantyclaims" and is in the Hive "default" database.
- The data set with a display name "MassTowns" has a Dgraph database name "default_edp_da5ff7d5-521e-4851-a9c8-2755802f3053" and its collection name is the same. The Hive source table is named "masstowns" and is also in the Hive "default" database.

Similarly, you can retrieve one data set (via its collection name) and then output its name and source:

```
>>> ds = dss.dataset('default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3')
>>> ds

WarrantyClaims  default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3
default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3   Hive   default.warrantyclaims
>>>
```

## Displaying the data set metadata

You can use the `BddDataset` `properties()` function to print the data set's metadata:

```
>>> ds = dss.dataset('default_edp_4f6c159c-1042-4cd5-a6b2-e567e5cd03d3')
>>> ds.properties()
{'timesViewed': '0', 'sourceName': 'default.warrantyclaims', 'attributeDisplayNames':
'Vehicle_Dealer',
...
'fullDataSet': 'true', 'collectionIdToBeReplaced': None, 'authorizedGroup': None}
>>>
```

To display only one property, you can use this syntax:

```
ds.properties ['propName']
```

For example:

```
>>> ds.properties() ['displayName']
'WarrantyClaims'
>>>
```

To produce a more readable output, import the `json` module and then use the `print` function:

```
>>> dss = bc.datasets()
>>> ds = dss.dataset('default_edp_65e296e7-52b5-4e3e-b837-3386cb3ec079')
>>> import json
>>> print json.dumps(ds.properties(), indent=2, sort_keys=True, ensure_ascii=False)
{
  "accessType": "private",
  "attributeCount": "24",
  "attributeDisplayNames": "Vehicle_Dealer",
  ...
  "transformed": "false",
  "uploadUserId": "10098",
  "uploadUserName": "Admin Admin",
  "version": "3"
}
>>>
```

## Displaying Views

A View provides a way of accessing data in a data set.

BDD Shell implements two types of Views for a data set:

- `DgraphView` represents a Dgraph View of the data set. When this View is printed, it lists the Dgraph database name and collection name of a data set, as well as the host name and port of the machine on which the data resides.
- `HdfsView` represents an HDFS view of the data set. Internally, this view refers to the data set sample files on HDFS. When this View is printed, it lists the HDFS location of the data set.

Note that these views are not the same as the views in Studio, and they have nothing to do with each other.

### Printing all views in a data set

You can use the `BddDataset properties()` function to print the data set's metadata:

```
>>> dss = bc.datasets()
>>> ds = dss.dataset('edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066')
>>> views = ds.views()
>>> for view in views:
...     print view
...
DgraphView
edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066.edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066
   bus14.example.com 7003
HdfsView        /user/bdd/edp/data/.collectionData
/edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066.edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066
```

Note that in the `DgraphView`, the data set's name is made up of the Dgraph database name and the collection name, separated with a period. The data set name in the `HdfsView`, likewise, has the databaseName.collectionName syntax.

### Finding a view type

First import the two view types:

```
>>> from bdd_sdk import DgraphView
>>> from bdd_sdk import HdfsView
>>> dgraph_views = views.find_views(view_type=DgraphView)
>>> hdfs_views = views.find_views(view_type=HdfsView)
```

Next, print out each view type:

```
>>> for view in dgraph_views:
...     print view
...
DgraphView
edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066.edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066
   bus14.example.com 7003
>>> for view in hdfs_views:
...     print view
...
HdfsView        /user/bdd/edp/data/.collectionData
/edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066.edp_cli_edp_80ec0018-05f4-4af4-b055-d0f058d04066
```

## Working with Spark DataFrames

BDD Shell lets you convert a BDD data set into a Spark DataFrame.

You use Spark DataFrames to handle records in your BDD instance. After converting a BDD data set to a Spark DataFrame, you can use the Spark API methods on the DataFrame. Some of these methods are used in the examples below.

For details on the Spark API, see:
*http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame*

### Converting a BDD data set into a Spark DataFrame

Use the `dataset.to_spark()` method to convert the data set to a DataFrame:

```
>>> ds = dss.dataset('default_edp_d147818d-fac2-479b-b48b-28160ae290d0')
>>> df = ds.to_spark()
16/03/30 16:04:07 INFO HiveContext: Initializing execution hive, version 1.2.1
16/03/30 16:04:07 INFO ClientWrapper: Inspected Hadoop version: 2.6.0
16/03
/30 16:04:07 INFO ClientWrapper: Loaded org.apache.hadoop.hive.shims.Hadoop23Shims for Hadoop
version 2.6.0
16/03/30 16:04:07 WARN HiveConf: HiveConf of name hive.enable.spark.execution.engine does not exist
16/03/30 16:04:07 INFO metastore: Trying to connect to metastore with URI thrift:/
/busgg2014.us.oracle.com:9083
16/03/30 16:04:08 INFO metastore: Connected to metastore.
...
16/03
/30 16:04:20 INFO ParseDriver: Parsing command: SELECT ORIGINAL_RECORD.`vin`[0]
`vin`,ORIGINAL_RECORD.`production_country`[0]
`production_country`,ORIGINAL_RECORD.`production_region`[0]
`production_region`,ORIGINAL_RECORD.`make`[0] `make`,ORIGINAL_RECORD.`manufacturer`[0]
`manufacturer`,ORIGINAL_RECORD.`model`[0] `model`,ORIGINAL_RECORD.`model_year`[0]
`model_year`,ORIGINAL_RECORD.`claim_date`[0] `claim_date`,ORIGINAL_RECORD.`dealer_geocode`[0]
`dealer_geocode`,ORIGINAL_RECORD.`vehicle_dealer`[0]
`vehicle_dealer`,ORIGINAL_RECORD.`dealer_state`[0] `dealer_state`,ORIGINAL_RECORD.`dealer_city`[0]
`dealer_city`,ORIGINAL_RECORD.`labor_description`[0]
`labor_description`,ORIGINAL_RECORD.`commodity`[0] `commodity`,ORIGINAL_RECORD.`complaint`[0]
`complaint`,ORIGINAL_RECORD.`part_number`[0] `part_number`,ORIGINAL_RECORD.`sale_date`[0]
`sale_date`,ORIGINAL_RECORD.`supplier_country`[0] `supplier_country`,ORIGINAL_RECORD.`supplier`[0]
`supplier`,ORIGINAL_RECORD.`supplier_state`[0] `supplier_state`,ORIGINAL_RECORD.`labor_amount`[0]
`labor_amount`,ORIGINAL_RECORD.`part_amount`[0] `part_amount`,ORIGINAL_RECORD.`claim_amount`[0]
`claim_amount`,ORIGINAL_RECORD.`PRIMARY_KEY` `PRIMARY_KEY` FROM temp1
16/03/30 16:04:21 INFO ParseDriver: Parse Completed
>>>
```

### Counting the DataFrame records

Use the Spark `count()` command to show the number of records in this DataFrame:

```
>>> df.count()
16/03/30 16:08:44 INFO MemoryStore: ensureFreeSpace(97136) called with curMem=274561, maxMem
=556038881
16/03
/30 16:08:44 INFO MemoryStore: Block broadcast_2 stored as values in memory (estimated size 94.9 KB,
free 529.9 MB)
...
16/03
/30 16:08:51 INFO DAGScheduler: Job 1 finished: count at NativeMethodAccessorImpl.java:-2, took
6.763105 s
16/03
/30 16:08:51 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 3) in 266 ms on
busgg2014.us.oracle.com (1/1)
```

```
16/03/30 16:08:51 INFO YarnScheduler: Removed TaskSet 2.0, whose tasks have all completed, from pool
9983
>>>
```

As the command result shows, there are 9983 records in the DataFrame.

## Printing the DataFrame schema

Use the Spark `printSchema()` method to print the schema in a tree format:

```
>>> df.printSchema()
root
 |-- vin: string (nullable = true)
 |-- production_country: string (nullable = true)
 |-- production_region: string (nullable = true)
 |-- make: string (nullable = true)
 |-- manufacturer: string (nullable = true)
 |-- model: string (nullable = true)
 |-- model_year: long (nullable = true)
 |-- claim_date: string (nullable = true)
 |-- dealer_geocode: string (nullable = true)
 |-- vehicle_dealer: string (nullable = true)
 |-- dealer_state: string (nullable = true)
 |-- dealer_city: string (nullable = true)
 |-- labor_description: string (nullable = true)
 |-- commodity: string (nullable = true)
 |-- complaint: string (nullable = true)
 |-- part_number: string (nullable = true)
 |-- sale_date: string (nullable = true)
 |-- supplier_country: string (nullable = true)
 |-- supplier: string (nullable = true)
 |-- supplier_state: string (nullable = true)
 |-- labor_amount: double (nullable = true)
 |-- part_amount: double (nullable = true)
 |-- claim_amount: double (nullable = true)
 |-- PRIMARY_KEY: string (nullable = false)

>>>
```

## Printing the rows in the table

Use the `df.show()` command to print the first five rows:

```
>>> df.show(5)
16/03/30 16:18:09 INFO MemoryStore: ensureFreeSpace(247720) called with curMem=698025, maxMem
=556038881
...
+-----------------+------------------+-----------------+---------+--------------------+------
+----------+-----------+----------------+--------------------+------------+-----------
+--------------------+--------------------+--------------------+-------------------+-----------
+----------------+--------------------+--------------+------------+-----------+------------
+-----------+
|              vin|production_country|production_region|     make|        manufacturer|
model|model_year| claim_date|  dealer_geocode|     vehicle_dealer|dealer_state|dealer_city|
labor_description|           commodity|           complaint|       part_number|
sale_date|supplier_country|
supplier|supplier_state|labor_amount|part_amount|claim_amount|PRIMARY_KEY|
+-----------------+------------------+-----------------+---------+--------------------+------
+----------+-----------+----------------+--------------------+------------+-----------
+--------------------+--------------------+--------------------+-------------------+-----------
+----------------+--------------------+--------------+------------+-----------+------------
+-----------+
|2U9SF69Q77E237441|           Germany|           Europe|CHEVROLET|GENERAL MOTORS CORP.|MALIBU|
2009|2011-10-300| 30.107 -81.7167|   COGGIN CHEVEROLET|          FL|Orange Park|Replace Filter
an...|POWER TRAIN:AUTOM...|WE UNFORTUNATELY ...|   p6J1RLSXXKE-1186|2009-10-278|
Canada|Magna Internation...|       Ontario|    219.2096|   319.2618|    538.4714|      0-0-0|
|3B2GL69N870242937|            Canada|    North America|   SATURN|GENERAL MOTORS CORP.|   ION|
2009|2011-10-274|27.9475 -82.4588|     SATURN OF TAMPA|          FL|      Tampa|Replace Filter
an...|POWER TRAIN:AUTOM...|2003 SATURN ION I...|p1468Z160GWVVA-1444|2009-09-253|          France|
   Hutchinson SA|         Paris|    135.5176|   340.9534|     476.471|      0-0-1|
|3F2DQ69P670264048|            Canada|    North America|   TOYOTA|TOYOTA MOTOR CORP...|TACOMA|
2009|2011-09-248|34.5476 -82.6276|  TOYOTA OF ANDERSON|          SC|   Anderson|Replace ABS
Contr...|SERVICE BRAKES, H...|DECEMBER 16, 2006...|   p6J1RLSXXKE-1186|2009-08-226|          France|
           Faurecia|      Nanterre|     78.8486|   327.8384|     406.687|      0-0-2|
|3P2KN692570293433|            Canada|    North America|    VOLVO|VOLVO CARS OF N.A...|   S80|
2009|2011-12-335|            null|SAND BERG NORTH W...|          WA|   Lindwood|Replace Control
M...|VEHICLE SPEED CON...|DT: 2000 VOLVO S8...| p2617B6326AID-1248|2009-11-305|
Italy|Magneti Marelli H...|        Milano|       83.57|   340.9534|    424.5234|      0-0-3|
|3X2VM69H770222629|            Canada|    North America| CADILLAC|GENERAL MOTORS CORP.|   CTS|
2009|2011-09-271|40.7702 -73.7108|NORTH BAY CADILLA...|          NY| Great Neck|     Replace
Harness|            AIR BAGS|THE CONSUMER STOP...|   p6J1RLSXXKE-1186|2009-08-226|
US|Key Safety System...|      Michigan|    295.2644|   327.8384|    623.1028|      0-0-4|
+-----------------+------------------+-----------------+---------+--------------------+------
+----------+-----------+----------------+--------------------+------------+-----------
+--------------------+--------------------+--------------------+-------------------+-----------
+----------------+--------------------+--------------+------------+-----------+------------
+-----------+
only showing top 5 rows

>>>
```

Some other commands that work with records are:

- `df.first()` gets the first record
- `df.head(n=1)` gets the first n records, n is optional and defaults to 1
- `df.collect()` gets all the records; note that this is an expensive operation, especially for large data sets.

## Working with Hive data sources

With BDD Shell, you can access the data in the Hive tables that are the data set sources.

All BDD data sets are sourced from Hive tables.

### Getting BddDataSource from BddDataset

Get the data source for the data set and print it:

```
>>> # get all data sets
>>> dss = bc.datasets()
>>> # get the WarrantyClaims data set
>>> ds = dss.dataset('default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620')
>>> # get the data source for WarrantyClaims
>>> bds = ds.source()
>>> # print out the source Hive table for the data set
>>> bds
Hive    default.warrantyclaims
```

The output shows that the data set's source is a Hive table named "warrantyclaims" and is stored in the Hive "default" database.

## Converting to a Spark DataFrame

Convert the `BddDataSource` to a Spark DataFrame:

```
>>> df = bds.to_spark()
16/03/24 16:21:22 INFO ParseDriver: Parsing command: SELECT * FROM default.warrantyclaims
16/03/24 16:21:22 INFO ParseDriver: Parse Completed
16/03/24 16:21:23 INFO AvroSerDe: Avro schema is {"type":"record","name":"schema",
"namespace":"com.oracle.eid.appwizard","doc":"schema for upload file",
"fields":[{"name":"VIN","type":["null","string"],"doc":"VIN","default":null},
...
{"name":"Claim_Amount","type":["null","string"],"doc":"Claim_Amount","default":null}]}
>>> # print out the DataFrame's schema
>>> df.printSchema()
root
 |-- vin: string (nullable = true)
 |-- production_country: string (nullable = true)
 |-- production_region: string (nullable = true)
 |-- make: string (nullable = true)
 |-- manufacturer: string (nullable = true)
 |-- model: string (nullable = true)
 |-- model_year: string (nullable = true)
 |-- claim_date: string (nullable = true)
 |-- dealer_geocode: string (nullable = true)
 |-- vehicle_dealer: string (nullable = true)
 |-- dealer_state: string (nullable = true)
 |-- dealer_city: string (nullable = true)
 |-- labor_description: string (nullable = true)
 |-- commodity: string (nullable = true)
 |-- complaint: string (nullable = true)
 |-- part_number: string (nullable = true)
 |-- sale_date: string (nullable = true)
 |-- supplier_country: string (nullable = true)
 |-- supplier: string (nullable = true)
 |-- supplier_state: string (nullable = true)
 |-- labor_amount: string (nullable = true)
 |-- part_amount: string (nullable = true)
 |-- claim_amount: string (nullable = true)
```

## Saving the DataFrame as a new Hive table

Create a new Hive table that contains only vehicle dealer information. The four dealer-related columns (as shown in the schema above) are:

- `vehicle_dealer`
- `dealer_state`
- `dealer_city`
- `dealer_geocode`

The new Hive table is named "dealers_info" and is stored in the Hive "default" database. Before creating the table, make sure that a table of that name does not already exist in the Hive database.

Select the four columns from the DataFrame and then save it as a new Hive table:

```
>>> # select the columns
>>> df2 = df.select('vehicle_dealer','dealer_state','dealer_city','dealer_geocode')
>>> # write the table
>>> df2.write.saveAsTable("default.dealers_info")
```

Note that if a table of that name already exits, you should see the following exception messages:

```
>>> df2.write.saveAsTable("default.dealers_info")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/localdisk/hadoop/spark-1.5.0-bin-hadoop2.6/python/pyspark/sql
/readwriter.py", line 370, in saveAsTable
    self._jwrite.saveAsTable(name)
  File "/scratch/localdisk/Oracle/Middleware/BDD-1.2.0.31.801/bdd-shell/py4j
/java_gateway.py", line 537, in __call__
    self.target_id, self.name)
  File "/localdisk/hadoop/spark-1.5.0-bin-hadoop2.6/python/pyspark/sql/utils.py", line 40, in deco
    raise AnalysisException(s.split(': ', 1)[1])
AnalysisException: path hdfs://bus014.example.com:8020/user/hive/warehouse
/dealers_info already exists.;
```

Assuming that the `write.saveAsTable` operation did not return an error, you can verify the operation by first querying the new table:

```
>>> # query the new table
>>> df3 = sqlContext.sql("select * from default.dealers_info")
16/04/08 13:01:21 INFO ParseDriver: Parsing command: select * from default.dealers_info
16/04/08 13:01:21 INFO ParseDriver: Parse Completed
16/04/08 13:01:21 INFO ParquetRelation: Listing hdfs://bus014.example.com:8020/user/hive/warehouse
/dealers_info on driver
...
>>>
```

Then print the new table's schema:

```
>>> df3.printSchema()
root
 |-- vehicle_dealer: string (nullable = true)
 |-- dealer_state: string (nullable = true)
 |-- dealer_city: string (nullable = true)
 |-- dealer_geocode: string (nullable = true)
```

As the results show, the table does have the four expected columns.

Finally, print the first three rows from the table:

```
>>> df3.show(3)
+--------------------+------------+-----------+--------------------+
|      vehicle_dealer|dealer_state|dealer_city|      dealer_geocode|
+--------------------+------------+-----------+--------------------+
|NORTH GATE LINCOL...|          FL|      Tampa|27.947500 -82.458800|
|               KERRY|          OH| Cincinnati|39.161600 -84.456900|
|        MANKATO FORD|          MN|    Mankato|44.115600 -93.998400|
+--------------------+------------+-----------+--------------------+
only showing top 3 rows
```

As the results show, each row has assignments from the four columns.

## Creating a new BDD data set from BDD Shell

From within BDD Shell, you can call the Data Processing CLI via Python code. Running the DP CLI creates a new BDD data set from the Hive table you just created.

The DP CLI is called via a Python `os.system()` call. The syntax is:

```
os.system("/localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli
/data_processing_CLI -d dbName -t tableName")
```

where:

- `-t` (or `--table`) specifies the name of the Hive table to process.
- `-d` (or `-database`) specifies the Hive database where the table is stored.

For example:

```
>>> os.system("/localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli
/data_processing_CLI -d default -t dealers_info")
```

For information on the Data Processing CLI, see the *Data Processing Guide*.

## Pandas integration

Pandas is a Python package that provides powerful data structures for data analysis, time series, and statistics.

This topic provides a quick overview of how you can use Pandas to manipulate a DataFrame. To access the Pandas documentation, see: *http://pandas.pydata.org/pandas-docs/version/0.17.1*

### Installing Pandas

Pandas is included in the Anaconda 2.5 version, and does not need a separate installation.

If Pandas is not installed in your version of Python, install Pandas via `pip` or `conda` first, re-start BDD Shell, and then run this command in the Python interpreter:

```
$ conda install pandas
```

### Converting a Spark DataFrame into a Pandas DataFrame

First import the Pandas package and then do the conversion:

```
>>> ds = dss.dataset('default_edp_75f94d7b-dea9-4d77-8e66-ed1bf981f615')
>>> df = ds.to_spark()
>>> import pandas as pd
>>> pdf = df.toPandas()
16/03/28 11:52:33 INFO MemoryStore: ensureFreeSpace(97136) called with curMem=269241, maxMem
=556038881
...
16/03/28 11:52:41 INFO YarnScheduler: Removed TaskSet 1.0, whose tasks have all completed, from pool
```

### Manipulating data in Pandas DataFrame

Add a new column and change a column:

```
pdf['tip_rolling_mean'] = pd.rolling_mean(pdf.tip_amount, window=10)
pdf['fare_tip_rolling_corr'] = pd.rolling_corr(pdf.fare_amount, pdf.tip_amount, window=10)
```

### Converting the Pandas DataFrame back into a Spark DataFrame

```
>>> df2 = sqlContext.createDataFrame(pdf)
```

### Persisting the Spark DataFrame into a new Hive table

First, create the new Hive table. The table is named "new_taxi_data" and resides in the Hive "default" database:

```
>>> df2.write.saveAsTable('default.new_taxi_data')
...
16/03/28 15:12:28 INFO SparkContext: Starting job: saveAsTable at NativeMethodAccessorImpl.java:-2
```

```
16/03
/28 15:12:28 INFO DAGScheduler: Got job 2 (saveAsTable at NativeMethodAccessorImpl.java:-2) with 2
output partitions
16/03
/28 15:12:28 INFO DAGScheduler: Final stage: ResultStage 2(saveAsTable at
NativeMethodAccessorImpl.java:-2)
...
16/03
/28 15:12:32 INFO HiveContext$$anon$1: Persisting data source relation with a single input path into
Hive metastore in Hive compatible format. Input path: hdfs://bus14.example.com:8020/user/hive
/warehouse/new_taxi_data
>>>
```

Finally, print out the schema of the new Hive table to verify that the new columns have been added:

```
>>> df2.printSchema()
root
 |-- trip_distance: double (nullable = true)
 ...
 |-- PRIMARY_KEY: string (nullable = true)
 |-- tip_rolling_mean: double (nullable = true)
 |-- fare_tip_rolling_corr: double (nullable = true)

>>>
```

Chapter 3

# BDD Shell SDK Reference

This section provides a reference of the BDD Shell SDK classes and methods.

## BddContext class

The `BddContext` class provides the main entry point of BDD Shell functionality and provides the runtime context of BDD Shell.

### datasets()

The `datasets()` method retrieves the BDD data sets. The syntax is:

```
datasets()
```

The method returns all `BddDatasets` instances that belong to this connected BDD environment, as in this example:

```
>>> dss = bc.datasets()
```

## BddDatasets class

The `BddDatasets` class is a container of `BddDataset`, and provides an iterator to access all data sets of BDD.

### dataset()

The `dataset()` method retrieves the specified data set. The syntax is:

```
dataset(collectionKey, databaseKey)
```

where:

- *collectionKey* is the name of the data set (Dgraph collection).
- *databaseKey* is the name of the Dgraph database to which the data set belongs.

Data sets created in Studio use the same name for both the database and collection keys. You can therefore supply only the collection key as an argument.

For example:

```
>>> dss = bc.datasets()
>>> claims_ds = dss.dataset('default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620')
>>> claims_ds

WarrantyClaims  default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620
default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620  Hive  default.warrantyclaims
```

The first command retrieves the "default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620" data set, while the second one prints the contents of `claims_ds`.

### next()

The `next()` method retrieves the next data set. For example, assume there are two data sets:

```
>>> all_ds = bc.datasets()
>>> all_ds.next()

WarrantyClaims  default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620
default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620  Hive  default.warrantyclaims
>>> all_ds.next()

Taxi_Data       edp_cli_edp_f28fb378-333a-4e01-8b6e-01dc06ba1e14
edp_cli_edp_f28fb378-333a-4e01-8b6e-01dc06ba1e14  Hive  default.taxi_data
>>>
```

## BddDataset class

The methods in the `BddDataset` class return information about a specific data set.

A `BddDataset` instance represents a data set of BDD.

### attribute_config()

The `attribute_config()` method returns the attributes (including their values) of the data set.

Example:

```
>>> ds.attribute_config()
{u'labor_amount': {u'averageLength': 8, u'defaultGroup': u'General', u'eidDimension': False,
u'hasCustomAttribDescription': False, u'displayName': u'Labor_Amount',
...
```

Note the letter "u" prefixed to the attribute names. In Python, the letter "u" means that the string is a Unicode string. In the `attribute_config()` output, the Python Interpreter will print the result automatically, and it will have that "u" for Unicode strings.

### properties()

The `properties()` method returns the data set's metadata properties from the DataSetInventory.

Example:

```
>>> ds.properties()
{'timesViewed': '0', 'sourceName': 'default.warrantyclaims', 'attributeDisplayNames':
'Vehicle_Dealer', 'accessType': 'private', 'tag': None, 'databaseKey':
```

```
...
'fullDataSet': 'true', 'collectionIdToBeReplaced': None, 'authorizedGroup': None}
```

## source()

The `source()` method retrieves the data source of the data set. The data source is from a Hive table, whose name shown by the method result.

Example:

```
>>> ds.source()
Hive    default.warrantyclaims
```

In this example, the `warrantyclaims` Hive table (which is in the `default` database) is the source.

## to_spark()

The `to_spark()` method returns a Spark DataFrame with the source data from the Hive table.

Example:

```
>>> ds.to_spark()
16/03/11 17:05:57 INFO HiveContext: Initializing execution hive, version 1.2.1
16/03/11 17:05:57 INFO ClientWrapper: Inspected Hadoop version: 2.6.0
...
16/03/11 17:06:23 INFO ParseDriver: Parse Completed
DataFrame[vin: string, production_country: string, production_region: string,
make: string, manufacturer: string, model: string, model_year: bigint,
claim_date: string, dealer_geocode: string, vehicle_dealer: string,
dealer_state: string, dealer_city: string, labor_description: string,
commodity: string, complaint: string, part_number: string, sale_date: string,
supplier_country: string, supplier: string, supplier_state: string,
labor_amount: double, part_amount: double, claim_amount: double, PRIMARY_KEY: string]
```

## transform_script()

The `transform_script()` method returns the transformation script that is associated with the data set. If the data set does not have a transformation script, then the operation returns nothing.

## views()

The `views()` method retrieves all the views belonging to a data set. The syntax is:

```
views()
```

For example:

```
>>> ds.views()
default_edp_79f6509b-2773-48b8-988f-e1ed51d649ba   2 views
```

The result includes the name of the data set and the number of views it has.

## BddDataSource class

The `BddDataSource` class represents the data source of a data set.

### to_spark()

The `to_spark()` method creates a Spark DataFrame containing the data of this data source.

Example:

```
>>> claims_ds = dss.dataset('default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620')
>>> claims_ds.to_spark()
16/03/31 16:49:21 INFO HiveContext: Initializing execution hive, version 1.2.1
16/03/31 16:49:21 INFO ClientWrapper: Inspected Hadoop version: 2.6.0
...
16/03/31 16:49:41 INFO ParseDriver: Parse Completed
DataFrame[vin: string, production_country: string, production_region: string,
make: string, manufacturer: string, model: string, model_year: bigint,
claim_date: string, dealer_geocode: string, vehicle_dealer: string,
dealer_state: string, dealer_city: string, labor_description: string,
commodity: string, complaint: string, part_number: string, sale_date: string,
supplier_country: string, supplier: string, supplier_state: string,
labor_amount: double, part_amount: double, claim_amount: double, PRIMARY_KEY: string]
```

## BddViews class

The `BddViews` class is a container of `BddViews`, and provides an iterator to access all views of a dataset.

### find_views()

The `find_views()` method retrieves a list of views specified by the view type. The syntax is:

```
find_views(view_type=type)
```

where *type* is one of:

- `DgraphView` represents a Dgraph View of the data set.
- `HdfsView` represents an HDFS View of the data set.

For example, to return a `DgraphView`:

```
>>> claims_ds = dss.dataset('default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620')
>>> views = claims_ds.views()
>>> from bdd_sdk import DgraphView
>>> dgraph_views = views.find_views(view_type=DgraphView)
>>> dgraph_views
[DgraphView
default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620.default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620
bus20.example.com 7003]
>>>
```

A similar procedure would apply to an `HdfsView`.

### next()

The `next()` method retrieves the next view. For example:

```
>>> claims_ds = dss.dataset('default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620')
>>> views = claims_ds.views()
>>> views.next()
DgraphView
default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620.default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620
bus20.example.com 7003
>>> views.next()
HdfsView        /user/bdd/edp/data/.collectionData
/default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620.default_edp_e35f9cbe-96c7-4183-8485-71459b8bd620
>>>
```

## DgraphView class

The `DgraphView` class represents a Dgraph View of the data set.

### view_type()

The `view_type()` method returns a string that represents a Dgraph View.

## HdfsView class

The `HdfsView` class represents an HDFS View of the data set.

### view_type()

The `view_type()` method returns a string that represents an HDFS View.

Appendix A

# Installation

This section describes how to install BDD Shell.

## Prerequisites

This topic lists the software requirement for the installation of BDD Shell.

You must install the following components on the Admin Server on which BDD Shell will be running.

### Installing Python

Python 2.7 must be installed on the Admin Server and the YARN NodeManager servers.

The Python installation procedure varies slightly depending on whether you are installing on a BDA (Big Data Appliance) or non-BDA platform.

#### Installing Python on non-BDA platforms

Python 2.7 must be installed on the Admin Server and the YARN NodeManager servers.

You can install either the Anaconda version of the Python distribution or the Miniconda version that contains the conda package manager and Python. If you intend to use 3rd-party packages for Python, then Anaconda is recommended as it includes Python, pandas, Jupyter, and over 150 other Python modules. Using Anaconda thus simplifies the installation of Python and those 3rd-party modules.

To install the Python package on the Admin Server:

1. Download the Python 2.7 installer:
   - For the Anaconda 2.5 version, download from *https://www.continuum.io/downloads#_unix*.
   - For the Miniconda version, download from *http://conda.pydata.org/miniconda.html*.
2. Run the Python installer, as documented in the download page.

As a result, you should have Python installed on the Admin Server machine. For example, it could be installed in the `/localdisk/anaconda2` directory.

After the Anaconda directory is created, you set that directory as the `LOCAL_PYTHON_HOME` in the `bdd-shell.conf` file, as in this example:

```
## Path to the python 2.7 and 3rd party libs on the server running BDD Shell
## Suggest to use Anaconda 2.5
LOCAL_PYTHON_HOME=/localdisk/anaconda2
```

Also set the location of the Python executable as the `SPARK_EXECUTOR_PYTHON` property in the `bdd-shell.conf` file, as in this example:

```
## Path to the python 2.7 binary on the Yarn Node Manager servers
SPARK_EXECUTOR_PYTHON=/localdisk/anaconda2/bin/python
```

Optionally, install any 3rd-party Python packages based on your needs, such as Pandas and Jupyter. Installing Anaconda 2.5, can simplify the installation of Python and 3rd-party packages.

### Installing Python on BDA platforms

You can install Python 2.7 in addition to the default Python 2.6 on a BDA platform.

By default, BDA has Python 2.6 installed. However, BDA can support two versions of Python: 2.6 for Mammoth and 2.7 for BDD Shell. Note that Python 2.6 must remain as the default version used on that node.

To install Python 2.7 for BDD Shell on BDA:

1. Download the latest Anaconda from: *https://www.continuum.io/downloads*

   ```
   $ wget https://repo.continuum.io/archive/Anaconda2-4.2.0-Linux-x86_64.sh
   ```

2. Install Anaconda in the same path on the entire cluster.

   ```
   $ bash Anaconda2-4.2.0-Linux-x86_64.sh
   ```

   Note that at the last step of the installation, type "no" to keep the original Python environment:

   ```
   installation finished.
   Do you wish the installer to prepend the Anaconda2 install location
   to PATH in your /home/oracle/.bashrc ? [yes|no]
   [no] >>> no
   ```

3. Export Anaconda to the PATH environment variable:

   ```
   $ export PATH=[anaconda_home]/bin:$PATH
   ```

## Spark 1.5 or 1.6 installation

Apache Spark 1.5.x or 1.6.x must be installed on the machine that is configured as the BDD Admin Server.

### Installing Spark on CDH and HDP

This topic describes how to install Apache Spark on a CDH or HDP instance.

You must download the Apache Spark which matches the Spark version of your CDH (Cloudera Distribution for Hadoop) or HDP (Hortonworks Data Platform) cluster. You can get the Spark version information from your installed cluster or from the CDH or HDP official website.

To install the Spark 1.5.x or 1.6.x component:

1. Create a directory on the Admin Server machine to store the Spark software component.

   For example, create a `/localdisk/hadoop` directory.

2. Download the Spark version which matches your version of Spark:

   (a) In a browser, go to: *http://archive.apache.org/dist/spark/*

   (b) Navigate to the directory that matches your version of Spark.

   (c) Download the `<spark-version>-bin-hadoop2.6.tgz` file.

   For example, for CDH with Spark 1.6, download *http://archive.apache.org/dist/spark/spark-1.6.0/spark-1.6.0-bin-hadoop2.6.tgz*

3. Unpack the archive file into the `/localdisk/hadoop` directory.

   When the file is unpacked, it produces a Spark directory. For example, the CDH version will produce a `spark-1.6.0-bin-hadoop2.6` directory.

After the Spark directory is created, you set that directory as the `SPARK_HOME` property in the `bdd-shell.conf` file, as in this CDH example:

```
## Path to the Spark installation on the server running BDD Shell
SPARK_HOME=/localdisk/hadoop/spark-1.6.0-bin-hadoop2.6
```

## Installing Spark on MapR

This topic describes how to install Apache Spark on a MapR instance.

You must download the Apache Spark which matches the Spark version of your MapR (MapR Converged Data Platform) cluster. You can get the Spark version information from your installed cluster or from the MapR official website.

To install the Spark 1.5.x or 1.6.x component:

1. Make sure you have a MapR client on the BDD Shell machine.

   For instructions, see:
   *http://maprdocs.mapr.com/51/index.html#AdvancedInstallation/SettingUptheClient.html*

2. Get Spark in one of two ways:

   - Either install MapR's Spark:

     ```
     yum install mapr-spark mapr-spark-historyserver
     ```

   - Or copy Spark from the MapR cluster:

     ```
     scp -r bdduser@abc.us.example.com:/opt/mapr/spark/ /opt/mapr/spark/
     ```

3. Copy MapR's `hive-site.xml` file:

   ```
   scp -r bdduser@abc.us.example.com:/opt/mapr/hive/hive-1.2/conf/hive-site.xml /opt/mapr/spark
   /spark-1.6.1/conf/
   ```

## Security considerations

You can control who can run BDD Shell and whether it can operate in a Kerberos-enabled environment.

The use of a custom Linux group and/or Kerberos support will enhance the security of BDD Shell.

### Group access control for BDD Shell users

The person who installs BDD Shell will be given Owner permissions to it.

Optionally, you can create a Linux group to provide access control to the BDD Shell scripts. Users who are members of this group will be able to run BDD Shell. The group name can be any of your choosing. Note that for security purposes, it is recommended that you create a dedicated group with limited permissions.

After the group is created, you set the group name as the `GROUP` property in the `bdd-shell.conf` file, as in example for a group named "bdd-shell":

```
#
# OS group name whose member could run the BDD Shell. The group must exist before installation.
(Optional)
GROUP=bdd-shell
```

As the property description notes, the group must exist before you can run the BDD Shell installer, as the installers validates the existence of the group.

### Kerberos support

BDD Shell can run in a BDD cluster that has been enabled for Kerberos support. The BDD cluster must be enabled for Kerberos before you can the BDD Shell installer. The reason is that the BDD Shell installer picks up its Kerberos settings from the cluster's `bdd.conf` configuration file.

For information on enabling Kerberos at BDD installation time, see the *Installation Guide*. For information on enabling Kerberos after BDD has been installed, see the *Administrator's Guide*.

## Installer configuration file

The BDD Shell installation script reads properties from a predefined configuration file for the installation.

The `bdd-shell.conf` configuration file is located in the `BDD_HOME/bdd-shell` directory. You can use any text editor to edit the configuration file, modifying the parameters listed in the following table.

You provide the name of this file when you run the orchestration script.

### Configuration settings

The settings for the configuration parameters are as follows. The listed defaults are provided in the configuration file.

| Configuration parameter | Possible settings |
|---|---|
| `SPARK_HOME` | The absolute path to the Spark installation on the server running BDD Shell. Default: none |

| Configuration parameter | Possible settings |
|---|---|
| SPARK_EXTRA_CLASSPATH | You can specify the absolute paths of other extra JAR files, with each JAR path separated by a colon(":"). For example, if you want to operate on CSV files, you can configure the CSV JARs here. The setup script will copy the JARs to `BDD_HOME/common/bdd-shell/lib` on localhost and every YARN NodeManager server. Default: none |
| SPARK_DRIVER_CORES | The number of CPU cores used by the BDD Shell Spark job driver. Default: 1 |
| SPARK_DRIVER_MEMORY | The maximum memory heap size to use for the BDD Shell job driver process, in the same format as JVM memory strings (such as 512m, 2g, 10g, and so on). Default: 1g |
| SPARK_EXECUTER_INSTANCES | The number of Spark job executors to use by BDD Shell. Default: 2 |
| SPARK_EXECUTOR_CORES | The maximum number of CPU cores to use for the BDD Shell Spark job executor. Default: 1 |
| SPARK_EXECUTOR_MEMORY | The maximum memory heap size to use for the BDD Shell job executor, in the same format as JVM memory strings (such as 512m, 2g, 10g, and so on). Default: 1g |
| SPARK_EXECUTOR_PYTHON | The absolute path to the Python 2.7 binary on the YARN NodeManager servers. Default: none |
| LOCAL_PYTHON_HOME | The absolute path to the Python 2.7 and 3rd party libraries on the server running BDD Shell. Default: none |
| GROUP | The Linux group name whose members can run BDD Shell. The group must exist before installation. Default: none |

## Configuration file example

The following is an example of the installation configuration file::

```
## Path to the Spark installation on the server running BDD Shell
SPARK_HOME=/localdisk/hadoop/spark-1.5.0-bin-hadoop2.6

## Absolute paths of extra jars on the server running BDD Shell.
SPARK_EXTRA_CLASSPATH=

## The number of cores used by the BDD Shell Spark job driver.
SPARK_DRIVER_CORES=1

## The maximum memory heap size for BDD Shell Spark job driver.
SPARK_DRIVER_MEMORY=1g

## The number of Spark job executors used by the BDD Shell
SPARK_EXECUTER_INSTANCES=2

## The number of cores used by the BDD Shell Spark job executor
```

```
SPARK_EXECUTOR_CORES=1

## The maximum memory heap size for BDD Shell Spark job executor.
SPARK_EXECUTOR_MEMORY=1g

## Path to the python 2.7 binary on the Yarn Node Manager servers.
SPARK_EXECUTOR_PYTHON=/localdisk/anaconda2/bin/python

## Path to the python 2.7 and 3rd party libs on the server running BDD Shell.
LOCAL_PYTHON_HOME=/localdisk/anaconda2

## OS group name whose members can run BDD Shell.
GROUP=bdd-shell
```

## Running the installation script

Once you have satisfied the BDD Shell requirements and updated the `bdd-shell.conf` configuration file, you can run the installation script.

After you have installed the BDD Shell prerequisites and updated the

To run the BDD Shell installation script:

1. From the Linux command prompt, navigate to the `BDD_HOME/bdd-shell` directory.
2. Run the `setup.sh` script:

   ```
   ./setup.sh
   ```

If successful, the installation script outputs these messages:

```
-bash-4.1$ ./setup.sh
[bus2014] Validating pre-requisites...
[bus2014] Validation Success
[bus2014] Setting up BDD Shell...
[bus2014] Setup Success
```

You can start BDD Shell as described in *Starting BDD Shell on page 8*.

## Post-installation tasks

The following sections describe tasks you may perform after you install BDD Shell.

*MapR post-installation tasks*

*Configuring KMS high availability*

*Configuring spark-defaults.conf*

*Workaround for topology issue*

## MapR post-installation tasks

These MapR tasks should be done after installing BDD Shell.

### Copy the ResourceManager proxy JAR

Copy the `hadoop-yarn-server-web-proxy-2.7.0-mapr-<version>.jar`:

```
scp bdduser@abc.us.example.com:/opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/yarn
/hadoop-yarn-server-web-proxy-2.7.0-mapr-1602.jar /opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/yarn/
```

Also make sure that the BDD Shell user has write privileges for the Hive warehouse folder.

### Workaround for BddDataset conversion error

When converting a `BddDataset` into a Spark DataFrame, you may experience this error:

```
>>> dss = bc.datasets()
>>> ds = dss.dataset('default_edp_xxx')
>>> df = ds.to_spark()

Exception: ("You must build Spark with Hive. Export 'SPARK_HIVE=true' and run build/sbt assembly",
Py4JJavaError(u'An error occurred while calling None.org.apache.spark.sql.hive.HiveContext.\n',
JavaObject id=o59))
```

To workaround this problem, open the `hive-site.xml` file in the `$BDD_HOME/bdd-shell/hadoop-conf` directory. Check to see if the file has these two property settings:

```
hive.metastore.schema.verification = true
hive.execution.engine = tez
```

If so, then change those properties to:

```
hive.metastore.schema.verification = false
hive.execution.engine = spark
```

## Configuring KMS high availability

BDD Shell can be configured to run in a Hadoop environment that has been set up for Key Trustee KMS high availability (HA).

This procedure assumes that you have already set up Key Trustee KMS (Key Management Server) high availability and that you have already installed BDD.

The Apache Spark that is used by BDD Shell does not support the semi-colon delimited list of KMS nodes that are set up for the HA purpose. Apache Spark only supports one KMS node. The workaround is to modify the `core-sites.xml` and `hdfs-sites.xml` configuration files used by BDD Shell and replace the list of KMS nodes with only one KMS node.

To modify the BDD Shell Hadoop configuration files for KMS high availability:

1. Install BDD Shell.

   The installation creates a directory named `hadoop-conf` in the BDD Shell root directory, into which the Hadoop configuration files are copied from the `$BDD_HOME/common/hadoop/conf` directory.

2. In the hadoop-conf directory, use a text editor to open the core-site.xml file and adjust the hadoop.security.key.provider.path property to refer to only one machine:

   Before:

   ```
   ...
      <property>
         <name>hadoop.security.key.provider.path</name>
         <value>kms://http@kms01.example.com;kms02.example.com:16000/kms</value>
      </property>
   ...
   ```

   After:

   ```
   ...
      <property>
         <name>hadoop.security.key.provider.path</name>
         <value>kms://http@kms01.example.com:16000/kms</value>
      </property>
   ...
   ```

3. In the same hadoop-conf directory, open the hdfs-site.xml file and make a similar adjustment to the hadoop.security.key.provider.path property to refer to only one machine:

   Before:

   ```
   ...
      <property>
         <name>dfs.encryption.key.provider.uri</name>
         <value>kms://http@kms01.example.com;kms02.example.com:16000/kms</value>
      </property>
   ...
   ```

   After:

   ```
   ...
      <property>
         <name>dfs.encryption.key.provider.uri</name>
         <value>kms://http@kms01.example.com:16000/kms</value>
      </property>
   ...
   ```

## Configuring spark-defaults.conf

This topic describes how to prevent Spark errors when making OS system calls from BDD Shell.

You can make call() or os.system() calls from BDD Shell. For example, you can call the DP CLI, as described in *Creating a new BDD data set from BDD Shell on page 17*.

However, if the spark-defaults.conf is misconfigured, making such a call may result in an error similar to this example:

```
org.apache.spark.SparkException: Job aborted due to stage failure:
Exception while getting task result: com.esotericsoftware.kryo.KryoException:
java.lang.UnsupportedOperationException
```

The error occurs because the Spark spark-defaults.conf file contains the following property:

```
spark.serializer=org.apache.spark.serializer.KryoSerializer
```

Remove this property and re-try the operation.

## Workaround for topology issue

A cluster topology problem can occur on some Hadoop installations. It can also occur on installations configured for rack awareness.

An example of this issue is if BDD Shell fails to start and throws an error similar to this example:

```
16/06/13 22:42:27 WARN ScriptBasedMapping: Exception running /etc/hadoop/conf
/topology_script.py 10.152.248.55
java.io.IOException: Cannot run program "/etc/hadoop/conf/topology_script.py" (in directory
"/scratch/bddSetup/Oracle/Middleware/BDD-1.4.0.37.1220/bdd-shell"): error
=2, No such file or directory
        at java.lang.ProcessBuilder.start(ProcessBuilder.java:1047)
        at org.apache.hadoop.util.Shell.runCommand(Shell.java:485)
        at org.apache.hadoop.util.Shell.run(Shell.java:455)
        at org.apache.hadoop.util.Shell$ShellCommandExecutor.execute(Shell.java:715)
        at
org.apache.hadoop.net.ScriptBasedMapping$RawScriptBasedMapping.runResolveCommand(ScriptBasedMapping.j
ava:251)
        at
org.apache.hadoop.net.ScriptBasedMapping$RawScriptBasedMapping.resolve(ScriptBasedMapping.java:188)
        at org.apache.hadoop.net.CachedDNSToSwitchMapping.resolve(CachedDNSToSwitchMapping.java:119)
        at org.apache.hadoop.yarn.util.RackResolver.coreResolve(RackResolver.java:101)
        at org.apache.hadoop.yarn.util.RackResolver.resolve(RackResolver.java:81)
...
```

The issue is caused because the path value of topology in `core-site.xml` does not exist. To solve the issue, delete the topology setting item from the `core-site.xml` file.

Alternatively, you can copy the corresponding file to the related path assigned in the topology setting.

# Index