

Dell EMC Ready Solutions for Data Analytics - Spark on Kubernetes

Version 1.0.0.0

Reference Architecture

H18107

February 2020

Copyright © 2020 Dell Inc. or its subsidiaries. All rights reserved.

Dell believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS-IS.” DELL MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. USE, COPYING, AND DISTRIBUTION OF ANY DELL SOFTWARE DESCRIBED IN THIS PUBLICATION REQUIRES AN APPLICABLE SOFTWARE LICENSE.

Dell Technologies, Dell, EMC, Dell EMC and other trademarks are trademarks of Dell Inc. or its subsidiaries. Other trademarks may be the property of their respective owners. Published in the USA.

Dell EMC
Hopkinton, Massachusetts 01748-9103
1-508-435-1000 In North America 1-866-464-7381
www.DellEMC.com

CONTENTS

Figures		5
Tables		7
Chapter 1	Executive summary	9
	Introduction.....	10
	Why Spark and Hadoop?.....	10
	Why Kubernetes?.....	11
Chapter 2	Inventory management use case	13
	Overview.....	14
	A use case story.....	14
	Starting point.....	15
	Proposed solution.....	15
	New challenges.....	16
	Use case workflows.....	16
	Query a data lake.....	16
	Run a data preparation job.....	17
	Create a model.....	18
	Validate the model.....	20
Chapter 3	Spark environment	23
	Overview.....	24
	Clustered execution.....	24
	Spark and Kubernetes.....	25
	Red Hat OpenShift Container Platform.....	25
	Running Spark on Kubernetes.....	26
	Spark-submit.....	27
	Spark Operator.....	28
	Interactive and batch execution.....	29
	Scalability and resource management.....	29
	Storage interfaces.....	30
	Kubernetes volumes.....	30
	HDFS storage.....	30
	Database storage.....	30
	Spark image.....	30
	Building Spark.....	31
	Building the image.....	31
	Pushing the image to a repository.....	31
	Additional images.....	32
	Jupyter image.....	32
	Spark history server image.....	33
	Monitoring.....	34
Chapter 4	Infrastructure summary	37
	Software.....	38
	Hardware infrastructure.....	38

	Master node server configuration.....	39
	Compute node server configuration.....	39
	Network switches.....	40
Chapter 5	Conclusions	41
	Document summary	42
Chapter 6	References	43
	Dell EMC documentation.....	44
	Apache Spark documentation.....	44
	Dell EMC Customer Solution Centers.....	44
	Dell Technologies InfoHub.....	44
	More information.....	44

FIGURES

1	Reading datafiles from a data lake.....	17
2	Datafiles loaded into Spark memory.....	17
3	Join the datasets.....	17
4	Parse the input columns.....	18
5	Parse the input columns (continued).....	18
6	Split the dataset.....	18
7	Add new columns.....	19
8	Code for checking correlation of output to input.....	19
9	Resulting graph.....	19
10	Select the high correlation input features.....	19
11	Train the model.....	20
12	Apply the trained model.....	20
13	Plot a graph.....	21
14	Spark execution model.....	24
15	OpenShift architecture.....	25
16	Typical Kubernetes cluster.....	26
17	Spark history server.....	34
18	OpenShift pod monitoring.....	35
19	Analytics infrastructure.....	38

TABLES

1	Software components.....	38
2	Master node configuration.....	39
3	Compute node configuration.....	40

CHAPTER 1

Executive summary

Many lessons learned in the last four to five decades of using computers to process data have led to Dell EMC's current enthusiasm for advanced analytics solutions. Machine learning, deep learning, and artificial intelligence are beginning to solve significant business and societal challenges.

- [Introduction](#)..... 10

Introduction

This guide outlines a practical approach to tool selection and model creation for projects with machine learning and deep learning requirements, using technology available today.

This guide describes how to use a combination of open-source projects and commercial software, including:

- Spark
- Spark SQL
- MLib
- BigDL
- Kubernetes

It covers both issues of interest to both data scientists and IT professionals. It also describes how to use Docker and Kubernetes to manage the infrastructure that can be integrated with software developers working on cloud-native applications. It uses an inventory management use case story for a fictitious large online and physical retailer. This use case story solves the technology puzzle one analytics pipeline stage at a time.

Why Spark and Hadoop?

One of the main reasons for Dell EMC's continued interest in Hadoop for advanced analytics is its breadth and flexibility.

Organizations can combine a few projects that handle everything from data ingestion to data cleansing, to model development and model hosting for inference. A platform that enables IT, data engineers, business analysts, and data scientists to focus investments and share resources could couple:

- Spark for in-memory for compute scale
- HDFS for on-disk storage scale

The two main challenges when moving from data analytics on a single system to a scale-out approach are:

- How to distribute the computations along with a subset of the data to a coordinated and interconnected set of compute nodes
- How to scale out data storage to keep up with the I/O demands of multiple compute nodes

The original two projects that made up the first version of Hadoop solved both problems in the mid to late 2000s. The original distributed compute engine, MapReduce, was incapable of performing advanced analytics such as machine learning or deep learning. It has been largely replaced and augmented with new distributed computing frameworks, with Spark being the most important for data science.

Spark started with a concept familiar to everyone that works in data science - the data frame. An approach was then devised to distribute it across many systems. This took advantage of the combined memory and computing cores so that data scientists did not have to change the way they traditionally work with data.

Why Kubernetes?

Data science has expanded beyond the computing power of a single machine to escape the memory and computing core limits of ubiquitous, inexpensive x86 systems.

Managing massive, distributed systems that handle the scaling need of data scientists working with increasing data volumes presents new challenges for architects and IT operations professionals. This area is where Kubernetes makes a significant contribution.

The relationship between Spark and Kubernetes is conceptually simple. Data scientists want to run many Spark processes that are distributed across multiple systems to have access to more memory and computing cores. Using container virtualization like Docker to host those Spark processes, Kubernetes orchestrates the creation, placement, and life cycle management of those processes across a cluster of x86 servers. There are many Kubernetes implementations available today, so later chapters discuss using OpenShift Kubernetes with Kubernetes, and how that compares to other options.

CHAPTER 2

Inventory management use case

This use case story is common to many large retail organizations who require demand forecasting and inventory management.

- [Overview](#) 14
- [A use case story](#) 14
- [Starting point](#) 15
- [Proposed solution](#) 15
- [New challenges](#) 16
- [Use case workflows](#) 16

Overview

This use case could focus on any industry that wrestles with vast amounts of structured and unstructured data every day, including:

- Manufacturing
- Health care
- Financial
- And many more

Any of those industries would make an equally compelling story about the reasons for using Spark. One of the primary reasons this guide focuses on the retail use case is the availability of a large, multi-table, simulated dataset. This dataset has orders with line items, customers, and sales regions (geography).

Also, broad familiarity with the retail experience (everyone shops and knows about stock outages) is an attractive feature that makes storytelling easier. Finally, the generated data schema could be staged in two or more simulated source systems. Then, it could be brought back together to demonstrate the use of a multi-stage analytics pipeline.

This use case does not claim that the data engineering or data science that it demonstrates is representative of the challenges that must be solved. The simulated data is not intended to evaluate the relative robustness of any machine learning or deep learning techniques. Its goal is to place realistic demands on lab resources while processing the pipeline.

Dell EMC built the lab demonstration to showcase the entire Spark and Hadoop toolset for implementing full-featured data pipelines. As in the later discussion of using Spark for interactive analysis, the relationships between variables in the dataset are artificial and simplistic. That does not stop the use case from showing how the platform handles distributed analytics for large datasets. It only impacts the ability to *find* interesting associations in the data.

This guide discusses the important elements of a typical analytics data pipeline, including:

- Data ingestion
- Joining and filtering large related tables
- Restructuring data to fit the input requirements of common machine learning models
- Distributed model training
- Configuring hardware resources
- Installing software tools

A use case story

The analytics *pipeline* is a useful metaphor for how data engineers and data scientists work through:

1. Data ingestion
2. Data cleansing and transformation
3. Data merging and testing

Dell EMC developed a story for explaining both the challenges and most successful approaches for each of these steps in the analytic pipeline. That required a large and complex dataset to showcase real-world challenges and solutions, without incurring the cost and time of solving an enterprise-class problem.

Dell EMC built a story describing a machine learning model-based approach to inventory management for a retailer with hundreds of thousands of individual Stock Keeping Units (SKUs). It

shows how Spark and related technologies provide a total solution for solving real-world data analytics problems, by developing individual pieces using a simplified dataset and requirements.

Starting point

This use case story begins with a fictitious retail company that has a catalog with hundreds of thousands of SKUs, grouped into five market segments:

- Automobile care and accessories
- Building materials
- Home furnishings
- Power tools
- Household goods and appliances

Their sales demand forecast used for product ordering is based on a manual roll-up of segment managers' estimates, in all sales areas. This roll-up is based on experience and knowledge of local conditions. This process is slow, and often produces incomplete forecasts when staff miss submission deadlines. The company is also experiencing high inventory carrying costs. The estimates of both the area and segment managers and the purchasers tend to overestimate demand in order to avoid stock-outs.

Management wants to add a model-based demand forecasting option to the planning process. They want it to be based on data from their sales order and supply purchasing systems. They were told that estimating individual models for each product is challenging, given the number of SKUs they manage and sales sparsity of many catalog items. The company hired an inventory management consultant who suggested a process called *hierarchical forecasting*, that is common for organizations with so many products. The consultant also recommended that the company may need new technology to implement the new modeling system.

The company has extensive experience with enterprise-class relational database management systems including a Massively Parallel Processing (MPP) database. They have recently started using a Hadoop Distributed File System (HDFS) data lake to offload some analytics from the overloaded Relational Database Management System (RDBMS). Spark is the most popular tool for accessing data in HDFS. Management wants any new analytics processing programs for the inventory planning system to be developed using primarily Hadoop and Spark, if possible. IT management wants to avoid bringing in new technology silos.

Proposed solution

The consultants have confirmed that product forecasting with such a large catalog is complicated.

They considered implementing a multi-tier approach where high-profit products would be modeled individually, and the current system would be kept for everything else. Management rejected that for the first round of development and choose to instead implement a consistent two-stage approach for all products. This approach is based on first developing a model to forecast aggregate sales for each market segment.

The proposed design uses a classical time-series approach for the aggregate forecast. The aggregate forecasts of daily sales dollars are allocated to individual products, and based on their historical contribution to total segment sales. Sales for each product are converted from dollar values to units based on recent average prices. The individual product forecasts are compared against current inventory to estimate when the next out of stock event is likely to occur for each product.

New challenges

The amount of data and number of computations that are required for an organization of this size exceeds the capability of any single scale-up machine architecture.

The data preparation, model training, and subsequent allocation and conversion calculations all involve tables with tens of millions of rows. That complexity makes operations like joining, aggregating, and filtering processor and memory demanding. The team is confident that choosing Spark for data ingestion and transformation workload is their best option. They have no experience with data modeling or model inferencing, using projects from the Spark ecosystem, like MLib for machine learning or Intel BigDL for deep learning. Most of the team has been working with Python and R model training on smaller datasets, using a single workstation.

The team also lacks experience in managing applications that use machine learning models in production. Stock outages are known to influence customer perceptions that can last long into the future. The data science team must define a workflow that incorporates new information into their models within days of becoming available. They must have automation that can run without human intervention for normal operations. They must be notified quickly if the models produce forecasts that differ significantly from historical trends. The platform and tools they choose for this project must be easily automated, and incorporate monitoring.

Use case workflows

The inventory management use case relies upon four workflows to obtain its results.

These workflows, in the order they are performed, include:

1. [Query a data lake](#) on page 16
2. [Run a data preparation job](#) on page 17
3. [Create a model](#) on page 18
4. [Validate the model](#) on page 20

Query a data lake

This topic describes how to obtain data from a data lake using Spark.

Before you begin

Before querying a data lake with Spark you must:

1. Connect to the data lake platform.
2. Locate the datafiles in the data lake platform.
3. Log in to the Kubernetes platform.
4. Verify network connectivity between the data lake and the Kubernetes platform.

About this task

This procedure loads datafiles from the data lake into the Spark memory.

 **Note:** The use case workflows in this reference architecture use the `python` programming language.

Procedure

1. Create a `SparkContext` object for the entry point to Spark.
2. Use that object to perform SQL operations that read the datafiles, in `.csv` format, from the data lake.

Figure 1 Reading datafiles from a data lake

```
In [ ]: #####sparkContext#####
sc= SparkContext.getOrCreate()
sqlContext = SQLContext(sc)

###Data Source Paths
path_to_customer_file = 'hdfs://192.168.11.70:/user/root/customer.csv'
path_to_lineitem_file = 'hdfs://192.168.11.70:/user/root/lineitem.csv'
path_to_order_file = 'hdfs://192.168.11.70:/user/root/orders.csv'

###Reading of CSV data #####
customer = sqlContext.read.csv(path_to_customer_file, header=True, inferSchema = True)
lineitem = sqlContext.read.csv(path_to_lineitem_file, header=True, inferSchema = True)
order = sqlContext.read.csv(path_to_orders_file, header=True, inferSchema = True)
```

Results

The datafiles are now loaded into the Spark memory.

Figure 2 Datafiles loaded into Spark memory

```
In [4]: customer.show(n=5)
```

	c_custkey	c_mktsegment	c_nationkey	c_name	c_address	c_phone	c_acctbal	c_comment
e...	1	BUILDING	8	Customer#000000001	KwX3hMHjZ6 937-241-3198	3560.03		ironic excuses d
c...	2	MACHINERY	16	Customer#000000002	ioUn,eqTTXOdo 906-965-7556	7550.21		final express ac
s...	3	FURNITURE	11	Customer#000000003	YddJqmIdouNT9Yj 328-750-7603	-926.96		carefully expres
s...	4	FURNITURE	24	Customer#000000004	ie7PADWuxr4pR5f9e... 127-505-7633	-78.75		silent packages
a...	5	MACHINERY	4	Customer#000000005	h3yhvBTvbF2IJPzTK... 322-864-6707	7741.9		sllyly special fr

only showing top 5 rows

Run a data preparation job

This topic describes how to prepare the data now in Spark dataframes for model testing and training.

Before you begin

Before transforming the dataframes, you must ensure that they are properly loaded and visible in Spark.

About this task

After all the dataframes are loaded, this procedure performs Extract, Transform, and Load (ETL) operations to create one coherent file from three different datafiles. It then splits the data into test and train sets. The train set is used to train the model; its performance is measured on the test dataset.

Procedure

1. Join the various dataframes on the join key to create a single data table.

Figure 3 Join the datasets

```
In [3]: sales = order.join(customer, order.o_custkey == customer.c_custkey, how = 'inner')
sales = sales.join(lineitem, lineitem.l_orderkey == sales.o_orderkey, how = 'full')
sales = sales.where('c_mktsegment == "BUILDING").select('l_quantity', 'o_orderdate')

sales = sales.groupBy('o_orderdate').agg({'l_quantity': 'sum'}) .withColumnRenamed("sum(l_quantity)", "TOTAL_SALES") .wi
sales2 = sales.withColumn("ORDERDATE", sales["ORDERDATE"]).withColumn("TOTAL_SALES", sales["TOTAL_SALES"].cast("float"))
sales = sales2
```

2. Parse the input columns to add additional input columns.

Figure 7 Add new columns

```
In [6]: training = training.withColumn('Class', lit(10))
testing = testing.withColumn('Class', lit(10))

#Add Running Mean of the distribution as the input to the model
windowval = (Window.partitionBy('Class').orderBy('DATE')
              .rangeBetween(Window.unboundedPreceding, 0))
training = training.withColumn('Cumi_Avg', f.avg('PREV_SALES').over(windowval))
testing = testing.withColumn('Cumi_Avg', f.avg('PREV_SALES').over(windowval))

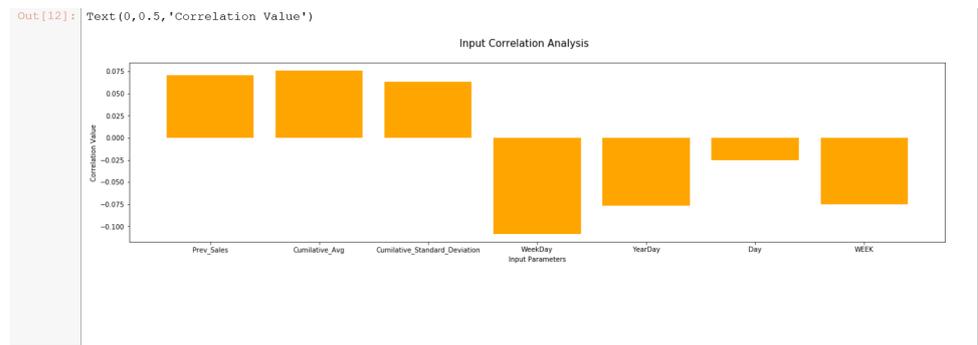
#Adding Running Standard Deviation of the distribution as the input to the model
training = training.withColumn('Cumi_Var', f.stddev('PREV_SALES').over(windowval))
testing = testing.withColumn('Cumi_Var', f.stddev('PREV_SALES').over(windowval))
columns = training.columns
for column in columns:
    training = training.withColumn(column, F.when(F.isnan(F.col(column)), None).otherwise(F.col(column)))
columns = testing.columns
for column in columns:
    testing = testing.withColumn(column, F.when(F.isnan(F.col(column)), None).otherwise(F.col(column)))
training = training.where(training["Cumi_Var"].isNotNull())
testing = testing.where(testing["Cumi_Var"].isNotNull())
```

2. Check the correlation of the output variable to all the input features.

Figure 8 Code for checking correlation of output to input

```
In [12]: print(df.corr())
z = df.corr()
z = z.iloc[1]
z = z[['Prev_Sales', 'Cumulative_Avg', 'Cumulative_Standard_Deviation', 'WeekDay', 'YearDay', 'Day', 'WEEK']]
fig = plt.figure(figsize=(22,5))
plt.bar(['Prev_Sales', 'Cumulative_Avg', 'Cumulative_Standard_Deviation', 'WeekDay', 'YearDay', 'Day', 'WEEK'], z, color='orange')
fig.suptitle('Input Correlation Analysis', fontsize=15)
plt.xlabel('Input Parameters', fontsize=10)
plt.ylabel('Correlation Value', fontsize=10)
```

Figure 9 Resulting graph



3. Select the input features with high correlation values, and define a model.

Figure 10 Select the high correlation input features

```
In [13]: featuresCols = training.columns
featuresCols.remove('TOTAL_SALES')
featuresCols.remove('DATE')
#featuresCols.remove("DAY")
featuresCols.remove("YDAY")
#featuresCols.remove("WEEK")
featuresCols.remove("Class")
#featuresCols.remove("WDAY")
featuresCols.remove("EXPPP")

#lr = GeneralizedLinearRegression(maxIter=1000, featuresCol = "features", labelCol = "TOTAL_SALES", predictionCol="ESTIMATED_SALES")
#rfr = RandomForestRegressor(featuresCol = "features", labelCol = "TOTAL_SALES", predictionCol="ESTIMATED_SALES")
```

Results

The Machine Learning (ML) model is ready to be used for training on train data.

Validate the model

This topic describes how to validate the model on a test dataset, and view the results.

Before you begin

Before validating the model, you must:

1. Ensure that the test and train data have the exact same columns that are selected to be fetched into the model.
2. Ensure that the statistical columns have correct numerical values in each row.

About this task

After the test and train datasets have been created, this procedure:

1. Defines a pipeline for the model
2. Runs the model on the training data for the model to learn
3. Uses the results on the test set to see how it performs

Procedure

1. Concatenate all the input columns as a single vector for the model, and pass it to the model for training.

 **Note:** `pyspark` only accepts a single vector of all the input at once in its ML models.

2. Define the parameters of the model. For example, the depth of the decision tree.
3. Train the model on the training set.

Figure 11 Train the model

```
In [13]: featuresCols = training.columns
featuresCols.remove("TOTAL_SALES")
featuresCols.remove("DATE")
#featuresCols.remove("DAY")
featuresCols.remove("YDAY")
#featuresCols.remove("WEEK")
featuresCols.remove("Class")
#featuresCols.remove("WIDAY")
featuresCols.remove("EXPPP")

va = VectorAssembler(inputCols = featuresCols, outputCol = "features")

#lr = GeneralizedLinearRegression(maxIter=10000, featuresCol = "features", labelCol = "TOTAL_SALES", predictionCol="ESTIMATED_SALES")
rfr = RandomForestRegressor(featuresCol = "features", labelCol = "TOTAL_SALES", predictionCol="ESTIMATED_SALES")
#ir = IsotonicRegression(featuresCol = "features", labelCol = "TOTAL_SALES", predictionCol="ESTIMATED_SALES")

paramGrid = ParamGridBuilder()\
    .addGrid(rfr.maxDepth, [2, 30])\
    .build()
evaluator = RegressionEvaluator(predictionCol="TOTAL_SALES", labelCol = "ESTIMATED_SALES", metricName="rmse")
cv = CrossValidator(estimator=rfr, evaluator=evaluator, estimatorParamMaps=paramGrid)
pipeline = Pipeline(stages=[va, cv])
pipelineModel = pipeline.fit(training)
```

4. Use the trained model on the test set, and check the Root Mean Squared Error (RMSE) value.

Figure 12 Apply the trained model

```
predictions = pipelineModel.transform(testing)
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data (RFR) = %g" % rmse)

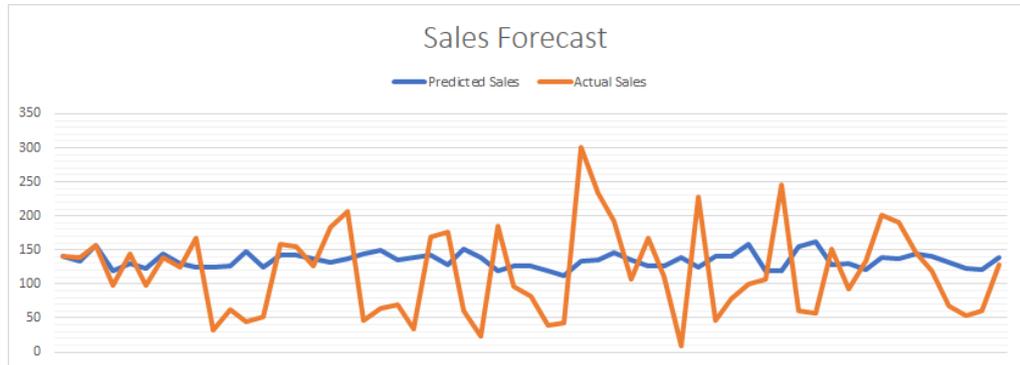
< >
Root Mean Squared Error (RMSE) on test data (RFR) = 70.5023
```

Results

Now you can plot a graph to get better insights on the model results.

Note: The data that is represented in [Figure 13](#) on page 21 was randomly generated. The model found no seasonality patterns, so it produced a flat line forecast.

Figure 13 Plot a graph



CHAPTER 3

Spark environment

This chapter provides guidance on a reference architecture that Dell EMC considers appropriate for general-purpose data analytics involving all stages of an analytics pipeline using Apache Spark.

- [Overview](#) 24
- [Clustered execution](#) 24
- [Spark and Kubernetes](#) 25
- [Red Hat OpenShift Container Platform](#) 25
- [Running Spark on Kubernetes](#) 26
- [Storage interfaces](#) 30
- [Spark image](#) 30
- [Additional images](#) 32
- [Monitoring](#) 34

Overview

Dell EMC used an Inventory management scenario as an example, but many additional analytics scenarios can also be supported with the same environment.

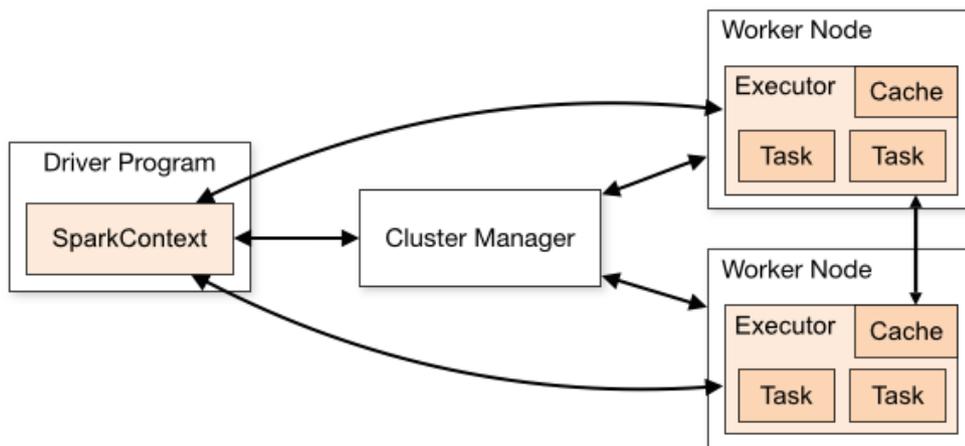
Apache Spark is a unified analytics engine for large-scale data processing. Spark is a flexible, general-purpose analytics framework with language APIs for Scala, Java, Python, and R developers. This design enables Spark to run on many processor architectures. It is compatible with multiple operating systems, and supports several cluster managers for parallel execution. It is extensible and allows additional libraries to be added. It supports diverse data sources. However, this flexibility requires you to make many infrastructure and environment decisions.

Clustered execution

Spark applications run as a collection of multiple processes. Each application consists of a process for the main program (the driver program), and one or more executor processes that run Spark tasks.

These processes are multithreaded. [Figure 14](#) on page 24 illustrates the general Spark execution model.

Figure 14 Spark execution model



Spark relies on a separate Cluster Manager to allocate and manage resources for these processes in multinode environments. Spark supports several cluster managers:

- Spark Standalone - A simple cluster manager included with Spark
- Apache Mesos - A general-purpose cluster manager, invented at the UC Berkeley AMPLab, which invented Spark
- Apache Hadoop YARN - A resource manager developed for the Apache Hadoop project
- Kubernetes – A system for automating deployment, scaling, and management of containerized applications

This reference architecture uses Kubernetes as the cluster manager.

Spark and Kubernetes

Kubernetes has become a popular open-source platform for running containerized workloads. This reference architecture uses Kubernetes for many reasons. The primary factors were:

- Kubernetes can scale from small to large cluster implementations.
- Kubernetes supports a diverse set of workloads, and many vendors are adding support for it to their products.
- Kubernetes can support many of the workloads and software systems that typically complement a Spark analytics environment.
- A containerized, cloud-native model aligns with the developer-centric environment typical of Spark application development.
- There are multiple Kubernetes distributions available, including open source and commercial.
- Spark has support for running under Kubernetes.
- Kubernetes has a vibrant ecosystem.

From a software perspective, Kubernetes has some parallels with the Linux kernel. Although capable on its own, it is highly configurable and extensible. Some extensions are typically desirable for a fully functional production system. There are plug-in components with multiple alternative implementations, and some integration with external systems (such as authentication) is typically required.

This similarity has led to the development of multiple Kubernetes distributions, which provide different perspectives on how Kubernetes should be configured, deployed, managed, and integrated with existing systems. Some of these distributions include Pivotal Container Service (PKS), Docker Enterprise, Rancher Kubernetes Engine, Ubuntu Kubernetes, and Red Hat OpenShift Container Platform.

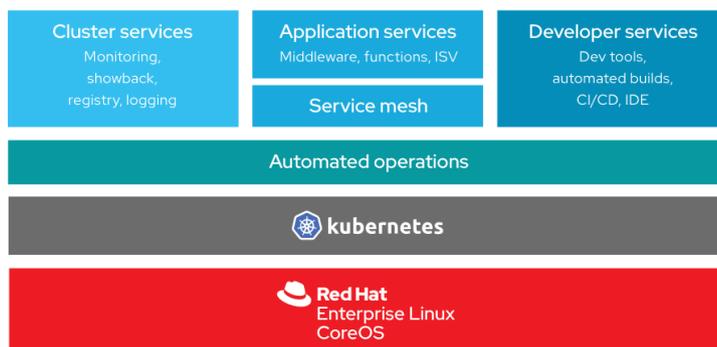
Spark can be run under any of these Kubernetes distributions. In general, the Spark runtime environment is uniform across all the Kubernetes platforms, although the underlying details are different. This reference architecture uses Red Hat OpenShift Container Platform as its reference platform.

Red Hat OpenShift Container Platform

Red Hat OpenShift Container Platform is an enterprise Kubernetes platform.

[Red Hat OpenShift Container Platform](#) extends Kubernetes with cluster services, developers services, and automated operations. [Figure 15](#) on page 25 illustrates the OpenShift architecture.

Figure 15 OpenShift architecture



Dell EMC deployed an OpenShift instance by following the guidelines in the [Dell EMC Ready Stack for Red Hat OpenShift Container Platform 4.2 Design Guide](#). The physical deployment consists of

master nodes that host the platform control plane, and compute nodes that run application containers and supporting OpenShift services. OpenShift master nodes run Red Hat CoreOS. Compute nodes can run either Red Hat Enterprise Linux Server or Red Hat CoreOS. This reference architecture uses Red Hat Enterprise Linux Server.

Some of the high level OpenShift architecture decisions include:

- CRI-O (crio) is used as the container engine.
- Kubernetes Operators are used to implement most OpenShift system services.
- An operator provides and manages an integrated container image repository.

Note: More details about OpenShift can be found in the [OpenShift documentation](#).

Running Spark on Kubernetes

A Spark application generally runs on Kubernetes the same way as it runs under other cluster managers, with a driver program, and executors. Under Kubernetes, the driver program and executors are run in individual Kubernetes pods.

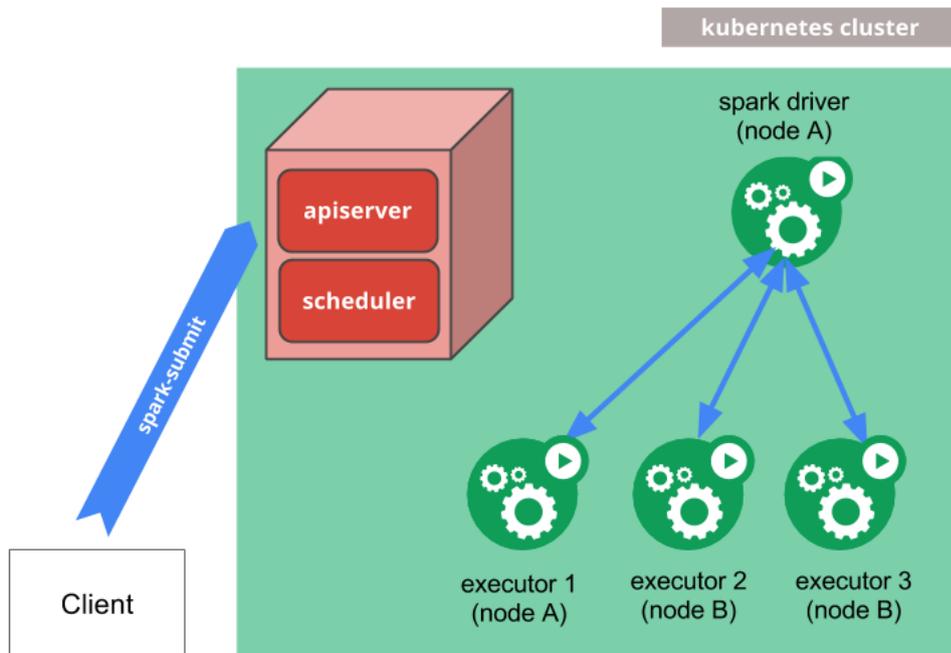
All containers within a Kubernetes pod:

- Are co-located on the same physical node.
- Share an IP address and pod space.
- Can share any Kubernetes volumes.

The Kubernetes scheduler allocates the pods across available nodes in the cluster.

[Figure 16](#) on page 26 illustrates a typical Kubernetes cluster.

Figure 16 Typical Kubernetes cluster



Kubernetes requires users to provide a Kubernetes-compatible image file that is deployed and run inside the containers. This image should include:

- The Spark executable files

- The JVM
- All the other components necessary to run the application

Dell EMC built a custom image for the inventory management example. See [Spark image](#) on page 30 for the details.

Launching a Spark program under Kubernetes requires a program or script that uses the Kubernetes API (using the Kubernetes *apiserver*) to:

- Allocate resources.
- Create the pods.
- Generally bootstrap the program execution.

There are two ways to launch a Spark program under Kubernetes:

- `spark-submit`
- The Spark Operator for Kubernetes

Spark-submit

Dell EMC uses `spark-submit` as the primary method of launching Spark programs.

The `spark-submit` script that is included with Apache Spark supports multiple cluster managers, including Kubernetes. Most Spark users understand `spark-submit` well, and it works well with Kubernetes.

With Kubernetes, the `-master` argument should specify the Kubernetes API server address and port, using a `k8s://` prefix. The `--deploy mode` argument should be `cluster`. All the other Kubernetes-specific options are passed as part of the Spark configuration. You can run `spark-submit` of outside the cluster, or from a container running on the cluster.

[spark-submit](#) on page 27 shows an example of using `spark-submit` to launch a time-series model training job with Spark on Kubernetes.

Example 1 `spark-submit`

```
#!/bin/bash
~/SparkOperator/demo/spark01/spark-2.4.5-SNAPSHOT-bin-spark/ \
bin/spark-submit \
--master k8s://https://100.84.118.17:6443/ \
--deploy-mode cluster \
--name tsmodel \
--conf spark.executor.extraClassPath=/opt/spark/examples/jars/ \
scopt_2.11-3.7.0.jar \
--conf spark.driver.extraClassPath=/opt/spark/examples/jars/ \
scopt_2.11-3.7.0.jar \
--conf spark.eventLog.enabled=true \
--conf spark.eventLog.dir=hdfs://isilon.tan.lab/history/spark-logs \
--conf spark.kubernetes.namespace=spark-jobs \
--conf \
spark.kubernetes.authenticate.driver.serviceAccountName=spark \
--conf spark.executor.instances=4 \
--conf spark.kubernetes.container.image.pullPolicy=Always \
--conf spark.kubernetes.container.image=infra.tan.lab/tan/ \
spark-py:v2.4.5.1 \
--conf spark.kubernetes.authenticate.submission.caCertFile=/etc/ \
kubernetes/pki/ca.crt \
hdfs://isilon.tan.lab/tpch-s1/tsmodel.py
```

Spark Operator

Operators are software extensions to Kubernetes that are used to manage applications and their components. Operators all follow the same design pattern and provide a uniform interface to Kubernetes across workloads.

The Spark Operator for Kubernetes can be used to launch Spark applications. The Spark Operator uses a declarative specification for the Spark job, and manages the life cycle of the job. Internally, the Spark Operator uses `spark-submit`, but it manages the life cycle and provides status and monitoring using Kubernetes interfaces.

Dell EMC also uses the Kubernetes Operator to launch Spark programs. It works well for the application, but is relatively new and not as widely used as `spark-submit`.

The following examples describe using the Spark Operator:

- [Operator YAML file \(sparkop-ts_model.yaml\) used to launch an application](#) on page 28
- [Launching a Spark application](#) on page 28
- [Checking status of a Spark application](#) on page 29
- [Checking Spark application logs](#) on page 29

Example 2 Operator YAML file (sparkop-ts_model.yaml) used to launch an application

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: sparkop-tsmodel
  namespace: spark-jobs
spec:
  type: Python
  mode: cluster
  image: "infra.tan.lab/tan/spark-py:v2.4.5.1"
  imagePullPolicy: Always
  mainApplicationFile: "hdfs://isilon.tan.lab/tpch-s1/tsmodel.py"
  sparkConfigMap: sparkop-cmap
  sparkVersion: "2.4.5"
  restartPolicy:
    type: Never
  driver:
    cores: 1
    memory: "2048m"
    labels:
      version: 2.4.4
  serviceAccount: spark
  executor:
    cores: 1
    instances: 8
    memory: "4096m"
    labels:
      version: 2.4.4
```

Example 3 Launching a Spark application

```
k8s1:~/SparkOperator$ kubectl apply -f sparkop-ts_model.yaml
sparkapplication.sparkoperator.k8s.io/sparkop-tsmodel created
k8s1:~/SparkOperator$ k get sparkApplications
```

Example 3 Launching a Spark application (continued)

```
NAME AGE
sparkop-tsmodel 7s
```

Example 4 Checking status of a Spark application

```
k8s1:~/SparkOperator$ kubectl describe sparkApplications sparkop-
tsmodel
Name: sparkop-tsmodel
Namespace: spark-jobs
Labels: <none>
Annotations: kubectl.kubernetes.io/last-applied-configuration:
{"apiVersion":"sparkoperator.k8s.io/
v1beta2","kind":"SparkApplication","metadata":{"annotations":
{},"name":"sparkop-tsmodel","namespace":"...
API Version: sparkoperator.k8s.io/v1beta2
Kind: SparkApplication
...

Normal SparkExecutorPending 9s (x2 over 9s) spark-operator Executor
sparkop-tsmodel-1575645577306-exec-7 is pending
Normal SparkExecutorPending 9s (x2 over 9s) spark-operator Executor
sparkop-tsmodel-1575645577306-exec-8 is pending
Normal SparkExecutorRunning 7s spark-operator Executor sparkop-
tsmodel-1575645577306-exec-7 is running
Normal SparkExecutorRunning 7s spark-operator Executor sparkop-
tsmodel-1575645577306-exec-6 is running
Normal SparkExecutorRunning 6s spark-operator Executor sparkop-
tsmodel-1575645577306-exec-8 is running
```

Example 5 Checking Spark application logs

```
k8s1:~/SparkOperator$ kubectl logs tsmodel-1575512453627-driver
```

Interactive and batch execution

The prior examples include both interactive and batch execution.

Dell EMC uses Jupyter for interactive analysis and connects to Spark from within Jupyter notebooks. The image that was created earlier includes Jupyter. The Jupyter image runs in its own container on the Kubernetes cluster independent of the Spark jobs.

Scalability and resource management

When a job is submitted to the cluster, the OpenShift scheduler is responsible for identifying the most suitable compute node on which to host the pods. The default scheduler is policy-based, and uses constraints and requirements to determine the most appropriate node.

Platform administrators can control the scheduling with advanced features including pod and node affinity, node selectors, and overcommit rules. The user specifies the requirements and constraints

when submitting the job, but the platform administrator controls how the request is ultimately handled.

In general, the scheduler abstracts the physical nodes, and the user has little control over which physical node a job runs on. OpenShift *MachineSets* and node selectors can be used to get finer grained control over placement on specific nodes.

Storage interfaces

The on-disk files in a Kubernetes container are ephemeral.

A container always starts with a clean state, including after a crash and restart. The files cannot be shared with other containers in the same pod. This ephemeral storage is allocated from the local storage of the host on which the container is running.

A typical Spark environment needs additional storage beyond the ephemeral storage that is provided, including persistent storage. The environment also needs access to external big data stores, data lakes, and databases.

Kubernetes volumes

This reference architecture uses Kubernetes Volumes to provide access to any additional run-time storage that is needed.

The Kubernetes Container Storage Interface (CSI) enables storage from local drives, and supported external storage systems, to be *mapped* as volumes into a container at run time. These volumes can be ephemeral or persistent. A full list of the CSI-compatible storage systems that OpenShift supports can be found in the [Dell EMC Ready Stack for Red Hat OpenShift Container Platform 4.2 Design Guide](#). This reference architecture uses volumes from local storage and Dell EMC Isilon for the work.

HDFS storage

The bulk of the source data and results for this reference architecture are stored on HDFS.

This reference architecture uses native Spark support for HDFS to connect to a data lake hosted on Isilon. This data path goes directly over TCP/IP, and does not require any special Kubernetes support beyond the network layer. The necessary HDFS support libraries are compiled into Spark, and are in the image file.

Database storage

This reference architecture uses native Spark support for database connections over JDBC to access external databases.

Like the HDFS path, this capability does not require any special Kubernetes support, and the necessary libraries are in the image file.

Spark image

Dell EMC built an image to run Spark under Kubernetes, using:

- The Spark source distribution
- The `docker-image` tool included with Spark

Building Spark

Dell EMC built Spark 2.4.5 from source using the following command:

```
$ ./dev/make-distribution.sh
--name spark --tgz \
-Pkubernetes \
-Phadoop-2.7
```

Building the image

Dell EMC built the container image using the compiled Spark binaries and the `docker-image` tool, with the following commands:

```
k8s1:$ tar -xvf spark-2.4.5-SNAPSHOT-bin-spark.tgz
k8s1:$ cd spark-2.4.5-SNAPSHOT-bin-spark/
k8s1:$ ./bin/docker-image-tool.sh -n -r infra.tan.lab/tan \
-p kubernetes/dockerfiles/spark/bindings/python/Dockerfile \
-R kubernetes/dockerfiles/spark/bindings/R/Dockerfile \
-t v2.4.5-test \
build
```

Pushing the image to a repository

About this task

Dell EMC pushed the final image to the image repository using the following commands:

```
k8s1:$ ./bin/docker-image-tool.sh -n -r infra.tan.lab/tan\
-p kubernetes/dockerfiles/spark/bindings/python/Dockerfile \
-R kubernetes/dockerfiles/spark/bindings/R/Dockerfile \
-t v2.4.5-test\
push
```

If the images are stored in an image repository outside the OpenShift environment, the external repository can be registered in OpenShift using the web interface:

Procedure

1. Log in to the OCP console
For example, <https://console-openshift-console.apps.orange.bda.labs>.
2. Select your credential provider.
For example, `my-htpasswd-provider`.
3. Enter your **username** and **password**.
4. Create the Project for running the Spark application in **Home > Projects >**.
5. Once the project is created, scroll to the **Details** section.
6. Click the **Edit** icon for the Default Pull Secret.
7. Choose **Enter Username/Password**, and then enter the fields for your container registry:
For example, **registry address** (Dell EMC uses `infra.tan.lab`), and **username**.
8. Click the **Save** button.

Additional images

Dell EMC built two additional images for Jupyter and the Spark history server. These images run independently of the Spark jobs on the cluster.

Jupyter image

For Jupyter, this reference architecture uses the `jupyter/all-spark-notebook` image from Docker Hub. The reference architecture uses the following Kubernetes YAML file to deploy the image:

```

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jupyter-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jupyter-server
  revisionHistoryLimit: 3
  template:
    metadata:
      name: jupyter-server
    labels:
      app: jupyter-server
    spec:
      containers:
      - name: jupyter-server
        image: infra.tan.lab/tan/spark-jupyter-server
        ports:
        - containerPort: 8888
---
apiVersion: v1
kind: Service
metadata:
  name: jupyter-svc
spec:
  selector:
    app: jupyter-server
  ports:
  - port: 80
  targetPort: 8888
  name: jupyter
---
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: "jupyter-ingress"
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: jupyter.tan.lab
  http:
    paths:
    - backend:
        serviceName: jupyter

```

Spark history server image

The Spark history server image is based on the Spark image. It starts a server that saves its data in a predefined HDFS folder, so that all the Spark jobs (Operator or `spark-submit`) can write to it.

This reference architecture uses the following *dockerfile* for the history server:

```
FROM infra.tan.lab/tan/spark:v2.4.5
LABEL maintainer="spark_wizard@dell.com"
LABEL version = "1.0"
LABEL description="spark history server on hdfs backend"
# add Tini
ENV TINI_VERSION v0.18.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /
sbin/tini
RUN chmod +x /sbin/tini
```

The reference architecture uses an *entrypoint* script to start the history server when the container is created, and writes history logs to HDFS:

```
$ export eventsDir=\
"hdfs://isilon.tan.lab/history/spark-logs"
export SPARK_HISTORY_OPTS="$SPARK_HISTORY_OPTS -
Dspark.history.fs.logDirectory=$eventsDir";
exec /sbin/tini -s -- /opt/spark/bin/spark-class
org.apache.spark.deploy.history.HistoryServer
```

The history server starts with the following command:

```
$ kubectl apply -f sparkhs.yaml
```

The Kubernetes YAML file is:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spark-history-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sparkhs
  revisionHistoryLimit: 3
  template:
    metadata:
      name: spark-history-server
    labels:
      app: sparkhs
    spec:
      containers:
      - name: spark-hs
        image: infra.tan.lab/tan/spark-history-server
        ports:
        - containerPort: 18080
---
apiVersion: v1
kind: Service
metadata:
  name: sparkhs-svc
```

```
spec:
  selector:
    app: sparkhs
  ports:
  - port: 80
  targetPort: 18080
  name: sparkhs
  ---
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: "spark-history-ingress"
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: sparkhs.tan.lab
  http:
    paths:
    - backend:
        serviceName: sparkhs-svc
        servicePort: 80
```

Monitoring

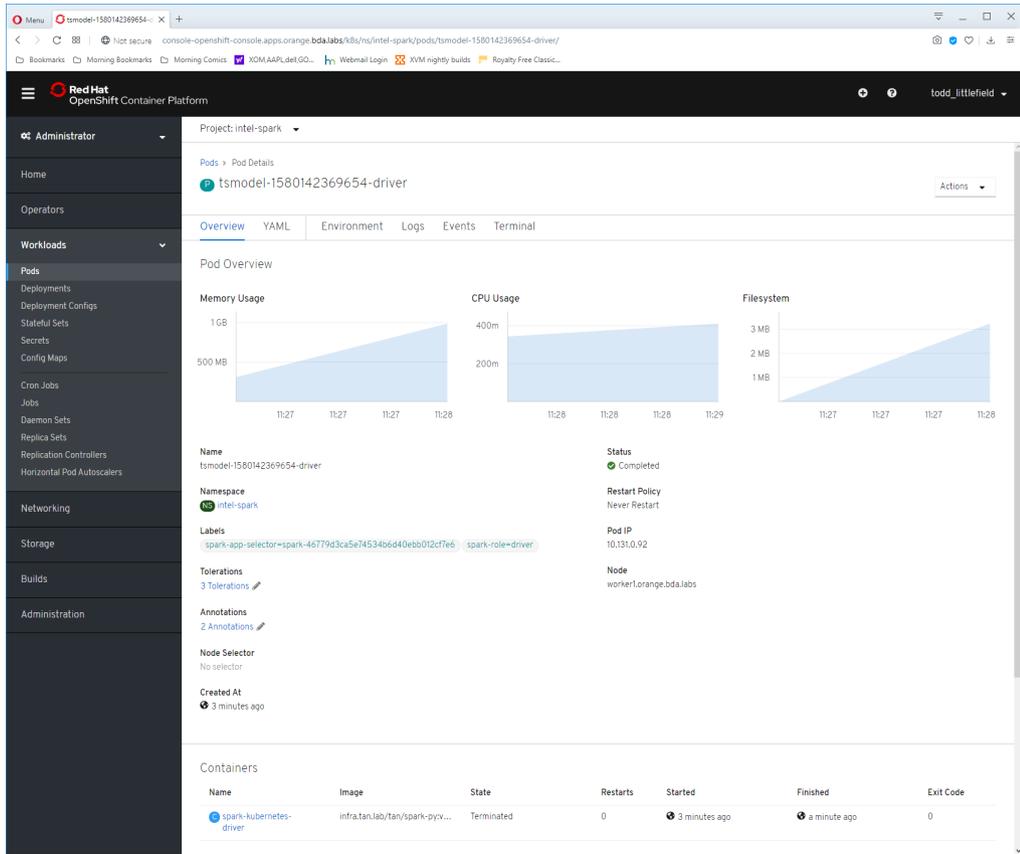
The standard Spark history server can be used to monitor a Spark job while it is running. See [Figure 17](#) on page 34.

Figure 17 Spark history server

Name	Value
spark.driver.host	tamode1-1575467908917-driver-svc.spark-jobs.svc
spark.serializer.objectStreamReset	100
spark.kubernetes.namespace	spark-jobs
spark.event.log.enabled	true
spark.driver.port	7078
spark.rdd.compress	True
spark.driver.blockManager.port	7079
spark.kubernetes.authenticate.submission.caCertFile	/etc/kubernetes/pki/ca.crt
spark.app.name	tamode1
spark.kubernetes.submitDriver	true
spark.scheduler.mode	FIFO
spark.executor.instances	4
spark.kubernetes.memoryOverheadFactor	0.4
spark.driver.bindAddress	192.168.219.49
spark.kubernetes.python.mainAppResource	hdfs://slon.tan.lab/tpch-s1/tamode1.py
spark.kubernetes.container.image.pullPolicy	Always
spark.kubernetes.resource.type	python
spark.files	hdfs://slon.tan.lab/tpch-s1/tamode1.py
spark.kubernetes.container.image	infra.tan.labs/openshift-py-v2.4.5.1
spark.executor.id	driver
spark.submit.deployMode	client
spark.master	h5a://https://100.84.118-17.2443/
spark.kubernetes.authenticate.driver.serviceAccountName	spark
spark.kubernetes.executor.podNamePrefix	tamode1-1575467908917
spark.driver.extraClassPath	/opt/spark/examples/jars/scopt_2.11-3.7.0.jar
spark.event.log.dir	hdfs://slon.tan.lab/history/spark-logs
spark.executor.extraClassPath	/opt/spark/examples/jars/scopt_2.11-3.7.0.jar
spark.kubernetes.python.pyFiles	
spark.app.id	spark-a802b13d5a9840dc83ab23c9094d607b
spark.kubernetes.driver.pod.name	tamode1-1575467908917-driver

The OpenShift web console can also be used to monitor overall cluster activity, including Spark jobs and other activity. See [Figure 18](#) on page 35.

Figure 18 OpenShift pod monitoring



CHAPTER 4

Infrastructure summary

This chapter describes recommended software and hardware infrastructure for Data Analytics with Spark.

- [Software](#)..... 38
- [Hardware infrastructure](#)..... 38

Software

Table 1 on page 38 lists the software components and version that Dell EMC used for Data Analytics with Spark.

Table 1 Software components

Component	Version
Red Hat OpenShift Container Platform	4.2
Apache Spark	2.4.5
Java/Open JDK	1.8
Red Hat Enterprise Linux Server	7.6
Red Hat CoreOS	4.2

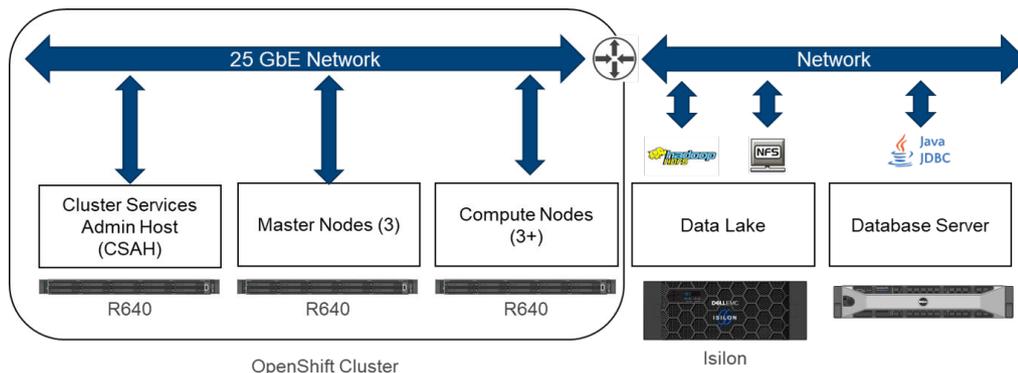
Hardware infrastructure

The primary references for the OpenShift cluster installation are:

- [Dell EMC Ready Stack for Red Hat OpenShift Container Platform 4.2 Design Guide](#)
- [Dell EMC Ready Stack for Red Hat OpenShift Container Platform 4.2 Deployment Guide](#)

Figure 19 on page 38 illustrates the main components of the OpenShift analytics infrastructure.

Figure 19 Analytics infrastructure



The minimum OpenShift deployment consists of seven nodes:

- One cluster services administration host
- Three master nodes
- Three compute nodes

The cluster is scaled primarily by adding additional compute nodes. Extra master nodes should be added for clusters over 250 nodes. Physical scaling of the cluster adds additional resources to the cluster that are available to Kubernetes without changes to job configuration.

The Dell EMC design for OpenShift is flexible and can be used for a wide variety of workloads. The server infrastructure can be customized. The design guide includes general recommendations for server infrastructure.

Dell EMC recommends the following master nodes and compute node server configurations as a starting point for a Spark analytics environment. These configurations have been selected to provide a solid base for a modern analytics environment.

Master node server configuration

This master node configuration, as shown in [Table 2](#) on page 39, is adequate for clusters up to 250 nodes and rarely must be customized.

Table 2 Master node configuration

Component	Details
Platform	Dell EMC PowerEdge R640
Chassis	2.5 in. chassis with up to 10 hard drives, 8 NVMe drives, and 3 PCIe slots, 2 CPU only
Power supply	Dual, hot-plug, redundant power supply (1+1), 750 W
Processor	Dual Intel Xeon Gold 6226 2.7 G, 12C/24T, 10.4 GT/s, 19.25M cache, Turbo, HT (125 W)
RAM	192 GB (12 x 16 GB 293 3MT/s) DDR4-2933
Network daughter card	Mellanox ConnectX-4 Lx dual port 25 GbE SFP 28 rNDC
Network - additional	Mellanox ConnectX-4 Lx dual port 25 GbE SFP 28 network interface controller, low profile
Storage controller	Dell EMC HBA330 12 Gbps SAS HBA Controller, minicard
Storage - operating system	2 x 800 GB SSD SAS Mix Use 12 Gbps e 2.5 in Hot-plug AG Drive, 3 DWPD, 4380 TBW
Storage - data	1 x Dell 1.6 TB, NVMe, Mixed Use Express Flash, 2.5 SFF Drive, U.2, P4610 with Carrier

Compute node server configuration

This compute node configuration, as shown in [Table 3](#) on page 40, provides a good balance of compute and memory for typical Spark programs. There is enough local storage to support temporary files, Spark disk cache, and storage for other applications.

The storage is not intended to be primary storage for applications that require significant storage capacity, such as databases or web applications. The CSI interface to external storage systems is expected to be used for those applications.

OpenShift supports heterogenous node configurations. You can create specialized node configurations. Contact your Dell EMC representative for assistance in sizing and customizing any of these configurations.

Table 3 Compute node configuration

Component	Details
Platform	Dell EMC PowerEdge R640
Chassis	2.5 in. chassis with up to 10 hard drives, 8 NVMe drives, and 3 PCIe slots, 2 CPU only
Power supply	Dual, hot-plug, redundant power supply (1+1), 750 W
Processor	Dual Intel Xeon Gold 6238 2.1 G, 22C/44T, 10.4 GT/s, 30.25 M Cache, Turbo, HT (140 W)
RAM	384 GB (12 x 32 GB 293 3MT/s) DDR4-2933
Network daughter card	Mellanox ConnectX-4 Lx dual port 25 GbE SFP 28 rNDC
Network - additional	Mellanox ConnectX-4 Lx dual port 25 GbE SFP28 network interface controller, low profile
Storage controller	Dell EMC HBA330 12 Gbps SAS HBA Controller, minicard
Storage - operating system	2 x 800 GB SSD SAS Mix Use 12 Gbps e 2.5 in Hot-plug AG Drive, 3 DWPD, 4380 TBW
Storage - data	3 x Dell 1.6 TB, NVMe, Mixed Use Express Flash, 2.5 SFF Drive, U.2, P4610 with Carrier

Network switches

The cluster data network uses a leaf and spine architecture. Dell EMC recommends the following switches:

- Leaf Top of Rack (ToR) - Dell EMC PowerSwitch S5248F-ON
- Spine - Dell EMC PowerSwitch S5232F-ON
- iDRAC network and Out of Band (OOB) management - Dell EMC PowerSwitch S3048-ON

 **Note:** A typical installation uses two leaf switches per rack for high availability.

CHAPTER 5

Conclusions

The proliferation of data science tools and platforms over the last decade has left many organizations stagnated during long periods of software and hardware evaluation. Dell EMC has shown how Spark hosted in a Kubernetes managed cluster can provide a wide range of data science capability that IT can install, configure, and manage.

- [Document summary](#)42

Document summary

The use case showed that data scientists and data engineers can collaborate to build a full analytics pipeline without having to go outside the Spark ecosystem, for:

- Data ingestion
- Data cleansing
- Data merging
- Model training

The demonstration of Jupyter notebooks and server adds rapid prototyping and visualization capabilities to the data science team. This method uses the same container or Kubernetes management tool set as all the other Spark-specific services.

This reference architecture also showed IT professionals how they can leverage the growing capabilities of Kubernetes to manage infrastructure for Spark analytics. Distributed analytics workloads have been difficult to migrate to containers and Kubernetes. This document offered lab-tested recommendations that should save an organization new to Spark with Kubernetes countless hours of research and prototyping, including:

- Submitting jobs
- Managing storage
- Building Spark image files (containers)

Finally, all the specifics of the reference lab hardware, software, and configurations used for the retail inventory management use case demonstration were documented. This document provided guidance for a representative reference architecture that Dell EMC considers appropriate for general-purpose data analytics involving all stages of an analytics pipeline using Apache Spark.

Dell EMC chose the [Dell EMC Ready Stack for Red Hat OpenShift Container Platform 4.2 Design Guide](#) as a base implementation for Kubernetes. The detailed Red Hat OpenShift Container Platform information, and Spark background and configuration details in this reference architecture can jumpstart any organization wanting to test an analytics platform that enthruses data scientists and IT professionals.

CHAPTER 6

References

Additional information can be obtained at the [Dell EMC InfoHub for Big Data Analytics](#). If you need additional services or implementation help, contact your Dell EMC sales representative.

- [Dell EMC documentation](#)..... 44
- [Apache Spark documentation](#)..... 44
- [Dell EMC Customer Solution Centers](#)..... 44
- [Dell Technologies InfoHub](#)..... 44
- [More information](#)..... 44

Dell EMC documentation

The following Dell EMC documentation provides additional and relevant information. Access to these documents depends on your login credentials. If you do not have access to a document, contact your Dell EMC representative.

- [Data Analytics with Spark Reference Architecture \(this guide\)](#)
- [Dell EMC PowerEdge R740xd Spec Sheet](#)
- [Dell EMC PowerEdge R640 Manuals and Documents](#)
- [Dell EMC PowerEdge R740xd Manuals and Documents](#)
- [Dell EMC PowerSwitch S5248F-ON Manuals and Documents](#)
- [Dell EMC PowerSwitch S5232F-ON Manuals and Documents](#)
- [Dell EMC PowerSwitch S3048-ON Manuals and Documents](#)
- [Isilon Site Preparation and Planning Guide](#)

Apache Spark documentation

More information about Apache Spark can be found on the [Spark](#) website.

Dell EMC Customer Solution Centers

Our global network of dedicated [Dell EMC Customer Solution Centers](#) are trusted environments where world class IT experts collaborate with customers and prospects to share best practices; facilitate in-depth discussions of effective business strategies using briefings, workshops, or Proofs of Concept (PoCs); and help business become more successful and competitive.

Dell EMC Customer Solution Centers reduce the risks that are associated with new technology investments, and can help improve speed of implementation.

All of the services of the Customer Solution Centers are available to all Dell Technologies customers at no charge. Contact your account team today to submit an engagement request.

Dell Technologies InfoHub

The [Dell Technologies InfoHub](#) is your one-stop destination for the latest information about Dell EMC Solutions and Networking products. New material is frequently added, so visit often to keep up to date on our expanding portfolio of cutting-edge products and solutions.

More information

For more information, contact your Dell EMC or authorized partner sales representative.