phoenixNAP GLOBAL IT SERVICES intel

White Paper by phoenixNAP
May 2021

Big Data Processing on Bare Metal Cloud with Apache Spark

How to automatically deploy a Spark cluster on non-virtual bare metal machines Powered by Intel

Written by:
Sergio Muriana, DevOps at phoenixNAP
With assistance of Seow Lim, PhD, VP of Architecture & Platform

Introduction

We are creating massive amounts of data at an unprecedented rate. By 2025, it is estimated that the world will generate 463 exabytes[1] of data each day, and at least 80%[2] of that data will be unstructured.

As opposed to structured data, which is easily organized in relational databases, unstructured data is difficult to organize and process with traditional technologies. This is because unstructured data comes in many different forms, such as transactions, logs, sales figures, email, social media, videos, photos, web pages, and IoT device sensors.

Vast amounts of unstructured data are what we call big data. Processing and contextualizing these massive datasets requires high-performance data processing technologies, including software and infrastructure solutions capable of delivering the necessary compute power on demand. According to Gartner[3] , public cloud services will be essential for 90% of data and analytics innovations by 2022.

Apache Spark is one of the leading software solutions for big data processing. Running this open-source data processing application in the public cloud brings many benefits such as scalability, flexibility, and ability. However, performance and security issues in virtualized, multi-tenant public cloud environments intensify as data processing workloads grow.

Big Data Processing Challenges

For decades, businesses have mainly collected and stored structured data in relational databases. However, due to the sheer size and diversity of data that we collect today, traditional analytics practices and tools are not capable of meeting the demands of modern businesses.

Extracting actionable insights from unstructured data is achieved through the four Vs of big data. The four Vs describe the speed (velocity) at which large amounts (volume) of diverse data (variety) from various sources (veracity) must be verified, organized, and processed to extract actionable insights.

[1] https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/

[2] https://www.ibm.com/blogs/watson/2019/12/ibm-watson-enables-your-business-to-get-started-with-text-analytics/

[3] https://www.gartner.com/smarterwithgartner/gartner-top-10-trends-in-data-and-analytics-for-2020/

When dealing with big data, the biggest roadblocks modern enterprises face are velocity and variety. This is because querying dissimilar data sources to extract meaningful results in real-time while ensuring seamless scalability and performance simply cannot be achieved with traditional relational databases and infrastructure solutions.

Other challenges with big data include storage and processing costs, as well as proprietary technologies. With on-premises solutions, organizations often have to spend capital on software licenses, hardware upgrades, and infrastructure investments to support growth. And with the amount of information that needs to be stored and processes, implementing strict data security policies has also become increasingly complex.

Open-source big data processing technologies and cloud-native bare metal infrastructure solutions can help businesses overcome these challenges. Running data processing frameworks such as Apache Spark on phoenixNAP's Bare Metal Cloud (BMC) infrastructure gives modern enterprises the opportunity to capitalize on big data in multiple ways.

Memory, CPU, Network, and Storage Performance

BMC servers leverage the latest-generation Intel® Xeon® processors to deliver exceptional speed when performing CPU-heavy data calculations. There is no pre-installed hypervisor on BMC servers. Data processing workloads run on physical hardware and in a single-tenant environment, which offsets the negative effects of other tenants using up valuable resources.

Providing access to the latest **3rd Gen Intel Xeon Scalable processors** (codenamed Ice Lake) as of April 2021, Bare Metal Cloud further expands the options for performance-oriented workloads. The increased core count and memory capacity bring significant performance gains performance gains over previous generation CPUs, ensuring seamless processing of data-hungry workloads. BMC also includes instances with Intel Software Guard Extensions (SGX) enabled to provide confidential computing capabilities.

In addition to processing power, BMC servers are equipped with the latest NVMe SSD drives by Intel. This ensures faster I/O operations when reading or writing large-scale datasets from the disk. And with network capacity of up to 50 Gbps, users benefit from ultra-low latency when transferring data between Apache Spark nodes.

Apache Spark leverages in-memory computing to deliver faster data processing. It stores data in RAM rather than on the disk, which is critical for achieving faster processing speeds with machine learning and micro-batch processing workloads. BMC servers go hand-in-hand with Apache Spark's in-memory computing architecture by offering up to 768 GB of RAM with select server instances. This amount of RAM is sufficient for handling memory-intensive data processing operations.

For customers who want to operate at very large data sets to avoid memory and I/O bottlenecks, phoenixNAP offers configurations with additional memory capacity through Intel Optane Persistent Memory. This allows improving I/O intensive SPARK queries by moving all the processed data closer to the CPU for analytics processing.

Automated Deployments and Scaling

Scaling BMC servers is as easy as scaling virtual public cloud servers. With more than 20 server instance types to choose from, users have the freedom to deploy and scale server configurations that best fit their data processing needs.

On top of this, BMC servers are compatible with the most popular infrastructure as code tools such as Terraform, Ansible, and Pulumi. This means users can fully automate server deployments and scaling operations with only a couple of lines of code.

BMC servers typically deploy in under two minutes. This fast and automated server provisioning capability supports unpredictable data processing workloads, especially real-time data crunching and analysis. Users can deploy and destroy servers on-demand, launch new Apache Spark worker nodes on those servers, and scale them across the US, Europe, and Asia.

More Control Over Security

When processing large datasets of sensitive information, maintaining strong data privacy and security is of paramount importance. While the public cloud delivers many benefits, concerns related to security in shared virtual environments still abound. BMC servers solve these data security and privacy issues by giving users maximum control over their environments.

phoenixNAP maintains the availability and physical security of the global BMC infrastructure, while the user is responsible for everything running on the machines. Since there are no pre-installed hypervisors or other virtualization layers, no resources are shared with other tenants on the same machine. As opposed to the public cloud which relies on software-defined environment isolation, each BMC server is physically isolated. This means each physical machine is allocated to a single user.

Teams and organizations looking to virtualize their resources can install custom hypervisors or container engines to segment physical server resources to fit their needs.

Cloud-Native Infrastructure

BMC was built as a cloud-native, ready, dedicated server platform that offers the flexibility and agility of the cloud coupled with the raw power of physical hardware. The platform delivers a cloud-like approach to server provisioning via API and CLI tools. In addition to the web-based dashboard, BMC servers can be deployed by making simple HTTP calls to its API. This enables the platform to integrate with various automated provisioning and orchestration engines such as Terraform, Ansible, Pulumi, Puppet, Chef, and many more. The BMC development team is actively releasing new infrastructure as code modules and automation scripts to make it easier for users to deploy servers with different technologies.

Along with cloud-like server deployments, BMC has also adopted the pay-per-use pricing model of the public cloud. Users only pay for the compute resources they consume on an hourly basis. However, those that require guaranteed resources available for large-scale data processing workloads can take advantage of monthly and yearly reservation options. Available plans include month-to-month, 12, 24, and 36-month reservations.

How to Deploy a Spark Cluster on Bare Metal Cloud

BMC exposes a RESTful API interface which enables developers to automate server deployments. These Python code examples demonstrate how to leverage the BMC API to automate the provisioning of a Spark cluster.

Steps that will be covered:

1. Generating BMC API access tokens
2. Deploying three BMC servers running the Ubuntu OS
3. Deploying a Spark cluster on the servers
4. Accessing the Spark dashboard

Generating BMC API Access Tokens

Before sending requests to the BMC API, you need to obtain an OAuth access token using the *client_id* and *client_secret* commands. These values are generated in the BMC portal.

To learn more about how to register these access tokens, refer to the Bare Metal Cloud API Quick Start guide.

This is the Python function that gets the access token for the API.

```
def get _ access _ token(client _ id: str, client _ secret: str) -> str:
“””Retrieves an access token from BMC auth by using the client ID and the
client Secret.”””
    credentials = “%s:%s” % (client _ id, client _ secret)
    basic _ auth = standard _ b64encode(credentials.encode(“utf-8”))
    response = requests.post(‘ https://api.phoenixnap.com/bmc/v0/servers’,
        headers={
            ‘Content-Type’: ‘application/x-www-form-urlencoded’,
            ‘Authorization’: ‘Basic %s’ % basic _ auth.decode(“utf-8”)},
        data={‘grant _ type’: ‘client _ credentials’})

    if response.status _ code != 200:
        raise Exception(‘Error: {}. {}’.format(response.status _ code, response.json()))
    return response.json()[‘access _ token’]
```

Deploying BMC Servers Running the Ubuntu OS

Use POST/servers REST API calls to deploy BMC server instances. For each POST/server request, specify the required parameters, such as the data center location, instance type, and operating system. This is the Python function that makes a call to the BMC API to create a BMC server:

```
def _ _ do _ create _ server(session, server):
    response = session.post(‘https://api.phoenixnap.com/bmc/v0/servers’),
                            data=json.dumps(server))
    if response.status _ code != 200:
        print(“Error creating server: {}”.format(json.dumps(response.json())))
    else:
        print(“{}”.format(json.dumps(response.json())))
        return response.json()
```

In this example, three ***s1.c1.small*** BMC server instances are created, as specified in the server-settings.conf file.

```
{
“ssh-key” : “ssh-rsa xxxxxx== username”,
“servers _ quantity” : 3,
“type” : “s1.c1.small”,
“hostname” : “spark”,
“description” : “spark”,
“public” : True,
“location” : “PHX”,
“os” : “ubuntu/bionic”
}
```

The expected output from the Python script that generates the token and provisions the servers should look like this:

```
Retrieving token
Successfully retrieved API token
Creating servers...
{
  “id”: “5ee9c1b84a9ca71ea6b9b766”,
  “status”: “creating”,
  “hostname”: “spark-1”,
  “description”: “spark-1”,
  “os”: “ubuntu/bionic”,
  “type”: “s1.c1. small “,
  “location”: “PHX”,
  “cpu”: “E-2276G”,
  “ram”: “128GB RAM”,
  “storage”: “2x 960GB NVMe”,
  “privateIpAddresses”: [
    “10.0.0.11”
  ],
  “publicIpAddresses”: [
    “131.153.143.250”,
    “131.153.143.251”,
    “131.153.143.252”,
    “131.153.143.253”,
    “131.153.143.254”
  ]
}
Server created, provisioning spark-1...
{
  “id”: “5ee9c1b84a9ca71ea6b9b767”,
  “status”: “creating”,
  “hostname”: “spark-0”,
  “description”: “spark-0”,
  “os”: “ubuntu/bionic”,
  “type”: “s1.c1.small”,
  “location”: “PHX”,
  “cpu”: “E-2276G”,
  “ram”: “128GB RAM”,
  “storage”: “2x 960GB NVMe”,
  “privateIpAddresses”: [
    “10.0.0.12”
  ],
  “publicIpAddresses”: [
    “131.153.143.50”,
    “131.153.143.51”,
    “131.153.143.52”,
    “131.153.143.53”,
    “131.153.143.54”
  ]
}
Server created, provisioning spark-0...
{
  “id”: “5ee9c1b84a9ca71ea6b9b768”,
  “status”: “creating”,
  “hostname”: “spark-2”,
  “description”: “spark-2”,
  “os”: “ubuntu/bionic”,
  “type”: “s1.c1. small “,
  “location”: “PHX”,
  “cpu”: “E-2276G”,
  “ram”: “128GB RAM”,
  “storage”: “2x 960GB NVMe”,
  “privateIpAddresses”: [
    “10.0.0.13”
  ],
  “publicIpAddresses”: [
    “131.153.142.234”,
    “131.153.142.235”,
    “131.153.142.236”,
    “131.153.142.237”,
    “131.153.142.238”
  ]
}
Server created, provisioning spark-2...
Waiting for servers to be provisioned...
```

Once server provisioning is initiated, the script communicates with the BMC API to check the servers' status until provisioning is completed and the servers are powered on.

Hostname	Location	Status	CPU	Memory	Storage	Type	Manage
spark-1	Phoenix	Powered On	E-2276G	64GB RAM	1x 960GB NVMe	s1.c1.small	Actions
spark-0	Phoenix	Powered On	E-2276G	64GB RAM	1x 960GB NVMe	s1.c1.small	Actions
spark-2	Phoenix	Powered On	E-2276G	64GB RAM	1x 960GB NVMe	s1.c1.small	Actions

Provisioning a Spark cluster

With the servers in the "powered-on" state, the Python script establishes an SSH connection using the servers' public IP address. The script then installs Spark on the Ubuntu servers along with the JDK, Scala, and Git.

To begin the installation process, execute the all_hosts.sh file on all servers. This script contains download and installation instructions as well as the environment configuration needed to prepare the cluster for use.

Apache Spark includes scripts which configure the servers as master and worker nodes. The first server to be provisioned is assigned as the Spark master node.

The following Python function performs that task:

```
def wait_server_ready(function_scheduler, server_data):
    json_server = bmc_api.get_server(REQUEST, server_data['id'])
    if json_server['status'] == "creating":
       main_scheduler.enter(2, 1, wait_server_ready, (function_scheduler, server_data))
    elif json_server['status'] == "powered-on" and not data['has_a_master_server']:
        server_data['status'] = json_server['status']
        server_data['master'] = True
        server_data['joined'] = True
        data['has_a_master_server'] = True
        data['master_ip'] = json_server['publicIpAddresses'][0]
        data['master_hostname'] = json_server['hostname']
        print("ASSIGNED MASTER SERVER: {}".format(data['master_hostname']))
```

Execute the master_host.sh file to configure the first server as the master node.

```
#!/bin/bash
echo "Setting up master node"
/opt/spark/sbin/start-master.sh
```

Once the master node is assigned and configured, the other two nodes are added to the Spark cluster as worker nodes. This is the contents of the ***worker_host.sh*** file:

```
#!/bin/bash
echo "Setting up master node on /etc/hosts"
echo "$1 $2 $2" | sudo tee -a /etc/hosts
echo "Starting worker node"
echo "Joining worker node to the cluster"
/opt/spark/sbin/start-slave.sh spark://$2:7077
```

If the Spark cluster provisioning is successful, the output will look like this:

```
ASSIGNED MASTER SERVER: spark-2
Running all_host.sh script on spark-2 (Public IP: 131.153.142.234)
Setting up /etc/hosts
Installing jdk, scala and git
Downloading spark-2.4.5
Unzipping spark-2.4.5
Setting up environment variables

Running master_host.sh script on spark-2 (Public IP: 131.153.142.234)
Setting up master node
starting org.apache.spark.deploy.master.Master, logging to /opt/spark/logs/spark-ubuntu-org.apache.spark.deploy.master.
Master-1-spark-2.out
Master host installed

Running all_host.sh script on spark-0 (Public IP: 131.153.143.170)
Setting up /etc/hosts
Installing jdk, scala and git
Downloading spark-2.4.5
Unzipping spark-2.4.5
Setting up environment variables

Running all_host.sh script on spark-1 (Public IP: 131.153.143.50)
Setting up /etc/hosts
Installing jdk, scala and git
Downloading spark-2.4.5
Unzipping spark-2.4.5
Setting up environment variables

Running slave_host.sh script on spark-0 (Public IP: 131.153.143.170)
Setting up master node on /etc/hosts
10.0.0.12 spark-2 spark-2
Starting worker node
Joining worker node to the cluster
starting org.apache.spark.deploy.worker.Worker, logging to /opt/spark/logs/spark-ubuntu-org.apache.spark.deploy.worker.
Worker-1-spark-0.out

Running slave_host.sh script on spark-1 (Public IP: 131.153.143.50)
Setting up master node on /etc/hosts
10.0.0.12 spark-2 spark-2
Starting worker node
Joining worker node to the cluster
starting org.apache.spark.deploy.worker.Worker, logging to /opt/spark/logs/spark-ubuntu-org.apache.spark.deploy.worker.
Worker-1-spark-1.out

Setup servers done
Master node UI: http://131.153.142.234:8080
```

Accessing the Spark dashboard

To access the Apache Spark dashboard, follow the link generated by the Python script. This is what the Spark dashboard looks like:

GLOBALLY CONNECTED. LOCALLY AVAILABLE.

2.35 Tbps Bandwidth Capacity | 20,000+ Servers Available Worldwide

100% Network Uptime with World-Class Carrier Blend

May, 2021