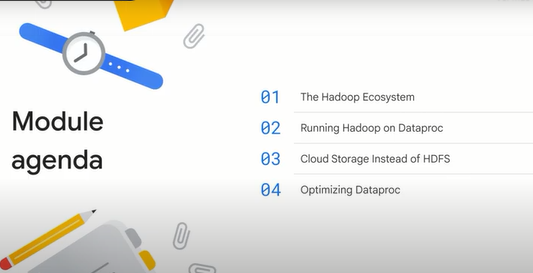
**Executing Spark on DataProc.**

**Module Introduction.**

In this module, we will discuss Dataproc, Google Cloud's managed Hadoop service, and in particular, Apache Spark.

In this module, we'll cover the Hadoop Ecosystem, learn about running Hadoop on Dataproc, understand the benefits of Cloud Storage instead of HDFS, learn about optimizing Dataproc, and complete a hands-on lab with Apache Spark on Dataproc.



**The Hadoop ecosystem.**

Let's start by looking at the Hadoop ecosystem in a little more detail.

It helps to place the services you'll be learning about in historical context.

Before 2006, big bata meant big databases.

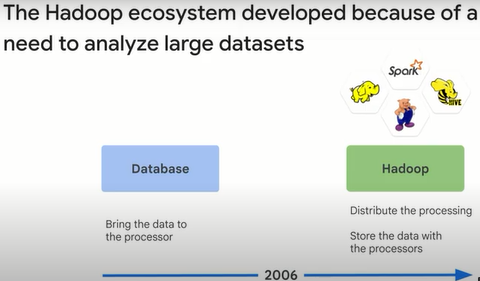
Database design came from a time when storage was relatively cheap and processing was expensive, so it made sense to copy the data from its storage location to the processor to perform data processing, then the result will be copied back to storage.

Around 2006, distributed processing of big data became practical with Hadoop.

The idea behind Hadoop is to create a cluster of computers and leverage distributed processing.

HDFS, the Hadoop Distributed File System, stored the data on the machines in the cluster and MapReduce provided distributed processing of the data.

A whole ecosystem of Hadoop related software grew up around Hadoop, including Hive, Pig and Spark.



Organizations use Hadoop for on-premises big data workloads.

They make use of a range of applications that run on Hadoop clusters, such as Presto, but a lot of customers use Spark.

Apache Hadoop is an open source software project that maintains the framework for distributed processing of large datasets across clusters of computers using simple programming models.

HDFS is the main file system Hadoop uses for distributing work to nodes on the cluster.

Apache Spark is an open source software project that provides a high performance analytics engine for processing batch and streaming data.

Spark can be up to 100 times faster than equivalent Hadoop jobs because it leverages in memory processing.

Spark also provides a few for dealing with data including resilient distributed datasets and data frames.

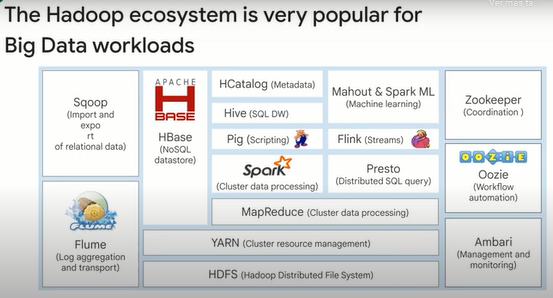
Spark in particular is very powerful and expressive and used for a lot of workloads.

A lot of the complexity and overhead of OSS Hadoop has to do with assumptions in the design that existed in the data center.

Relieved of those limitations, data processing becomes a much richer solution with many more options.

There are two common issues with OSS Hadoop: tuning and utilization.

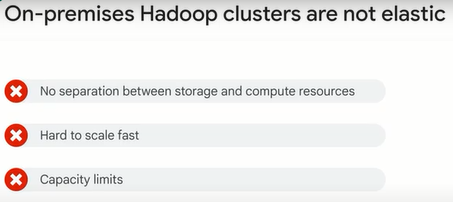
In many cases, using Dataproc as designed will overcome these limitations.



On-premises Hadoop clusters, due to their physical nature, suffer from limitations.

The lack of separation between storage and compute resources results in capacity limits and an inability to scale fast.

The only way to increase capacity is to add more physical servers.



There are many ways in which using Google Cloud can save you time, money and effort compared to using an on-premises Hadoop solution.

In many cases, adopting a Cloud based approach can make your overall solution simpler and easier to manage.

Built-in support for Hadoop.

Dataproc is a managed Hadoop and Spark environment.

You can use Dataproc to run most of your existing jobs with minimal alteration, so you don't need to move away from all of the Hadoop tools you already know.

Managed hardware and configuration.

When you run Hadoop on Google Cloud, you never need to worry about physical hardware.

You specify the configuration of your cluster and Dataproc allocates resources for you, you can scale your cluster at any time.

Simplified version management.

Keeping open source tools up to date and working together is one of the most complex parts of managing a Hadoop cluster.

When you use Dataproc, much of that work is managed for you by Dataproc versioning.

Flexible job configuration.

A typical on-premises Hadoop setup uses a single cluster that serves many purposes.

When you move to Google Cloud, you can focus on individual tasks, creating as many clusters as you need.

This removes much of the complexity of maintaining a single cluster with growing dependencies and software configuration interactions.



Running MapReduce directly on top of Hadoop is very useful, but it has the complication that the Hadoop system has to be tuned for the kind of job being run to make efficient use of the underlying resources.

A simple explanation of Spark is that it is able to mix different kinds of applications and to adjust how it uses the available resources.

Spark uses a declarative programming model.

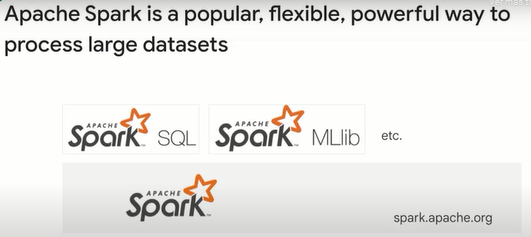
In imperative programming, you tell the system what to do and how to do it.

In declarative programming, you tell the system what you want and it figures out how to implement it.

You will be learning to work with Spark in the labs in this course.

There is a full SQL implementation on top of Spark.

There is a common data frame model that works across Scala, Java, Python, SQL and R. And there is a distributed machine learning library called Spark ML Lib.



**Running Hadoop on Dataproc.**

Next, we'll discuss how and why you should consider processing your same Hadoop job code in the Cloud using Dataproc on Google Cloud.

Dataproc lets you take advantage of open source data tools for batch processing, querying, streaming, and machine learning.

Dataproc automation helps you create clusters quickly, manage them easily and save money by turning clusters off when you don't need them.

When compared to traditional on-premises products and competing Cloud services, Dataproc has unique advantages for clusters of three to hundreds of nodes.

There is no need to learn new tools or APIs to use Dataproc, making it easy to move existing projects into Dataproc without redevelopment.

Spark, Hadoop, Pig and Hive are frequently updated.

Here are some of the key features of Dataproc.

Low cost, Dataproc is priced at one cent per virtual CPU per cluster per hour on top of the other Google Cloud resources you use.

In addition, Dataproc clusters can include preemptable instances that have lower compute prices.

You use and pay for things only when you need them.

So Dataproc charges second by second billing with a one-minute minimum billing period.

Superfast, Dataproc clusters are quick to start, scale, and shut down, with each of these operations taking 90 seconds or less on average.

Resizable clusters.

Clusters can be created and scaled quickly with a variety of virtual machine types, disk sizes, number of nodes and networking options.

Open source ecosystem.

You can use Spark and Hadoop tools, libraries, and documentation with Dataproc.

Dataproc provides frequent updates to native versions of Spark, Hadoop, Pig and Hive.

So there is no need to learn new tools or APIs.

And it is possible to move existing projects or ETL pipelines without redevelopment.

Integrated.

Built-in integration with Cloud Storage, BigQuery and Cloud Bigtable ensures data will not be lost.

This together with Cloud Logging and Cloud Monitoring provides a complete data platform and not just a Spark or Hadoop cluster.

For example, you can use Dataproc to effortlessly ETL terabytes of raw log data directly into BigQuery for business reporting.

Managed.

Easily interact with clusters and Spark or Hadoop jobs without the assistance of an administrator or special software through the Cloud Console, the Cloud SDK, or the Dataproc REST API.

When you're done with a cluster, simply turn it off so money isn't spent on an idle cluster.

Versioning.

Image versioning allows you to switch between different versions of Apache Spark, Apache Hadoop and other tools.

Highly available.

Run clusters with multiple primary nodes and set jobs to restart on failure to ensure your clusters and jobs are highly available.

Developer Tools.

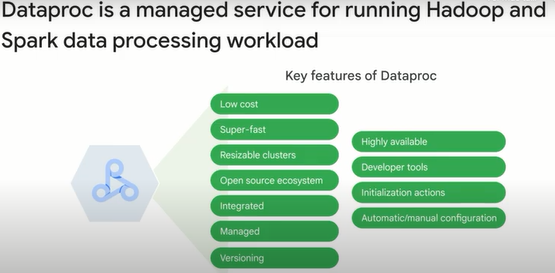
Multiple ways to manage a cluster including the Cloud Console, the Cloud SDK, restful API's and SSH access.

Initialization actions.

Run initialization actions to install or customize the settings and libraries you need when your cluster is created.

And automatic or manual configuration.

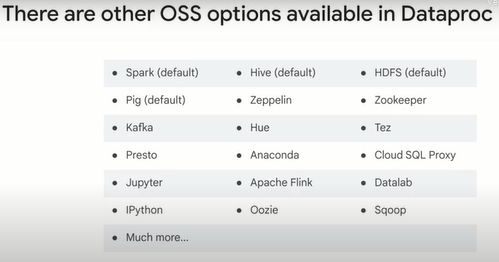
Dataproc automatically configures hardware and software on clusters for you while also allowing for manual control.



Dataproc has two ways to customize clusters: optional components and initialization actions.

Pre-configured optional components can be selected when deploying from the console or via the command line and include Anaconda, Hive, WebHCat, Jupyter notebook, Zeppelin notebook, Druid, Presto, and Zookeeper.

Initialization actions let you customize your cluster by specifying executables or scripts that Dataproc will run on all nodes in your Dataproc cluster immediately after the cluster is set up.



Here's an example of how you can create a Dataproc cluster using the Cloud SDK.

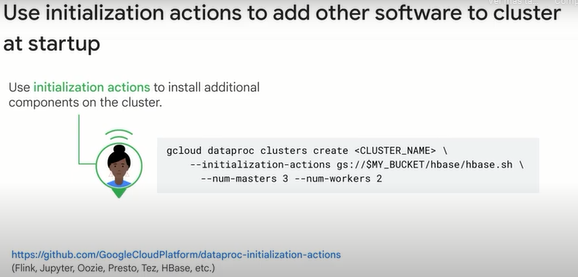
And we're going to specify a HBase shell script to run on the cluster's initialization.

There are a lot of pre-built startup scripts that you can leverage for common Hadoop cluster setup tasks, like Flink, Jupyter and more.

You can check out the GitHub repo link in the course resources to learn more.

Do you see the additional parameter for the number of master and worker nodes in the script?

Let's talk more about the architecture of the cluster.



A Dataproc cluster can contain either preemptable secondary workers or non-preemptable secondary workers, but not both.

The standard set of architecture is much like you would expect on premise.

You have a cluster of virtual machines for processing, and then persistent disks for storage via HDFS.

You've also got your primary node VMs in a set of worker nodes.

Worker nodes can also be part of a managed instance group, which is just another way of ensuring that VMs within that group are all of the same template.

The advantage is that you can spin up more VMs than you need to automatically resize your cluster based on the demands.

It also only takes a few minutes to upgrade or downgrade your cluster.

Generally, you shouldn't think of a Dataproc cluster as long lived.

Instead, you should spin them up when you need compute processing for a job, and then simply turn them down.

You can also persist them indefinitely if you want to.

What happens to HDFS storage on disk when you turn those clusters down?

The storage would go away too, which is why it's a best practice to use storage that's off cluster by connecting to other Google Cloud products.

Instead of using native HDFS on a cluster, you could simply use a cluster of buckets on Cloud storage via the HDFS connector.

It's pretty easy to adopt existing Hadoop code to use Cloud Storage instead of HDFS.

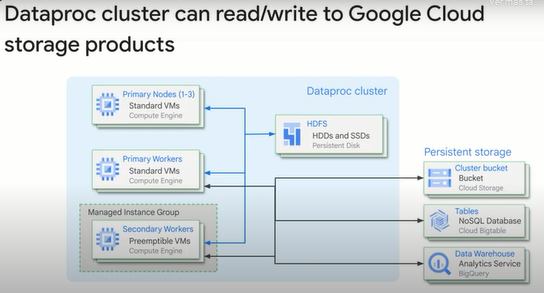
Change the prefix for this storage from HDFS// to GS//.

What about HBase off cluster?

Consider writing in the Cloud Bigtable instead.

What about large analytical workloads?

Consider reading that data into BigQuery and doing those analytical workloads there.



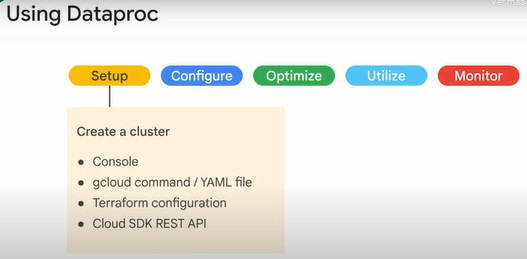
Using Dataproc involves this sequence of events, setup, configuration, optimization, utilization, and monitoring.

Setup means creating a cluster.

And you can do that through the Cloud Console or from the command line using the G Cloud command.

You can also export a YAML file from an existing cluster or create a cluster from a YAML file.

You can create a cluster from a Terraform configuration, or use the REST API.



The cluster can be set as a single VM, which is usually to keep costs down for development and experimentation.

Standard is with a single primary node and high availability has three primary nodes.

You can choose a region and zone or select a global region and allow the service to choose the zone for you.

The cluster defaults to a global endpoint, but defining a regional endpoint may offer increased isolation and in certain cases, lower latency.

The primary node is where the HDFS Name Node runs, as well as the Yarn node and job drivers.

HDFS replication defaults to two in Dataproc.

Optional components from the Hadoop ecosystem include Anaconda, Python distribution and package manager, Hive WebHCat, Jupyter notebook, and Zeppelin notebook.

Cluster properties are runtime values that can be used by configuration files for more dynamic startup options, and user labels can be used to tag the cluster for your own solutions or reporting purposes.

The primary node worker nodes and preemptable worker nodes if enabled have separate VM options such as VCPU, memory and storage.

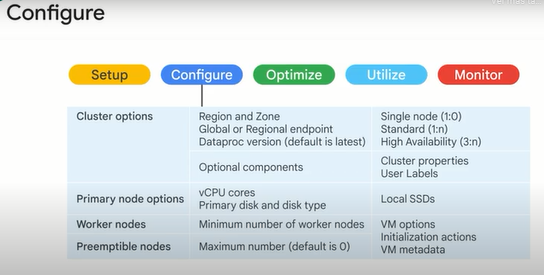
Preemptable nodes include ER node manager, but they do not run HDFS.

There are a minimum number of worker nodes, the default is two.

The maximum number of worker nodes is determined by a quota and the number of SSDs attached to each worker.

You can also specify initialization actions, such as initialization scripts that can further customize the worker nodes.

And metadata can be defined so that the VMs can share state information.

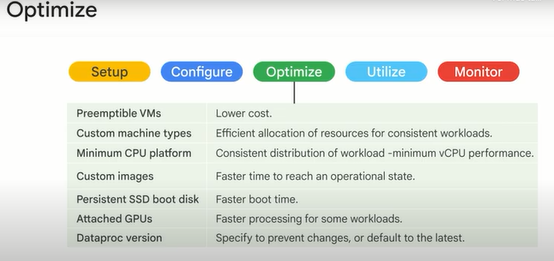


Preemptable VMs can be used to lower costs, just remember that they can be pulled from service at any time and within 24 hours, so your application might need to be designed for resilience to prevent data loss.

Custom machine types allow you to specify the balance of memory in CPU to tune the VM to the load so you are not wasting resources.

A custom image can be used to pre-install software so that it takes less time for the customized node to become operational than if you installed the software at boot time using an initialization script.

You can also use a persistent SSD boot disk for faster cluster startup.



Jobs can be submitted through Console, the G Cloud command or the REST API.

They can also be started by orchestration services such as Dataproc workflow and Cloud composer.

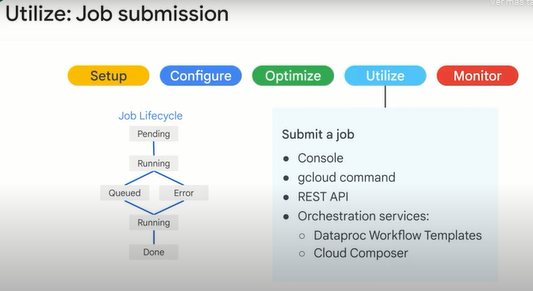
Don't use Hadoop's direct interfaces to submit jobs because the metadata will not be available to Dataproc for job and cluster management.

And for security, they are disabled by default.

By default, jobs are not restartable.

However, you can create restartable jobs through the command line or REST API.

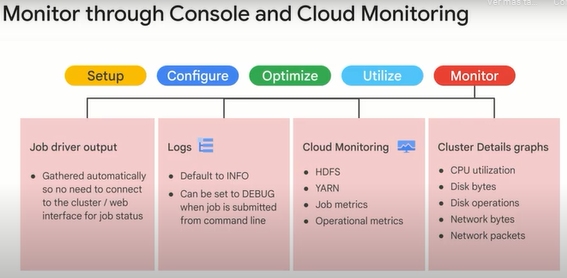
Restartable jobs must be designed to be item potent, and to detect successorship and restore state.



Lastly, after you submit your job, you'll want to monitor it, you can do so using Cloud Monitoring.

Or you can also build a custom dashboard with graphs and set up monitoring of alert policies to send emails for example, where you can notify if incidents happen.

Any details from HDFS, Yarn, metrics about a particular job or overall metrics for the cluster like CPU utilization, disk and network usage, can all be monitored and alerted on with Cloud Monitoring.



**Cloud Storage instead of HDFS.**

Let's discuss more about using Google Cloud Storage instead of the native Hadoop file system or HDFS.

Network speeds were slow originally.

That's why we kept data as close as possible to the processor.

Now, with petabit networking, you can treat storage and compute independently and move traffic quickly over the network.

Your on-premise Hadoop clusters need local storage on its disk since the same server runs computes on stores jobs.

That's one of the first areas for optimization.

You can run HDFS in the Cloud just by lifting and shifting your Hadoop workloads to Dataproc.

This is often the first step to the Cloud and requires no code changes.

It just works.

But HDFS on the Cloud is a subpar solution in the long run.

This is because of how HDFS works on the clusters with block size, the data locality and the replication of the data in HDFS.

For block size in HDFS, you're tying the performance of input and output to the actual hardware the server is running on.

Again, storage is not elastic in this scenario, you're in the cluster.

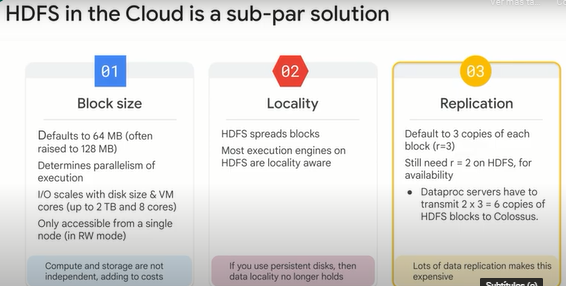
If you run out of persistent disk space on your cluster, you will need to resize even if you don't need the extra compute power.

For data locality, there are similar concerns about storing data on individual persistent disks.

This is especially true when it comes to replication.

In order for HDFS to be highly available, it replicates three copies of each block out to storage.

It would be better to have a storage solution that's separately managed from the constraints of your cluster.



Google's network enables new solutions for big data.

The Jupyter networking fabric within a Google data center delivers over one petabit per second of bandwidth.

To put that into perspective, that's about twice the amount of traffic exchanged on the entire public internet.

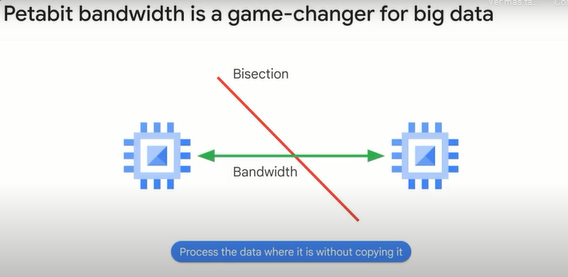
See Cisco's annual estimate of all internet traffic.

If you draw a line somewhere in a network, bisectional bandwidth is the rate of communication at which servers on one side of the line can communicate with servers on the other side.

With enough bisectional bandwidth, any server can communicate with any other server at full network speeds.

With petabit bisectional bandwidth, the communication is so fast that it no longer makes sense to transfer files and store them locally.

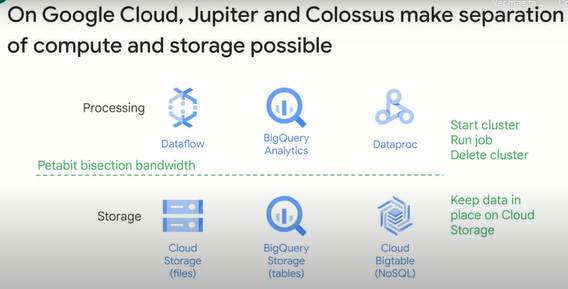
Instead, it makes sense to use the data from where it is stored.



Inside of a Google data center, the internal name for the massively distributed storage layer is called Colossus.

Under the network inside the data center is Jupyter.

Dataproc clusters get the advantage of scaling up and down VMs that they need to do the compute while passing off persistent storage needs with the ultra-fast Jupyter network to a storage products like Cloud Storage, which is controlled by Colossus behind the scenes.



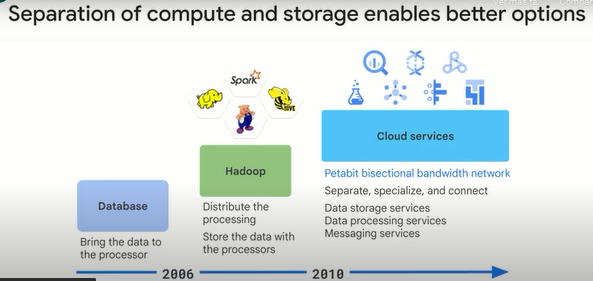
A historical continuum of data management is as follows.

Before 2006, big data meant big databases, database design came from a time when storage was relatively cheap, and processing was expensive.

Around 2006, distributed processing of big data became practical with Hadoop.

Around 2010, BigQuery was released, which was the first of many big data services developed by Google.

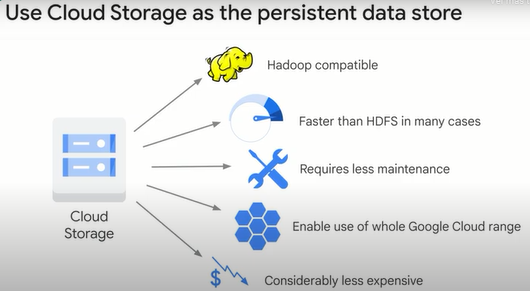
Around 2015, Google launched Dataproc, which provides a managed service for creating Hadoop and Spark clusters and managing data processing workloads.



One of the biggest benefits of Hadoop in the Cloud is that separation of compute and storage.

With Cloud Storage as the backend, you can treat clusters themselves as ephemeral resources, which allows you not to pay for compute capacity when you're not running any jobs.

Also, Cloud Storage is its own completely scalable and durable storage service, which is connected to many other Google Cloud projects.



Cloud storage could be a drop-in replacement for your HDFS backend for Hadoop, the rest of your code would just work.

Also, you can use the Cloud storage connector manually on your non-Cloud Hadoop clusters if you didn't want to migrate your entire cluster to the Cloud yet.

With HDFS, you must over-provision for current data and for data you might have, and you must use persistent disks throughout.

With Cloud Storage however, you pay for exactly what you need when you use it.

Cloud Storage is optimized for large bulk parallel operations.

It has very high throughput, but it has significant latency.

If you have large jobs that are running lots of tiny little blocks, you may be better off with HDFS.

Additionally, you want to avoid iterating sequentially over many nested directories in a single job.

Using Cloud Storage instead of HDFS provides some key benefits due to the distributed service including eliminating bottlenecks and single points of failure.

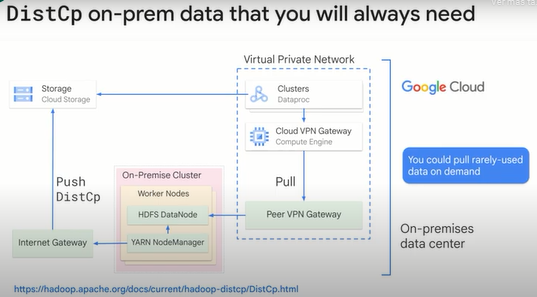
However, there are some disadvantages to be aware of, including the challenges presented by renaming objects and the inability to append to objects.

Cloud Storage is at its core an object store, it only simulates a file directory.

So directory renames in HDFS are not the same as they are in Cloud Storage, but new objects store oriented output committers mitigate this as you see here.

Disk CP is a key tool for moving data.

In general, you want to use a push-based model for any data that you know you will need while pull-based may be a useful model if there is a lot of data that you might not ever need to migrate.



**Optimizing Dataproc.**

Next, let's look at optimizing Dataproc.

Where is your data and where is your cluster?

Knowing your data locality can have a major impact on your performance.

You want to be sure that your data's region and your cluster zone are physically close in distance.

When using Dataproc you can omit the zone and have the Dataproc Auto Zone feature select a zone for you in the region you choose.

While this handy feature can optimize on where to put your cluster it does not know how to anticipate the location of the data you're cluster will be accessing.

Make sure that the Cloud storage bucket is in the same regional location as your Dataproc region.

Is your network traffic being funneled?

Be sure that you do not have any network rules or roots that funnel Cloud storage traffic through a small number of VPN gateways before it reaches your cluster.

There are large network pipes between Cloud Storage and Compute Engine.

You don't want to throttle your bandwidth by sending traffic into a bottleneck in you're google Cloud networking configuration.

How many input files and Hadoop partitions are you trying to deal with?

Make sure you are not dealing with more than around 10,000 input files.

If you find yourself in this situation try to combine or union the data into larger file sizes.

If this file volume reduction means that now you are working with larger datasets more than approximately 50,000 Hadoop partitions you should consider adjusting the setting fs.gs.block.size to a larger value accordingly.

Is the size of your persistent disk limiting your throughput?

Oftentimes when getting started with Google Cloud you may have just a small table that you want to benchmark.

This is generally a good approach as long as you do not choose a persistent disk that assigns to such a small quantity of data, it will most likely limit your performance.

Standard persistent disk scale linearly with volume size.

Did you allocate enough virtual machines to your cluster?

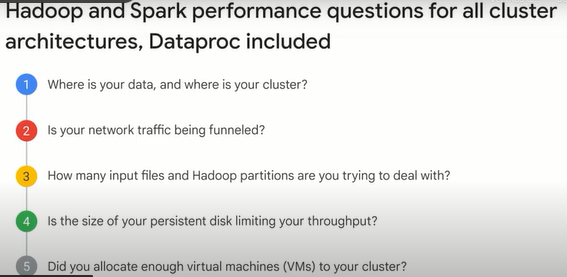
A question that often comes up when migrating from on-premises hardware to Google Cloud is how to accurately size the number of virtual machines needed.

Understanding your workloads is key to identifying a cluster size.

Running prototypes and benchmarking with real data and real jobs is crucial to informing the actual VM allocation decision.

Locally, the ephemeral nature of the Cloud makes it easy to write size clusters for the specific task at hand instead of trying to purchase hardware upfront, thus, you can easily resize your cluster as needed.

Employing job scoped clusters is a common strategy for Dataproc clusters



**Optimizing Dataproc storage.**

Local HDFS is a good option if your jobs require a lot of metadata operations, for example, you have thousands of partitions and directories, and each file size is relatively small.

You modify the HDFS data frequently or you rename directories.

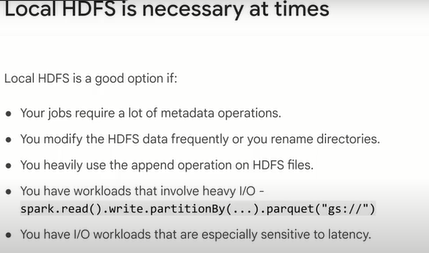
Cloud storage objects are immutable.

So renaming a directory is an expensive operation because it consists of copying all objects to a new key and deleting them afterwards.

You heavily use the append operation on HDFS files, you have workloads that involve heavy IO, for example, you have a lot of partitioned writes such as in this example.

You have IO workloads that are especially sensitive to latency.

For example, you require single digit millisecond latency per storage operation.



In general, we recommend using Cloud Storage as the initial and final source of data in a big data pipeline.

For example, if a workflow contains five Spark jobs in series, the first job retrieves the initial data from Cloud Storage, and then writes shuffled data and intermediate job output to HDFS.

The final Spark job writes its results to Cloud Storage.

Using Dataproc with Cloud storage allows you to reduce the disk requirements and save costs by putting your data there instead of in the HDFS.

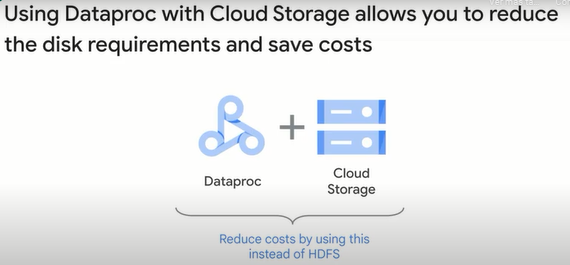
When you keep your data on Cloud storage and don't store it on the local HDFS, you can use smaller disks for your cluster.

By making your cluster truly on demand, you're also able to separate storage and compute as noted earlier which helps you reduce costs significantly.

Even if you store all of your data in Cloud Storage, your Dataproc cluster needs HDFS for certain operations, such as storing control and recovery files, or aggregating logs.

It also needs non-HDFS local disk space for shuffling.

You can reduce the disk size per worker if you are not heavily using the local HDFS.



Here are some options to adjust the size of the local HDFS.

Decrease the total size of the local HDFS by decreasing the size of primary persistent disks for the primary and workers.

The primary persistent disk also contains the boot volume and system libraries.

So allocate at least 100 gigabytes.

Increase the total size of the local HDFS by increasing the size of the primary persistent disk for workers.

Consider this option carefully.

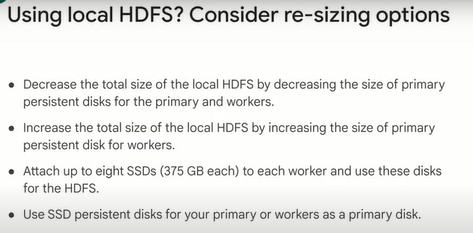
It's rare to have workloads that get better performance by using HDFS with standard persistent disks in comparison to using Cloud storage or local HDFS with SSD.

Attach up to eight SSDs to each worker and use these disks for the HDFS.

This is a good option if you need to use the HDFS for IO intensive workloads, and you need single digit millisecond latency.

Make sure that you use a machine type that has enough CPUs and memory on the worker to support these disks.

And use SSD persistent disks for your primary or workers as a primary disk.



You should understand the repercussions of geography and regions before you configure your data and jobs.

Many Google Cloud services require you to specify regions or zones in which to allocate resources.

The latency of requests can increase when the requests are made from a different region than the one where the resources are stored.

Additionally, if the service is resources and your persistent data are located in different regions, some calls to Google Cloud services might copy all of the required data from one zone to another before processing.

This can have a severe impact on performance.

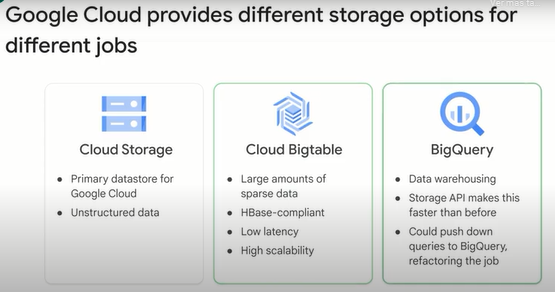
Cloud Storage is the primary way to store unstructured data in Google Cloud, but it isn't the only storage option.

Some of your data might be better suited to storage in products designed explicitly for big data.

You can use Cloud Bigtable to store large amounts of sparse (*dispersa, escasa*) data.

Cloud Bigtable is an HBase compliant API that offers low latency and high scalability to adapt to your jobs.

For data warehousing, you can use BigQuery.



Because Dataproc runs Hadoop on Google Cloud, using a persistent Dataproc cluster to replicate your on-premises setup might seem like the easiest solution.

However, there are some limitations to that approach.

Keeping your data in a persistent HDFS cluster using Dataproc is more expensive than storing your data in Cloud storage, which is what we recommend.

Keeping data in an HDFS cluster also limits your ability to use your data with other Google Cloud products.

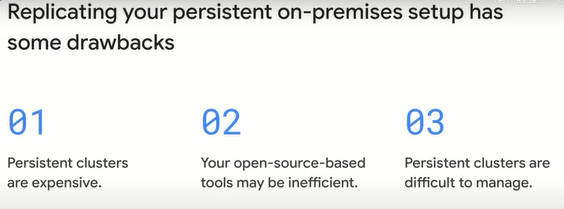
Augmenting or replacing some of your open source based tools with other related Google Cloud services can be more efficient or economical for particular use cases.

Using a single persistent Dataproc cluster for your jobs is more difficult to manage than shifting to targeted clusters that serve individual jobs or job areas.

The most cost-effective and flexible way to migrate your Hadoop system to Google Cloud is to shift away from thinking in terms of large, multi-purpose persistent clusters, and instead think about small, short-lived clusters that are designed to run specific jobs.

You store your data in Cloud storage to support multiple temporary processing clusters.

This model is often called the ephemeral model, because the clusters you use for processing jobs are allocated as needed and are released as jobs finish.



If you have efficient utilization, don't pay for resources that you don't use, employ scheduled deletion.

A fixed amount of time after the cluster enters an idle state, you can automatically set a timer, you can give it a timestamp, and the count starts immediately once the expiration has been set.

You can set a duration, the time in seconds to wait before automatically turning down the cluster.

You can range from 10 minutes as a minimum to 14 days as a maximum at a granularity of one second.

The biggest shift in your approach between running an on-premises Hadoop workflow and running the same workflow on Google Cloud is the shift away from monolithic persistent clusters to specialized ephemeral clusters.

You spin up a cluster when you need to run a job and then delete it when the job completes.

The resources required by your jobs are active only when they're being used, so you only pay for what you use.

This approach enables you to tailor cluster configurations for individual jobs.

Because you aren't maintaining and configuring a persistent cluster, you reduce the costs of resource use and cluster administration.

This section describes how to move your existing Hadoop infrastructure to an ephemeral model.

To get the most from Dataproc, customers need to move to an ephemeral model of only using clusters when they need them.

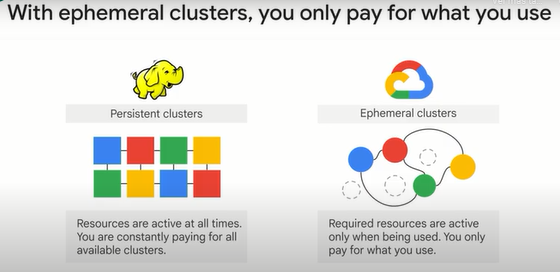
This can be scary because a persistent cluster is comfortable.

With Cloud Storage data persistence and fast boot of Dataproc however, a persistent cluster is a waste of resources.

If a persistent cluster is needed, make it small.

Clusters can be resized anytime.

Ephemeral model is the recommended route, but it requires storage to be decoupled from compute.

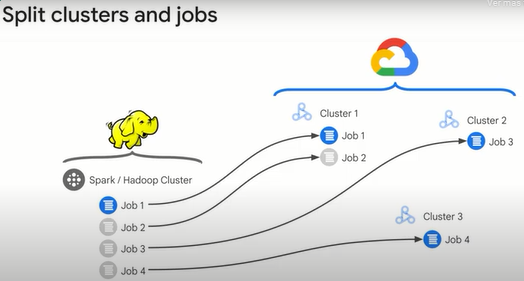


Separate job shapes and separate clusters, decompose even further with job-scoped clusters.

Isolate Dev, staging and production environments by running on separate clusters.

Read from the same underlying data source on Cloud Storage.

Add appropriate ACLs to service accounts to protect data.



The point of ephemeral clusters is to use them only for the job's lifetime.

When it's time to run a job, follow this process.

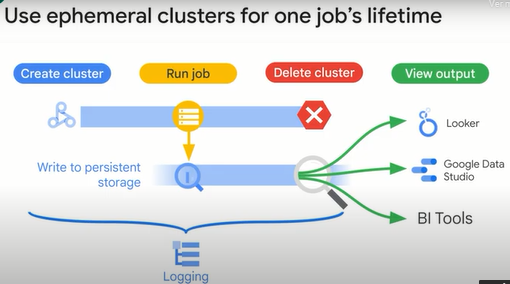
Create a properly configured cluster.

Run your job, sending output to Cloud Storage or another persistent location.

Delete the cluster.

Use your job output however you need to.

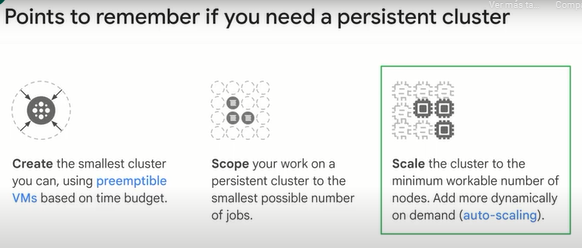
View logs in Cloud Logging or Cloud storage.



If you can't accomplish your work without a persistent cluster, you can create one.

This option may be costly and isn't recommended if there is a way to get your job done on ephemeral clusters.

You can minimize the cost of a persistent cluster by creating the smallest cluster you can, scoping your work on that cluster to the smallest possible number of jobs, and scaling the cluster to the minimum workable number of nodes, adding more dynamically to meet demand.



**Optimizing Dataproc templates and autoscaling.**

The dataproc workflow template is a YAML file that is processed through a directed acyclic graph, or DAG.

It can create a new cluster, select from an existing cluster, submit jobs, hold jobs for submission until dependencies can complete, and it can delete a cluster when the job is done.

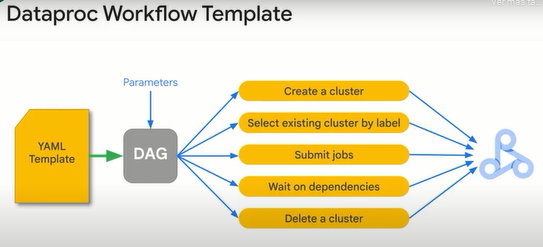
It's currently only available through the gcloud command and the REST API.

It cannot be accessed through the Cloud Console.

The workflow template becomes active when it is instantiated into the DAG.

The template can be submitted multiple times with different parameter values.

You can also write a template inline in the gcloud command, and you can list workflows and workflow metadata to help diagnose issues.



Here is an example of a dataproc workflow template.

First, we get all the things that need to be installed in the cluster using our startup scripts and manually echoing pip install commands like the ones seen here to install matplotlib.

You can have multiple startup shell scripts run like you see in this example.

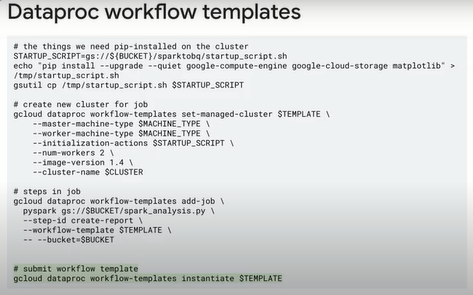
Next, we use the gcloud command for creating a new cluster in advance of running our job.

We specify cluster parameters like the template to be used in our desired architecture and what machine types and image versions we want for hardware and software.

After that, we need to add a job to the newly created cluster.

In this example, we have a Spark job written in Python that exists in a cloud storage bucket that we control.

Lastly, we need to submit this template itself as a new workflow template as you see with the last command.



Dataproc autoscaling provides clusters that size themself to the needs of the enterprise.

Key features include jobs are fire and forget.

There is no need to manually intervene when a cluster is over or undercapacity.

You can choose between standard and preemptible workers, and you can save resources, quota and costs at any point in time.

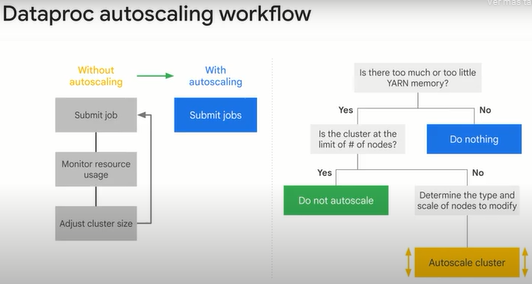
Autoscaling policies provide fine-grained control.

This is based on the difference between YARN, pending and available memory.

If more memory is needed, then you scale up.

If there is excess memory, you scale down.

Obey VM limits and scale based on scale factor.



Autoscaling improvements can be summarized as follows.

Even more fine-grained controls.

Autoscaling policies can be updated or removed at any time.

The minimum scaling interval has been reduced from 10 minutes to 2 minutes, and autoscaling policies can be shared between multiple clusters.

Autoscaling is now easier to understand.

YARN and HDFS dashboards can be viewed in a cluster page, and the autoscaling decision history is available in cloud logging.

And job stability is provided through the ability to scale, map reduce and Spark jobs without losing progress.

Dataproc autoscaling provides flexible capacity for more efficient utilization, making scaling decisions based on Hadoop YARN metrics.

It's designed to be used only with off-cluster persistent data, not on-cluster HDFS or H-Based.

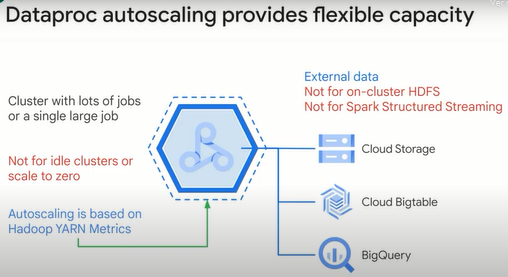
It works best with a cluster that processes a lot of jobs or that processes a single large job.

It doesn't support Spark Structured Streaming, a streaming service built on top of Spark SQL.

It's also not designed to scale to zero, so it's not the best for sparsely utilized or idle clusters.

In these cases, it's equally fast to terminate a cluster that's idle and create a new cluster when it's needed.

For that purpose, you would look at dataproc workflows or cloud composer and cluster schedule deletion.



One of the things that you want to consider when working with autoscaling is setting the initial workers.

The number of initial workers is set from worker nodes' nodes minimum.

Setting this value ensures that the cluster comes up to basic capacity faster than if you let autoscaling handle it because autoscaling might require multiple autoscale periods to scale up.

The primary minimum number of workers may be the same as the cluster nodes' minimum.

There is a maximum that caps the number of worker nodes.

If there is a heavy load on the cluster, autoscaling determines it is time to scale up.

The scale-up factor determines how many nodes to launch.

This will commonly be one node.

But if you knew that a lot of demand would occur at once, maybe you want to scale up faster.

After the action, there is a cooldown period to let things settle before autoscaling evaluation occurs again.

The cooldown period reduces the chances that the cluster will start and terminate nodes at the same time.

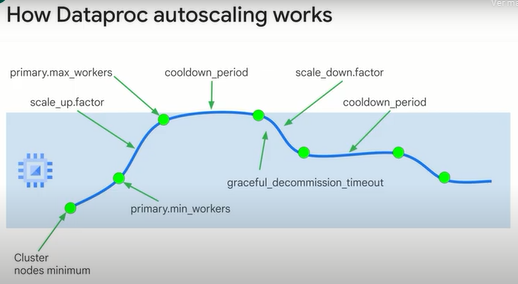
In this example, the extra capacity isn't needed, and there is a graceful decommission timeout to give running jobs a chance to complete before the node goes out of service.

Notice there is a scaledown factor.

In this case, it is scaling down by one node at a time for a more leisurely reduction in capacity.

After the action, there is another cooldown period, and a second scaledown, resulting in a return to the minimum number of workers.

A secondary minimum number of workers, a maximum number of workers controls the scale preemptible (preferencial) workers.



**Optimizing Dataproc monitoring.**

In Google Cloud, you can use Cloud Logging and Cloud Monitoring to view and customize logs, and to monitor jobs and resources.

The best way to find what error caused a Spark job failure is to look at the driver output and the logs generated by the Spark executioners.

Note, however, that if you submit a Spark job by connecting directly to the primary node using SSH, it's not possible to get the driver output.

You can retrieve the driver program output by using the Cloud Console or by using gcloud command.

The output is also stored in the Cloud storage bucket of the Dataproc cluster.

All other logs are located in different files inside the machines of the cluster.

It's possible to see the logs for each container from the spark app Web UI, or from the history server after the program ends in the executer's tab.

You need to browse through each Spark container to view each log.

If you write logs or print to standard out or standard air in your application code, the logs are saved in the redirection of standard out or standard air.

In a Dataproc cluster, Yarn is configured to collect all these logs by default, and they're available in Cloud Logging.

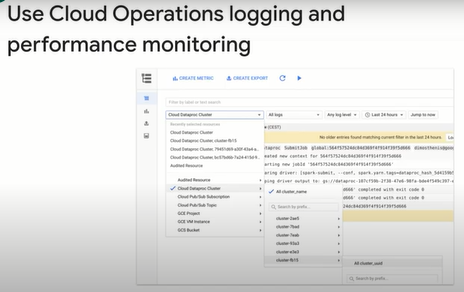
Logging provides a consolidated and concise view of all logs so that you don't need to spend time browsing among container logs to find errors.

This screen shows the login page in the Cloud Console.

You can view all logs from your Dataproc cluster by selecting the cluster's name in the selector menu.

Don't forget to expand the time duration in the time range selector.

You can get logs from a Spark application by filtering by its ID, you can get the application ID from the driver output.



To find logs faster, you can create and use your own labels for each cluster or for each Dataproc job.

For example, you can create a label with the key environment or ENV as the value in the exploration and use it for your data exploration job.

You can then get logs for all exploration job creations by filtering with the label environment with a value exploration in logging.

Note that this filter will not return all logs for this job, only the resource creation logs.

You can set the driver log level using the following G Cloud command: G Cloud, Dataproc, jobs, submit, Hadoop, with the parameter driver log levels.

You set the log level for the rest of the application from the spark context, for example, Spark dot Spark context dot set log level.

And for here, we'll just say the example is debug.

Cloud Monitoring can monitor the cluster's CPU, disk, network usage and Yarn resources.

You can create a custom dashboard to get up to date charts for these and other metrics.

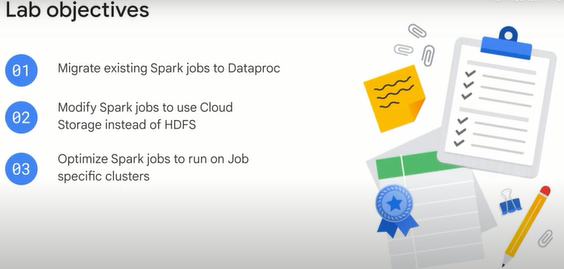
Dataproc runs on top of Compute Engine.

If you want to visualize CPU usage, disk IO or networking metrics in a chart, you need to select a Compute Engine VM instance as the resource type, and then filter by the cluster name.

This diagram shows an example of the output.

To view metrics for Spark queries, jobs, stages, or tasks, connect to the spark applications Web UI.

**Lab: Running Apache Spark jobs on Cloud Dataproc.**



**Task 1. Lift and shift**

Migrate existing Spark jobs to Cloud Dataproc

You will create a new Cloud Dataproc cluster and then run an imported Jupyter notebook that uses the cluster's default local Hadoop Distributed File system (HDFS) to store source data and then process that data just as you would on any Hadoop cluster using Spark. This demonstrates how many existing analytics workloads such as Jupyter notebooks containing Spark code require no changes when they are migrated to a Cloud Dataproc environment.

Configure and start a Cloud Dataproc cluster

1. In the GCP Console, on the **Navigation menu**, in the **Analytics** section, click **Dataproc**.
2. Click **Create Cluster**.
3. Click **Create** for the item **Cluster on Compute Engine**.
4. Enter sparktodp for **Cluster Name**.
5. In the **Versioning** section, click **Change** and select **2.0 (Debian 10, Hadoop 3.2, Spark 3.1)**.

This version includes Python3, which is required for the sample code used in this lab.

1. Click **Select**.
2. In the **Components** > **Component gateway** section, select **Enable component gateway**.
3. Under **Optional components**, Select **Jupyter Notebook**.
4. Click **Create**.

The cluster should start in a few minutes. You can proceed to the next step without waiting for the Cloud Dataproc Cluster to fully deploy.

Clone the source repository for the lab

In the Cloud Shell you clone the Git repository for the lab and copy the required notebook files to the Cloud Storage bucket used by Cloud Dataproc as the home directory for Jupyter notebooks.

1. To clone the Git repository for the lab enter the following command in Cloud Shell:

git -C ~ clone https://github.com/GoogleCloudPlatform/training-data-analyst

1. To locate the default Cloud Storage bucket used by Cloud Dataproc enter the following command in Cloud Shell:

export DP\_STORAGE="gs://$(gcloud dataproc clusters describe sparktodp --region=us-central1 --format=json | jq -r '.config.configBucket')"

1. To copy the sample notebooks into the Jupyter working folder enter the following command in Cloud Shell:

gsutil -m cp ~/training-data-analyst/quests/sparktobq/\*.ipynb $DP\_STORAGE/notebooks/jupyter

Log in to the Jupyter Notebook

As soon as the cluster has fully started up you can connect to the Web interfaces. Click the refresh button to check as it may be deployed fully by the time you reach this stage.

1. On the Dataproc Clusters page wait for the cluster to finish starting and then click the name of your cluster to open the **Cluster details** page.
2. Click **Web Interfaces**.
3. Click the **Jupyter** link to open a new Jupyter tab in your browser.

This opens the Jupyter home page. Here you can see the contents of the /notebooks/jupyter directory in Cloud Storage that now includes the sample Jupyter notebooks used in this lab.

1. Under the **Files** tab, click the **GCS** folder and then click **01\_spark.ipynb** notebook to open it.
2. Click **Cell** and then **Run All** to run all of the cells in the notebook.
3. Page back up to the top of the notebook and follow as the notebook completes runs each cell and outputs the results below them.

You can now step down through the cells and examine the code as it is processed so that you can see what the notebook is doing. In particular pay attention to where the data is saved and processed from.

* The first code cell fetches the source data file, which is an extract from the KDD Cup competition from the Knowledge, Discovery, and Data (KDD) conference in 1999. The data relates to computer intrusion detection events.

!wget https://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/kddcup.data\_10\_percent.gz

* In the second code cell, the source data is copied to the default (local) Hadoop file system.

!hadoop fs -put kddcup\* /

* In the third code cell, the command lists contents of the default directory in the cluster's HDFS file system.

!hadoop fs -ls /

Reading in data

The data are gzipped CSV files. In Spark, these can be read directly using the textFile method and then parsed by splitting each row on commas.

The Python Spark code starts in cell In[4].

* In this cell Spark SQL is initialized and Spark is used to read in the source data as text and then returns the first 5 rows.

from pyspark.sql import SparkSession, SQLContext, Row

spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext

data\_file = "hdfs:///kddcup.data\_10\_percent.gz"

raw\_rdd = sc.textFile(data\_file).cache()

raw\_rdd.take(5)

* In cell In [5] each row is split, using , as a delimiter and parsed using a prepared inline schema in the code.

csv\_rdd = raw\_rdd.map(lambda row: row.split(","))

parsed\_rdd = csv\_rdd.map(lambda r: Row(

duration=int(r[0]),

protocol\_type=r[1],

service=r[2],

flag=r[3],

src\_bytes=int(r[4]),

dst\_bytes=int(r[5]),

wrong\_fragment=int(r[7]),

urgent=int(r[8]),

hot=int(r[9]),

num\_failed\_logins=int(r[10]),

num\_compromised=int(r[12]),

su\_attempted=r[14],

num\_root=int(r[15]),

num\_file\_creations=int(r[16]),

label=r[-1]

)

)

parsed\_rdd.take(5)

Spark analysis

In cell In [6] a Spark SQL context is created and a Spark dataframe using that context is created using the parsed input data from the previous stage.

1. Row data can be selected and displayed using the dataframe's .show() method to output a view summarizing a count of selected fields:

sqlContext = SQLContext(sc)

df = sqlContext.createDataFrame(parsed\_rdd)

connections\_by\_protocol = df.groupBy('protocol\_type').count().orderBy('count', ascending=False)

connections\_by\_protocol.show()

The .show() method produces an output table similar to this:

+-------------+------+

|protocol\_type| count|

+-------------+------+

| icmp|283602|

| tcp|190065|

| udp| 20354|

+-------------+------+

SparkSQL can also be used to query the parsed data stored in the Dataframe.

1. In cell In [7] a temporary table (connections) is registered that is then referenced inside the subsequent SparkSQL SQL query statement:

df.registerTempTable("connections")

attack\_stats = sqlContext.sql("""

SELECT

protocol\_type,

CASE label

WHEN 'normal.' THEN 'no attack'

ELSE 'attack'

END AS state,

COUNT(\*) as total\_freq,

ROUND(AVG(src\_bytes), 2) as mean\_src\_bytes,

ROUND(AVG(dst\_bytes), 2) as mean\_dst\_bytes,

ROUND(AVG(duration), 2) as mean\_duration,

SUM(num\_failed\_logins) as total\_failed\_logins,

SUM(num\_compromised) as total\_compromised,

SUM(num\_file\_creations) as total\_file\_creations,

SUM(su\_attempted) as total\_root\_attempts,

SUM(num\_root) as total\_root\_acceses

FROM connections

GROUP BY protocol\_type, state

ORDER BY 3 DESC

""")

attack\_stats.show()

You will see output similar to this truncated example when the query has finished:

+-------------+---------+----------+--------------+--

|protocol\_type| state|total\_freq|mean\_src\_bytes|

+-------------+---------+----------+--------------+--

| icmp| attack| 282314| 932.14|

| tcp| attack| 113252| 9880.38|

| tcp|no attack| 76813| 1439.31|

...

...

| udp| attack| 1177| 27.5|

+-------------+---------+----------+--------------+--

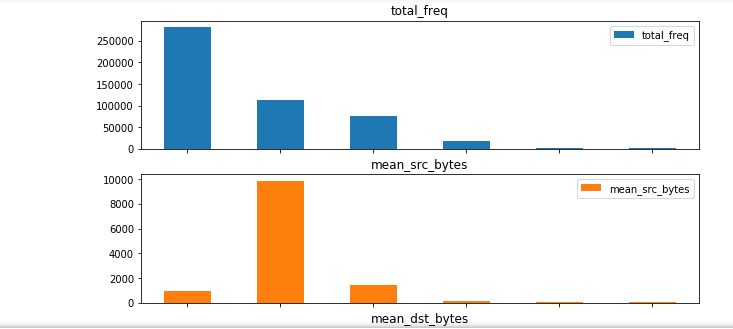
And you can now also display this data visually using bar charts.

1. The last cell, In [8] uses the %matplotlib inline Jupyter magic function to redirect matplotlib to render a graphic figure inline in the notebook instead of just dumping the data into a variable. This cell displays a bar chart using the attack\_stats query from the previous step.

%matplotlib inline

ax = attack\_stats.toPandas().plot.bar(x='protocol\_type', subplots=True, figsize=(10,25))

The first part of the output should look like the following chart once all cells in the notebook have run successfully. You can scroll down in your notebook to see the complete output chart.



**Task 2. Separate compute and storage**

Modify Spark jobs to use Cloud Storage instead of HDFS

Taking this original 'Lift & Shift' sample notebook you will now create a copy that decouples the storage requirements for the job from the compute requirements. In this case, all you have to do is replace the Hadoop file system calls with Cloud Storage calls by replacing hdfs:// storage references with gs:// references in the code and adjusting folder names as necessary.

You start by using the cloud shell to place a copy of the source data in a new Cloud Storage bucket.

1. In the Cloud Shell create a new storage bucket for your source data:

export PROJECT\_ID=$(gcloud info --format='value(config.project)')

gsutil mb gs://$PROJECT\_ID

1. In the Cloud Shell copy the source data into the bucket:

wget https://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/kddcup.data\_10\_percent.gz

gsutil cp kddcup.data\_10\_percent.gz gs://$PROJECT\_ID/

Make sure that the last command completes and the file has been copied to your new storage bucket.

1. Switch back to the 01\_spark Jupyter Notebook tab in your browser.
2. Click **File** and then select **Make a Copy**.
3. When the copy opens, click the **01\_spark-Copy1** title and rename it to De-couple-storage.
4. Open the Jupyter tab for 01\_spark.
5. Click **File** and then **Save and checkpoint** to save the notebook.
6. Click **File** and then **Close and Halt** to shutdown the notebook.

* If you are prompted to confirm that you want to close the notebook click **Leave** or **Cancel**.

1. Switch back to the De-couple-storage Jupyter Notebook tab in your browser, if necessary.

You no longer need the cells that download and copy the data onto the cluster's internal HDFS file system so you will remove those first.

To delete a cell, you click in the cell to select it and then click the **cut selected cells** icon (the scissors) on the notebook toolbar.

1. Delete the initial comment cells and the first three code cells ( In [1], In [2], and In [3]) so that the notebook now starts with the section **Reading in Data**.

You will now change the code in the first cell ( still called In[4] unless you have rerun the notebook ) that defines the data file source location and reads in the source data. The cell currently contains the following code:

from pyspark.sql import SparkSession, SQLContext, Row

spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext

data\_file = "hdfs:///kddcup.data\_10\_percent.gz"

raw\_rdd = sc.textFile(data\_file).cache()

raw\_rdd.take(5)

1. Replace the contents of cell In [4] with the following code. The only change here is create a variable to store a Cloud Storage bucket name and then to point the data\_file to the bucket we used to store the source data on Cloud Storage:

from pyspark.sql import SparkSession, SQLContext, Row

gcs\_bucket='[Your-Bucket-Name]'

spark = SparkSession.builder.appName("kdd").getOrCreate()

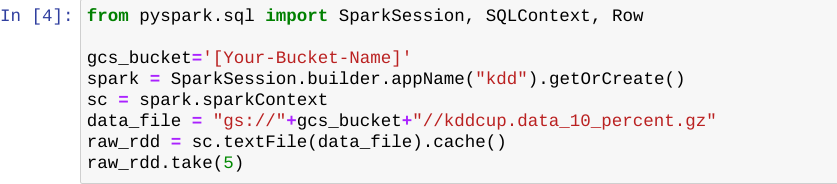
sc = spark.sparkContext

data\_file = "gs://"+gcs\_bucket+"//kddcup.data\_10\_percent.gz"

raw\_rdd = sc.textFile(data\_file).cache()

raw\_rdd.take(5)

When you have replaced the code the first cell will look similar to the following, with your lab project ID as the bucket name:



1. In the cell you just updated, replace the placeholder [Your-Bucket-Name] with the name of the storage bucket you created in the first step of this section. You created that bucket using the Project ID as the name, which you can copy here from the Qwiklabs lab login information panel on the left of this screen. Replace all of the placeholder text, including the brackets [].
2. Click **Cell** and then **Run All** to run all of the cells in the notebook.

You will see exactly the same output as you did when the file was loaded and run from internal cluster storage. Moving the source data files to Cloud Storage only requires that you repoint your storage source reference from hdfs:// to gs://.

**Task 3. Deploy Spark jobs**

Optimize Spark jobs to run on Job specific clusters

You now create a standalone Python file, that can be deployed as a Cloud Dataproc Job, that will perform the same functions as this notebook. To do this you add magic commands to the Python cells in a copy of this notebook to write the cell contents out to a file. You will also add an input parameter handler to set the storage bucket location when the Python script is called to make the code more portable.

1. In the De-couple-storage Jupyter Notebook menu, click **File** and select **Make a Copy**.
2. When the copy opens, click the **De-couple-storage-Copy1** and rename it to PySpark-analysis-file.
3. Open the Jupyter tab for De-couple-storage.
4. Click **File** and then **Save and checkpoint** to save the notebook.
5. Click **File** and then **Close and Halt** to shutdown the notebook.

* If you are prompted to confirm that you want to close the notebook click **Leave** or **Cancel**.

1. Switch back to the PySpark-analysis-file Jupyter Notebook tab in your browser, if necessary.
2. Click the first cell at the top of the notebook.
3. Click **Insert** and select **Insert Cell Above**.
4. Paste the following library import and parameter handling code into this new first code cell:

%%writefile spark\_analysis.py

import matplotlib

matplotlib.use('agg')

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("--bucket", help="bucket for input and output")

args = parser.parse\_args()

BUCKET = args.bucket

The %%writefile spark\_analysis.py Jupyter magic command creates a new output file to contain your standalone python script. You will add a variation of this to the remaining cells to append the contents of each cell to the standalone script file.

This code also imports the matplotlib module and explicitly sets the default plotting backend via matplotlib.use('agg') so that the plotting code runs outside of a Jupyter notebook.

1. For the remaining cells insert %%writefile -a spark\_analysis.py at the start of each Python code cell. These are the five cells labelled **In [x]**.

%%writefile -a spark\_analysis.py

For example the next cell should now look as follows:

%%writefile -a spark\_analysis.py

from pyspark.sql import SparkSession, SQLContext, Row

spark = SparkSession.builder.appName("kdd").getOrCreate()

sc = spark.sparkContext

data\_file = "gs://{}/kddcup.data\_10\_percent.gz".format(BUCKET)

raw\_rdd = sc.textFile(data\_file).cache()

#raw\_rdd.take(5)

1. Repeat this step, inserting %%writefile -a spark\_analysis.py at the start of each code cell until you reach the end.
2. In the last cell, where the Pandas bar chart is plotted, remove the %matplotlib inline magic command.

**Note:**You must remove this inline matplotlib Jupyter magic directive or your script will fail when you run it.

1. Make sure you have selected the last code cell in the notebook then, in the menu bar, click **Insert** and select **Insert Cell Below**.
2. Paste the following code into the new cell:

%%writefile -a spark\_analysis.py

ax[0].get\_figure().savefig('report.png');

Add another new cell at the end of the notebook and paste in the following:

%%writefile -a spark\_analysis.py

import google.cloud.storage as gcs

bucket = gcs.Client().get\_bucket(BUCKET)

for blob in bucket.list\_blobs(prefix='sparktodp/'):

blob.delete()

bucket.blob('sparktodp/report.png').upload\_from\_filename('report.png')

1. Add a new cell at the end of the notebook and paste in the following:

%%writefile -a spark\_analysis.py

connections\_by\_protocol.write.format("csv").mode("overwrite").save(

"gs://{}/sparktodp/connections\_by\_protocol".format(BUCKET))

Test automation

You now test that the PySpark code runs successfully as a file by calling the local copy from inside the notebook, passing in a parameter to identify the storage bucket you created earlier that stores the input data for this job. The same bucket will be used to store the report data files produced by the script.

1. In the PySpark-analysis-file notebook add a new cell at the end of the notebook and paste in the following:

BUCKET\_list = !gcloud info --format='value(config.project)'

BUCKET=BUCKET\_list[0]

print('Writing to {}'.format(BUCKET))

!/opt/conda/miniconda3/bin/python spark\_analysis.py --bucket=$BUCKET

This code assumes that you have followed the earlier instructions and created a Cloud Storage Bucket using your lab Project ID as the Storage Bucket name. If you used a different name modify this code to set the BUCKET variable to the name you used.

1. Add a new cell at the end of the notebook and paste in the following:

!gsutil ls gs://$BUCKET/sparktodp/\*\*

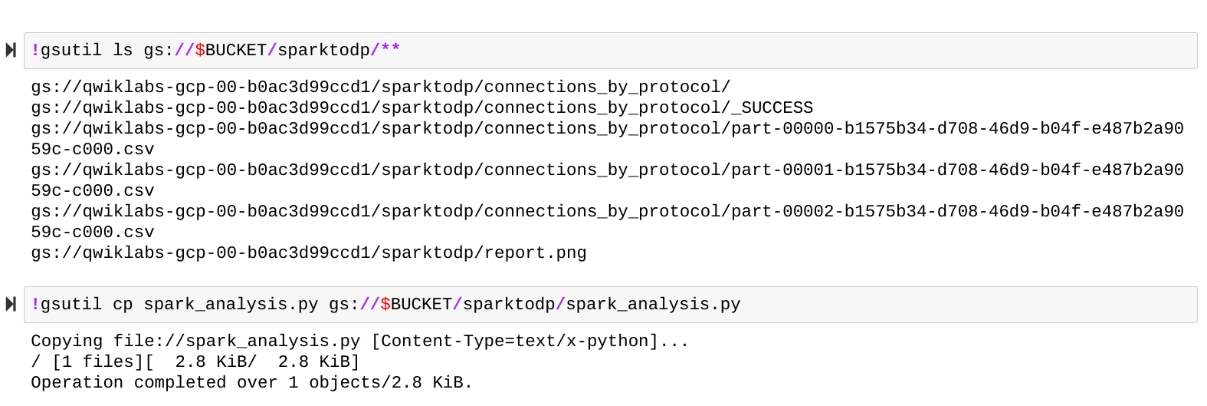
This lists the script output files that have been saved to your Cloud Storage bucket.

1. To save a copy of the Python file to persistent storage, add a new cell and paste in the following:

!gsutil cp spark\_analysis.py gs://$BUCKET/sparktodp/spark\_analysis.py

Click **Cell** and then **Run All** to run all of the cells in the notebook.

If the notebook successfully creates and runs the Python file you should see output similar to the following for the last two cells. This indicates that the script has run to completion saving the output to the Cloud Storage bucket you created earlier in the lab.



**Note:**The most likely source of an error at this stage is that you did not remove the matplotlib directive in **In [7]**. Recheck that you have modified all of the cells as per the instructions above, and have not skipped any steps.

Run the Analysis Job from Cloud Shell.

1. Switch back to your Cloud Shell and copy the Python script from Cloud Storage so you can run it as a Cloud Dataproc Job:

gsutil cp gs://$PROJECT\_ID/sparktodp/spark\_analysis.py spark\_analysis.py

Create a launch script:

nano submit\_onejob.sh

Paste the following into the script:

#!/bin/bash

gcloud dataproc jobs submit pyspark \

--cluster sparktodp \

--region us-central1 \

spark\_analysis.py \

-- --bucket=$1

1. Press CTRL+X then Y and Enter key to exit and save.
2. Make the script executable:

chmod +x submit\_onejob.sh

1. Launch the PySpark Analysis job:

./submit\_onejob.sh $PROJECT\_ID

1. In the Cloud Console tab navigate to the **Dataproc** > **Clusters** page if it is not already open.
2. Click **Jobs**.
3. Click the name of the job that is listed. You can monitor progress here as well as from the Cloud shell. Wait for the Job to complete successfully.
4. Navigate to your storage bucket and note that the output report, /sparktodp/report.png has an updated time-stamp indicating that the stand-alone job has completed successfully.

The storage bucket used by this Job for input and output data storage is the bucket that is used just the Project ID as the name.

1. Navigate back to the **Dataproc** > **Clusters** page.
2. Select the **sparktodp** cluster and click **Delete**. You don't need it any more.
3. Click **CONFIRM**.
4. Close the **Jupyter** tabs in your browser.

**Quizz: Executing Spark on Dataproc.**

1.Which of the following statements are true about Dataproc? (Select all 2 correct answers)

**Lets you run Spark and Hadoop clusters with minimal administration**

**Helps you create job-specific clusters without HDFS**

Streamlined API for Spark and Hadoop programming

2. Match each of the terms with what they do when setting up clusters in Dataproc

|  |  |
| --- | --- |
| Term | Definition |
| \_\_ 1. Zone | A. Costs less but may not be available always |
| \_\_ 2. Standard Cluster mode | B. Determines the Google data center where compute nodes will be |
| \_\_ 3. Preemptible | C. Provides 1 primary and N workers |

A-B-C

C-B-A

C-A-B

**B-C-A**

3. Dataproc provides the ability for Spark programs to separate compute and storage by:

**Reading and writing data directly from/to Cloud Storage**

Setting individual zones for compute and storage

Mirroring data on both Cloud Storage and HDFS

Pre-copying data from Cloud Storage to persistent disk on cluster startup