dremio

Dremio Software

Cluster Configuration

Introduction

This document outlines the best practices to establish the initial configuration of a Dremio cluster.

Dremio Workload Management can be customized to improve wait times and query execution times. These configurations can be tuned as we build the use case or as the platform evolves. However, we should start with some baseline configurations. This document describes those initial baseline configurations.

Node Configuration

Regardless of the deployment model, Dremio recommends the node configurations below:

Component	vCPU	RAM (GB)	Disk Size (GB)
Coordinator	16	64	500
Executor	16	128	250
ZooKeeper	2	8	10

If you notice that the Command Pool Wait (which can be seen in the Job Profile/Job Summary) time is increasing, the number of cores should be increased for the current concurrency of the queries. We recommend scaling the coordinator to 32 cores.

In Kubernetes deployments, you can create node pools with appropriate system configuration and number of nodes for each component.

Here is a sample values.yaml with three node pools for Coordinator, Executor, and Zookeeper.

```
nodeSelector:
    cloud.google.com/gke-nodepool: coordpool
nodeSelector:
    cloud.google.com/gke-nodepool: executorpool
nodeSelector:
    cloud.google.com/gke-nodepool: zkpool
```

In AWSE deployments, by default, the coordinator uses m5d.2xlarge, and you can choose a bigger instance type if needed. Also, in this deployment type, ZooKeeper is embedded in the coordinator. For executors, you can select an instance type and number of instances. Each engine can have its own instance type and the number of executors.

CPU Configuration

For Kubernetes deployments, the number of CPUs for each component can be specified in the Helm chart. Reserve 2 CPUs for Kubernetes system pods and operating system; e.g., if the node has 16 CPUs, you would allocate 14 to Dremio. If the node has 32 CPUs, you would allocate 30 CPUs.

Dremio uses the available CPUs for other deployment types, and CPU configuration is not required.

Here is a sample values.yaml highlighted with CPU settings in Kubernetes deployments.

```
# Dremio Coordinator
coordinator:
  # CPU & Memory
  # Memory allocated to each coordinator, expressed in MB.
  # CPU allocated to each coordinator, expressed in CPU cores.
  cpu: 14
  memory: 122800

# Dremio Executor
executor:
  # CPU & Memory
  # Memory allocated to each executor, expressed in MB.
  # CPU allocated to each executor, expressed in CPU cores.
  cpu: 14
  memory: 122800

# Zookeeper
zookeeper:
  # The Zookeeper image used in the cluster.
  image: zookeeper
  imageTag: 3.8-temurin

  # CPU & Memory
  # Memory allocated to each zookeeper, expressed in MB.
  # CPU allocated to each zookeeper, expressed in CPU cores.
  cpu: 0.5
  memory: 1024
  count: 3
```

Memory Configuration

In the Dremio cluster, memory is configured for the Coordinator and Executors. The memory requirements for each node type are different. The Coordinator needs more heap memory than direct memory, whereas Executors need more direct memory than heap memory. The memory must be configured in /etc/dremio/dremio-env as described here.

For Kubernetes deployments, memory can be configured in values.yaml, as shown below. In calculating memory settings, multiply the total available memory by 0.875; e.g., if the machine has 128 GB of physical memory, you would allocate 112 GB of memory to Dremio.

As described here, Dremio automatically allocates this value to HEAP and DIRECT memory separately for JVM.

Here is a sample values.yaml highlighted with memory settings in Kubernetes deployments.

```
# Dremio Coordinator
coordinator:
  # CPU & Memory
  # Memory allocated to the coordinator, expressed in MB.
  # CPU allocated to the coordinator, expressed in CPU cores.
  cpu: 14
  memory: 61800

# Dremio Executor
executor:
  # CPU & Memory
  # Memory allocated to each executor, expressed in MB.
  # CPU allocated to each executor, expressed in CPU cores.
  cpu: 14
  memory: 122800

# Zookeeper
zookeeper:
  # The Zookeeper image used in the cluster.
  image: zookeeper
  imageTag: 3.8-temurin

  # CPU & Memory
  # Memory allocated to each zookeeper, expressed in MB.
  # CPU allocated to each zookeeper, expressed in CPU cores.
  cpu: 0.5
  memory: 1024
  count: 3
```

For Standalone deployments, make sure that dremio-env is updated in all executors.

Disk Configuration

There are three different disk types in Dremio:

- **Coordinator - Metadata store disk / KVstore:** There will be lots of random IO reads and writes, so the disk requires lots of IOPS and throughput. If this disk is slow, the entire query planning might be slow and slow down all queries.
- **Executor - Spill disk:** This disk is used when the data does not fit into memory and it spills the data to the disk. Usually this disk should see sequential write and read access and a high throughput is required.
- **Executor - C3 cache disk:** Data of former or current queries are cached on this disk. This disk

sees a high throughput and a lot of random access. The disk should provide many IOPS, otherwise it might slow down queries. C3 can also be disabled if there are only slow disks available.

- **Zookeeper - data disk:** The disk has no high demands on throughput and performance. Only little data is stored.

AWS Cloud Disks

The coordinator disk size should start from 500 GB and use IO2 storage with 5000 IOPS (10 IOPS per GB in the storage class). IO2 storage is recommended because of the higher durability and performance.
The executor spilling disk and C3 disk should use gp3 storage starting with 300 GB storage and use the free 3000 IOPS and 125 MB/s throughput or more.
For Zookeeper a 16 GB disk of gp2 or gp3 storage is recommended.

For Kubernetes / EKS: We recommend EBS CSI. To add the storage class, follow this document: https://docs.aws.amazon.com/eks/latest/userguide/managing-ebs-csi.html. The storage classes for io2, gp2 and gp3 need to be created, since they are not available out-of-the-box.

More information:
https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-storage-compare-volume-types.html

Azure Managed Disks

The coordinator disk should use a minimum of 512 GB premium storage, but 1 TB is recommended (P15 with 5000 IOPS). **Do not use standard disks**, because they are capped at 60 MB/s throughput and only have 500 IOPS available.
The executor spilling and C3 cache should use managed premium disks as well with a minimum size of 256 GB. More storage is better, since the performance correlates with the disk size.
For Zookeeper a 16 GB disk of standard storage is recommended.

For Kubernetes / AKS: Use the storage class ‘managed-premium’ with the sizes described above.

More information:
https://azure.microsoft.com/en-us/pricing/details/managed-disks/

Google Cloud

The SSDs disks in Google Cloud are extremely performant. SSD PD provides 30 IOPS per GB.
For the coordinator use a Zonal SSD PD with 256 GB or more. The executor spill and C3 disk should also use Zonal SSD PD disks starting from 256 GB.
Zookeeper can use a Zonal Premium Disk with 16 GB (standard disks are not performant enough with this small size).

For Kubernetes / GKE: The storage classes are already available. For Zookeeper, use “premium-rwo” and 16 GB. For coordinators and executors use the storage class “premium-rwo” (PD SSD) with the sizes above.
https://cloud.google.com/kubernetes-engine/docs/how-to/persistent-volumes/ssd-pd

More information:
https://cloud.google.com/compute/docs/disks/performance#performance_limits

https://cloud.google.com/compute/docs/disks

On-premise Deployments

The coordinator needs a disk with a minimum throughput of 100 MB/s. The disk should also provide 5000 IOPS or more.
The executor disks for C3 and spilling should provide a minimum throughput of 100 MB/s and at least 3000 IOPS. ZooKeeper has no high requirements. A disk with 30 MB/s throughout and 500 IOPS should be sufficient.

Monitor Disk Saturation

It is important to monitor the disk saturation. It tells you when the IOPS and throughput need to be increased.

The command below returns the disk statistics every second. Once the command gets cancelled using Ctrl+C, it prints the average for the entire time.

The two most important fields are:

- **tps**: Total number of transfers per second issued to physical devices. (IOPS)
- **aqu-sz**: The average queue length of the requests issued to the device. If the value exceeds 3 to 4, you might consider adding more IOPS to the disk.

```
$ sar -p -d 1  -u

Linux 5.15.0-1051-azure (aks-coordpool-35637789-vmss000029)     01/03/24        _x86_64_        (4 CPU)

13:34:33        CPU     %user     %nice   %system   %iowait    %steal     %idle
13:34:34        all      3.27      0.00      0.75      0.25      0.00     95.73

13:34:33          tps     rkB/s     wkB/s     dkB/s   areq-sz    aqu-sz     await     %util DEV
13:34:34         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sr0
13:34:34         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sdb
13:34:34         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sda
13:34:34         2.00      0.00     32.00      0.00     16.00      0.00      1.50      0.40 sdc

Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all      3.27      0.00      0.75      0.25      0.00     95.73

Average:          tps     rkB/s     wkB/s     dkB/s   areq-sz    aqu-sz     await     %util DEV
Average:         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sr0
Average:         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sdb
Average:         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sda
Average:         2.00      0.00     32.00      0.00     16.00      0.00      1.50      0.40 sdc
```

Here is a nicer screenshot of the output above:

```
root@aks-coordpool-35637789-vmss000029:~# sar -p -d 1  -u
Linux 5.15.0-1051-azure (aks-coordpool-35637789-vmss000029)     01/03/24        _x86_64_        (4 CPU)

13:34:33        CPU     %user     %nice   %system   %iowait    %steal     %idle
13:34:34        all      3.27      0.00      0.75      0.25      0.00     95.73

13:34:33          tps     rkB/s     wkB/s     dkB/s   areq-sz    aqu-sz     await     %util DEV
13:34:34         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sr0
13:34:34         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sdb
13:34:34         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sda
13:34:34         2.00      0.00     32.00      0.00     16.00      0.00      1.50      0.40 sdc
^C


Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all      3.27      0.00      0.75      0.25      0.00     95.73

Average:          tps     rkB/s     wkB/s     dkB/s   areq-sz    aqu-sz     await     %util DEV
Average:         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sr0
Average:         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sdb
Average:         0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00 sda
Average:         2.00      0.00     32.00      0.00     16.00      0.00      1.50      0.40 sdc
root@aks-coordpool-35637789-vmss000029:~#
```

Distributed Storage Configuration

The distributed storage stores the data set's metadata (in Iceberg format), reflections, uploads, downloads, and scratch directories. Dremio supports a variety of storage options, as documented here.

If you are using cloud object stores like S3/ADLS/GCS, you don't need to worry about the size of the distributed storage. For AWSE, EKS, AKS, and GKE deployments, the distributed storage is commonly used as the cloud storage as well. Hence, size doesn't need to be further configured.

For Standalone deployments, ensure enough storage is allocated if you use a NAS. If you don't have enough storage, Dremio can't perform well. The size of the distributed storage depends on the size of the data sets, the number of reflections, uploads, downloads, and the number of CTAS in your scratch directory.

Here is a sample values.yaml configured with S3 as a distributed store.

```
aws:
  bucketName: "my-bucket"
  path: "/xxxxx"
  authentication: "accessKeySecret"
  # If using accessKeySecret for authentication against S3, uncomment the lines below and use the values
  # to configure the appropriate credentials.
  credentials:
    accessKey: "xxxxxx"
    secret: "yyyyyyyyyyyyyyyy"
```

Cloud Cache Configuration

Dremio supports Columnar Cloud Cache(C3) for Parquet files on the Cloud Data Sources, such as S3/ADLS/GCS/HDFS, as mentioned here. C3 caches Parquet files on executors' disks. Dremio recommends using SSD/NVMe for best C3 performance. C3 is temporary, and the data is discarded when an executor is restarted. Follow the steps here to configure C3.

For AWSE deployment, C3 is automatically configured based on the EC2 instance type without manual configuration.

For Kubernetes installs, C3 is configured using values.yaml. Set cloudCache.enabled to true. Administrators can configure volumes in the file volumes.size.

Here is a sample values.yaml configured to enable Columnar Cloud Cache:

```
# Dremio C3
  cloudCache:
    enabled: true

    volumes:
    - size: 100Gi
```

If you are seeing a lot of cache misses in the query profile (Go to a job > Click on raw profile > Click on the operator called "TABLE FUNCTION" > there will be a section called "Operator Metrics" and under that, you can see the cache hits and cache misses columns) you can adjust the size of the columnar cache.

By enabling the caching of reflection data in distributed stores, the performance of the queries that use reflections is improved. If you are using a cloud storage provider, such as AWS, GCP, or Microsoft Azure, as a distributed store, it is enabled by default.

If you use HDFS as a distributed store, set dist.caching.enabled support key to true in dremio.conf and restart the cluster.

Follow the best practices to set up cloud caching as described here for the type of disks, size, etc.

Log Directory Configuration

For AWSE and Standalone deployments, the log directory is set to /var/log/dremio by default. Unless you want to change this, no further configuration is needed.

For Kubernetes installs, by default, the log is set to STDOUT. The log directory is configured in values.yaml as follows.

```
extraStartParams: >-
    -Ddremio.log.path=/opt/dremio/data/log
```

If the engine is already running, set this configuration and upgrade the Helm chart.

Authentication Configuration

Dremio supports multiple Authentication types, as described here. By default, Dremio supports Local authentication. In Local authentication, you must create users and groups in Dremio and login using password/PAT. Personal Access Tokens do not need any external authentication mechanism.

We recommend starting with Local authentication, and while the use case development is in progress, you can plan to integrate with external identity providers, including LDAP, AzureAD, and OpenID.

Dremio supports more than one authentication method in a single Dremio cluster. For example, you can use local authentication for some users and LDAP authentication for others. However, Dremio supports only one external identity provider per Dremio instance. For example, you can use local and Azure AD authentication but not Azure AD and LDAP.

Configuring LDAP - https://docs.dremio.com/24.0.x/get-started/cluster-deployments/customizing-configuration/ldap

Configuring AAD/OpenID - https://docs.dremio.com/24.0.x/get-started/cluster-deployments/customizing-configuration/dremio-conf/sso-config

TLS Configuration

Dremio supports encryption for the web, clients, and intra-cluster connections, as described here.

For AWSE deployments, it is recommended to configure web and client TLS termination at load balancer (like NLB) wherever possible. If a load balancer is not used, here is a simple configuration for TLS encryption for web, client, and flight connections.

For AWSE deployments with no load balancer and Standalone deployments, the TLS can be configured as described here. Assuming a valid key and certificate, this creates a pkcs12 file.

```
openssl pkcs12 -export -inkey xxx.key -in yyyy.crt -out /opt/dremio/tls/dremio.pkcs12 -passout pass:
```

Update dremio.conf as follows:

```
eservices.coordinator.web.ssl.enabled: true
services.coordinator.web.ssl.auto-certificate.enabled: false
services.coordinator.web.ssl.keyStore: "/opt/dremio/tls/dremio.pkcs12"

services.coordinator.client-endpoint.ssl.enabled: true
services.coordinator.client-endpoint.ssl.auto-certificate.enabled: false
services.coordinator.client-endpoint.ssl.keyStore: "/opt/dremio/tls/dremio.pkcs12"

services.flight.ssl.enabled: true
services.flight.ssl.auto-certificate.enabled: false
services.flight.ssl.keyStore: "/opt/dremio/tls/dremio.pkcs12"
```

For Kubernetes deployments, the key and certificate should be in pem format. Follow the steps here for configuration.

Workload Management Configuration

The Workload Management (WLM) feature provides the capability to manage cluster resources and workloads. WLM is configured by defining a queue with specific characteristics (such as memory limits, CPU priority, queueing, and runtime timeouts) and then defining rules that specify which query is assigned to which queue and assigning this queue to an engine.

An engine is a group of executor(s) that runs in isolation without sharing their resources (CPU, memory, C3) with other isolated engines. When a query is executed on an engine, it is executed only by the nodes assigned to that engine. This isolation allows the creation of

exclusive engines for isolated workloads. For example, you have a requirement to create a dashboard for CXOs that should respond interactively at any time they use it. In this scenario, you can create an exclusive engine (with an appropriate size based on your sets) and route these dashboard queries via a queue that is dedicated to this workload. This way, irrespective of the concurrency of the queries, the dashboard queries are routed to its exclusive engine.

When Dremio is first installed, no guardrails are put in place out of the box to restrict how much memory any queue or query in a queue can consume out of the total amount of memory available. This leaves Dremio open to potential out-of-memory issues if a user issues a large query that requires more memory than is available on the Dremio executors. In addition, the default queue concurrencies are a little high and could lead to memory exhaustion if many smaller queries (or up to 10 large ones) also consume more memory than is available on the Dremio Executors. By default, one engine is created.

Here is the default out-of-the-box WLM configuration. Details about it can be found here.

Queues

Average node memory: 9096.00 MB Add Queue

Name ↑	CPU Priority	Concurrency Limit	Queue Memory Limit per Node	Job Memory Limit per Node	Engine Name
High Cost Reflections	Background	1	-	-	-
High Cost User Queries	Medium	10	-	-	-
Low Cost Reflections	Background	10	-	-	-
Low Cost User Queries	Medium	100	-	-	-
UI Previews	Critical	100	-	-	-

One crucial value to make note of is the Average node memory, which in this example is 9096 (~96 GB). This value tells us the maximum amount of direct memory available on any Executor.

As part of the initial settings (when we don't have much information on the number of concurrent users, queries, reflections, data sets, and their sizes), we can set the initial concurrency settings as described below to avoid Out Of Memory (OOM) errors thus avoiding the executor's instability. We can start with the default engine and configure the limits.

Queue Memory Limit per Node defines the sum of the maximum memory of all running jobs from the queue that can be used in a node. For example, if you set this memory limit to 90 GB and three jobs (J1, J2, J3) are running from this queue, the maximum memory J1+J2+J3 can use on any node is 90 GB. If J1 may take 70 GB for a given time (t1), at that time (t1), the sum of J2 and J3 can’t exceed 20 GB, otherwise, it will throw an OOM error.

As a rule of thumb, the total queue memory limit per node summed across the Low and High cost user queries queues should be at most 120% of the Average node memory value. We allow this to exceed 100% because it is unlikely that both queues will experience maximum memory usage simultaneously; therefore, we allow some overlap.

The low- and high-cost reflections queue memory limit per node should be set to, at most, the same values as the queue memory limit for the low- and high-cost user queries. Reflections

typically run far less frequently than other query types, and often, they are triggered to run outside of normal working hours. Therefore, again, we allow the sum of these values to exceed the average node memory value.

However, after making changes to conform to the rule of thumb above, too many queries fail due to insufficient memory available to a particular queue. In that case, increasing the amount of memory allocated to the queue where queries are failing is safe. However, it should only go up to 95% of the average node memory on any queue.

Job Memory Limit per Node defines the maximum direct memory a job can use in a node while running a query. Let's say you have a 3-node engine, each with 90GB of direct memory, and you set the per-job limit of 50 GB. If two jobs run concurrently, each job can take up to 50 GB. If both request 50GB simultaneously, it will throw an OOM error. Note that it is unlikely that both of them request maximum memory at the same time.

In terms of job memory limits, for high-cost user queries, we want to allow Dremio to execute the biggest queries. Therefore, depending on the Average node memory setting, we will let the biggest job consume up to approximately 50-70% of the total memory available. Low-cost user queries typically consume far less memory, and at most, we would set a job memory limit of 50% of the queue memory limit or 5GB for one of these jobs, whichever is lower.

For UI Previews, Dremio recommends both the queue and job memory limit be set to the maximum memory allocated to a job in the high-cost user queries queue. It is doubtful that these memory limits will ever get reached, but this provides guardrails in case they do.

ding concurrency limits, Dremio recommends the following initial concurrency settings, regardless of what the memory settings are:

Queue Name	Max Concurrency Limit
High Cost Reflections	2
High Cost User Queries	3
Low Cost Reflections	5
Low Cost User Queries	20
UI Previews	100

Here are the recommended settings for a cluster with ~110 GB of direct memory.

Queue Name	Max Concurrency Limit	Queue Memory Limit Per Node (GB)	Job Memory Limit Per Node (GB)

High Cost Reflections	2	60	30
High Cost User Queries	3	80	40
Low Cost Reflections	5	25	5
Low Cost User Queries	20	45	5
UI Previews	100	40	40

Multiple Engine Configuration

Dremio supports creating multiple engines in all flavors of deployments - Kubernetes, AWSE, and Standalone deployments.

You can create and configure isolated engines for specific workloads. For example, you have many reflections that are being refreshed multiple times in a day. In that case, you can create and configure an isolated engine for creating and refreshing reflections. One caution here, though, is that if you may have multiple reflections and if they refresh once a day during non-business hours, creating an isolated engine may not be a good idea as it will not be used during business hours for user queries.

Similarly, you can create an isolated engine for Metadata refresh too. Ideally, if you have many Parquet file data sets in your data lake and if your data lake is ingested multiple times a day, you should create an engine specifically for Metadata. A separate engine for Metadata refreshes is not required if most of your data sets are Iceberg or external data sources.

Summary

In this document, we have provided the Best Practices for configuring Dremio after a vanilla installation. Following these Best Practices provides the best experience when operating Dremio in the future.