Performance Considerations

Introduction

Apache NiFi is an outstanding tool for moving and manipulating a multitude of data sources. It provides a robust interface for monitoring data as it moves through the configured NiFi system as well as the ability to view data provenance during each step.

While NiFi can be extremely performant in transitioning data from one source to another, it can be tricky to configure for optimal performance. This writeup will attempt to explain some of the basic and "advanced" configuration options and performance considerations in an effort to help developers configure the system for optimal performance.

Note: the vast majority of this information can be found by searching the web for various topics regarding NiFi. The information here attempts to provide a relatively consolidation of many of these sorts of posting found on the web.

Topics for Discussion

NiFi Configuration Files

NiFi contains basically two configuration files (`bootstrap.conf & nifi.properties`) and they live in the `conf` folder. These files allow you to provide basic settings for the NiFi application itself as well as the JVM environment NiFi runs in.

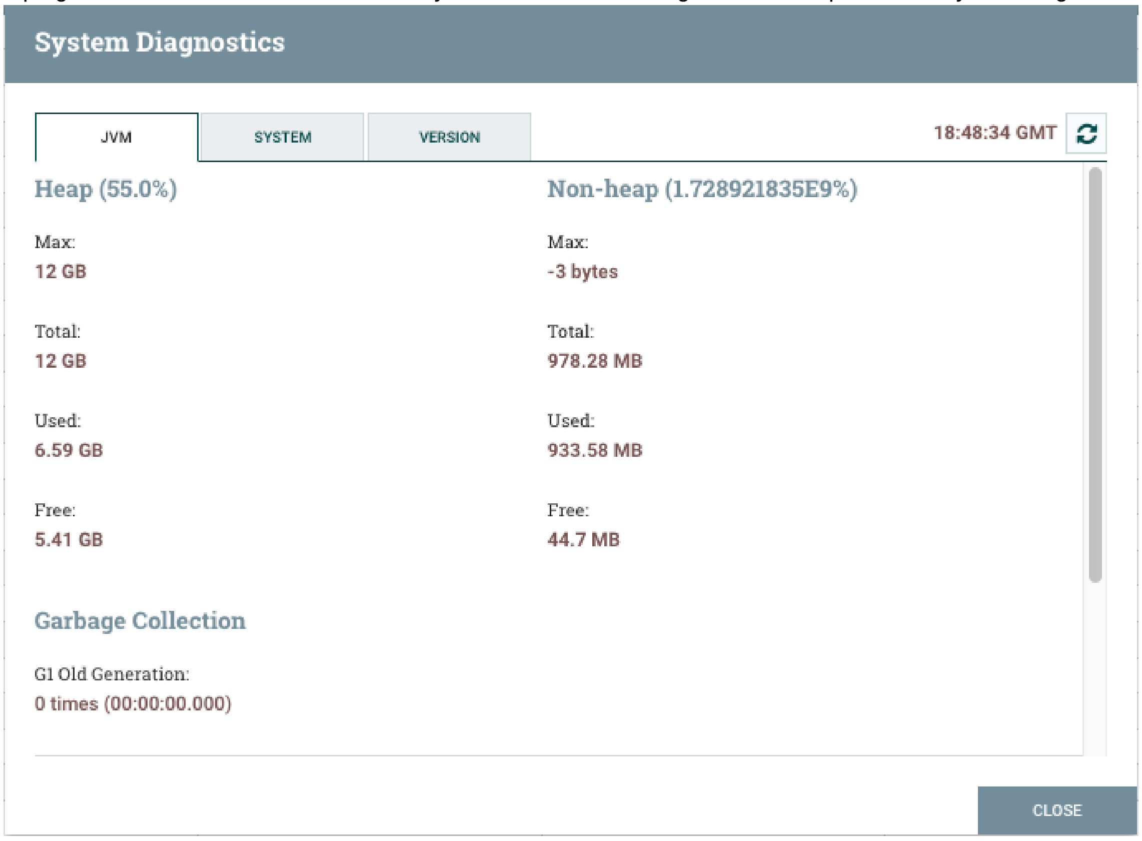
Bootstrap.conf

As mentioned earlier, the bootstrap.conf file contains various configuration settings allowing you to optimize the JVM environment for NiFi. Based upon the server capabilities (i.e. memory, processing speed, threads available, OS, etc.) you can tweak the JVM settings to get the most out of available system resources.

JVM Memory Settings

Argument	Description
`java.arg.2=-Xms512m`	Controls the minimum (or starting) amount of memory dedicated to the JVM heap space
`java.arg.3=-Xmx512m`	Controls the maximum amount of memory allowed to be consumed by the JVM

Keep in mind that the more flow files you anticipate having in flight at any one time will require increasing these settings. Additionally, if you're incorporating processors into your NiFi flow that are resource intensive, you may also want to consider increasing these values. To evaluate if you have enough memory allocated to the JVM you can open up the System Diagnostics pane and view how much memory is being used at any given time along with how much space your flow files are taking up. Click the "Hamburger" menu at the top right of the NiFi UI and select Summary. Then on the bottom right side of the pane, click system diagnostics.


System Diagnostics

JVM | SYSTEM | VERSION

18:48:34 GMT

Heap (55.0%)

Max:
12 GB

Total:
12 GB

Used:
6.59 GB

Free:
5.41 GB

Non-heap (1.728921835E9%)

Max:
-3 bytes

Total:
978.28 MB

Used:
933.58 MB

Free:
44.7 MB

Garbage Collection

G1 Old Generation:
0 times (00:00:00.000)

CLOSE

Garbage Collection

Argument	Description
`java.arg.13=-XX:+UseG1GC`	This setting tells the JVM which Garbage Collection to use during runtime. Recent versions of NiFi current use the configuration above, but older ones may not so it's best to validate this setting while also considering the version of Java you're running to ensure whatever setting you choose is compatible. Keep in mind that smaller Java Memory Heap settings may result in more frequent garbage collection being performed.

Running Java 8 or Later?

Consider adding the following lines to your bootstrap.conf file

Argument	Description
`java.arg.7=-XX:ReservedCodeCacheSize=256m`	Helps to prevent compiler from switching off when the code cache fills up
`java.arg.8=-XX:CodeCacheMinimumFreeSpace=10m`	Establishes a boundary for how much of the code cache can be used before flushing of the code cache will occur to prevent it from filling and resulting in the stoppage of the compiler
`java.arg.9=-XX:+UseCodeCacheFlushing`	

nifi.properties

Most of these settings described below are the default values for an OOTB NiFi instance. You should still validate these settings.

Argument	Description
`nifi.bored.yield.duration=10 millis`	Designed to help with CPU utilization by preventing processors using the timer driven scheduling strategy from using excessive CPU when there is no work to do. Smaller values equate to lower latency but higher CPU utilization. Increasing this value will cut down overall CPU utilization.
`nifi.ui.autorefresh.interval=30 sec`	Controls the interval between refreshing the latest statistics, bulletins, and flow revisions in the browser
`nifi.database.directory=./database_repository`	Recommend moving the location of this repo to outside the root directory of NiFi to simplify upgrading to future NiFi version. See Database Repository section for additional information.
`nifi.flowfile.repository.directory=./flowfile_repository`	Recommend moving the location of this repo to outside the root directory of NiFi to simplify upgrading to future NiFi version. See Flowfile Repository section for additional information
`nifi.content.repository.directory.default=./content_repository`	Recommend moving the location of this repo to outside the root directory of NiFi to simplify upgrading to future NiFi version. See Content Repository section for additional information.
`nifi.provenance.repository.directory.default=./provenance_repository`	Recommend moving the location of this repo to outside the root directory of NiFi to simplify upgrading to future NiFi version. See Provenance Repository section for additional information.

`nifi.queue.swap.threshold=20000`	This setting sets the default maximum queue size for all queues between NiFi components. Increasing this value will require additional memory be allocated to NiFi since these Flow Files are stored in memory. If a queue exceeds this size, the files are swapped from memory to disk and will later be retrieved when the queue size drops resulting in increased I/O.
`nifi.provenance.repository.query.threads=2`	Adjusts the number of system threads available for searching the Provenance Repository
`nifi.provenance.repository.index.threads=1`	Adjusts the number of system threads available for indexing the Provenance Repository. If a significant number of Flow Files are being operated upon this setting can become a bottleneck. If you see a warning bulletin stating “The rate of the dataflow is exceeding the provenance recording rate. Slowing down flow to accommodate.” you may need to increase the number of threads available. Remember though that when you increase the number of threads available for one process you reduce the number available for other processes.
`nifi.provenance.repository.index.shard.size=500 MB`	Sets the amount of Java heap space used for Provenance Repository indexing. Larger values will result in better performance and will utilize more Java Heap space. ***IMPORTANT****: configured shard size MUST be 50% smaller than the total configured Provenance Repository max storage size (nifi.provenance.repository.max.storage.size)* Only set this value above 500 MB if you have also increased the max storage size above 1 GB.

NiFi Repositories

Flow File Repository

This is the most important repository within NiFi. The Flow File repository contain the information for ALL flow files in flight. Each entry contains all the attributes associated with a given flow file as well as a pointer to where the Flow File’s actual content is stored within the Content Repository. It is a “Write-Ahead Log” of the metadata of each Flow File that currently exist within the system (all modifications are written to a log before they are applied). This repository provides the resiliency necessary to handle system restarts and failures.

It should be noted that Flow Files are never “modified”, but rather the original flow file is maintained during each update/modification to it and a copy is created with the updated version. This technique helps to support robust data provenance throughout the system.

Ideally, this repository should be located on a high-performance RAID that is not shared with other high I/O software and should never be located on the same RAID as the Content or Provenance repositories whenever possible. It should ideally use disk that does not compete for I/O with MarkLogic itself.

Note: if this repository become corrupted or runs out of disk space the state of flow files may become permanently lost!

Database Repository

This repository contains 2 H2 Databases. nifi-flow-audit.h2.db is used by NiFi to keep track of all configuration changes made within the NiFi UI & nifi-user-keys.h2.db which is only used when NiFi has been secured and it contains information about who has logged into NiFi.

Content Repository

The content repository is where the content, to include multiple historical versions of the content (if data archiving has been enabled), pointed to by each Flow File is stored. When a Flow File's content is determined to be no longer needed the content is either deleted or archived (deletion and archival are configured in the NiFi properties file). NiFi documentation strongly recommends this repository be moved to its own hard disk/RAID. Additional performance gains can be obtained by defining multiple Content Repositories per NiFi instance.

To do this, comment out the *nifi.content.repository.directory.default* setting and then add a new setting for each Content Repository you want to define (i.e. *nifi.content.repository.directory.contentS1R1 = …*, *nifi.content.repository.directory.contentS1R2 = …*, *nifi.content.repository.directory.contentS1R3 = …*).

In this example the *S#R#* represents the system number and repository number respectively. By splitting the content across multiple disks, you can significantly increase I/O performance while also improving durability in the event of failures.

Provenance Repository

This repository contains the historical records for each Flow File. This is basically the data lineage from the point NiFi created the Flow File to the point when NiFi finished processing the it, aka Chain of Custody, and contains a record of each Flow File modification. It uses a rolling group of 16 provenance log files (default) to track events and increase throughput. The provenance data is stored in a Lucene index broken into multiple shards to support indexing and search.

Data in this repository is updated frequently and results in a lot of disk I/O. As such, you can significantly increase performance and durability by spitting the repository into multiple repositories. See Content Repository for information on how to accomplish this. Additionally, updating the *nifi.provenance.repository.implementation* setting in the nifi.properties file to use the `org.apache.nifi.provenance.WriteAheadProvenanceRepository` instead of the default PersistentProvenanceRepository.

When switching to the WriteAheadProvenanceRepository, you may wish to consider additional changes to the nifi.properties as outlined in the Hortonworks DataFlow Documentation (https://docs.hortonworks.com/HDPDocuments/HDF3/HDF-3.1.1/bk_user-guide/content/system-properties.html).

NiFi Clustering

In large Production deployments, NiFi should be run in a clustered configuration. Doing so effectively multiplies the number of resources available for processing NiFi flows. Therefore, clustering allows NiFi to scale "out." However, clustering does not provide High Availability, so is purely a scaling approach. If tuning and scaling "up" with more CPU or I/O work, there is no need for the additional complexity of clustering.

Clustering will not have any impact if NiFi is running on a server that is not maxed out and the bottleneck for your process is MarkLogic instead. Clustering is a "scaling" approach to achieve additional throughput, not for achieving higher reliability.

In a clustered environment, each member of the cluster is responsible for processing its own Flow Files and has its own individual Repositories (discussed above) while sharing the same Flow File flow configuration.

Clustering in NiFi is accomplished by using ZooKeeper. NiFi comes with and embedded ZooKeeper already included in the NiFi distribution.

- ZooKeeper: Provides distributed configuration service, synchronization service, and naming registry for large distributed systems. Is also responsible for electing the Primary Node
- Zero-Master Configuration: Each node performs the same task but on a different set of data. The elected Master determines what nodes can connect to the cluster based upon the node's configured FlowFile. Each node within the cluster sends a Heartbeat status to the Master node
- Cluster Coordinator: Responsible for carrying out the tasks to manage which nodes are allowed to join the cluster and provides the most up to date data flow to new nodes joining the cluster

Clustering Considerations

- When a node within the cluster goes down or becomes disconnected from the cluster, the remaining nodes DO NOT pick up or process the Flow Files from the missing node since each node is responsible for handling its own set of the data. That said, when the failed/disconnected node comes back online and joins the cluster, it will resume processing its own Flow Files (assuming there are no catastrophic failure such as a corrupted Flow File Repository or running out of disk space).
- High Availability (from the point of Automatic Failover) is not currently natively supported. However, HA could be set up using an external system to monitor nodes within the cluster and then when one node goes down, a backup node could be brought online. As long as the new node coming online references the failed node's repositories and flow file configuration, it should be able to join the cluster. (note this is a presumption, but has not been validated) See Figure 3 for a conceptual diagram of attempting HA.
- See *Figure* 1 - *Data Distribution in NiFi Cluster* in the Screen Shots section for a representation of distributing data across a NiFi clustered environment. In the screen shot, you'll notice a DistributeLoad processor distributing unique flow files to each of the three HTTPInvoke processors. Each one is responsible for sending data from the Primary node to all other nodes within the cluster (including the Primary node) Note that the DistributeLoad and InvokeHTTP processors only run on the Primary node. The single ListenHTTP processor runs on all three nodes (including the Primary) and receives the data sent from the Primary.

Tips & Tricks

Assigning Threads to NiFi

When you initially set up NiFi, there is a default number of threads assigned to the host. Specifically, *Maximum Timer Driven Thread Count* and *Maximum Event Driven Thread Count*. These settings can be found by clicking the hamburger menu at the top right and selecting *Controller Settings*. On the *General* tab you'll see the two aforementioned settings. Typically, you can leave the *Max Event Driven Thread Count* setting at 5 since the vast majority of processors are Timer Driven. For the *Max Timer Driven Thread Count* you'll want to set this value between 2 – 4 times the number of cores your host has. So, if your host has 30 cores you can set the value to be between 60 & 120. These setting configure the maximum number of threads that can be utilized by NiFi across all processors. So, as an example if you set the number of concurrent tasks for a given processor (i.e. 5) when that processor executes it will consume 5 threads of the total allotted for NiFi. All processors will cumulatively consume threads when they execute. Keep in mind that regardless of how many concurrent tasks are configured for all processors, the total concurrent tasks cannot exceed the total number of threads assigned. By increasing the threads as described above, you can dramatically increase NiFi throughput.

Avoid Unnecessary File Modifications

It is important to understand when your metadata in a FlowFile is changed, vs. when the content referenced by that FlowFile is changed.

If you remember our earlier discussion regarding what Flow Files are comprised of and where their content is stored you'll see that the actual content is placed on disk in the ContentRepository, with only a pointer to that content residing in the Flow File. If you run a processor that modifies the Flow File's underlying content, that content must be fetched from disk, updated, copied and then written back to disk. The Flow File's pointer to the content will be also updated to reference the updated content, requiring a FlowFile disk write. Additionally, additional space is taken up to store the original version of Flow File content as well as the Provenance information.

Dataflow Optimization

Ref: NiFi/HDF Dataflow Optimization (Part 1) (https://community.hortonworks.com/articles/9782/nifihdf-dataflow-optimization-part-1-of-2.html) & (Part 2 (https://community.hortonworks.com/content/kbentry/9785/nifihdf-dataflow-optimization-part-2-of-2.html))

The links above provide excellent insight into some optimizations that can be done to improve NiFi's performance. No need to copy that information here. Briefly, look for resource exhaustion (I/O or CPU) if your flow is not running fast, or some processor is backing up. If not (there are ample disk and CPU resources) consider adding more threads. But if resources are exhausted, you must optimize the flows, reduce threads, or cluster. More and better details are in the linked web pages.

Handling Large Files

Large files can take on many forms such as binary video files or text-based files like CSV, JSON, or XML. In the latter case, these files are typically "record-based" such that the single file represents 1 – N individual records. In this case, you should avoid trying to load the entire file into memory and then splitting it into individual flow files. Instead, use a "record-based" processor (i.e. ConvertRecord) that allows the file to be processed using streaming. This will preclude the need to load the entire file into memory. Here is a blog post (https://bryanbende.com/development/2017/06/20/apache-nifi-records-and-schema-registries) explaining the various Record Streaming processors and how to interact with them.

Additional things to consider are:

- Ensure individual nodes have enough memory configured to handle the anticipated number, type, and size of files
- Ensure the flow file and content repositories have enough disk space to accommodate the anticipated quantity and size of files to be processed

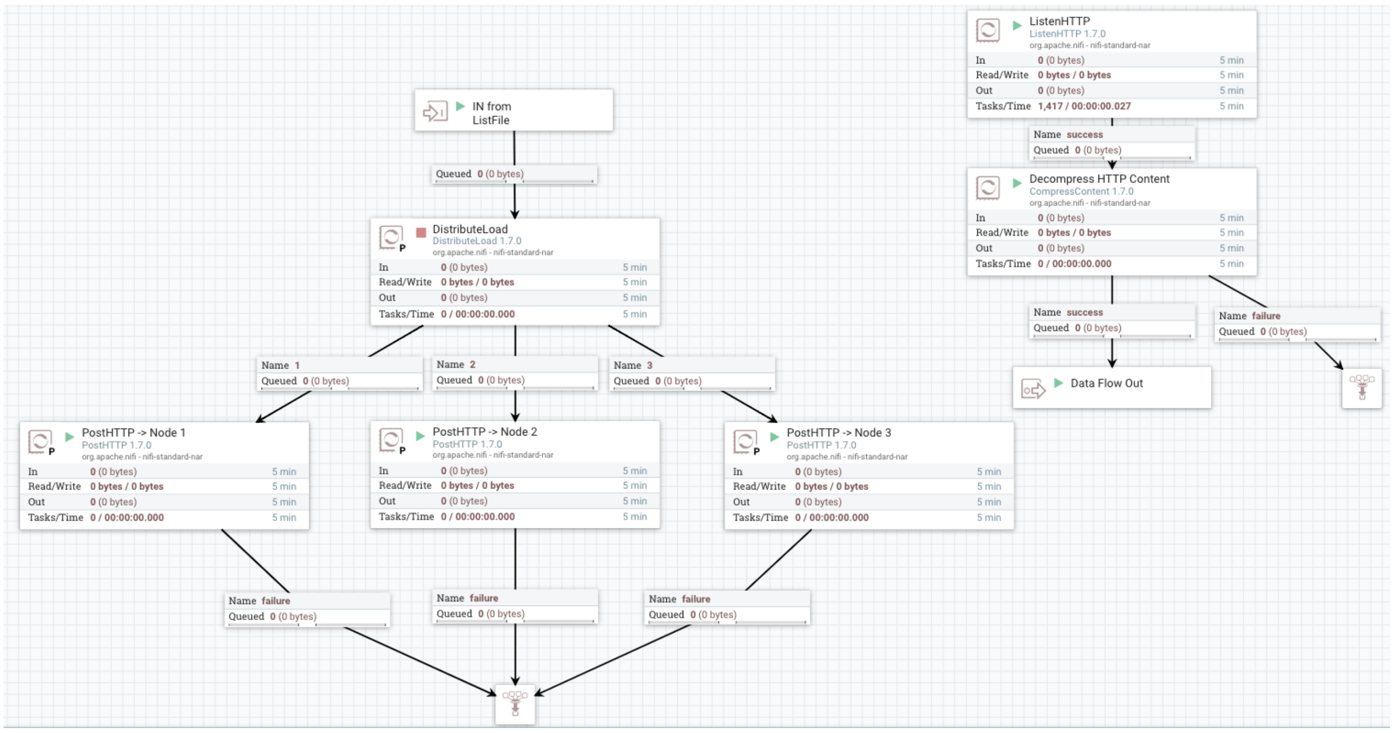
Handling Disconnected Nodes

If a node fails/becomes disconnected from the cluster and is unable to rejoin, look in the *nifi-1.x.x/logs* directory. The nifi-app.log file will log most all NiFi activity including errors. Here are some thing to look for:

- You may see something indicating the node’s Flow File configuration is out of date. This issue was largely seen with older versions of NiFI (previous to 1.0.0). In this case simply delete or rename the *flow.xml.gz* file in the *conf* directory and restart the node. When the node attempts to connect, the Primary Node (i.e. Cluster Coordinator) will see the rejoining node has no Flow File template and will then provide the most current one to that node.
- Ensure regular heartbeats are being sent to the Cluster Coordinator. If the Cluster Coordinator doesn’t receive regular heartbeats (interval configured in the conf/nifi.properties file) the node sending the heartbeat notifications will become disconnected. The Cluster Coordinator will wait 8x the configured *nifi.cluster.protocol.heartbeat.interval* before disconnecting a node. This can happen for numerous reasons such as network latency between the node and the Cluster Coordinator or excessive garbage collection. In the latter case, ensure you have configured enough Java Heap Memory for the node.

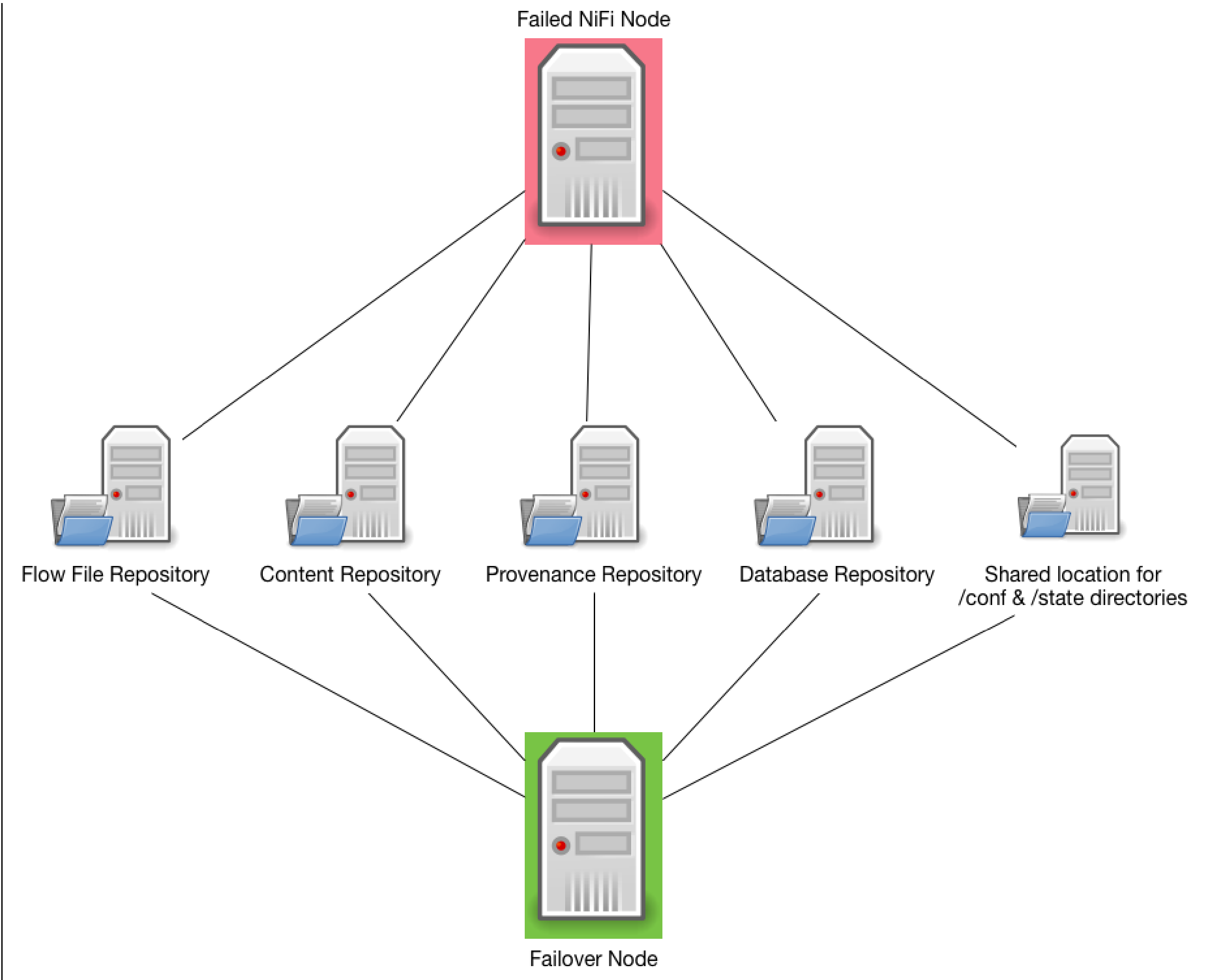
Figures

Figure 2 - Data Distribution in NiFi Cluster



Note: The Decompress HTTP Content processor is only needed if you have configured the PostHTTP processors on the Primary node to perform data compression (Compression Level property)

Figure 3 - Conceptual HA Diagram



In the diagram above, conceptually a standby node could be configured to reference the same Repositories as a currently running node as well as referencing the same configurations. When a node fails, the standby node could be brought online referencing all the same configuration and repositories. This, theoretically, could allow for some sort of HA. Note: It is extremely important that only a single node is allowed to run using this configuration. Multiple instances accessing the same repositories and configuration could result in corrupting the repositories.

NiFi/HDF Dataflow Optimization (Part 1 of 2)

Labels (2)

Apache NiFi Cloudera DataFlow (CDF)

Wynner Guru
Created on 01-13-2016 06:56 PM - edited 08-17-2019 01:31 PM

NiFi/HDF Dataflow Optimization

Who is This Guide For?

This guide is for Dataflow Managers (DFM) very comfortable with NiFi and a good knowledge of the underlying system on which NiFi is running. This guide also assumes that the user has applied the recommend changes in the Configuration Best Practices section of the Admin Guide https://nifi.apache.org/docs.html and applied appropriate changes recommended in the setup best practices here: https://community.hortonworks.com/content/kbentry/7882/hdfnifi-best-practices-for-setting-up-a-high-.... This guide is not intended to be an exhaustive manual for Dataflow Optimization, but more of points to consider when optimizing their own dataflow.

What is meant by dataflow optimization?

Dataflow Optimization isn't an exact science with hard and fast rules that will always apply. But more of a balance between system resources (memory, network, disk space, disk speed and CPU), the number of files and size of those files, the types of processors used, the size of dataflow that has been designed, and the underlying configuration of NiFi on the system.

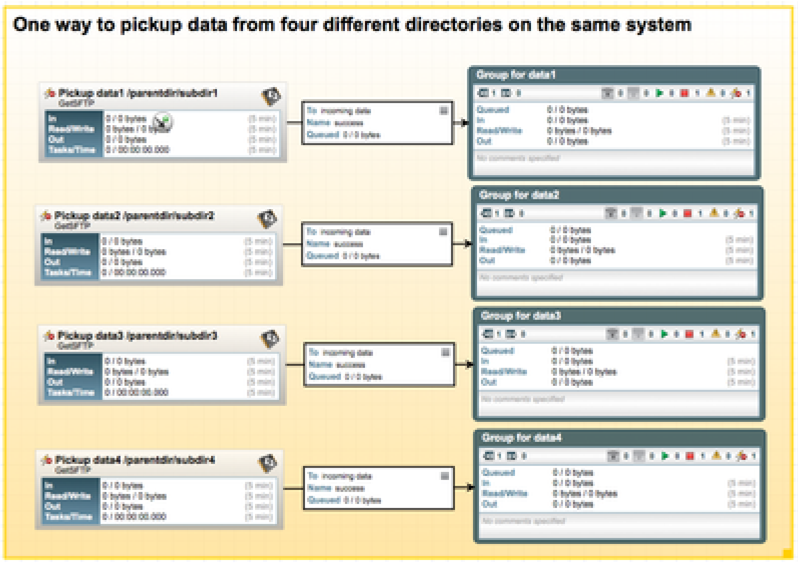
Group common functionality when and where it makes sense

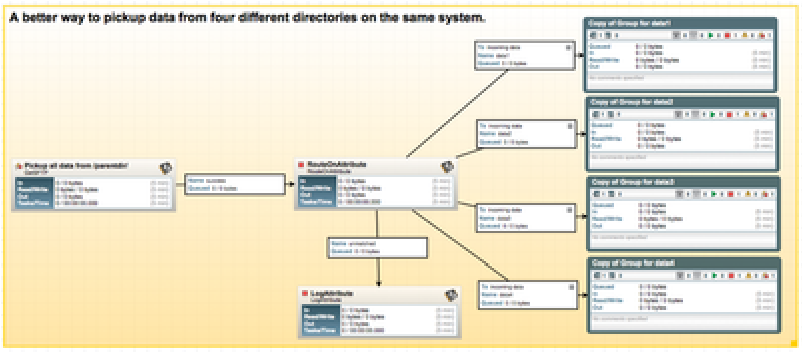
A simple approach to dataflow optimization is to group repeated operations into a Process Group. This will optimize the flow by removing redundant operations. Then pass the data through the group and then continue through the flow. When repeating the same process in multiple places on the graph, try to put the functionality into a single group.

Use the fewest number of processors

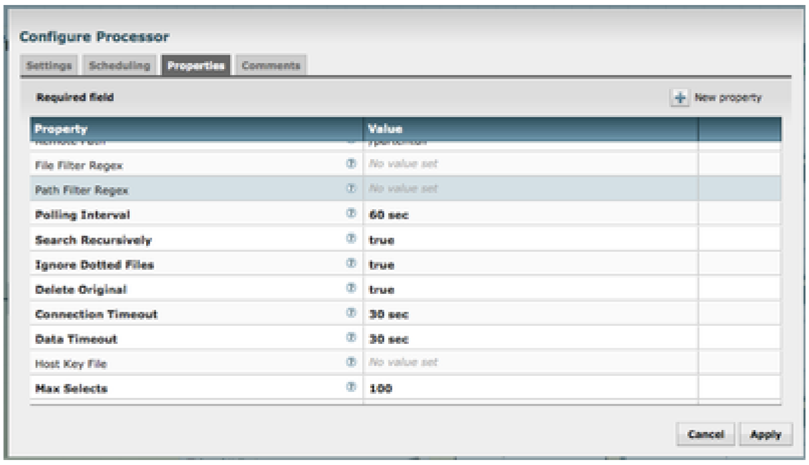
The simplest approach to dataflow optimization is to use the fewest number of processors possible to accomplish the task. How does this translate in to better NiFi performance? Many NiFi processors support batch processing per thread. With multiple processors working on smaller subsets of a large dataset you are not taking advantage of the batch processing capability. For example 4 GetSFTP processors each pulling from separate directories, with the same parent directory, with 20 files will use 4 threads. While 1 GetSFTP processor will get a listing, from the parent directory, for all files in those sub directories and then pull them all under 1 thread.

Here is a flow snippet illustrating this example:


One way to pickup data from four different directories on the same system
Pickup data1 /parentdir/subdir1
GetSFTP
Pickup data2 /parentdir/subdir2
GetSFTP
Pickup data3 /parentdir/subdir3
GetSFTP
Pickup data4 /parentdir/subdir4
GetSFTP
To incoming data
Name success
Group for data1
Group for data2
Group for data3
Group for data4
Queued
In
Read/Write
Out
Tasks/Time
No comments specified



In this case with a single GetSFTP processor, make sure the property **Search Recursively** is set to true:



Configure Processor

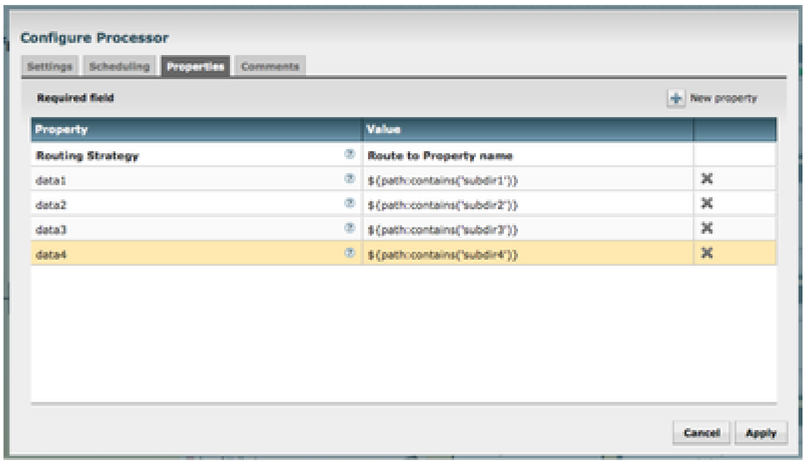
Settings | Scheduling | Properties | Comments

Required field — New property

Property	Value
File Filter Regex	No value set
Path Filter Regex	No value set
Polling Interval	60 sec
Search Recursively	true
Ignore Dotted Files	true
Delete Original	true
Connection Timeout	30 sec
Data Timeout	30 sec
Host Key File	No value set
Max Selects	100

Cancel | Apply

Configuration of RouteOnAttribute processor being used to separate data pulled by one GetSFTP processor from four different directories using the standard flow file attribute ${path}:



Configure Processor

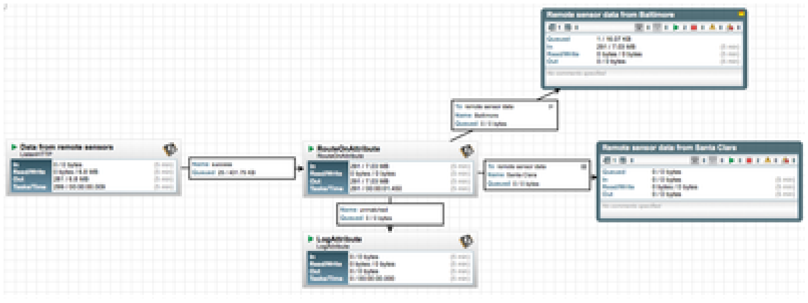
Settings | Scheduling | Properties | Comments

Required field — New property

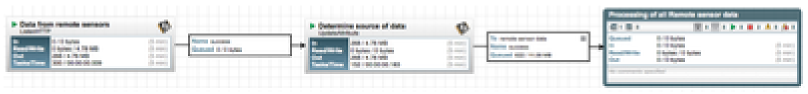
Property	Value
Routing Strategy	Route to Property name
data1	${path:contains('subdir1')}
data2	${path:contains('subdir2')}
data3	${path:contains('subdir3')}
data4	${path:contains('subdir4')}

Cancel | Apply

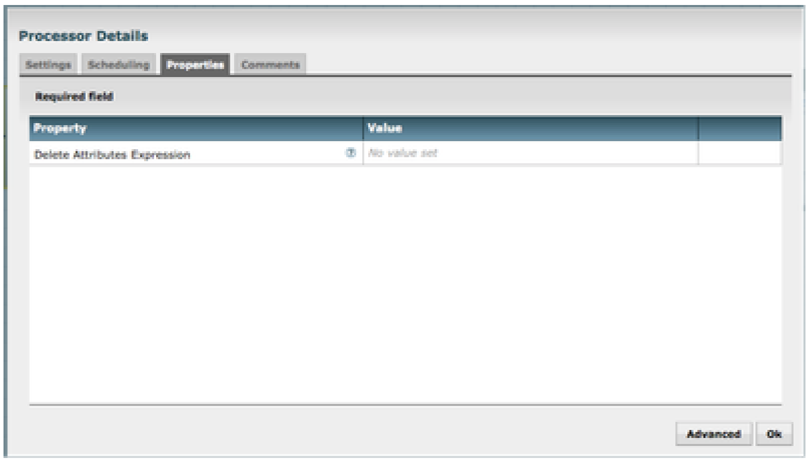
If data were being pushed to NiFi, a similar way to identify the data would be utilizing a user-supplied attribute. This method would work for data coming in from any type of processor that receives data in the flow. In the example below, the data is coming in via a ListenHTTP processor from another NiFi instance with a user added *${remote-sensor}* attribute. The receiving NiFi uses that attribute to make a routing decision.


Data from remote sensors
RouteOnAttribute
Remote sensor data from Baltimore
Remote sensor data from Santa Clara
LogAttribute
unmatched

In addition, you can tag data that is already combined using an UpdateAttribute processor so it can be managed through a shared downstream data path that will make use of batch processing. Using a very similar example, say the data is coming in from multiple sensor sources without the user-supplied tag already added. How would NiFi identify the data? In the flow figure below NiFi can use the attribute *restlistener.remote.user.dn* to identify the source of the data and add the appropriate tag:

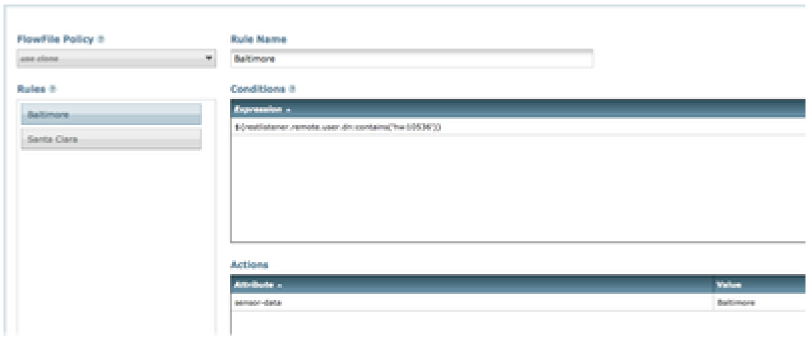


Configuration of an UpdateAttribute processor to use inherent attribute, *restlistener.remote.user.dn,* in the ListenHTTP processor, the first figure is the properties tab of the processor, the second figure shows the rules inside the advanced tab:



Required field

Property	Value
Delete Attributes Expression	No value set



FlowFile Policy: use clone

Rules: Baltimore, Santa Clara

Rule Name: Baltimore

Conditions

Expression
${restlistener.remote.user.dn:contains('hw10536')}

Actions

Attribute	Value
sensor-data	Baltimore

Part 2 of this tutorial can be found here: NiFi/HDF Dataflow Optimization part 2

13,693 Views 10 Kudos

NiFi/HDF Dataflow Optimization (Part 2 of 2)

Labels (2)

Apache NiFi Cloudera DataFlow (CDF)

Wynner Guru
Created on 01-13-2016 07:15 PM - edited 08-17-2019 01:28 PM

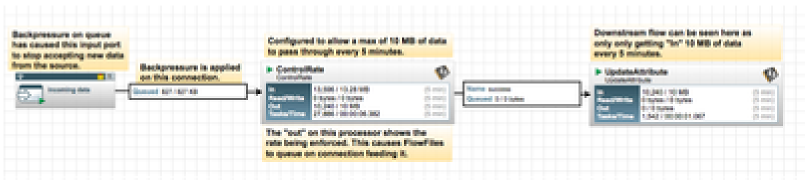
This is part 2 of the Dataflow Optimization tutorial. Part 1 can be found here: NiFi/HDF Dataflow Optimization part 1

How do I prevent my system/processors from becoming overwhelmed?

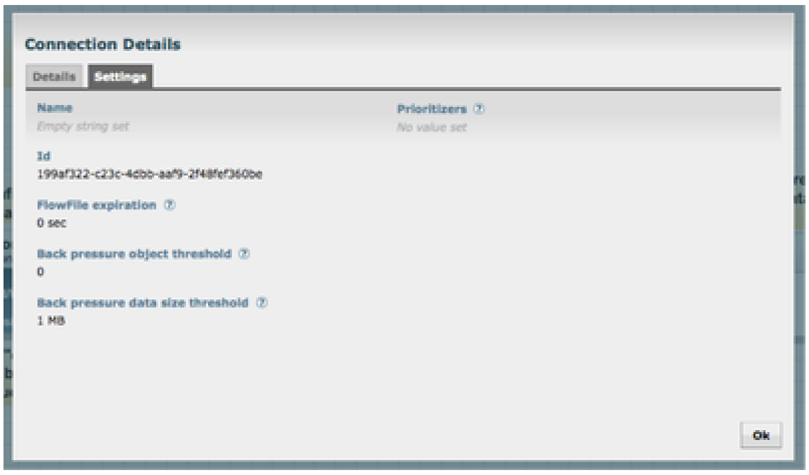
Another aspect of optimization is preventing the dataflows from overwhelming the underlying system and affecting NiFi software stability and/or performance. If allowed to run unconstrained, it is more likely to see a severe performance hit. You are more likely to see severe performance hit if dataflows are unbounded and the content repo fills 100%

For example, if you have 50MB for the content_repository partition and data normally is delivered via an input port in groups of 10MB. What would happen if the data source was down for a few hours and then came back online with a backlog of 75MB? Once the content_repository filled, the input port would start generating errors when trying to write the files it was receiving. The performance of NiFi would drop because the disk/disks where the content_repository resided would be trying write new files while at the same time NiFi would be trying to access the disk/s to deal with the current flowfiles. Backpressure can be configured based on number of objects and/or the size of the flowfiles in the connection.

Using backpressure would be one way to prevent this scenario, below is an example:


Backpressure on queue has caused this input port to stop accepting new data from the source.
Backpressure is applied on this connection.
Configured to allow a max of 10 MB of data to pass through every 5 minutes.
Downstream flow can be seen here as only only getting "in" 10 MB of data every 5 minutes.
The "out" on this processor shows the rate being enforced. This causes FlowFiles to queue on connection feeding it.
ControlRate
UpdateAttribute

Backpressure is set in the connection from the input port to the ControlRate processor:



Connection Details

Name
Empty string set

Prioritizers
No value set

Id
199af322-c23c-4dbb-aaf9-2f48fef360be

FlowFile expiration
0 sec

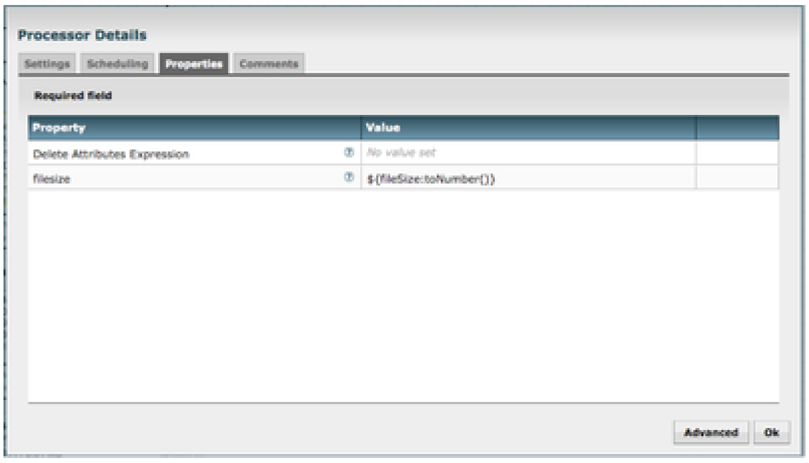
Back pressure object threshold
0

Back pressure data size threshold
1 MB

It can be seen that in the above figure that if the backlog of data reaches 1 MB, then the input port would not be able to receive data until the backlog dropped below the threshold and then incoming data from the source system would resume. This configuration allows NiFi to receive the backlog of data at a rate that won't over utilize the system resources. Additionally, adding the ControlRate processor to the flow will ensure that the backlog of data will not overwhelm any processors further down the flow path. This method of combining backpressure with the ControlRate processor is easier than trying to set backpressure in every connection through the complete flow path.

Using ControlRate processor to control the rate of data through a flow

The ControlRate processor is good example of using one processor to prevent another from becoming overwhelmed and/or overwhelming the overall flow. For example, the DFM is only able to allocate one concurrent task to the CompressContent processor, if the queue to the processor becomes too large, that one concurrent task would only be able to scan the queue to determine which flowfile should be compressed next. Meaning, it would spend all of its time looking at its queue and never actually compressing any flow files. Using the above example of the CompressContent processor, then we could use the ControlRate processor to prevent the processor from becoming overwhelmed, by using data rate, flowfile count or even a flowfile attribute to control the rate of data. The below example is using the added *filesize* attribute to control the rate of data, the first figure shows the configuration of the UpdateAttribute processor adding the filesize attribute:



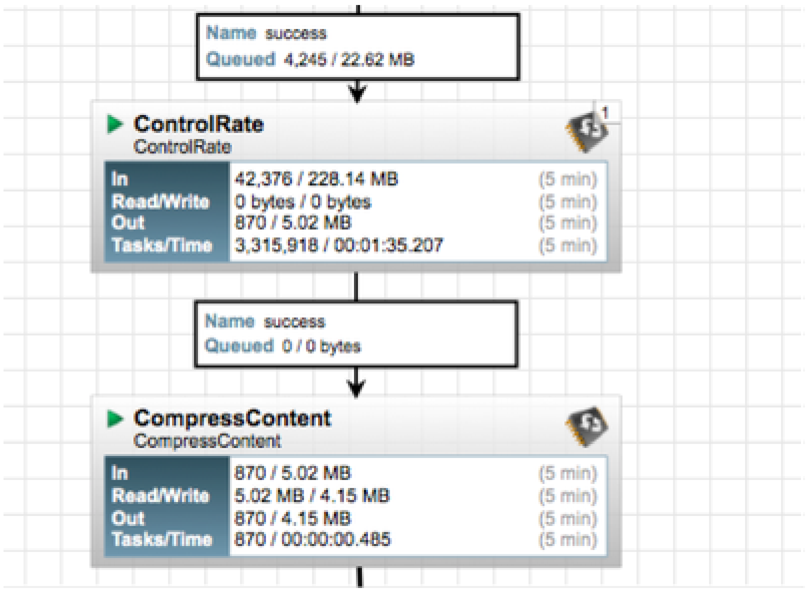
Processor Details

Settings | Scheduling | Properties | Comments

Required field

Property	Value
Delete Attributes Expression	No value set
filesize	${fileSize:toNumber()}

Advanced | Ok



Name success
Queued 4,245 / 22.62 MB

ControlRate
ControlRate

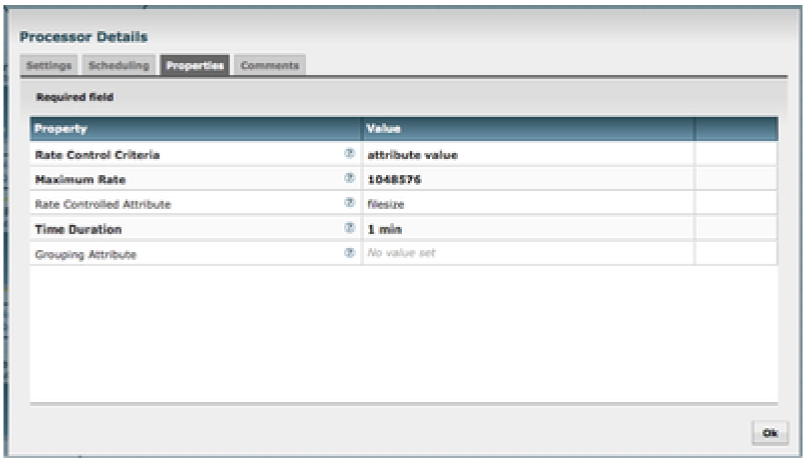
```
In           42,376 / 228.14 MB           (5 min)
Read/Write   0 bytes / 0 bytes            (5 min)
Out          870 / 5.02 MB                (5 min)
Tasks/Time   3,315,918 / 00:01:35.207     (5 min)
```

Name success
Queued 0 / 0 bytes

CompressContent
CompressContent

```
In           870 / 5.02 MB                (5 min)
Read/Write   5.02 MB / 4.15 MB            (5 min)
Out          870 / 4.15 MB                (5 min)
Tasks/Time   870 / 00:00:00.485           (5 min)
```

The configuration of the ControlRate processor is shown below. It is set to allow ~1 MB/minute, based on the cumulative size of files through the processor.



Processor Details

Settings | Scheduling | Properties | Comments

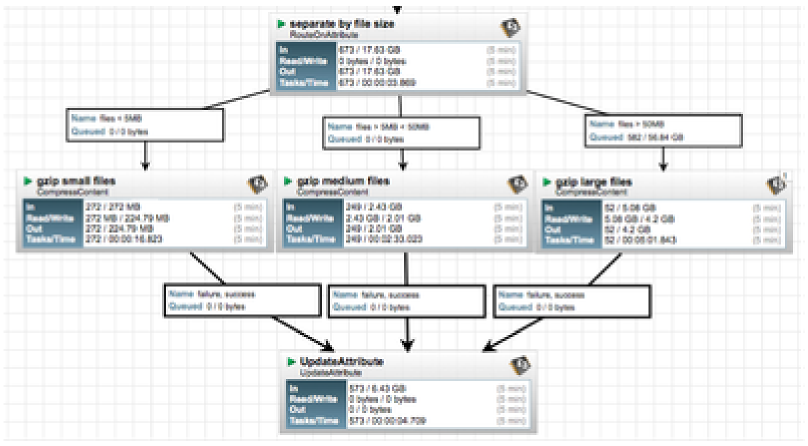
Required field

Property	Value
Rate Control Criteria	**attribute value**
Maximum Rate	**1048576**
Rate Controlled Attribute	filesize
Time Duration	**1 min**
Grouping Attribute	No value set

Ok

Understanding what resources a processor uses

Another aspect of dataflow optimization is to understand the resources needed by each processor. This will allow better dataflow performance. For instance, the CompressContent processor will use 1 CPU/concurrent task, so if this processor has 4 concurrent tasks and there are four files in the queue, then 4 CPUs will be utilized by this processor until the files have been compressed. For small files this becomes less of a resource bottleneck than dealing with large files. A good example of this would be to have small, medium and large files all go down a separate flow paths into 3 different CompressContent processors, each with their own number of concurrent tasks. In the example below, all three CompressContent processors have one concurrent task.



```
separate by file size
RouteOnAttribute
In          673 / 17.63 GB          (5 min)
Read/Write  0 bytes / 0 bytes       (5 min)
Out         673 / 17.63 GB          (5 min)
Tasks/Time  673 / 00:00:03.869      (5 min)

Name    files < 5MB
Queued  0 / 0 bytes

Name    files > 5MB < 50MB
Queued  0 / 0 bytes

Name    files > 50MB
Queued  582 / 56.84 GB

gzip small files
CompressContent
In          272 / 272 MB            (5 min)
Read/Write  272 MB / 224.79 MB      (5 min)
Out         272 / 224.79 MB         (5 min)
Tasks/Time  272 / 00:00:16.823      (5 min)

gzip medium files
CompressContent
In          249 / 2.43 GB           (5 min)
Read/Write  2.43 GB / 2.01 GB       (5 min)
Out         249 / 2.01 GB           (5 min)
Tasks/Time  249 / 00:02:33.023      (5 min)

gzip large files
CompressContent
In          52 / 5.08 GB            (5 min)
Read/Write  5.08 GB / 4.2 GB        (5 min)
Out         52 / 4.2 GB             (5 min)
Tasks/Time  52 / 00:05:01.843       (5 min)

Name    failure, success
Queued  0 / 0 bytes

Name    failure, success
Queued  0 / 0 bytes

Name    failure, success
Queued  0 / 0 bytes

UpdateAttribute
UpdateAttribute
In          573 / 6.43 GB           (5 min)
Read/Write  0 bytes / 0 bytes       (5 min)
Out         0 / 0 bytes             (5 min)
Tasks/Time  573 / 00:00:04.709      (5 min)
```

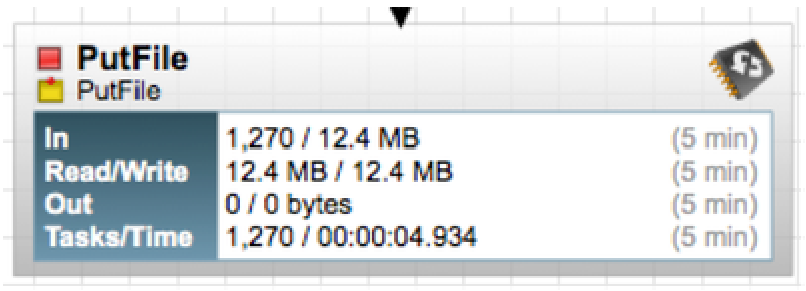
This diagram provides a good example of optimizing throughput. The "Tasks/Time" for each of the CompressContent processors clearly show that the amount of time required to compress larger files is exponentially bigger as the files increase in size. This provides better use of CPU to process the maximum number of files and not letting the arbitrary large files slow down the entire flow. This isn't specifically noted in the documentation, but is known by the fact that the only way to compress a file either inside or out of NiFi is to use a CPU per file being compressed/uncompressed on the system where the operation is being done.

Be aware that some of the documentation for the processors points out when a particular processor has a behavior that should be made aware of. For example the documentation for the MergeContent processor has the following line: *It is recommended that the Processor be configured with only a single incoming connection, as Group of FlowFiles will not be created from FlowFiles in different connections.*

The size of FlowFiles and the number of FlowFiles traversing various dataflow paths will have different impacts on some processors. Learning to read the information on a processor can help you determine when and where to apply some or all of the discussed optimization strategies.

How does NiFi help you find where/how you can optimize?

There is a great deal of information provided by each processor, that can assist the DFM in determining where the trouble spots could be in the flow; see the description below:



PutFile
PutFile

In	1,270 / 12.4 MB	(5 min)
Read/Write	12.4 MB / 12.4 MB	(5 min)
Out	0 / 0 bytes	(5 min)
Tasks/Time	1,270 / 00:00:04.934	(5 min)

In: The amount of data that the Processor has pulled from the queues of its incoming Connections based on a sliding 5-minute window. This value is represented as <count> / <size> where <count> is the number of FlowFiles that have been pulled from the queues and <size> is the total size of those FlowFiles' content. In this example, the Processor has pulled 1270 FlowFiles from the input queues, for a total of 12.4 megabytes (MB).

Read/Write: The total size of the FlowFile content that the Processor has read from disk and written to disk. This provides valuable information about the I/O performance that this Processor requires. Some Processors may only read the data without writing anything while some will not read the data but will only write data. Others will neither read nor write data, and some Processors will both read and write data. In this example, we see that in the past five minutes, this Processor has read 12.4 MB of the FlowFile content and has written 12.4 MB as well. This is what we would expect since this Processor simply copies the contents of a FlowFile to disk.

Out: The amount of data that the Processor has transferred to its outbound Connections. This does not include FlowFiles that the Processor removes itself, or FlowFiles that are routed to connections that are auto-terminated. Like the "In" metric above, this value is represented as <count> / <size> where <count> is the number of FlowFiles that have been transferred to outbound Connections and <size> is the total size of those FlowFiles' content. In this example, all of the files are written to disk and the connections are auto terminated, so no files have been moved out to downstream connections.

Tasks/Time: Reflects the number of tasks that completed their run in the last 5 minutes and the reported total amount of time those tasks took to complete. You may have 1 thread and a single task takes 20 minutes to complete. When it completes it is added to this cumulative report for the next 5 minutes. The format of the time is <hour>:<minute>:<second>. Note that the amount of time taken can exceed five minutes, because many tasks can be executed in parallel. For instance, if the Processor is scheduled to run with 60 Concurrent tasks, and each of those tasks takes one second to complete, it is possible that all 60 tasks will be completed in a single second. However, in this case we will see the Time metric showing that it took 60 seconds, instead of 1 second.

Utilize the information provided by the processors, number of reads/write and tasks/time per task to find "hot spots" on the graph. For instance, if a large number of tasks ran but the amount of data that passed through the processor is low, then the processor might be configured to run too often or with too many concurrent tasks. At the very least a closer look is warranted for the processor. A processor with few completed tasks along with a high task time indicates that this processor is CPU intensive. If the dataflow volume is high and a processor show a high number of completed threads and high task time, performance can be improved by increasing the run duration in the processor scheduling.

Are there places in the graph were data is backlogged or which processors are not processing the rate of files being passed to them?

If there is a connection in the flow where data is always backlogged, it can be a point of concern if any delay in processing the data is unacceptable. But, simply adding more concurrent tasks to the processor with the backlogged can lead to thread starvation in another part of the graph (covered in more detail below). Here again, the DFM must take care in understanding why the data is backing up at this particular point in the graph. It simply could be a processor that is very CPU intensive and there are only so many CPUs to be utilized throughout the graph. The files might be very large for the system and it might require a read and a write for each file, which are expensive operations. If resources aren't an issue, then add more concurrent tasks and see if the backlog is resolved. If resources are an issue, then either the flow will have to be redesigned to better utilize what is available or else the work load will have to be distributed across multiple systems, meaning clustering two or more systems to reduce the load on any one system. Files could be backlogging because the processor that is working on them is I/O intensive. Processor stats should show this. Check your disks; is I/O at or near 100%? Adding more threads will not help process files faster but instead will lead to thread starvation.

NiFi Thread starvation

Another aspect to optimization is how processors can be configured to take too many of the available resources in one area of the flow, and then thread starving another area of the flow.

Say, there is a processor that is CPU and disk intensive that requires 10 concurrent tasks to maintain the daily operational flow. Then the graph is modified to add additional dataflows, which require system resources. After the additional dataflow has been added to the graph, it is noticed that the processor with 10 concurrent tasks is unable to keep up with the data rate. So, an additional 5 concurrent tasks are added to the processor. Then another flow on the graph starts backing up data and then additional concurrent tasks are added to that flow and so and so on... Soon, too many concurrent tasks have been added to the graph that the dataflow never actually gets a thread, the system spends all of the resources deciding which processor should get the threads, but is never allowed to complete the task, and the system is stuck in this cycle.

To prevent this from happening to a dataflow, after each update or addition to the graph, the DFM should examine the current level of utilization of system resources. If after an update or addition the data begins to backlog in a point on the flow not previously seen, then the change has overwhelmed the system in some way and it is advised the added flow be turned off until it can determined what system resources are being over utilized. Another solution would be load balancing the flow over two or more systems by cluster the systems.

When to Cluster

Physical resource exhaustion can and does occur even with an optimized dataflow. When this occurs the best approach is to spread the data load out across multiple NiFi instances in a NiFi cluster.

NiFi Administrators or Dataflow Managers (DFMs) may find that using one instance of NiFi on a single server is not enough to process the amount of data they have. So, one solution is to run the same dataflow on multiple separate NiFi servers. However, this creates a management problem, because each time DFMs want to change or update the dataflow, they must make those changes on each server and then monitor each server individually. By clustering the NiFi servers, it’s possible to have that increased processing capability along with a single interface through which to make dataflow changes and monitor the dataflow. Clustering allows the DFM to make each change only once, and that change is then replicated to all the nodes of the cluster. Through the single interface, the DFM may also monitor the health and status of all the nodes.

The design of NiFi clustering is a simple master/slave model where there is a master and one or more slaves. In NiFi clustering, we call the master the NiFi Cluster Manager (NCM), and the slaves are called Nodes. While the model is that of master and slave, if the NCM dies, the nodes are all instructed to continue operating, as they were to ensure the dataflow remains live. The absence of the NCM simply means new nodes cannot join the cluster and cluster flow changes cannot occur until the NCM is restored.

16,490 Views **11 Kudos**