**HDF best practices guide to configuring your system and NiFi for high performance dataflows**

<https://community.hortonworks.com/articles/7882/index.html?mkt_tok=eyJpIjoiT1dSaU1HUXdaVGhpT1RGaCIsInQiOiIrVkVDYlh3bGREM1REb21zNURjM1ZUdlBSTFwvdW85T0NQTlM1SlNyMUJGZllTajhpOUlYQVlXV243QXlxQkpSTWhGaURhTXhmSUQ5eW5qczFram5wTkp5TXMzZHVYUm1ETUtRVE1yRFwvSnJKZ2NVcVJRQllzd295SmVlUmJvSEtpIn0%3D>

NiFi is pre-configured to run with very minimal configuration needed out of the box. Simply edit the **nifi.properties** file located in the conf directory by adding a sensitive props key (used to encode all sensitive properties in flow) and you are ready to go. This may get you up and running, but that basic configuration is far from ideal for high volume/high performance dataflows. While the NiFi core itself does not have much **impact** on the system in terms of memory, **disk/network I/O**, or **CPU**, the dataflows that you build have the potential of impacting those performance metrics quite a bit. Some NiFi processors can be **CPU**, **I/O** or memory intensive. In fact some can be intensive in all three areas. We have tried to identify if a processor is intensive in any of these areas within the processor documentation (Processor documentation can be found by clicking on Help in the upper right corner of the NiFi UI or right clicking an instance and selecting Usage) This guide is not intended to cover how to optimize your data flow design or how to fine-tune your individual processors.

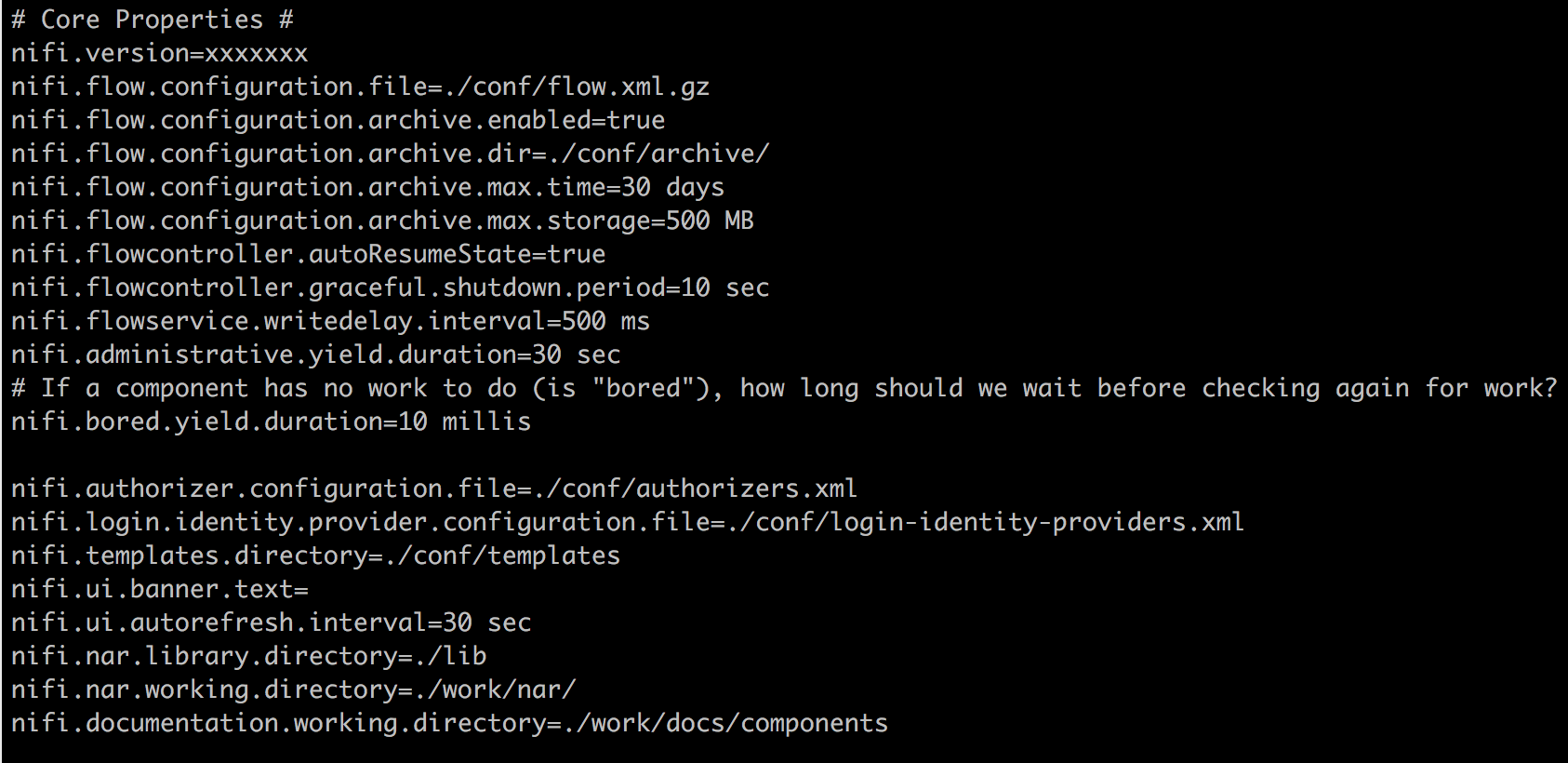
In this guide, we will focus on the initial setup of the application itself and focus on the areas where you can improve performance by changing some out of the box default values. We will take a closer look at many properties in both the **nifi.properties** and the **bootstrap.conf** files. We will point out the properties that should be changed from their default values and what the changes will gain you. For those properties that we do not cover in this guide, please refer to **Admin Guide** (This can be found by clicking on Help in the upper right corner of your **NiFi UI** or on the Apache website at <https://nifi.apache.org/docs.html>) for more information.

Before we dive into these two configuration files lets take a 5,000-foot view of the **NiFi** application. **NiFi** is a Java application and therefore runs inside a JVM, but that is not the entire story. The application has numerous repositories that each have their own very specific function. The database repository keeps track of all changes made to the dataflow using the UI and all users who have authenticated in to the UI (Only when NiFi is setup securely using https). The **FlowFile** repository holds all the FlowFile attributes about every **FlowFile** that is being processed within the data flows built using NiFi UI. This is one of the most important repositories. If it becomes corrupt or runs out of disk space, state of the **FlowFiles** can become lost (This guide will address ways to help avoid this from occurring). This repository holds information like where a **FlowFile** is currently in the dataflow, a pointer to location of the content in the content repository, and any **FlowFile** attributes associated or derived from the content. It is this repository that allows **NiFi** to pickup where it left off in the event of a restart of the application (Think unexpected power outage, inadvertent user/software restart, or upgrade). The content repository is where all the actual content of the file being processed by **NiFi** resides. It is also where multiple historical versions of the content are held if data archiving has been enabled. This repository is can be very I/O intensive depending one the type of data flows the user constructs. Finally, we have the provenance repository, which keeps track of the lineage on both current and past **FlowFiles**. Through the use of the provenance UI data can be downloaded, replayed, tracked and evaluated at numerous points along the dataflow path. Download and replay are only possible if a copy of the content still exists in the content repository. Like any application, the overall performance is governed by the performance of its individual components. So we will focus on getting the most out of these components in this guide.

**nifi.properties file:**

We will start by looking at the **nifi.properties** file located in the conf directory of the **NiFi** installation. The file is broken up in to sections (Core Properties, State Management, H2 Settings, **FlowFile Repository**, **Content Repository**, **Provenance Repository**, **Component Status Repository**, Site to Site properties, Web Properties, Security properties, and **Cluster properties**). The various properties that make up each of these sections come pre-configured with default values.

**# Core Properties #:**



There is only one property in this section that has an impact on **NiFi**, performance.

**nifi.bored.yield.duration=10 millis**

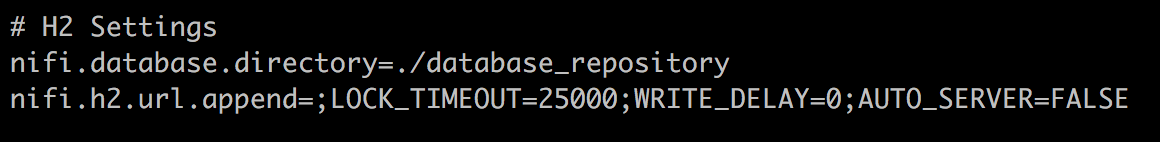
This property is designed to help with **CPU** utilization by preventing processors, that are using the timer driven scheduling strategy, from using excessive CPU when there is no work to do. The default 10-millisecond value already makes a huge impact on cutting down on **CPU** utilization. Smaller values equate to lower latency, but higher CPU utilization. So depending on how important latency is to your overall dataflow, increasing the value here will cut down on overall **CPU** utilization even further.

There is another property in this section that can be changed. It does not have an impact on **NiFi** performance but can have an impact browser performance.

**nifi.ui.autorefresh.interval=30 sec**

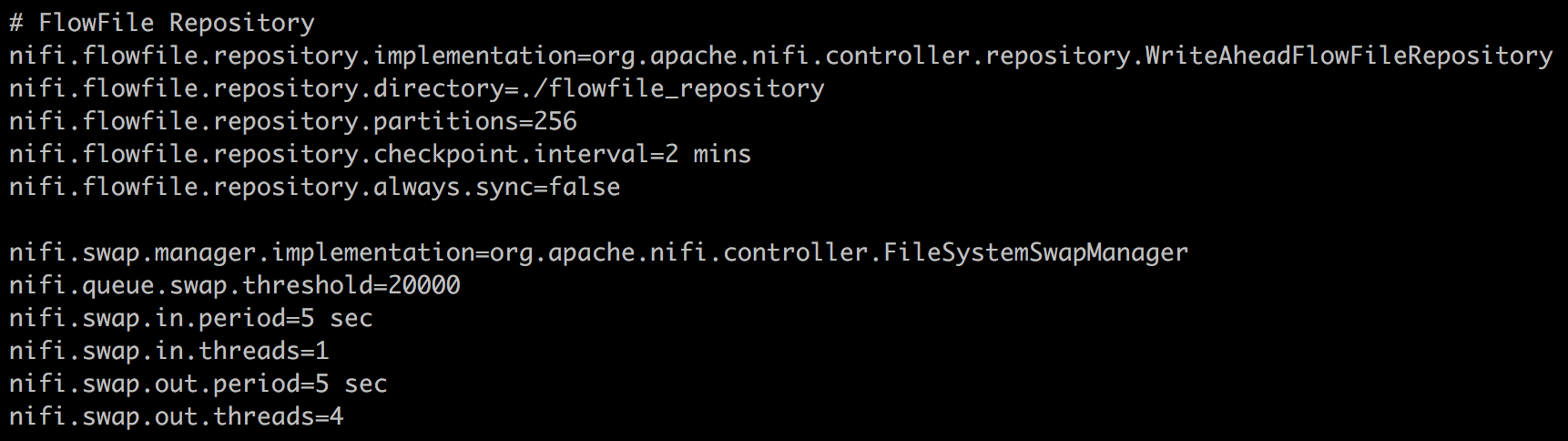
This property sets the value at which the latest statistics, bulletins and flow revisions will be refreshed pushed to connected browser sessions. In order to reload the complete **dataflow** the user must trigger a refresh. Decreasing the time between refreshes will allow bulletins to present themselves to the user in a timelier manner; however, doing so will increase the network bandwidth used. The number of concurrent users accessing the **U**I compounds this. We suggest keeping the default value and only changing it if closer to real–time bulletin or statistics reporting in the UI is needed. The user can always manually trigger a refresh at any time by right clicking on any open space on the graph and selecting “refresh status”.

**# H2 Settings**



There are two **H2 databases** used by NiFi. A user DB (keeps track of user logins when the **NiFi** is secured) and history DB (keeps track of all changes made on the graph) that stay relatively small and require very little hard drive space. The default installation path of **<root-level-nifi-dir>/database\_repository** would result in the directory being created at the root level of your **NiFi** installation (same level as conf, bin, lib, etc directories). While there is little to no performance gain by moving this to a new location, we do recommend moving all repositories to a location outside of the NiFi install directories. It is unlikely that the user or history DBs will change between NiFi releases so moving it outside of the base install path can simplify upgrading; allowing you retain the user and component history information after upgrading.

**# FlowFile Repository**



This is by far the most important repository in **NiFi**. It maintains state on all **FlowFiles** currently located anywhere in the data flows on the **NiFi UI**. If it should become corrupt, the potential exists that you could lose access to some or all of the files currently being worked on by NiFi. The most common cause of corruption is the result of running out of disk space. The default configuration again has the repository located in <root-level-nifi-dir>/flowfile\_repository.

**nifi.flowfile.repository.directory=./flowfile\_repository**

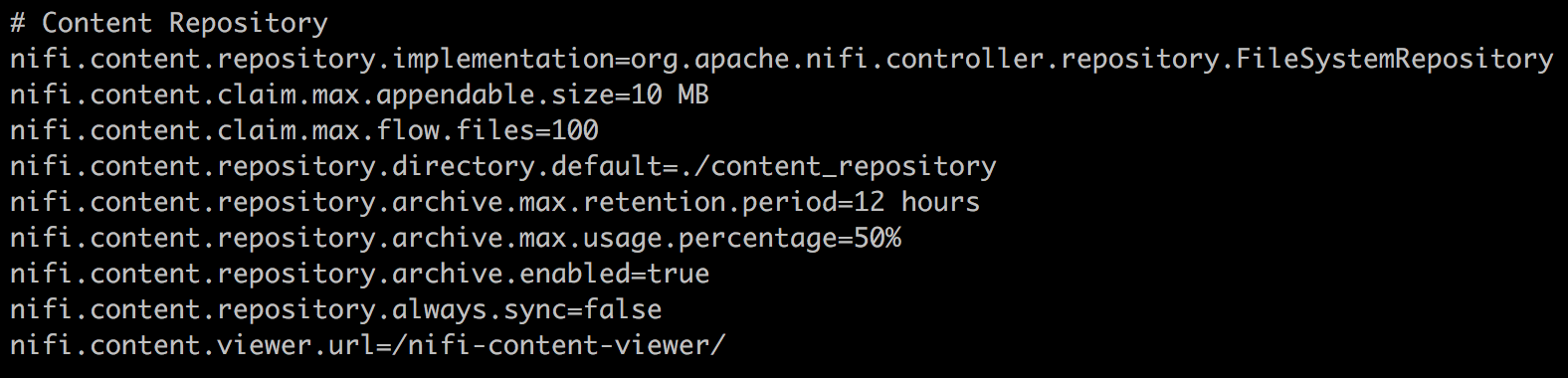
For the same reason as mention for the database repository, we recommend **moving this repository out of the base install path**. You will also want to have the **FlowFile** repository located on a disk (high performance RAID preferably) that is not shared with **other high I/O software**. On high performance systems, the **FlowFile** repository should **never be located on the same hard disk/RAID** as either the **content repository** or provenance repository if at all possible.

NiFi does not move the physical file (content) from processor to processor, FlowFiles serve as the unit of transfer from one processor to the next. In order to make that as fast as possible, FlowFiles live inside the JVM memory. This is great until you have so many FlowFiles in your system that you begin to run out of JVM memory and performance takes a serious dive. To reduce the likelihood of this happening, NiFi has a threshold that defines how many FlowFiles can live in memory on a single connection queue before being swapped out to disk. Connections are represented in the UI as arrows connecting two processors together. FlowFiles can queue on these connections while waiting for the follow-on processor.

**nifi.queue.swap.threshold=20000**

If the number of total FlowFiles in any one-connection queue exceeds this value, swapping will occur. Depending on how much swapping is occurring, performance can be affected. If queues having an excess of **20,000 FlowFiles** is the norm rather then the occasional data surge for your data flow, it may be wise to increase this value. You will need to closely monitor your heap memory usage and increase it as well to handle these larger queues. Heap size is set in the **bootstrap.conf** file that will be discussed later.

**# Content Repository**



Since the content (the physical file that was ingested by NiFi) for every **FlowFile** that is consumed by NiFi is placed inside the content repository, the hard disk that this repository is loaded on will experience high I/O on systems that deal with high data volumes. As you can see, once again the repository is created by default inside the NiFi installation path.

**nifi.content.repository.directory.default=./content\_repository**

The content repository should be moved to its own hard disk/RAID. Sometimes even having a single **dedicated high performance RAID** is not enough. **NiFi** allows you to configure multiple content repositories within a single instance of NiFi. NiFi will then round robin files to however many content repositories are defined. Setting up additional content repositories is easy. First remove or comment out the line above and then add a new line for each content repository you want to add. For each new line you will need to change the word “default” to some other unique value:

**nifi.content.repository.directory.contS1R1=/cont-repo1/content\_repository**

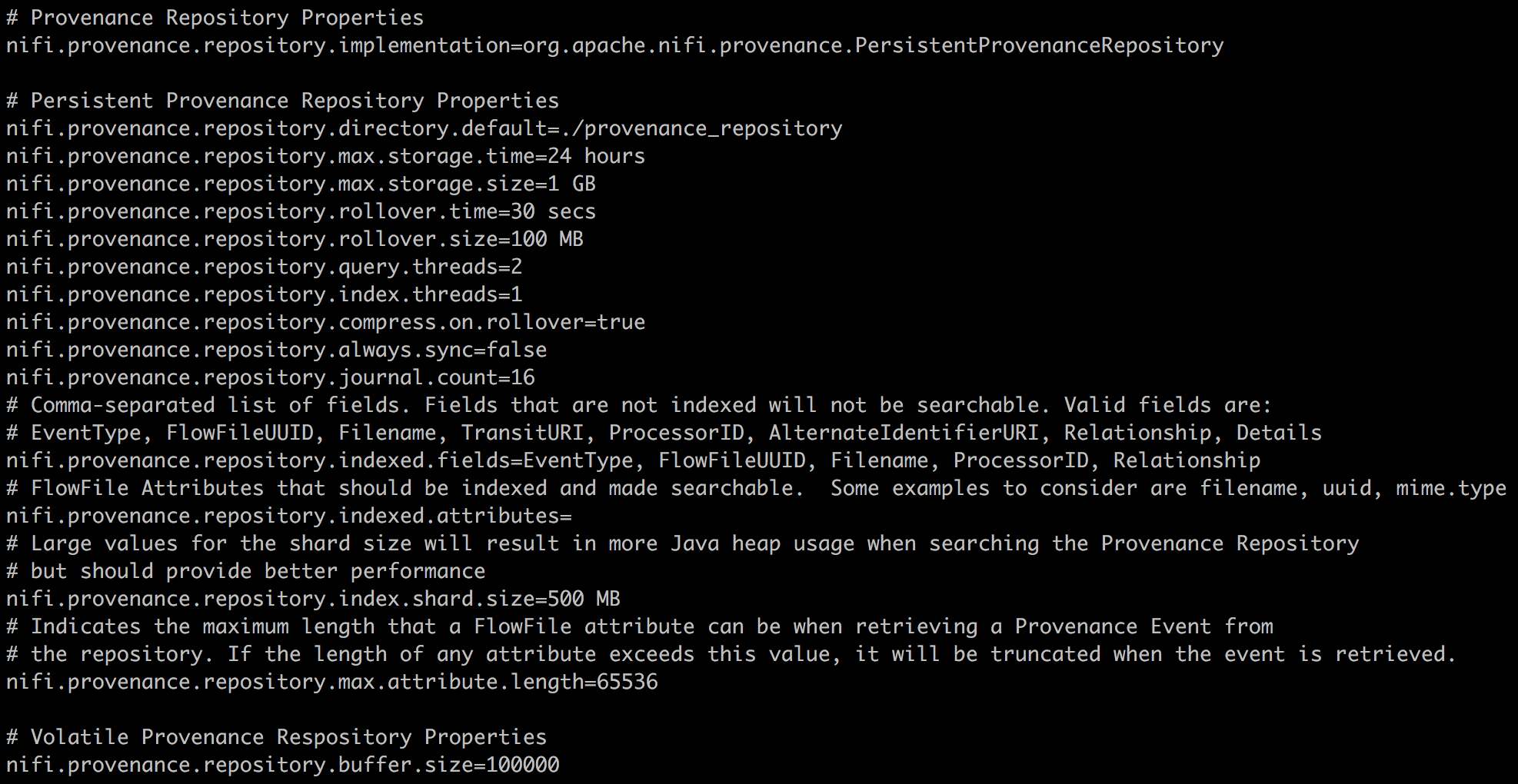
**nifi.content.repository.directory.contS1R2=/cont-repo2/content\_repository**

**nifi.content.repository.directory.contS1R3=/cont-repo3/content\_repository**

In the above example, you can see the default was changed to **contS1R1**, **contS1R2**, and **contS1R3**. The ‘S’ stands for System and ‘R’ stands for Repo. The user can use whatever names they prefer as long as they are unique to one another. Each hard disk / RAID would be mapped to either /cont-repo1, /cont-repo2, or /cont-repo3 using the above example. This division of I/O load across multiple disks can result in significant performance gains and durability in the event of failures.

\*\*\* In a **NiFi cluster**, every Node can be configured to use the same names for their various repositories, but it is recommend to use different names. It makes monitoring repo disk utilization in a cluster easier within the **NiFi UI.** Multiple repositories with the same name will be summed together under the system diagnostics window. So in a cluster you might use **contS2R1**, **contS2R2**, **contS3R1**, **contS3R2**, and so on.

**# Provenance Repository**



Similar to the content repository, the provenance repository can use a large amount of disk I/O for writing and reading provenance events because every transaction within the entire dataflow the affects either the **FlowFile** or content has a provenance event created The UI does not restrict the number of users who can make simultaneous queries against the provenance repository. I/O increases with the number of concurrent queries taking place. The default configuration has the provenance repository being created inside the NiFi installation path just like the other repositories:

**nifi.provenance.repository.directory.default=./provenance\_repository**

It is recommended that the provenance repository is also located on its own hard disk /RAID and does not share its disk with any of the other repositories (database, **FlowFile,** or content). Just like the content repository, multiple provenance repositories can be defined by providing unique names in place of ‘default’ and paths:

**nifi.provenance.repository.directory.provS1R1=/prov-repo1/provenance\_repository**

**nifi.provenance.repository.directory.provS1R2=/prov-repo2/provenance\_repository**

In the above example, you can see the default was changed to **provS1R1** and **provS1R2**. The ‘**S**’ stands for System and ‘**R**’ stands for Repo. The user can use whatever names they prefer as long as they are unique to one another. Each hard disk / RAID would be mapped to either **/prov-repo1** or **/prov-repo2** using the above example. This division of **I/O** load across multiple disks can result in significant performance gains and limited durability in the event of failure when dealing with large amounts of provenance data.

The ability to query provenance information on files has great value to a variety of users. The number of threads that are available to conduct provenance queries is defined by:

**nifi.provenance.repository.query.threads=2**

On systems where numerous users may be making simultaneous queries against the provenance repository, it may be necessary to increase the number of threads allocated to this process. The default value is 2.

The number of threads to use for indexing Provenance events so that they are searchable can be adjusted by editing this line:

**nifi.provenance.repository.index.threads=1**

The **default value is 1**. For flows that operate on a very high number of **FlowFiles**, the indexing of Provenance events could become a bottleneck. If this is the case, a bulletin will appear indicating, "The rate of the dataflow is exceeding the provenance recording rate. Slowing down flow to accommodate." If this happens, increasing the value of this property may increase the rate at which the Provenance Repository is able to process these records, resulting in better overall throughput. Keep in mind that as you increase the number of threads allocated to one process, you reduce the number available to another. So you should leave this at one unless the above error message is encountered.

When provenance queries are performed, the configured shard size has an impact on how much of the heap is used for that process:

**nifi.provenance.repository.index.shard.size=500 MB**

Large values for the shard size will result in more Java heap usage when searching the Provenance Repository but should provide better performance. The default value is 500 MB. This does not mean that 500 MB of heap is used. Keep in mind that if you increase the size of the shard, you may also need to increase the size of your overall heap, which is configured in the **bootstrap.conf** file discussed later.

NOTE: While NiFi provenance cannot be turned off, the implementation can be changed from "**PersistentProvenanceRepository**" to "**VolatileProvenanceRepository**". This move all provenance storage into heap memory. There is one config setting for how much heap can be used:

**nifi.provenance.repository.buffer.size=100000**

This value is set it bytes. Of course by switching to Volatile you will have very limited provenance history and all provenance is lost anytime the **NiFi JVM** is restarted, but it remove some of the overhead associate to provenance if you have no need to retain this information.

------------------------------------

The following is just an example system configuration for a high performance Linux system. In a NiFi cluster, every Node would be configured the same way. (The hardware specs are the same for other operating systems; however, disk partitioning will vary):

CPU: 24 - 48 cores

Memory: 64 -128 GB

Hard Drive configuration:

(1 hardware RAID 1 array)

(2 or more hardware RAID 10 arrays)

RAID 1 array (This could also be a RAID 10) logical volumes:

* -/
* -/boot
* -/home
* -/var
* -/var/log/nifi-logs <-- point all your NiFi logs (logback.xml) here
* -/opt <-- install NiFi here under a sub-directory
* -/database-repo <-- point NiFi database repository here
* -/flowfile-repo <-- point NiFi flowfile repository here

1st RAID 10 array logical volumes:

* -/cont-repo1 <-- point NiFi content repository here

2nd RAID 10 array logical volumes:

- /prov-repo1 <-- point NiFi provenance repository here

3rd RAID 10 array logical volumes (recommended):

- / cont-repo2 <-- point 2nd NiFI content repository here

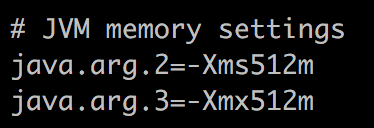
4th + RAID arrays logical volumes (optional):

\*\*\* Use additional RAID arrays to increase the number of content and/or provenance repositories available to your NiFi instance.

**Bootstrap.conf file:**

The bootstrap.conf file in the conf directory allows users to configure settings for how NiFi should be started. This includes parameters, such as the size of the Java Heap, what Java command to run, and Java System Properties. This files comes pre-configured with default values. We will focus on just the properties that should be changed when installing NiFi for the purpose of high volume / high performance dataflows. The bootstrap.conf file is broken in to sections just like the nifi.properties file. Since NiFi is a Java application it requires that Java be installed. NiFi requires Java 7 or later. Some recommended settings only apply if you are using Java 7. We will only highlight the sections that need changing.

**# JVM memory settings**



This section is used to control the amount heap memory to use by the JVM running NiFi. Xms defines the initial memory allocation, while Xmx defines the maximum memory allocation for the JVM. As you can see the default values are very small and not suitable for dataflows of any substantial size. We recommend increasing both the initial and maximum heap memory allocations to at least 4 GB or 8 GB for starters.

**java.arg.2=-Xms8g**

**java.arg.3=-Xmx8g**

If you should encounter any “out of memory” errors in your NiFi app log, this is an indication that you have either a memory leak or simply insufficient memory allocation to support your dataflow. When using large heap allocations, garbage collection can be a performance killer (most noticeable when Major Garbage Collection occurs against the Old Generation objects). When using larger heap sizes it is recommended that a more efficient garbage collector is used to help reduce the impact to performance should major garbage collection occur.

Be mindful that setting an extremely large HEAP size can have its pluses and minuses. It give you more room to process large batches of FlowFiles concurrently, but if your flow has Garbage collection issues it could take considerable time for a full garbage collection to complete. If these stop the world events (Garbage collection) takes longer then the configured heartbeat threshold configured in a NiFi cluster setup, you will end up with nodes becoming disconnected form your cluster.

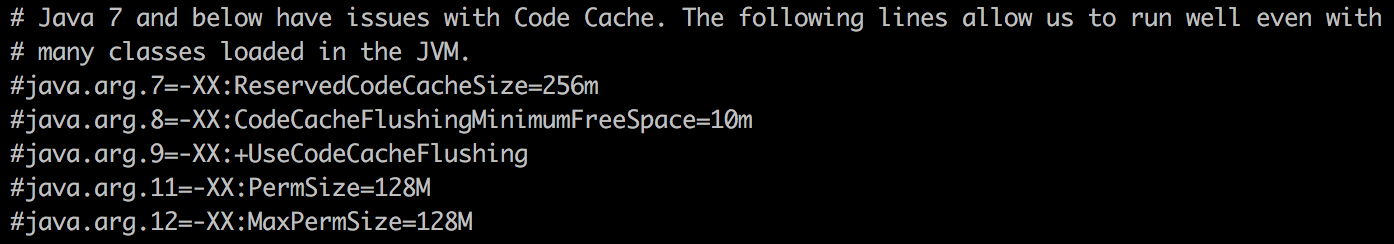
You can configure your NiFi to use the G1 garbage collector by uncommenting the above line. (G1GC is default in HDF 2.x versions)

**java.arg.13=-XX:+UseG1GC**

**# Java 7 and below only (This applies to HDF 1.x or NiFi 0.x versions only)**

**There is a section in the bootstrap.conf file that includes several commented out Java parameters that can improve performance if you are running with a version of Java 7. (This section is only in HDF 1.x versions) We do not set these values by default because Java 8 uses slightly different values and the PermGen settings do not apply to Java 8.**

**JAVA 7 tuning:**



**Increase the Code Cache size by uncommenting this line:**

**java.arg.7=-XX:ReservedCodeCacheSize=256m**

**The code cache is memory separate from the heap that contains all the JVM bytecode for a method compiled down to native code. If the code cache fills, the compiler will be switched off and will not be switched back on again. This will impact the long running performance of NiFi. The only way to recover performance is a restart the JVM (restart NiFi). So by removing the comment on this line, the code cache size is increased to 256m,which should be sufficient to prevent the cache from filling up. The default code cache is defined by the Java version you are running, but can be as little as 32m.**

**An additional layer of protection comes by removing the comment on the following 2 lines:**

**java.arg.8=-XX:CodeCacheFlushingMinimumFreeSpace=10m**

**java.arg.9=-XX:+UseCodeCacheFlushing**

**This parameter 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.**

**The final two lines cover increasing the default size of the permgen space. This can be set by removing the comments on the following two lines:**

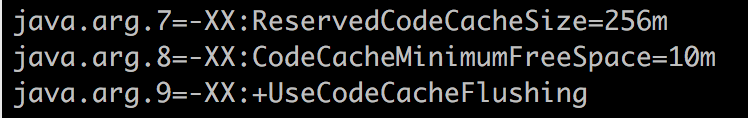
**java.arg.11=-XX:PermSize=128M**

**java.arg.12=-XX:MaxPermSize=128M**

**The permgen space is where Java 7 class definitions and NiFi class definitions are loaded on startup. The default permgen size is often not large enough to accommodate all these classes being loaded on startup. Depending on the number of classes being loaded, more or less space may be allocated. The value provided above is more then enough to accommodate NiFi’s current classes; however, if you have added additional custom processors/classes, permgen size may need to be increased.**

**JAVA 8 tuning:**

**# Java 8 (This applies to all versions of HDF 1.x, HDF 2.x, NiFi 0.x, and NiFi 1.x when running with Java 8. HDF 2.x and NiFi 1.x require a minimum of Java 8)**



**You can add the above lines to your NiFi bootstrap.conf file to improve performance if you are running with a version ofJava 8.**

**Increase the Code Cache size by uncommenting this line:**

**java.arg.7=-XX:ReservedCodeCacheSize=256m**

**The code cache is memory separate from the heap that contains all the JVM bytecode for a method compiled down to native code. If the code cache fills, the compiler will be switched off and will not be switched back on again. This will impact the long running performance of NiFi. The only way to recover performance is a restart the JVM (restart NiFi). So by removing the comment on this line, the code cache size is increased to 256m,which should be sufficient to prevent the cache from filling up. The default code cache is defined by the Java version you are running, but can be as little as 32m.**

**An additional layer of protection comes by removing the comment on the following 2 lines:**

**java.arg.8=-XX:CodeCacheMinimumFreeSpace=10m**

**java.arg.9=-XX:+UseCodeCacheFlushing**

**This parameter 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.**

**Conclusion:**

**At this point you should have an environment configured so it that will support the building of a high volume / high performance data flow using NiFi. While this guide has set the stage for a solid performing system, performance can also be significantly governed by the configuration of your dataflows themselves. It is very important to understand what the various configuration parameters on every processor mean and how to read the information the processors are providing during run time (In, Read/Write, Out and Tasks/Time). Through the interpretation of this provided information, a user can adjust the parameters so that a dataflow is optimized as well, but that is information for another guide.**