Hammerscript and the Hammerspace Toolkit Guide

Table of Contents

About This Documentation

Hammerspace presents a cross-platform global namespace where users and applications can have direct, multi-protocol access to all files, regardless of which storage type, cloud vendor, or location they are in today, or move to in the future. Within this namespace, the Hammerscript language allows you to define and work with customized rich metadata by allowing you to express things in a semi-English language that are evaluated in an intuitive way.

Hammerspace uses all the usual metadata you'd expect in a traditional POSIX-style file system such as name, permissions and access controls. Additionally, it expands the types of relationships you can express with richer metadata.

Our Hammerscript language syntax is modeled after the formula syntax used by spreadsheets such as OpenOffice Calc, Google Sheets, or Excel. If you know how to write a formula in a spreadsheet cell, you already know many of the basics of Hammerscript. The customized metadata embedded in a Hammerspace namespace by Hammerscript is typically a kind of business logic, yet extremely powerful and flexible to meet the many ways data is managed and moved.

This document is an introduction to the concepts of Hammerspace metadata, and the tools to manipulate, and use metadata and put it to work to achieve business objectives for data. The reader will learn what metadata is, what it includes or can include, and how it is managed. The Hammerscript language is introduced, along with tools that use Hammerscript to query and orchestrate data management functions. These tools include Hammerspace Toolkit (HSTK) and the Hammerspace Metadata Plugin for Windows File Explorer.

For support resources, see the Hammerspace Support Hub and the Hammerspace Licensing and Delivery Portal.

Do not install additional or third-party software, agents, or packages directly on Hammerspace Anvil (metadata) or DSX (data) nodes. These nodes run a managed appliance software stack; anything installed outside that stack is unsupported and is removed during a software upgrade. Such software can also interfere with node operation. If you need monitoring or integration with an external tool, contact Hammerspace Support for a supported approach.

Chapter 1. Leveraging Hammerscript and the HSTK for Data Management

Hammerscript is the proprietary query language used to query the metadata of files stored on the Hammerscript platform. These queries are used for a variety of purposes including defining data orchestration policies and for any manner of file system reporting.

In this document, we will provide a variety of examples of how you use the Hammerspace Toolkit and Hammerscript to query and manipulate file system metadata.

Some of the examples provided in this document include:

- Identify data older than X years, add it to a Hammerspace collection, and then delete it.
- List the number of files and total space used, per file owner
- Quickly output the file count and space used for each of your user home directories
- Quickly output the file count and space used for each file owner
- Output the file count and space used for all your Hammerspace shares (in one view)
- Output the top 10 files by space used for all your Hammerspace shares (in one view)
- Output the file instance count and space used for each cluster volume, on a per share basis

1.1. What Is the Hammerspace Toolkit (HSTK)?

The Hammerspace Toolkit (HSTK) is a collection of Python-based extensions developed for Linux Bash. However, it can also be used in the Windows command line (CMD or PowerShell) if either Python 3 or Windows Services for Linux (WSL) installed. Furthermore, the Hammerspace Toolkit additionally offers the ability to simplify common commands using the system wide bash completions outlined in the documentation.

There are many possible uses for the HSTK. Some of the examples featured in this document include:

- Developing highly customized filesystem reports, which are delivered quickly and use real time data
- Reviewing and managing file and directory metadata
- Reviewing and managing file and directory objectives
- Offloading resource intensive filesystem operations to the cluster directly

1.2. What Is the Hammerspace Metadata Plugin for Windows File Explorer?

The Hammerspace Metadata Plugin for Windows File Explorer enables users to right click a file or directory on a Hammerspace share, and then access and manipulate the more common custom metadata values without the need to use the HSTK. For the full power of Hammerspace advanced metadata, the HSTK should be used.

- The Hammerspace Metadata Plugin can be downloaded from the Hammerspace licensing portal.
- Additional information about the plugin is available in the **Hammerspace Metadata Plugin for Windows** section of this document.

1.3. What Is Metadata?

Simply put, metadata is data or information about other data. Anybody who has used a modern computer and operating system is familiar with concepts of files and directories within a file system. Directories and file names are the most basic metadata that we use daily, and are the most basic structures of a POSIX file system. POSIX also includes basic file properties like:

- Size
- Ownership (user and group)
- Permissions (Unix style rwx or an Access Control List)
- Timestamps including last access time (atime), last time the file contents were modified (mtime), and last time the file metadata or inode (think fileheader) was changed (ctime). Some also include a birth or create time.

Hammerspace manages and orchestrates unstructured data by separating metadata from data. Metadata is stored in a database on an Anvil node, rather than in inode, directory, and other structures in a volume as with other filesystems. Data, or file *contents,* is stored as an instance in one or more volumes on any or a combination of DSX nodes, NFS servers, or object storage.

1.4. Key Terms

Term	Definition
Anvil	A Hammerspace metadata server.
Clone	A duplicate of a file or a share.
DSX	Hammerspace data services node, which could be a direct storage or just as a data portal and a mover.
Hammerscript	A language for querying and manipulating metadata and related concepts and features of Hammerspace.
HSTK	Hammerspace Toolkit - Command line extensions that enable access to and manipulation of Hammerspace metadata.
Inode	A data structure in a Unix-style file system that describes a file-system object such as a file or a directory.

Term	Definition
Instance	When a file is placed on a Hammerspace share, one or more instances will be written to cluster volumes based on the objective requirements. In a global file share, the only instances may be on a volume at another site.
NAS	Network attached storage.
Objective	Uses metadata criteria to orchestrate data.
Retention policy	Specified length of time to keep a file system snapshot or file clone.
Schedule	Specified time and frequency to take a snapshot or make a file clone.
Shadow Namespace	The shadow Namespace provides hidden access to Hammerspace metadata and functionality through a mounted file system.
Share	A file system presented for file access by clients via NFS, SMB, and/or S3 protocols. When a client writes a file to a Hammespace share, the cluster then uses objectives to determine where to place the file.
SLO	Service-level objective. A target value or ranges of values measured by service-level indicators. In Hammerscript, SLO often refers to an objective to place file instances on a specific volume, volume group, or storage system.
SMB	Server message block. A network communication protocol for providing shared access to files, printers, and serial ports between nodes on a network.
Snapshot	A record of the state of the file system at a specific point in time.
Volume	A storage container used by Hammerspace to store instances of files. May be a disk device on a DSX node, an export on a third-party storage system, or a bucket in cloud or on-premises object storage. Volumes are selected based on the objectives that apply to the file.

Chapter 2. How Hammerspace Uses Metadata

Hammerspace metadata is actionable in that you can use it to orchestrate data. These orchestration use cases can be broken down into three broadly defined categories: presentation, placement, and protection. In this section, we will review examples of each of these categories.

2.1. Presentation

You can do ad-hoc queries, similar to the Linux find command, but with many additional options than typical POSIX metadata. The **File Metadata Example (Full)** section of this document contains an export of the full metadata values for a single value. Virtually every value seen there can be used when creating queries.

Hammerspace shares also contain a default set of collections, which are subsets of files determined by a predefined query of metadata updated on each pass of the sweeper process, a daemon that scans the files in all Hammerspace shares to determine if they are aligned (compliant) to the objectives. Collections are accessible in any directory in a Hammerspace share in **.collections**, for both Linux (NFS) and Windows (SMB).

2.2. Placement

Hammerspace metadata is used by objectives to determine tiering, replication for distributed workforce, cloud applications such as analytics and rendering, as well as disaster recovery.

2.3. Protection

Objectives use metadata to apply rules for mirroring, applying and complying with 9's of availability and durability, backup (including snapshots), and WORM.

Chapter 3. Hammerspace Metadata Fundamentals

In this section, we will review some key fundamental concepts regarding Hammerspace metadata.

3.1. Hammerspace Metadata

Hammerspace manages three classes of metadata:

- **POSIX** - includes the directory structure, file names, and basic file properties.
- **Internal** - includes additional statistics, file instance locations, etc.
  - More detailed file usage data (beyond atime and mtime)
  - Extensive inode information - e.g. volume(s) and path(s) to instances
- **Custom** - rich metadata that can be defined by the administrator or user.
  - Can be manually created or extracted and applied by script
  - Customer or third-party tools and applications can directly access and manipulate custom metadata

3.1.1. Custom Hammerspace Metadata

There are four types of custom metadata: tags, attributes, labels, and keywords. The acronym **TALK** can be used to remember them. The distinction between the types is two dimensional:

- Whether there is a predefined schema or set of metadata that the user can apply
- Whether the type uses a simple string or key-value pair

	Schema	Non-Schema
Simple Strings	Label	Keyword
Key-Value Pairs	Attribute	Tag

Attributes are defined by Hammerspace and are generally not configurable by the administrator, although they can be applied and values set by the user.

3.2. Interacting with Metadata

It is important to note that Hammerspace does not change the rules with regard to interacting with filesystem metadata. The following rules apply to Hammerspace just as they would with any other storage system:

- Users can only see metadata of files to which they have read access.
- Users can only manipulate metadata of files to which they have write access.

3.3. Shadow Namespace

Hammerspace metadata can be accessed and manipulated by Linux clients through the shadow namespace that exists in any mounted Hammerspace share. Hammerspace implements a proprietary file naming extension that invokes calls into the shadow namespace. If a user reads a file adding '?.' (question mark and dot) and some valid Hammerspace metadata reference to the filename, the filesystem will return metadata (or take action on metadata) based on the metadata reference rather than the file contents.

For example, if a file called telemetry.txt exists in the current directory, **cat telemetry.txt** would return the contents of the file. However, **cat telemetry.txt?.attribute=inode_info** would return extensive information about the inode (fileheader) and other internal Hammerspace metadata about the file.

Chapter 4. Hammerscript Fundamentals

Hammerscript is the query language used throughout Hammerspace to select a set of files and take a specified action on them. Some of the capabilities it enables includes:

- Determining which files to place on which storage volume(s), via one or more objectives
- Determining which data protection features to apply (WORM, write or read deny or block, versioning, undelete, antivirus)
- Determining which files to include in a collection (in the **.collection** directories)
- Running queries or setting metadata on a file or set of files, typically using HSTK

Hammerscript syntax is based on and meant to resemble Microsoft Excel formulas and Visual Basic. You can use simple expressions, or use an if-then-else structure. The structure also has shortcuts. For example:

```
If <condition> then <action1> else <action2>
```

Can be represented as:

```
<condition>?<action1>:<action2>
```

You can run queries using HSTK to find sets of files and other information using Hammerspace metadata. For example, this command finds files over 1 megabyte in size (**SIZE>1MBYTE**) and returns their paths relative to the share (**PATH**):

```
$ hs eval -r -e 'SIZE>1MBYTE?PATH' .
```

The previous command is equivalent to the Linux command **find . -size +1M** except for two differences:

- The output from the hs command is the path to the file relative to the share, not the pwd where the command was run.
- The command avoids making many calls to the physical file instance, but rather queries the Anvil metadata database directly. By separating the metadata from the files, you avoid the need to consume file storage system resources for metadata operations.
  - This is particularly important when a file has been tiered off to a remote object storage volume where, with other storage architectures, the metadata is not readily available to read.

The **Using the Hammerspace Toolkit** and **Advanced Filesystem Reporting** section of this document provide examples for every HS command.

Refer to the Appendix section **Common Operators Used With Hammerscript** for a list of operators used when creating Hammerscript queries.

4.1. Combining or Modifying Conditions

In this section, we will explore how we combine multiple conditions into longer statements, and how we apply a negative match to conditions. With just a basic understanding of how conditions can be combined or modified, you will quickly learn how to craft complex ones that target exactly the data you need.

4.1.1. Marking a Condition as a Negative

By using a **!**, you can apply a negative to many conditions. This is useful when you want to use a condition to exclude files, rather than target them.

Here are some examples of negative matches that will exclude files.

Match files that do not have the **Space Launch** tag:

```
# Match files that do NOT have the tag "Space Launch" (case sensitive)
!HAS_TAG("Space Launch")
```

Match files that do not start with **mars** (lowercase):

```
# Match files that do NOT have the name "mars*.*" (case sensitive)
!FNMATCH("mars*.*",NAME)
```

Match files that do not have the **Project Apollo** label:

```
# Match files that do NOT have the label "Project Apollo"
!HAS_LABEL(LABEL('Project Apollo'))
```

4.1.2. Build a Hammerscript Conditional Statement

Conditions do not need to be singular. You can combine them with **and** or **or** statements and even **( )**. In this section, we will review three different conditions that also leverage conditional statements.

It is important to note that plain language operators are also supported. For example, despite using different operator formats the following expressions will produce the same result:

```
# Hammerscript expression that uses symbols for all operators
IS_LIVE&&((HAS_LABEL("launch_report"))||LAST_USE_AGE<=1*DAYS)

# Hammerscript expression that uses plain language for some operators
IS_LIVE AND ((HAS_LABEL("launch_report")) OR LAST_USE_AGE<=1*DAYS)
```

Each of the examples provided uses ⇐ or **>=** operators for date statements, meaning that the times referenced will also match the condition. Remove the **=** if you do not want to match the time referenced, and only the value that precedes or follows it.

Conditional Statement Example 1

Desired match: Live files only (no snapshot data) **AND EITHER** used within the last 6 months and also has the "Space Launch" tag **OR** contained within any directory named jupiter and used within the last 30 days.

Equivalent condition written in Hammerscript:

```
# Match live files that were (used is last 6 months AND have the tag Space Launch) OR (are within a
directory named jupiter AND were used in the last 30 days)
IS_LIVE AND ((LAST_USE_AGE<6*MONTHS AND HAS_TAG("Space Launch")) OR (FNMATCH("*/jupiter/*",PATH) AND
LAST_USE_AGE<30*DAYS))
```

Conditional Statement Example 2

Desired match: All files that start with **mars** (all lowercase) **OR** files with the **orbit** extension (all lowercase) that were created less than 60 days ago.

Equivalent condition written in Hammerscript:

```
# Match all files whose name starts with mars OR (were created within the last 60 days AND match the file
name *.orbit)
FNMATCH("mars*.*",NAME) OR (CREATE_AGE < 60*DAYS AND FNMATCH("*.orbit",NAME))
```

Conditional Statement Example 3

Desired match: Live files last used between 6 months and 1 year ago, including the 6 month and 1 year mark.

Equivalent condition written in Hammerscript:

```
# Match live files that were last used between 6 months and 1 year ago (inclusive of the start and end
dates)
IS_LIVE AND (LAST_USE_AGE>=6*MONTHS AND LAST_USE_AGE<=1*YEARS)
```

Chapter 5. Hammerspace Toolkit - Overview

In this section we will discuss how to use the Hammerspace Toolkit, and outline all the command types. The HSTK commands can be arranged into groups of three based on their typical use case:

- **Metadata Operations** - Query or interact with filesystem metadata.
- **Filesystem Operations** - Perform typical filesystem operations, but offload the task to the cluster.
- **General** - Other commands not related to filesystem or metadata operations.

5.1. Obtaining Help

The HSTK provides contextual help, which allows you to obtain help for the actual command entered rather than only the base command. To display the help for a HSTK command, type the command and add the --help option.

To output the help for all top level hs commands, execute the **hs** command with the --cmd -tree option.

The following example shows the output of the hs attribute --help command:

```
# Help page for hs attribute
$ hs attribute --help
Usage: hs attribute [OPTIONS] COMMAND [ARGS]...

attribute: Manage Hammerspace embedded attribute metadata

Attributes must be pre-created on the anvil and currently requires
assistance from hammerspace support.

Attributes can hold a value. Some values are string type (use -s) others
are a number or expression (use -e), for example values true/false are -e

ex: hs attribute set CONSIDER_VOLATILE -e false path/to/file_or_dir
ex: hs attribute set COLOR -e 'COLORS_TABLE{COLOR_LOOKUP("RED")}'
path/to/file_or_dir

Options:
--help Show this message and exit.

Commands:
list list all attributes and values applied
get Get the attribute's value
has Is the inode's attribute value non-empty
delete remove attribute values from inode(s)
set Add/Set value of attribute on inode(s)
add Add/Set value of attribute on inode(s)
```

Contextual support allows us to add an additional command and get information for that option. In the below example, we will expand the previous command to hs attribute get --help:

```
# Help page for hs attribute get
$ hs attribute get --help
Usage: hs attribute get [OPTIONS] NAME paths

Get the attribute's value

Options:
-r, --recursive Apply recursively
--nonfiles Apply recursively to non files
--raw Print raw output
--compact Print compact output
-l, --local Show only settings directly applied to this file
-h, --inherited Show only settings inherited by this file from a parent
-o, --object Show only settings inherited by this file from the data
object

-u, --unbound Delay binding of any expression, it will be evaluated fresh
each time

--help Show this message and exit.
```

5.2. Command Syntax - General

If no path is provided, all **hs** commands (excluding hs eval) execute from within the current directory. Additionally, you can target the directory you are in. All examples are shown below.

While at times HSTK commands may be shown in uppercase letters, commands will work the same if entered in lowercase. This formatting is really just to make them easier to read. The only time that case matters is when you are trying to match a pattern, in which case you must use the exact case you are attempting to match.

IMPORTANT! If you are using the HSTK on a Windows client, it is recommended that you execute the commands from within a PowerShell command prompt. While the HSTK is not itself a PowerShell command set, the PowerShell interface processes input differently than an ordinary Windows command prompt. You may encounter errors when supplying expressions if you use the ordinary Windows command prompt.

In this example, no path is provided so the command will execute against the contents of the current directory:

```
# Returns the number of and space used by all files accessed within the last 7 days.
$ hs sum -e 'IS_FILE&&ACCESS_AGE<=7DAYS?{1FILE,SPACE_USED}'
{24 FILES, 13.894 MBYTES}
```

In this example the path is provided and the command executes against the contents of the specified directory:

```
# Same command as the previous example, but targets the /home/david directory
$ hs sum -e 'IS_FILE&&ACCESS_AGE<=7DAYS?{1FILE,SPACE_USED}' /home/david
{5 FILES, 8.494 MBYTES}
```

In this example a period was provided at the end of the command, which instructs it to return information about the directory you are currently in. Replace the period with a filename to target a specific file:

```
# Returns all objectives applied to the current directory (partial command output)
$ hs objective list .
SLOS_TABLE{
|OBJECTIVE = SLO('keep-online'),
|COUNT = EXPRESSION(IS_BEING_CREATED OR HAS_ONLINE_INSTANCE AND IS_RECENTLY_USED?ALWAYS);

|OBJECTIVE = SLO('durability-1-nine'),
|COUNT = EXPRESSION(DATA_ORIGIN_LOCAL?ALWAYS);

|OBJECTIVE = SLO('compress-on-object'),
|COUNT = TRUE;
```

- This example command is designed to target a specific file or directory, some others (like **hs eval** or **hs sum**) can be made recursive or are by default. These commands will be discussed later in this guide. When using a **.** as the target of the command, just remember that you are telling the command to start at the directory you are in.
- For more information regarding Hammerspace objectives, refer to the {kb-objectives-url}[{kb-objectives-title}] {kb-login-note}.

5.3. Command Syntax - Selecting Metadata Fields

The **File Metadata Example (Full)** section of this document shows the full output of metadata Hammerspace stores for a single file. Some common metadata values from that output are featured below with the value name on the left and the value on the right:

```
|NAME = "newfile1.txt",
TYPE = ITEM_TYPE('FILE'),
OWNER = USER('neil@catalyst.local'),
OWNER_GROUP = GROUP('Apollo_Astronauts@catalyst.local'),
SIZE = 48 BYTES,
SPACE_USED = 4.096 KBYTES,
CREATE_AGE = 01:03:52.44,
CHANGE_AGE = 01:03:52.15,
ACCESS_AGE = 01:03:52.44,
MODIFY_AGE = 01:03:52.15,
LAST_USE_AGE = 01:03:52.15,
```

Any number of these fields could be selected as is when querying metadata or designating which fields. The

following example shows these fields used both as part of the query (**IS_FILE&&ACCESS_AGE>=2DAYS**), and again when formatting the output (**NAME**):

```
# Return the name of all files last accessed in 2 or more days
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS?NAME'
"./telemetry.txt"
"log_mercury.aaab"
"log_mercury.aaaa"
"log_mercury.aaac"
"log_mercury.aaae"
```

The same rules do not apply when the fields you want to use are contained within tables inside of the metadata. Here is an example of one such table that will be used in later examples:

```
|DIRECTORY_STATS = DIRECTORY_STATS_TABLE{
|FILE_COUNT = 1 FILE,
|SPACE_USED = 1.234 MBYTES,
|ACTIVITY = 0.011574 OPERATIONS / SECOND,
|LAST_USE_TIME = LOCAL_TIME('1901-05-15 20:00:00'),
|LAST_USE_AGE = (123 YEARS+217 DAYS),
|ACTIVITY_LEVEL = IOPS_LEVEL('UNDER 10 IOPS'),
|HEAT = TEMPERATURE(|ACTIVITY=0.011574 OPERATIONS / SECOND),
|HEAT_LEVEL = TEMPERATURE_LEVEL('UNDER 10 IOPS')},
```

To query for values within the table, or specify that they be part of the output, we must use a format similar to ***table*[ROW].*VALUE*** as shown in the following example:

```
DIRECTORY_STATS[ROW].HEAT
```

The following example shows how these fields would be used in a simple query (**DIRECTORY_STATS[ROW].ACTIVITY>0.011IOPS**), and as part of the output (**PATH,DIRECTORY_STATS[ROW].ACTIVITY**):

```
#Return the full path and file name of all files whose last IOPS were greater than .011
$ hs eval -r -e 'IS_FILE&&DIRECTORY_STATS[ROW].ACTIVITY>0.011IOPS?{PATH,DIRECTORY_STATS[ROW].ACTIVITY}'
{"./mission-6/1char.txt", 0.011574 OPERATIONS / SECOND}
{"./mission-6/0byte.txt", 0.011574 OPERATIONS / SECOND}
{"./mission-6/data_gemini.aaaa", 0.011574 OPERATIONS / SECOND}
{"./mission-6/data_gemini.aaac", 0.011574 OPERATIONS / SECOND}
{"./mission-6/data_gemini.aaad", 0.011574 OPERATIONS / SECOND}
{"./mission-6/data_gemini.aaae", 0.011574 OPERATIONS / SECOND}
{"./mission-6/data_gemini.aaaf", 0.011574 OPERATIONS / SECOND}
```

5.3.1. Evaluating Advanced Metadata Values - Specifying the Correct Unit

In the metadata example in the previous section, the value in the table is listed as **OPERATIONS / SECOND**, but our expression has labeled the value **IOPS**.

Here is the value as seen when browsing the file metadata (**OPERATIONS / SECOND**):

```
|ACTIVITY = 0.011574 OPERATIONS / SECOND,
```

Here we see a HSTK command with an expression referencing that value using the correct name (**IOPS**):

```
# Return the full path and file name of all files whose last IOPS were greater than 0.011
$ hs eval -r -e
'IS_FILE&&DIRECTORY_STATS[ROW].ACTIVITY>0.011IOPS?{PATH,DIRECTORY_STATS[ROW].ACTIVITY}'
```

When viewing metadata, certain labels are expanded to make them easier to read. Note that when doing an evaluation, you must use the shortened label.

One way to learn what that value should be is simply to leave the label off and observe the error as shown in the following example:

```
$ hs eval -r -e 'IS_FILE&&DIRECTORY_STATS[ROW].ACTIVITY>0.01?{PATH,DIRECTORY_STATS[ROW].ACTIVITY}'
{"./mission-6/0byte.txt", 0.011574 OPERATIONS / SECOND}}
internal:1:25: WARNING: #B_OP>: A IOPS cannot be combined with a NUMBER
```

In this example, the evaluation included no label (**DIRECTORY_STATS[ROW].ACTIVITY>0.01**), but the resulting error told us what the label should be: *A IOPS cannot be combined with a NUMBER*. This tells us that the actual evaluation should be **DIRECTORY_STATS[ROW].ACTIVITY>0.01IOPS**.

5.3.2. Evaluating Advanced Metadata Values - Working with Levels

Some values you may wish to evaluate are not just numbers, but alphanumeric ranges named levels.

Here's a few examples from a random file:

```
|HEAT_LEVEL = TEMPERATURE_LEVEL('6 TO 12 HOURS OLD'),
SPACE_USED_LEVEL = SIZE_LEVEL('4 TO 32 KBYTES'),
ACCESS_AGE_LEVEL = TIMESPAN_LEVEL('3 TO 6 HOURS OLD'),
MODIFY_AGE_LEVEL = TIMESPAN_LEVEL('6 TO 12 HOURS OLD'),
CHANGE_AGE_LEVEL = TIMESPAN_LEVEL('6 TO 12 HOURS OLD'),
CREATE_AGE_LEVEL = TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'),
LAST_USE_AGE_LEVEL = TIMESPAN_LEVEL('6 TO 12 HOURS OLD'),
ACTUAL_ACCESS_AGE_LEVEL = TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'),
ACTUAL_MODIFY_AGE_LEVEL = TIMESPAN_LEVEL('6 TO 12 HOURS OLD'),
ACTUAL_CREATE_AGE_LEVEL = TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'),
ACTUAL_LAST_USE_AGE_LEVEL = TIMESPAN_LEVEL('6 TO 12 HOURS OLD'),
```

Levels are predefined and cannot be modified. When evaluating against these values, you must include the full text for the level value.

In the following example, we will list only files that fit within the **ACCESS_AGE_LEVEL** specified (**3 TO 6 HOURS**), though our output will list the access age in hours (**ACCESS_AGE/HOURS**):

```
# Return the full path and file name and last accessed age in hours of all files that were last accessed
between 3 and 6 hours ago
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE_LEVEL=TIMESPAN_LEVEL("3 TO 6 HOURS OLD")?{PATH,ACCESS_AGE/HOURS}'
{"./newout.txt", 5.8206624432932585}
{"./newfile1.txt", 4.987130687688477}
{"./mission-6/0byte.txt", 4.96975263801869}
{"./mission-6/1char.txt", 4.96974788652733}
{"./mission-6/data_gemini.aaaa", 4.969744433183223}
{"./mission-6/data_gemini.aaab", 4.969742385903373}
...
```

Best Practice: Levels are particularly useful with hs sum commands, as they enable you to summarize data on a per-level basis. By using the HSTK, you can recreate all of the reports you see (and more) in the GUI Share dashboard, and then import them into a spreadsheet for further analysis. HS sum commands are detailed in the **Reporting** section of this guide.

5.4. Command Syntax - Setting the Scope for Your Expressions

The sample expressions in this document typically use **IS_FILE**, which indicates that only files, not directories, will be examined. If you want your expression to target directories only, use **IS_FILE=FALSE** instead. If you want to examine both files and directories, omit **IS_FILE?** from the expression entirely.

The following example shows all three options: How to examine only files, only directories, or both files and directories:

```
# Examine only files
$ hs sum -e 'IS_FILE?SIZE<10KBYTES?{1FILE/FILES,SIZE/BYTES}' .

# Examine only directories
$ hs sum -e 'IS_FILE=FALSE?SIZE<10KBYTES?{1FILE/FILES,SIZE/BYTES}' .

# Examine both files and directories
$ hs sum -e 'SIZE<10KBYTES?{1FILE/FILES,SIZE/BYTES}' .
```

Best Practice: If you are trying to measure something like average file size, you should examine only files or else your average will likely change due to the space used by a directory itself (typically 4096 bytes).

5.5. Formatting HSTK Output

When working with HSTK commands that evaluate file metadata, it is important to note that they are queries, which may or may not evaluate an expression. Commands such as **hs sum** can *evaluate and summarize* their output, while commands like **hs eval** only *evaluate* an expression and return their result for every file or directory it targets.

Best Practice:

- The hs eval command output is typically line by line, making it easy to parse and import into spreadsheet tools.
- The hs sum command output is often broken up into individual tables, and requires more effort to parse for later analysis in spreadsheet tools.

5.5.1. Formatting Output for HS Eval Commands

Let's use the **hs eval** command recursively (**-r**) to look for files (**IS_FILE**) which were last accessed 2 days or less ago (**ACCESS_AGE⇐2DAYS**). Note that hs sum returns a **TRUE** for every file it finds that matches the expression, but we don't know the actual file name:

```
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS'
TRUE
TRUE
TRUE
TRUE
TRUE
```

The first modification we can make is to make the command return something more useful than just a **TRUE**.

We can do that by appending the expression with a **?** and then the file name field (**NAME**):

```
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS?NAME'
"./telemetry.txt"
"log_mercury.aaab"
"log_mercury.aaaa"
"log_mercury.aaac"
"log_mercury.aaae"
```

Next, let's switch **NAME** for path and space used (**PATH,SPACE_USED**), note that we now need to enclose the output fields in brackets:

```
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS?{PATH,SPACE_USED}'
{"telemetry.txt", 4.096 KBYTES}
{"./log_mercury.aaaa", 4.096 KBYTES}
{"./log_mercury.aaab", 4.096 KBYTES}
{"./log_mercury.aaac", 4.096 KBYTES}
```

```
{"./log_mercury.aaad", 4.096 KBYTES}
{"./log_mercury.aaae", 4.096 MBYTES}
```

Even if you run this command from within a subdirectory of a share, the path displayed will be relative to the share root.

If you intend to import this data into a later analysis, you may notice that some values are labeled (**KBYTES**, M**BYTES**).

By default, many numeric values will modify the units to keep the numbers themselves within a limited range. We can force the HSTK to standardize the units by "dividing" the output (**/BYTES**):

```
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS?{PATH,SPACE_USED/BYTES}'
{"telemetry.txt", 4096 BYTES}
{"./log_mercury.aaaa", 4096 BYTES}
{"./log_mercury.aaab", 4096 BYTES}
{"./log_mercury.aaac", 4096 BYTES}
{"./log_mercury.aaad", 4096 BYTES}
{"./log_mercury.aaae", 4096000000 BYTES}
```

You can convert the **SPACE_USED** value to whatever unit you want, but note that converting small values into larger units may result in output that is difficult to import. For example, 4096 Bytes would be 4.096e-6 Gigabytes. It is better to standardize on a smaller unit, which will be easier to use or modify later.

The same technique can be applied to values that use time, such as **LAST_USE_AGE**. Let's expand the previous command to include **LAST_USE_AGE** and see the standard output:

```
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS?{PATH,SPACE_USED/BYTES,LAST_USE_AGE}'
{"./log_mercury.aaaa", 4096, (3 DAYS+20:29:51)}
{"./log_mercury.aaac", 4096, (3 DAYS+20:29:50)}
{"./log_mercury.aaab", 4096, (3 DAYS+20:29:50)}
{"./log_mercury.aaad", 4096, (3 DAYS+20:29:50)}
{"./log_mercury.aaae", 4096, (3 DAYS+20:29:50)}
```

Once again we see that we have a value that would be difficult to use if imported into a spreadsheet. To solve that, we will convert **LAST_USE_AGE** to hours (**/HOURS**):

```
$ hs eval -r -e 'IS_FILE&&ACCESS_AGE>=2DAYS?{PATH,SPACE_USED/GBYTES,LAST_USE_AGE/HOURS}'
{"./telemetry.txt", 4096, 92.92196696950123}
{"./log_mercury.aaaa", 4096, 92.53868029313162}
{"./log_mercury.aaab", 4096, 92.53867905103834}
{"./log_mercury.aaac", 4096, 92.53867788839852}
{"./log_mercury.aaad", 4096, 92.53867630369496}
{"./log_mercury.aaae", 4096, 92.53867469943361}
```

Similar to working with file sizes, the time is now in a format we can easily import into a spreadsheet and then use or modify later.

5.5.2. Formatting Output for HS Sum Commands

The **hs sum** command is useful when you want the HSTK to perform the actual analysis of the data for you. Note that this requires a more complex syntax.

Let's start with an analysis of instance count and storage used by volume. This report will use the INSTANCES table, VOLUMES row in the file metadata (**KEY=INSTANCES[ROW].VOLUME**) as the key, as that is where the metadata stores the name of the volume where the instances are located. The report will then output the number of files and the total space consumed (**VALUE=\{1FILE,SPACE_USED}**):

```
$ hs sum -e
'IS_FILE&&ACCESS_AGE>=2DAYS?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE,SPACE
_USED}}[ROWS(INSTANCES)]'
SUMS_TABLE{
|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol0'),
|VALUE = {12 FILES, 49.152 KBYTES};

|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol1'),
|VALUE = {13 FILES, 53.248 KBYTES};

|KEY = STORAGE_VOLUME('NYC-dsx-2.catalyst.local::/hsvol0'),
|VALUE = {14 FILES, 57.344 KBYTES};

|KEY = STORAGE_VOLUME('NYC-dsx-2.catalyst.local::/hsvol1'),
|VALUE = {12 FILES, 49.152 KBYTES};

|KEY = STORAGE_VOLUME('Bucket-NYC'),
|VALUE = {51 FILES, 208.9 KBYTES}}
```

While an instance is technically a file, it's important to note that:

- A file may have more than one instance depending on the share objectives
- Files stored on object storage are broken into 4MB chunks, and (if enabled) the chunks are duplicated

We can take the previous expression and expand it to output a table that contains the the top 10 files names by space used per volume (**TOP10_TABLE\{\{SPACE_USED,DPATH}}**):

```
# Note: The table output was shortened
$ hs sum -e
'IS_FILE&&ACCESS_AGE>=2DAYS?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE,SPACE
_USED,TOP10_TABLE{{SPACE_USED,DPATH}}}}[ROWS(INSTANCES)]'
SUMS_TABLE{
|KEY = STORAGE_VOLUME('AZ3:Minio-NFS::/vols/NYC1'),
VALUE = {88 FILES, 68.387 MBYTES, TOP10_TABLE{
|KEY = {16.691 MBYTES, "./Launch00077.log"};
```

```
|KEY = {1.6425 MBYTES, "./countdown.log"};
|KEY = {1.1756 MBYTES, "./Launch00022.log.gz"};
|KEY = {1.151 MBYTES, "./Launch00018.log.gz"};
|KEY = {1.1469 MBYTES, "./Launch00011.log.gz"}}};

|KEY = STORAGE_VOLUME('Minio-S3::gfs'),
VALUE = {88 FILES, 68.387 MBYTES, TOP10_TABLE{
|KEY = {16.691 MBYTES, "./Launch00077.log"};
|KEY = {1.6425 MBYTES, "./countdown.log"};
|KEY = {1.1756 MBYTES, "./Launch00022.log.gz"};
|KEY = {1.151 MBYTES, "./Launch00018.log.gz"};
|KEY = {1.1469 MBYTES, "./Launch00011.log.gz"}}}}
```

You can add additional output fields to the **TOP10_TABLE** by simply inserting your desired fields within the **\{ }** brackets. In this example, we will add **LAST_USE_AGE**:

```
# Note: The table output was shortened
$ hs sum -e
'IS_FILE&&ACCESS_AGE>=2DAYS?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE,SPACE
_USED,TOP10_TABLE{{SPACE_USED,DPATH,LAST_USE_AGE}}}}[ROWS(INSTANCES)]'
SUMS_TABLE{
|KEY = STORAGE_VOLUME('AZ3:Minio-NFS::/vols/NYC1'),
VALUE = {88 FILES, 68.387 MBYTES, TOP10_TABLE{
|KEY = {16.691 MBYTES, "./Launch00077.log", (6 DAYS+17:44:50)};
|KEY = {1.6425 MBYTES, "./countdown.log", (3 DAYS+06:35:45)};
|KEY = {1.1756 MBYTES, "./Launch00022.log.gz", (9 DAYS+09:23:31)};
|KEY = {1.151 MBYTES, "./Launch00018.log.gz", (2 DAYS+12:16:28)};
|KEY = {1.1469 MBYTES, "./Launch00011.log.gz", (4 DAYS+15:08:08)}}};
```

Three "top" tables are available: **TOP10** (used in the previous example, **TOP100**, and **TOP1000**.

5.6. Options for Processing HSTK Output

While the output from **hs** commands is legible, it would be easier to work with in a spreadsheet. To facilitate this, there are multiple options for formatting the command output.

5.6.1. Formatting Output in JSON

Some hs commands have a **--json / -j** parameter that generates output more easily handled by a script, or imported using common JSON parsing tools.

In the following example, we see the how the output differs with JSON on or off:

```
# hs sum command executed with JSON output on (hs -j)
$ hs -j sum -e
'IS_FILE&&ACCESS_AGE>=2DAYS?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE,SPACE
```

```
_USED,TOP10_TABLE{{SPACE_USED,DPATH}}}}[ROWS(INSTANCES)]'
{"SUMS_TABLE":[{
"KEY":{"HAMMERSCRIPT":"STORAGE_VOLUME('la-a-dsx-1.catalyst.local::/hsvol0')"},
"VALUE":[[{"KFILES":3.56},{"MBYTES":28.34},{"TOP10_TABLE":[{
"KEY":[[{"MBYTES":13.763},"./telemetry_dump3/archive/librocksdbjni7424396042295213658.so"]]},{
"KEY":[[{"KBYTES":4.096},"./mission-6/telemetry.txt"]]},{
"KEY":[[{"KBYTES":4.096},"./mission-6/sensor7_apollo.aaak"]]},{

# hs sum command executed with default output (non-JSON)
$ hs sum -e
'IS_FILE&&ACCESS_AGE>=2DAYS?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE,SPACE
_USED,TOP10_TABLE{{SPACE_USED,DPATH,LAST_USE_AGE}}}}[ROWS(INSTANCES)]'
SUMS_TABLE{
|KEY = STORAGE_VOLUME('la-a-dsx-1.catalyst.local::/hsvol0'),
VALUE = {3.56 KFILES, 28.34 MBYTES, TOP10_TABLE{
|KEY = {13.763 MBYTES, "./telemetry_dump3/archive/librocksdbjni7424396042295213658.so", (3
DAYS+18:38:21)};
|KEY = {4.096 KBYTES, "./mission-6/telemetry.txt", (6 DAYS+17:44:58)};
|KEY = {4.096 KBYTES, "./mission-6/sensor7_apollo.aaak", (6 DAYS+17:44:58)};
|KEY = {4.096 KBYTES, "./mission-6/sensor7_apollo.aaaj", (6 DAYS+17:44:58)};
|KEY = {4.096 KBYTES, "./mission-6/sensor7_apollo.aaac", (6 DAYS+17:44:58)};
```

Hammerspace provides sample scripts in GitHub that run the **hs sum** or **hs usage** command, and then parse the output into an Excel workbook with title, headings, and filters. The following screenshot shows the appearance of the data once parsed and opened in Microsoft Excel.

	A	B	C	D	E
1		**Storage usage report for \\hammer\healthcare**			
2		07/30/2024 22:29:35			
3					
4	User	Group	Storage Volume	File	Bytes
5	root@localdomain	root@localdomain	pl-yvr-dsx1.selab.hammer.space::/hsvol0	2043	30851072
6	root@localdomain	root@localdomain	pl-yvr-dsx2.selab.hammer.space::/hsvol0	2036	303001600
7	root@localdomain	root@localdomain	ntap-sim-7m::/vol/assim_vol_unix/q_uu3	2038	19849216
8	root@localdomain	root@localdomain	sim910::ntap-sim910-svm1:unix1	17	0
9	root@localdomain	root@localdomain	AWS::peter-demo2	3	268509184
10	root@localdomain	root@localdomain	AWS::hspeterbucket1	5	319562400
11	root@localdomain	root@localdomain	AWS::peter3	3	17216496
12	peter@selab.hammer.space	root@localdomain	pl-yvr-dsx2.selab.hammer.space::/hsvol0	1	4096
13	peter@selab.hammer.space	ses@selab.hammer.space	pl-yvr-dsx1.selab.hammer.space::/hsvol0	1	8192
14	peter@selab.hammer.space	ses@selab.hammer.space	pl-yvr-dsx2.selab.hammer.space::/hsvol0	1	4096
15	peter@selab.hammer.space	ses@selab.hammer.space	ntap-sim-7m::/vol/assim_vol_unix/q_uu3	2	32768
16	peter@selab.hammer.space	ses@selab.hammer.space	pl-yvr-dsx1.selab.hammer.space::/hsvol0	1	4096
17	peter@selab.hammer.space	ses@selab.hammer.space	pl-yvr-dsx2.selab.hammer.space::/hsvol0	1	4096
18	gag@selab.hammer.space	root@localdomain	ntap-sim-7m::/vol/assim_vol_unix/q_uu3	1	4096
19	arthur@selab.hammer.space	root@localdomain	AWS::hspeterbucket1	1	4096
20	f2@selab.hammer.space	root@localdomain	AWS::peter3	1	0
21	f2@selab.hammer.space	ses@selab.hammer.space	pl-yvr-dsx1.selab.hammer.space::/hsvol0	1	4096

Figure 1. Formatting output in JSON

The scripts to parse JSON output are provided on a per-HSTK command basis. Customization will be required to use the scripts with other HSTK commands.

5.6.2. Formatting Standard Output

The amount of formatting required to be able to import HSTK output into spreadsheet tools will vary based on the report you are trying to generate.

Hammerspace provides a sample script in GitHub that enables you to run a variety of different reports, and then use the Linux **sed** command to place the data into a csv-formatted output that will be properly parsed when opened.

The following example, generated using the sample script, shows a report that returns the file count and capacity used in bytes for all directories in the current directory and the first level of subdirectories.

Folder	File Count	Capacity Used in Bytes
gfstest	14248	113407712
gfstest/prj_mercury	4289	17559552
gfstest/prj_gemini	5001	20484096
gfstest/prj_apollo	236	970752
gfstest/apollo_telemetry	51	208896
gfstest/mission-6	106	425984
gfstest/mission_cancel	4314	17661952
gfstest/telemetry_dump	58	233472
gfstest/telemetry_dump2	58	233472
gfstest/telemetry_dump3	73	55375584
SnapOnObject	201	104050688
SnapOnObject/folder	89	21237760
SnapOnGenNFS	60	40374272
OnlineDelay	33	192204800
OnlineDelayTier	22	13836288

Figure 2. Formatting standard output

- Depending on your directory names, Hammerspace-provided scripts may occasionally encounter issues parsing the raw HSTK output.
- The Hammerspace sample script includes some of the most common reporting requests. This script will continue to be expanded based on internal and external requests.

Chapter 6. Using the Hammerspace Toolkit

In this section, we will review all of the commands provided by the HSTK, and see common examples of how the commands can be used. The HSTK commands can be arranged into four sections based on their use:

- **Metadata Operations** - Interact with file metadata
- **Reporting** - Obtain detailed statistics and information about the filesystem
- **Filesystem Operations** - Perform a variety of common filesystem operations, but offload the execution to Hammerspace
- **General** - Other tasks that fall outside of the other categories

Keep in mind that this document only covers a small portion of the values you can query using Hammerscript. Consult the **File Metadata Example (Full)** section of this document for an example of the typical metadata generated for every file. Using the examples provided in this document, you can begin to build your own HSTK commands that provide whatever filesystem information that you require.

6.1. Running HS Commands from the Cluster Root Export

Mounting the cluster root export and running **hs** commands from there enables you to gather data on all cluster shares at once. Note that **hs** commands cannot natively traverse from the cluster root into share folders to summarize data. However, there are ways around this behavior that provide us with the simplest method for gathering data on all cluster shares at once.

In this guide, you will see examples where we use the Linux **ls** and **find** commands from the cluster root export to feed paths to **hs** commands, allowing them to directly target each share folder using one command execution. The collected output from these commands, whether viewed raw or exported into spreadsheet tools, enable us to analyze report data for all cluster shares using a single report.

Best Practice:

- If you mount the cluster root export to use the HSTK, for safety reasons, it should be for gathering information only.
- If you need to use the HSTK to make metadata or file changes, you should mount the share directly.

6.2. Commands - Metadata Operations

The most common use case for the HSTK is to query and manipulate file metadata for the purposes of reporting, data orchestration (via objectives), and data management. In this section, we will review the subset of HSTK commands that are used for these tasks. More advanced HSTK examples will be provided in the **Advanced Filesystem Reporting** section of this document.

Most of the Hammerscript examples below can be used for both file system reporting and as conditions for objectives. Consult the {kb-objectives-url}[{kb-objectives-title}] {kb-login-note} for examples about how you use these values as objective conditions.

6.2.1. Tag

The **hs tag** command enables you to view, set and remove Hammerspace custom metadata tags. Tags are a key-value pair, where the value can be specified or excluded. If no value is supplied, the tag will be set to **TRUE**. Tags can have a value added as a string (**-s 'some value'**) or a Hammescript expression (**-e 'some expression'**). In this document, all examples used are strings.

Tags are set using the **hs tag set <tag name> <file or directory name>** command, checked using the **hs tag list <file or directory name>**, and removed with **hs tag delete <tag name> <file or directory name>** as shown in the following examples:

```
# Apply tag Project with value Apollo to file recovery.txt
$ hs tag set Project -s 'Apollo' recovery.txt

# List tags applied to file recovery.txt
$ hs tag list recovery.txt
TAGS_TABLE{
|NAME = "Project",
|VALUE = "Apollo"}
# Remove tag Project from file recovery.txt, you do not need to specify the value
$ hs tag delete Project recovery.txt
```

As stated in the example command, you do not need to specify the tag value when removing tags.

Best Practice: Where possible, implement controls or standards regarding the usage of tags. Given that tags are often targeted by objectives or even HSTK queries, it's important that their spelling, case, and exact terms used are consistent.

You can also set tags using the Hammerspace Metadata Plugin for Microsoft Windows. Consult the **Hammerspace Metadata Plugin for Windows File Explorer** section of this document for information about using that tool to manage file or directory tags.

Using Numbers as Tag Values

In the previous section, we mentioned that tags can have a value, and that value can be set as an expression or a string. It is important to note that expressions include numbers, and there are reasons why you might want to set a numeric value as an expression rather than a string.

The following is an example of how a number set as an expression could be used. In this example, we will add a tag with a value of **3** set as an expression.

```
# Add tag launchphase with an expression of 3 to file launch-meco-tail.jpg
$ hs tag add launchphase -e 3 launch-meco-tail.jpg

# List tags for file launch-meco-tail.jpg
$ hs tag list launch-meco-tail.jpg
TAGS_TABLE{
|NAME = "launchphase",
```

```
|VALUE = 3;
```

Tag values set as strings have limited options as to how they can be evaluated, meaning the tag value either does or does not match the string provided. However, numeric tag values added as expressions can be evaluated using any available operator.

In this example, we see how we can use greater than(**>**) or less than or equal to(⇐) operators to find tagged files whose values meet the condition specified.

Specifically, we are using the label values to find photos (recursively, **-r**) that match certain phases of a rocket launch.

```
# Return files with tag launchphase whose value is greater than 3
$ hs eval -r -e 'GET_TAG("launchphase")>3'
launch-feco-tail.jpg
launch-orbit-side.jpg

# Return files with tag launchphase whose value is less than/equal to 3
$ hs eval -r -e 'GET_TAG("launchphase")<=3'
launch-meco-tail.jpg
launch-burn-tail.jpg
launch-liftoff-tail.jpg
launch-countdown-side.jpg
```

Best Practice: This example shows why it is important to consider not only the format of the tag names, but the values themselves. While the tag values can always be updated, you should consider how you plan to use them when setting the guidelines for tag use.

Integrating Tags into Other Workflows

Hammerspace has published example scripts that set tags on files based on embedded tags extracted from files or from cloud analytics tools, like Amazon Rekognition. Scripts available in GitHub include:

- **exif2hs** - Uses the open-source exiftool to extract standard tags from a wide set of file formats, then sets them as Hammerspace tags.
- **rek2hs** - Obtains the object reference for a file with an instance in an Amazon bucket, passes the object reference to Rekognition, then scrapes the Rekognition label and confidence, and sets them as Hammerspace tags.

6.2.2. Keyword

The **hs keyword** command can be used to apply keywords to files and directories, which allows them to be targeted by objectives or even HSTK queries. Keywords are set using the **hs keyword add <keyword name> <file or directory name>** command, checked using the **hs keyword list, and removed** removed with **hs keyword delete <keyword name> <file or directory name>** as shown in the following examples:

```
# Add keyword FAA to file telemetry.txt
```

```
$ hs keyword add FAA telemetry.txt

# List keywords added to file telemetry.txt
$ hs keyword list telemetry.txt
KEYWORDS_TABLE{
|KEYWORD = "FAA"}

# Remove keyword FAA from file telemetry.txt
$ hs keyword delete FAA telemetry.txt
```

Best Practice: Note that keywords do not use a schema, meaning they are free-form. Where possible, implement controls or standards regarding the usage of keywords. Given that keywords are often targeted by objectives or even HSTK queries, it's important that their spelling, case, and exact terms used are consistent.

You can also set keywords using the Hammerspace Metadata Plugin for Microsoft Windows. Consult the **Hammerspace Metadata Plugin for Windows File Explorer** section of this document for information about using that tool to manage file or directory keywords.

6.2.3. Label

The HSTK can be used to assign existing labels to files and directories as metadata. Labels have an administrator-defined schema that is set up using the Hammerspace CLI or API. Users can only apply labels that are already defined.

This type of metadata only lives within the file system itself, and does not change the contents of the file. The metadata is also replicated when used together with the Global File System feature.

- Labels must be created using the Hammerspace CLI.
- If you use global file shares, you must create the labels using the same letter case and spelling on each site, and verify that the label serial numbers must match across all sites.
- Refer to the {kb-objectives-url}[{kb-objectives-title}] {kb-login-note} for additional examples of how to create and manage labels, including verifying label serial numbers.
- Consult the **Hammerspace Metadata Plugin for Windows File Explorer** section of this document for information about using that tool to manage file and directory labels.

Apply a Label

As with other custom metadata, labels can be applied to a file or directory using the **hs label add** command:

```
# Add label Cape Canaveral to file telemetry.txt
$ hs label add "Cape Canaveral" telemetry.txt
```

Label "Cape Canaveral" was created to have an implied (parent) label of "US", so when we list the labels using **hs label list,** we see that both have been applied:

```
# List labels applied to file telemetry.txt
```

```
$ hs label list telemetry.txt
LABELS_TABLE{
|LABEL_ = LABEL('US'),
|COUNT = 1;
|LABEL_ = LABEL('Cape Canaveral'),
|COUNT = 1}
```

Remove a Label

The **hs label delete** command is used to remove labels. If the label removed has an implied (parent) label, that will also be removed.

```
# Delete the label Cape Canaveral from file telemetry.txt
$ hs label delete 'Cape Canaveral' telemetry.txt

# List labels applied to file telemetry.txt
$ hs label list telemetry.txt
LABELS_TABLE{}
```

6.2.4. Attribute

Attributes have a system-defined hierarchy. In other words, the set of attributes is predefined as part of the Hammerspace product. In this section, we will review some of the most basic uses of attributes. For a more advanced example that better illustrates how they might be used, refer to the **Hammerspace and Data Lifecycle Management** section of this document.

You can list the some of the defined attributes and their values for a file using the **hs attribute list** command:

```
# List attributes for file telemetry.txt
$ hs attribute list telemetry.txt
ITEM_ATTRIBUTES_TYPED_TABLE{
|VIRUS_SCAN = VIRUS_SCAN_STATE('UNSCANNED'),
MIME = MIME_TYPE_TABLE{
|STRING = "/"},
|DO_NOT_MOVE = FALSE,
|CONSIDER_VOLATILE = FALSE,
|RECENTLY_USED_DURATION = 00:05:00,
|SELECTED = FALSE,
|SELECTED2 = FALSE,
|PROMOTE = FALSE,
|DEMOTE = FALSE,
COLORS = COLORS_TABLE{},
CSI_DETAILS = CSI_DETAILS_TABLE{}}
```

While we won't go into all the attributes and their uses in this document, we will use "COLORS" as an example. The COLORS attribute contains a set of colors applied to a file. The allowed list of colors is predefined in the Hammerspace product, similar to how colors are available in Apple MacOS. The **hs attribute set colors**

command is used to set file colors:

```
# Set the color to RED for file telemetry.txt
$ hs attribute set colors -e 'COLORS_TABLE{COLOR_LOOKUP("RED")}' telemetry.txt

# List the attributes for file telemetry.txt
$ hs attribute list telemetry.txt
ITEM_ATTRIBUTES_TYPED_TABLE{
|VIRUS_SCAN = VIRUS_SCAN_STATE('UNSCANNED'),
MIME = MIME_TYPE_TABLE{
|STRING = "/"},
|DO_NOT_MOVE = FALSE,
|CONSIDER_VOLATILE = FALSE,
|RECENTLY_USED_DURATION = 00:05:00,
|SELECTED = FALSE,
|SELECTED2 = FALSE,
|PROMOTE = FALSE,
|DEMOTE = FALSE,
COLORS = COLORS_TABLE{
|COLOR = COLOR_LOOKUP('RED'),
|COUNT = 1},
CSI_DETAILS = CSI_DETAILS_TABLE{}}
```

Consult the **Hammerspace Metadata Plugin for Windows File Explorer** section of this document for information about using that tool to manage the color attribute.

You can set multiple colors by separating the lookups with a semicolon:

```
# Set the color to RED and PURPLE for file telemetry.txt
$ hs attribute set colors -e 'COLORS_TABLE{COLOR_LOOKUP("RED");COLOR_LOOKUP("PURPLE")}' telemetry.txt
```

Once set, you can use the **hs eval** command to query for files with a particular color selected:

```
# List all files that have the color set to GREEN, including path
$ hs eval -r -e 'HAS_COLOR("green")?PATH' .
"./telemetry.txt"
"./images/launch_pad.jpg"
```

6.2.5. Keep-On-Site

One of the default objectives applied to a Hammerspace share keeps files online that have the **keep-on-site** value set for that site. Assuming that the default objective is in place (or one similar to it), setting this value will keep the file (or contents of the directory) online using nothing more than the default share objectives.

When preparing to set this value, first list the sites using the **hs keep-on-site available** command:

```
# List all sites where the current share is available
$ hs keep-on-site available
NYC-HS-A
LA-HS-A
```

In this example, we see that the share is available on two sites - **NYC and LA**.

If the data is in a share that is not a global file share, only the local cluster name will be returned.

The value can then be set with the Hammerspace toolkit using the **hs keep-on-site** command specifying **add** or **delete** as needed:

```
# Add the keep-on-site designation for site NYC-HS-A to the current directory
$ hs keep-on-site add NYC-HS-A .
# Remove the keep-on-site designation for site NYC-HS-A to the current directory
$ hs keep-on-site delete NYC-HS-A .
```

In the previous example, we targeted the current directory using a period (.) - you can also specify a directory or file name.

Consult the **Hammerspace Administrators Guide** and {kb-objectives-url}[{kb-objectives-title}] {kb-login-note} for additional information about the default share objectives, and the keep-on-site feature.

6.2.6. Rekognition-Tag

The **hs rekognition-tag** command is also used to set file and directory tags, and follows the same format as the hs tag command. Consult the **tag** section of this document for the command syntax and examples.

6.3. Reporting

The HSTK and Hammerscript can be used for a wide variety of filesystem reporting needs, at whatever level of granularity that you require. In this section, we will review the subset of HSTK commands that are used for file system reporting.

More advanced examples are available in the **Advanced Filesystem Reporting** section of this document.

6.3.1. Frequently Used Metadata Values

In this section, we will outline some of the most commonly used metadata values within Hammerscript expressions, be it for objectives or while using the HSTK.

Many of the Hammerscript examples in this document leverage values such as **SPACE_USED** and **LAST_USE_AGE**. The former is self explanatory, while the latter represents the last time a file is closed and is the preferred value for measuring when a file was last used.

Additional metadata values are detailed in the Appendix A section **Additional Metadata Objects And Functions**. For a full list of metadata objects Hammerspace shares, refer to the **File Metadata Example (Full)** section of this document.

Timestamps

- **CREATE_AGE** = (3 DAYS+17:02:20) - When a file was created
- **CHANGE_AGE** = 2.50547 SECONDS - When a file was last modified, this timestamp also resets in response to a change in file permissions
- **ACCESS_AGE** = 38.5719 SECONDS - When a file was last opened; note that clients may cache reads which can cause this value to be out of date
- **MODIFY_AGE** = 38.5101 SECONDS - When a file was last modified, note that this timestamp does not reset if the file contents are not modified

File Details

- **NAME** = "telemetry.txt" - The file name
- **PATH** = "./mission-6/telemetry.txt" - The full path to the file, starts at the share root
- **OWNER** = USER('jason@catalyst.local') - The object user owner
- **OWNER_GROUP** = GROUP('Apollo_PrimeCrew@catalyst.local') - The object group owner
- **PARENT_SHARE** = SHARE('gfstest') - The share the file is in
- **SPACE_USED** = 4.096 KBYTES - The space used on disk, note that this may vary from SIZE as the default allocation unit is 4KB
- **SIZE** = 73 BYTES - The file size as reported to the client, note that this does not reflect the actual space used on the volumes, nor the total space used by all file instances
- **IS_FILE** = TRUE - TRUE indicates the object is a file, FALSE indicates it is either a symlink or directory
- **IS_SYMLINK** = FALSE - TRUE indicates the object is a symlink, FALSE indicates it is either a file or a directory
- **IS_DIRECTORY** = FALSE - TRUE indicates the object is a directory, FALSE indicates it is either a file or symlink

Tags, Labels, and Keywords are also commonly used values, though the expressions used to match them are unique. Consult the **Commands - Metadata Operations** section of this document for an overview of how to write expressions that target these values.

File Status

- **OVERALL_ALIGNMENT** = ALIGNMENT('ALIGNED') - The file alignment status
- **IS_ONLINE** = TRUE - TRUE indicates the file is available in a local online volume, FALSE indicates it is located on offline (object) storage or must be requested from another site
- **IS_OFFLINE** = FALSE - TRUE indicates the file is not available on a local online volume, FALSE indicates it is located in a local online volume

- **DATA_ORIGIN_LOCAL** = TRUE - TRUE indicates the current site was the last to edit the file and would be considered the owner, FALSE indicates another site is currently the owner
- **DATA_ORIGIN_REMOTE** = FALSE - FALSE indicates the current site was the last to edit the file and would be considered the owner, TRUE indicates another site is currently the owner
- **ORIGIN_SITE** = SITE('NYC-HS-A') - Indicates which site currently owns the file, meaning the file was last written to there

Directories and 0 byte files are stored entirely in metadata, which will cause them to always show as online.

6.3.2. Eval

The **hs eval** command allows us to evaluate Hammerscript expressions on a file, or simply return a specified metadata value. This can be used for a wide range of purposes such as reporting, testing objective conditions, or advanced searches.

Eval commands are not recursive by default. They will be performed on the current directory OR the path if one is provided. To make the command recursive, add the **-r** switch.

Best Practice: If you are looking for output suitable for later analysis in spreadsheet tools, **hs eval** is likely to require the least amount of effort to parse. That said, **hs eval** does not summarize output, but rather displays results for each file individually, so you may find that **hs sum** (described in the next section) is a more suitable command for some use cases. The same queries will often work for both commands, though it's important to note that **hs sum** is recursive, so it is easy enough to try both and compare the output.

HS Eval Command Examples

In this section we will review some common hs eval commands. More examples are provided in the **Advanced Filesystem Reporting** section of this document.

List All File Instance Volume Locations for the Specified File

The following example command will list all cluster volumes that have instances of a file (**telemetry.txt**) using the default **instances.volume** expression:

```
# List all volumes with instances of the telemetry.txt file.
$ hs eval -e instances.volume telemetry.txt
INSTANCES_TABLE{
|VOLUME = STORAGE_VOLUME('Minio-NFS::/vols/NYC1');
|VOLUME = STORAGE_VOLUME('Minio-S3::gfs')}
```

Display All Alignment Details for a File

The following example command will provide a detailed output of the alignment status for a file (**telemetry.txt**) using the default **alignment_details** expression (partial output shown):

```
# List the file alignment details for the telemetry.txt file
```

```
$ hs eval -e alignment_details telemetry.txt
ALIGNMENT_DETAILS_TABLE{
|CHANGE_TIME = LOCAL_TIME('1969-12-31 19:00:00'),
|LAST_CHECKED_TIME = LOCAL_TIME('1969-12-31 19:00:00'),
|OVERALL = ALIGNMENT('PARTIALLY ALIGNED'),
|PLACE_ON = ALIGNMENT('PARTIALLY ALIGNED'),
|EXCLUDE_FROM = ALIGNMENT('ALIGNED'),
|CONFINE_TO = ALIGNMENT('ALIGNED'),
```

Note that this file is not fully aligned. Based on the output, it would seem that all applicable place-on objectives have not yet been met.

Display the Site That Currently Owns a File

Hammerspace global file systems use a term called **owner** to denote which site was the last one to write to a file. While every participant site has the ability to keep an online instance of the latest version of the file, only one of those sites is designated as an owner.

Features such as the optional ***log-xfer*** objective, referred to as ***File conflict resolution*** in the Hammerspace share GUI, use the owner value to determine if a file version needs to be created when site ownership changes.

THe following example command queries the metadata value **DATA_ORIGIN_SITE.SITE_NAME** to return the current owning site.

```
# Display the current origin site for file telemetry.txt
$ hs eval -e 'DATA_ORIGIN_SITE.SITE_NAME' telemetry.txt
"NYC-HS-A"
```

In the **Total Number of Files Owned By Each Global File Share Site** section of this guide, we will show how to summarize ownership details for a collection of files.

6.3.3. Sum

The **hs sum** command is used to perform fast calculations on a set of files. Typically, a Hammerscript expression used with **hs eval** can be used with **hs sum**, but note that **hs sum** is recursive while hs eval is not (unless specified).

The **hs eval** command differs from **hs sum** in that you can sum and sort data using the HSTK itself, rather than needing to import it into spreadsheet tools to perform that task. Plus, **hs sum** also grants you access to built in reports for the top 10, 100, or 1000 examples for supported metadata objects.

- Summaries will be performed recursively starting at the specified directory.
- Summaries will be performed starting at the current directory OR the path if one is provided.

HS Sum Command Examples

In this section we will review some common hs sum commands. More examples are provided in the **Advanced Filesystem Reporting** section of this document.

Total Number of Files and Space Used - Specified Directory

The following command will return the file count and total amount of space used (**\{1FILE/FILE,SPACE_USED/BYTES}**) by all files within the specified directory (**telemetry_dump**).

```
# Return the file count and space used for all files in the telemetry_dump directory
$ hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES}' telemetry_dump
{4313, 17661952}
```

Commands like this can also use wildcards as a target (*). Note that if there were files and directories within the directory where the command was executed, both would be returned as shown in the following example where **apollo_telemetry**, **telemetry_dump2**, and **telemetry_dump3** are directories.

```
# Return the file count and space used for all files and directories within the current directory
$ hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES}' *
##### apollo_telemetry
{51, 208896}
##### data_gemini.aaaa
{1, 4096}
##### data_gemini.aaab
{1, 4096}
##### telemetry_dump2
{5001, 20484096}
##### telemetry_dump3
{235, 962560}
```

In the previous example, we see output for both files and directories. Files are identified by the first value being a **1** (for 1 file), while directories that contain more than 1 file would return the count of files within and their total space used.

Best Practice - Leveraging Linux to Enhance Your HSTK Commands

You can use ordinary Linux commands to narrow the scope of some commands, like in this case with directories. The following examples will reuse the command we learned in the previous section, but you will see how easy it is to customize how the HSTK command is executed.

The following example uses a Linux **ls -d** command to list only directories, and then pipes that output to the **hs sum** command:

```
# Use ls -d */ | xargs to pipe directory names to HSTK for execution
$ ls -d */ | xargs hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES}'
##### apollo_telemetry
```

```
{51, 208896}
##### mission-6
{106, 425984}
##### mission_cancel
{4314, 17661952}
##### prj_apollo
{196, 42594}
. . .
```

Note our output now excludes files, so there is less cleanup you would need to do before analyzing this data further. This technique can be applied to similar situations where your HSTK target is directories, not files.

In this example, a recursive operation is enabled. The command will again only execute against directories, but we are using a Linux **find . -type d** command to do a recursive search for directories. The resulting output is again piped to the same **hs sum** command, and the output now includes subdirectories:

```
# Execute the hs sum command shown for all folders within the current path
$ find . -type d | xargs hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES}'
##### .
{14247, 113399520}
##### prj_apollo
{235, 962560}
##### mission-6
{106, 425984}
##### mission-6/launch_details
{53, 212992}
##### telemetry_dump3
{73, 55375584}
##### telemetry_dump3/archive
{15, 55142112}
```

While running this report against a very large directory report will take some time, it provides a very deep level of insight into where your files are located.

Best Practice: Use a recursion limit to control how deep into the directory tree the find command will traverse. The following example limits find to 3 levels (level 1 is the current directory).

```
# Set Linux find command to a max depth of 3
$ find . -maxdepth 3 -type d | xargs hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES}'
```

Combining Linux tools with HSTK commands provides you with the maximum flexibility when it comes to performing tasks or gathering the data that you need.

Total Number of Files and Space Used - Optional Top 10 Table

The following example would return the total number of files (not directories, symlinks, or devices) and space used. The second command also includes a top 10 table, while the third includes only the table:

```
# Return the file count and space used for the contents of the current directory
$ hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES}'
{9711, 39772160}

# Return the file count and space used for the contents of the current directory, plus a top 10 file list
$ hs sum -e 'IS_FILE?{1FILE/FILE,SPACE_USED/BYTES,TOP10_TABLE{{SPACE_USED/BYTES,PATH}}}'
{9711, 39772160, TOP10_TABLE{
|KEY = {12288, "./.DS_Store"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiz"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiy"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaix"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiw"};
. . .

# Return the space used and file name with path for the top 10 files under the current directory
$ hs sum -e 'TOP10_TABLE{{SPACE_USED/BYTES,PATH}}'
TOP10_TABLE{
|KEY = {12288, "./.DS_Store"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiz"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiy"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaix"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiw"};
. . .
```

- Remember that a file can have multiple instances. A report like this would only count the number of files and not the actual number of file instances written to cluster volumes.
- A csv-formatted version of this report can be generated using the HSTK_Report_Builder.sh script.

Number of Files with the Specified Alignment Status

The following provides two different examples of commands that output stats on file alignment. The first example shows the total number of files that are aligned (**overall_alignment=ALIGNMENT("ALIGNED")**), while the second displays the count of files that are in any state other than aligned (**overall_alignment!=ALIGNMENT("ALIGNED")**).

```
# Return the count of aligned files for the content of the current directory
$ hs sum -e 'IS_FILE&&OVERALL_ALIGNMENT=ALIGNMENT("ALIGNED")' .
14247
0

# Display the count of files that are not in the aligned state (an ! was added)
$ hs sum -e 'IS_FILE&&OVERALL_ALIGNMENT!=ALIGNMENT("ALIGNED")' .
53
```

The possible alignment statuses include:

- **ALIGNED:** file instances are all where they should be
- **MISALIGNED:** all file instances are not where they should be
- **PARTIALLY ALIGNED:** some file instances are where they should be

If you want a count of files per alignment status, you can simply use the default **hs collsum** command discussed in the next section.

File Instance Count and Space Used - per Volume

The following **hs sum** command will return a per cluster volume (**KEY=INSTANCES[ROW].VOLUME**) report with file instance count and space used (**VALUE=\{1FILE/FILE,SPACE_USED/BYTES}**):

```
# Return the file instance count and space used in bytes for each cluster volume
$ hs sum -e
'IS_FILE?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE/FILE,SPACE_USED/BYTES}}[
ROWS(INSTANCES)]'
SUMS_TABLE{
|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol0'),
|VALUE = {2425, 9932800};

|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol1'),
|VALUE = {2431, 9957376};

|KEY = STORAGE_VOLUME('NYC-dsx-2.catalyst.local::/hsvol0'),
|VALUE = {2429, 9957376};

|KEY = STORAGE_VOLUME('NYC-dsx-2.catalyst.local::/hsvol1'),
|VALUE = {2423, 9924608};

|KEY = STORAGE_VOLUME('Bucket-NYC'),
|VALUE = {9708, 39772160}}
```

- A csv-formatted version of this report can be generated using the HSTK_Report_Builder.sh script; the script can generate this information for all shares.
- The previous example standardizes some of the output units (**/BYTES** and **/FILE**) using techniques described previously in this document. Remember, this is optional. If you omit those, the most appropriate unit for the value will be used (**1100** files would be labeled **1.1 KFILES** and **39772160** bytes would be labeled **39.772 MBYTES)**. As stated previously, this makes importing data for later analysis more difficult as the units could vary from one output line to the next.

You can return more values in the output by adding more metadata values within the brackets of the VALUE portion of the command (**VALUE=\{1FILE/FILE,SPACE_USED/BYTES}**); the following example would include a count of files that are considered a threat (**IS_THREAT**) by an external ICAP antivirus scan:

```
# Return the file instance count, count of files marked as a threat, and space used in bytes for each
cluster volume
```

```
$ hs sum -e
'IS_FILE?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE,IS_THREAT,SPACE_USED}}[R
OWS(INSTANCES)]'

SUMS_TABLE{
|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol0'),
|VALUE = {2.425 KFILES, 35, 9.9328 MBYTES};
. . .
```

The data is outputted in the order of the fields you provided. In this example, the IS_THREAT value is **35**.

File Instance Count, Total Space Used, and 10 Largest Files - per Volume

If you want the same report used in a previous example (file count and space used per volume), but also need a top 10 table detailing space used and file name and path (**TOP10_TABLE\{\{SPACE_USED/BYTES,PATH}}**) for each of the volumes, you can easily expand the command:

```
# Return the file instance count and space used in bytes for each cluster volume
$ hs sum -e
'IS_FILE?ROWS(INSTANCES)?SUMS_TABLE{|::KEY=INSTANCES[ROW].VOLUME,|::VALUE={1FILE/FILE,SPACE_USED/BYTES,TO
P10_TABLE{{SPACE_USED/BYTES,PATH}}}}[ROWS(INSTANCES)]'
SUMS_TABLE{
|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol0'),
VALUE = {2425, 9932800, TOP10_TABLE{
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiy"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaiw"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaip"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aail"};
|KEY = {4096, "./telemetry_dump3/telemetry_apollo.aaih"};
. . .
```

You can also use **TOP100** or **TOP1000** if you need a longer list of files.

Total Number of Files Owned by Each Global File Share Site

The **Display The Site That Currently Owns A File** section of this guide showed how to use **hs eval** to query a file to see which site currently owns it. In this example, we will summarize the count of files owned by each site in the global file share.

The following command will summarize the data for the current share recursively from the current folder:

```
# Return number of files owned for each site that is a member of the current global file share
$ hs sum -e
'IS_FILE?ROWS(DATA_ORIGIN_SITE.SITE_NAME)?SUMS_TABLE{|::KEY=DATA_ORIGIN_SITE[ROW].SITE_NAME,|::VALUE={1FI
LE/FILE}}[ROWS(SITE_NAME)]' .
SUMS_TABLE{
|KEY = "NYC-HS-A",
```

```
|VALUE = {34453};

|KEY = "LA-HS-A",
|VALUE = {14245}}
```

It is important to note that ownership can change at any moment. All that is required is for some other site to edit the file (opening a file does not change the owner). So while this report will give you some insight into where the latest versions of your files currently live, know that those stats can change frequently.

6.3.4. Collsum

The **hs collsum** command provides usage details about one or more of the default share collation summaries. While it is possible to create an unlimited number of custom summaries using **hs eval** or **hs sum**, the default collsums are automatically generated and cover many of the most common storage reports.

The following command leverages grep to obtain the standard list of collsums. A description of each is provided to the right of the name:

```
# List all standard collsums
$ hs collsum | grep SUMMATION
SUMMATION('BASIC'), {9.662 KFILES, 39.551 MBYTES, 0 OPERATIONS /
SECOND}; #Standard filesystem stats
SUMMATION('BY-HEAT'), { #File by access age and IOPS
SUMMATION('BY-ACCESS-AGE'), { #By access time
SUMMATION('BY-MODIFY-AGE'), { #By modify time
SUMMATION('BY-CHANGE-AGE'), { #By change time
SUMMATION('BY-CREATE-AGE'), { #By create time
SUMMATION('BY-SPACE-USED'), { #By space used
SUMMATION('BY-ACTUAL-MODIFY-AGE'), { #By modify time
SUMMATION('BY-VOLUME'), { #By instance location
SUMMATION('BY-ACTIVE-OBJECTIVE'), { #By active objective
SUMMATION('BY-ALIGNMENT'), INDEXED_TABLE{ #By alignment type
SUMMATION('BY-ALIGNMENT-STATE'), { #By alignment state
SUMMATION('BY-ERRORS'), { #By active error
SUMMATION('BY-TYPE'), { #File and directory count
SUMMATION('BY-VERSION'), { #Per snapshot version count
SUMMATION('BY-MIME'), {"/", {9.662 KFILES, 39.551 MBYTES, 0 OPERATIONS
/ SECOND}}; #Standard fs stats by mime type
SUMMATION('BY-VIRUS-SCAN'), {VIRUS_SCAN_STATE('UNSCANNED'), {9.662
KFILES, 39.551 MBYTES, 0 OPERATIONS / SECOND}}; #By scan state
SUMMATION('TOP-FILES'), TOP1000_TABLE{ #Top 100 files by size
```

Collsums are continually updated, and are the fastest method to obtain filesystem statistics. They will always return data for the entire share, regardless of where the command was actually executed from within the share.

Many collsums provide data similar to what you find in the Hammerspace GUI per-share dashboards, though if you are looking to prepare your own custom reports, the HSTK is the preferred method for obtaining this information.

Best Practice: Exploring the output of the various collsums is an easy way to learn about the different metadata values that you might want to use to create your own custom reports.

HS Collsum Examples

In this section we will review some of the available collsums. Many are self-explanatory, such as those based on space used or timestamps. If you are curious about ones not mentioned here, simply execute the command and observe what information is returned. If you execute just the **hs collsum** command, all summations for the share will be returned.

Basic

The **basic** collsum returns file instance count, total space used, and the IOPS.

```
# Use collsum to list the file instance count, total space used, and total IOPS
$ hs collsum --collation BASIC
INDEXED_TABLE{SUMMATION('BASIC'), {9.662 KFILES, 39.551 MBYTES, 0 OPERATIONS / SECOND}}
```

By-Alignment-State

The **by-alignment-state** collsum shows the number of and space consumed by files, broken down according to alignment. In the following example, you will see that a large number of files are waiting for a volume with sufficient space to place the files.

```
# Return the by-alignment-state collsum data
$ hs collsum --collation BY-ALIGNMENT-STATE
INDEXED_TABLE{SUMMATION('BY-ALIGNMENT-STATE'), {
ALIGNMENT_STATE('UNKNOWN'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
ALIGNMENT_STATE('ALIGNED'), {7 FILES, 4.096 KBYTES, 0 OPERATIONS / SECOND};
ALIGNMENT_STATE('WAITING_FOR_MOBILITY'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
ALIGNMENT_STATE('WAITING_FOR_TARGET_WITH_SPACE'), {9.655 KFILES, 39.547 MBYTES, 0 OPERATIONS / SECOND}}}
```

In this example, the Hammerspace cluster has a volume with available space to write files to, but it is not the volume that meets the share objective requirements. Rather than prevent the client from writing data, the cluster places files wherever it can, marks them as unaligned (and why), and will monitor the cluster to see if space eventually becomes available on a volume that meets the objective requirements.

By-Modify-Age

The following is a partial output of the **by-modify-age** collsum command. As with similar collsums, the full output extends as far as needed to break apart the values into smaller groups.

```
# Return the by-modify-age collsum data
$ hs collsum --collation BY-MODIFY-AGE
INDEXED_TABLE{SUMMATION('BY-MODIFY-AGE'), {
TIMESPAN_LEVEL('UNDER 1 MINUTE OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
```

```
TIMESPAN_LEVEL('1 TO 5 MINUTES OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
TIMESPAN_LEVEL('5 TO 15 MINUTES OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
TIMESPAN_LEVEL('15 TO 30 MINUTES OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
TIMESPAN_LEVEL('30 MINUTES TO 1 HOUR OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
TIMESPAN_LEVEL('1 TO 3 HOURS OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
TIMESPAN_LEVEL('3 TO 6 HOURS OLD'), {0 FILES, 0 BYTES, 0 OPERATIONS / SECOND};
```

Local-Objectives

The **local-objectives** collsum will return up to 1000 files that have objectives applied directly to them. Objectives applied in this way may be overriding those applied at the share root, so it is important to review where this has been done from time to time.

In the following example, we can see that the final line of the output starts the list of files that have objectives locally applied. Only 1 file is impacted in this example (**log_mercury.aaab**):

```
# Return the local-objectives collsum data
$ hs collsum local-objectives
INDEXED_TABLE{
SUMMATION('BASIC'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS / SECOND};
SUMMATION('BY-HEAT'), {TEMPERATURE_LEVEL('1 TO 2 WEEKS OLD'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS /
SECOND}};
SUMMATION('BY-ACCESS-AGE'), {TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS /
SECOND}};
SUMMATION('BY-MODIFY-AGE'), {TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS /
SECOND}};
SUMMATION('BY-CHANGE-AGE'), {TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS /
SECOND}};
SUMMATION('BY-CREATE-AGE'), {TIMESPAN_LEVEL('1 TO 2 WEEKS OLD'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS /
SECOND}};
SUMMATION('BY-SPACE-USED'), {SIZE_LEVEL('4 TO 32 KBYTES'), {1 FILE, 4.096 KBYTES, 0 OPERATIONS /
SECOND}};
SUMMATION('TOP-FILES'), TOP1000_TABLE{|KEY = {4.096 KBYTES, "./log_mercury.aaab"}}}
```

6.3.5. Usage

The **hs usage** command shows information about file instance count, capacity consumption, volume usage, and more about files in a given path.

```
# Help page for hs usage
$ hs usage --help
Usage: hs usage [OPTIONS] COMMAND [ARGS]...

Options:
--help Show this message and exit.

Commands:
alignment Alignment state of files each file(s) of files in dir(s)
```

```
virus-scan Virus scan state of files each file(s) of files in dir(s)
owner Owner state of files each file(s) of files in dir(s)
online Summary of files on NAS volumes in the dir
volume Usage for each volume backing each dir(s)
user Users consuming the most capacity in each dir(s)
objectives Objectives applied and capacity managed by dir(s)
mime_tags All tags added by mime discovery on dir(s)
rekognition_tags All tags added by Rekognition on dir(s)
dirs Number of subdirectories under specified directory(ies), not including that directory
```

HS Usage Examples

In this section we will review some of the **hs usage** command options. Many are self explanatory.

If you are curious about ones not mentioned here, just execute the command and observe what information is returned.

The **hs usage** command returns results based on where it was executed and is recursive. As such, it is a useful tool to get quick summary reports of the values it supports.

Volume

The **hs usage volume** command gives the file instance count and space usage of the specified path (in this case the current directory) on each volume used:

```
# List file instances per cluster volume for the current directory/subdirectories
$ hs usage volume
SUMS_TABLE{
|KEY = #EMPTY,
|VALUE = 3;
|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol0'),
|VALUE = 2425;
|KEY = STORAGE_VOLUME('NYC-dsx-1.catalyst.local::/hsvol1'),
|VALUE = 2431;
|KEY = STORAGE_VOLUME('NYC-dsx-2.catalyst.local::/hsvol0'),
|VALUE = 2429;
|KEY = STORAGE_VOLUME('NYC-dsx-2.catalyst.local::/hsvol1'),
|VALUE = 2423;
|KEY = STORAGE_VOLUME('Bucket-NYC'),
|VALUE = 9708}
```

In the above example, there are 3 files with an **#EMPTY** storage volume. These files have instances on another site in a global file system. The local site still knows about them, they are visible in the share, and users can still access them. Once a user attempts to open one of these files, a site to site mobility will occur and the files will now have a local instance.

Always remember that the number of instances is not necessarily equal to the number of files. If your objectives cause multiple instances of a file to be created for data protection or other reasons, then each

instance would show up in this report.

Owner

The **hs usage owner** command gives the file count by file owner within the specified path (in this case the current directory):

```
# List file count by file owner by user for the current directory/subdirectories
$ hs usage owner
SUMS_TABLE{
|KEY = USER('neil@catalyst.local'),
|VALUE = 9709;

|KEY = USER('buzz@catalyst.local'),
|VALUE = 2}
```

Alignment

The **hs usage alignment** command gives the file count by alignment state within the specified path (in this case, the current directory):

```
# List file count by alignment status for the current directory/subdirectories
$ hs usage alignment
SUMS_TABLE{
|KEY = ALIGNMENT('ALIGNED'),
|VALUE = 9711}
```

6.3.6. Perf

The **hs perf** command provides performance and operation stats on a per-share basis, and is out of scope for this document. Hammerspace recommends using the integrated Prometheus exporters to collect cluster performance statistics and display them using the custom Hammerspace dashboards for Grafana.

For information regarding the use of Grafana and Prometheus to gather Hammerspace cluster performance data, see *Monitoring Cluster Health* in the Hammerspace Administration Guide.

6.4. Advanced Filesystem Reporting

One of the most common reporting requests is for information regarding file ownership and capacity utilization. In this section, we will review some examples of HSTK commands used to create these reports.

When determining how to build your reports, you may wonder which of the commands that we have discussed is the best option.

For example, when using the same expression, the following commands will return the same data, but in two different formats:

- The **hs eval** command will return the path, name, and size of all files that meet the condition specified. The command description will explain exactly what the condition text is looking for.
- The **hs sum** command uses the same expression as hs-eval (though the output fields are modified: **\{1FILE/FILES,SIZE/BYTES}**), but instead returns a summary of the same information.

In the next section, we will review the results for various expressions. The only difference will be which command was used to perform the query.

6.4.1. Choosing Between HS Eval and HS Sum

The following three examples show the same Hammerscript expression being used with **hs eval** and then **hs sum**. When choosing which command, know that it will ultimately depend on the question you are trying to answer. That being said, it is usually quite easy to insert your expression into either one of those commands. While only **hs sum** can build the TOP tables, **hs eval** can do thorough reporting, and then analyze the data how you see fit using any number of tools.

The following examples show the difference in output between **hs eval** and **hs sum** for three different expressions.

```
# Return file name with path and size in bytes for all files less than 10kB
$ hs eval -r -e 'IS_FILE?SIZE<10KBYTES?{PATH,SIZE/BYTES}' .
{"./telemetry.txt", 535}
{"./log_mercury.aaaa", 1024}
{"./log_mercury.aaab", 1024}
{"./log_mercury.aaac", 1024}
{"./log_mercury.aaad", 1024}
. . .

# Return file count and total space used in bytes for all files less than 10kB
$ hs sum -e 'IS_FILE?SIZE<10KBYTES?{1FILE/FILES,SIZE/BYTES}' .
{4289, 4386138}

# Return file name with path and size in bytes for all files with the extension iso; update the value as
needed to match other file extensions or names
$ hs eval -r -e 'IS_FILE?FNMATCH("*.iso",NAME)?{PATH,SIZE/BYTES}' .
{"./Rocky_v9.iso", 20}
{"./telemetry_dump/Boot_Recovery.iso", 15}
. . .

# Return file count and total space used in bytes for all files with the extension iso; update the value
as needed to match other file extensions or names
$ hs sum -e 'IS_FILE?FNMATCH("*.iso",NAME)?{1FILE/FILES,SIZE/BYTES}' .
{2, 35}

# Return file name with path and size in bytes for all files contained within directories whose name
matches tele* (case sensitive)
$ hs eval -r -e 'IS_FILE?FNMATCH("*/tele*/*",PATH)?{PATH,SIZE/BYTES}' . | more
{"./telemetry_dump3/telemetry_apollo.aaaa", 1024}
{"./telemetry_dump6/telemetry_mercury.aaab", 1024}
```

```
{"./telemetry_dump11/telemetry_gemini.aaac", 1024}
. . .

# Return file count and total space used in bytes for all files contained within directories whose name
matches tele* (case sensitive)
$ hs sum -e 'IS_FILE?FNMATCH("*/tele*/*",PATH)?{1FILES/FILE,SIZE/BYTES}' .
{9525, 9743416}
```

Commands similar to this would make it easy to find those files which are consuming the most space.

Remember that **hs eval** is not recursive by default. You must add the **-r** switch to make it scan all files and directories within your current path. The **hs sum** command is recursive, so the **-r** switch is not required.

6.4.2. User Quota Reports

Hammerspace does not implement user or group quotas, though you can set a size limit on each share which functions as a share quota. However, Hammerspace does provide the ability to generate sophisticated usage reports as you might expect from a quota tool or system.

For these examples, we are mounted to a Hammerspace share, and the Python virtual environment is in the command search path. The commands work identically on Windows and Linux.

File Count and Total Space Used by File Owner

The command in the following example creates a simple quota report for all users with files under the current directory.

Note that unit conversions were not specified for files and space used, so the output for files and capacity use a variety of units (**FILES** vs **KFILES**, **BYTES** vs **KBYTES** vs **GBYTES**).

- The file owner is not necessarily the last person to edit a file.
- To standardize units for this command, you would replace **1FILES** with **1FILES/FILE**, and **SPACE_USED** with **SPACE_USED/BYTES**.

```
# Return file count and total space used by file owner
$ hs sum -e 'IS_FILE?SUMS_TABLE{|KEY={OWNER},|VALUE={1FILE,SPACE_USED}}'
SUMS_TABLE{
|KEY = {USER('neil@catalyst.local')},
|VALUE = {10.285 KFILES, 13.38 KBYTES};

|KEY = {USER('buzz@catalyst.local')},
|VALUE = {14.245 KFILES, 163.38 GBYTES};

|KEY = {USER('michael@catalyst.local')},
|VALUE = {9.115 KFILES, 113.38 KBYTES};
```

File Count and Total Space Used by File Owner per Volume

The following more sophisticated quota report breaks down each user's (based on file owner) usage under the current directory, broken down by storage volume. This includes volumes on DSX and NFS storage, as well as those located on object and cloud storage.

```
# Return the file count and total space used on a per-volume basis for each user
$ hs sum -e 'IS_FILE?SUMS_TABLE{|::KEY={OWNER,
OWNER_GROUP,INSTANCES[PARENT.ROW].VOLUME},|::VALUE={1FILE,SPACE_USED}}[ROWS(INSTANCES)]' .
SUMS_TABLE{
|KEY = {USER('neil@catalyst.local|neil@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('missionhq-dsx1.catalyst.local::/hsvol0')},
|VALUE = {13 FILES, 276.36 MBYTES};

|KEY = {USER('neil@catalyst.local|neil@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('missionhq-dsx2.catalyst.local::/hsvol0')},
|VALUE = {12 FILES, 287.88 MBYTES};

|KEY = {USER('neil@catalyst.local|neil@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('ntap-sim-7m::/vol/assim_vol_unix/q_uu3')},
|VALUE = {12 FILES, 12.792 MBYTES};

|KEY = {USER('neil@catalyst.local|neil@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('AWS::peter-demo2')},
|VALUE = {2 FILES, 268.51 MBYTES};

|KEY = {USER('neil@catalyst.local|neil@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('AWS::hspeterbucket1')},
|VALUE = {5 FILES, 319.56 MBYTES};

|KEY = {USER('neil@catalyst.local|neil@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('AWS::peter3')},
|VALUE = {3 FILES, 17.216 MBYTES};

|KEY = {USER('peter@catalyst.local|peter@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('missionhq-dsx2.catalyst.local::/hsvol0')},
|VALUE = {1 FILE, 4.096 KBYTES};

|KEY = {USER('ford@catalyst.local|ford@catalyst.local'), GROUP('Apollo_Astronauts@catalyst.local'),
STORAGE_VOLUME('ntap-sim-7m::/vol/assim_vol_unix/q_uu3')},
|VALUE = {1 FILE, 0 BYTES};

|KEY = {USER('ford@catalyst.local|ford@catalyst.local'), GROUP('nerds@catalyst.local'),
STORAGE_VOLUME('missionhq-dsx1.catalyst.local::/hsvol0')},
|VALUE = {1 FILE, 4.096 KBYTES};
```

6.4.3. General File and Directory Reporting

In this section, we will review a variety of reports used to report on file data for the entire cluster, find specific

file types, or files that meet certain characteristics.

Top Files - All Cluster Shares

The following command, designed to be executed from the Hammerspace cluster root export, leverages a Linux **ls** command to pipe a list of share directories to a **hs sum** command.

This command returns the Top 10 files with for every share, their parent share, and the file size in bytes (**\{PARENT_SHARE,SPACE_USED/BYTES,PATH}**).

A csv-formatted version of this report can be generated using the HSTK_Report_Builder.sh script.

```
# Return the top 10 table by file size for all folders under the current path
$ ls -d */ | xargs hs sum -e 'TOP10_TABLE{{PARENT_SHARE,SPACE_USED/BYTES,PATH}}'
##### gfstest
TOP10_TABLE{
|KEY = {SHARE('gfstest'), 13783920, "./telemetry_dump3/archive/librocksdbjni5683855077670294 94.so"};
|KEY = {SHARE('gfstest'), 13783920, "./telemetry_dump3/archive/librocksdbjni1540166046508206384.so"};
|KEY = {SHARE('gfstest'), 13762560, "./telemetry_dump3/archive/librocksdbjni8354332257291138511.so"};
. . .
##### OnlineDelay
TOP10_TABLE{
|KEY = {SHARE('OnlineDelay'), 13721600, "./more.file7"};
|KEY = {SHARE('OnlineDelay'), 13721600, "./more.file6"};
|KEY = {SHARE('OnlineDelay'), 13721600, "./more.file5"};
. . .
```

File Count and Total Space Used by Directory

In this command, useful for quickly assessing a collection of user home directories, we will use **find . -type d** command to get a list of directories, and then pipe that list into a **xargs hs sum** command to obtain the file count and total space used for each directory (**\{,1FILES/FILE,SPACE_USED/BYTES,}**), including the current directory.

```
# Return the path, file count, and space used for every directory recursively
$ find . -type d | xargs hs sum -e 'IS_FILE?{,1FILES/FILE,SPACE_USED/BYTES,}'
##### .
{, 14055, 57544704, }
##### .Lost+Found
{, 1, 4096, }
##### .Lost+Found/missing_dir__ino=102605
#EMPTY
##### telemetry_dump
{, 4288, 17555456, }
##### telemetry_dump2
{, 5001, 20484096, }
##### telemetry_dump3
```

```
{, 235, 962560, }
##### apollo_telemetry
{, 51, 208896, }
##### mission-6
{, 106, 425984, }
##### mission-6/launch_details
{, 53, 212992, }
##### mission_cancel
{, 4314, 17661952, }
```

In the example above, we are combining ordinary Linux commands with HSTK as described in the **Best Practice - Leveraging Linux To Enhance Your HSTK Commands** section of this document. Note that an additional comma was added to the command output string, directly in front of **1FILES** and after **BYTES**. This was intentional, and makes it easier to parse the data later. Parsing techniques are explained in greater detail in the **Options For Processing HSTK Output** of this document.

A csv-formatted version of this report can be generated using the HSTK_Report_Builder.sh script.

Matching Multiple File Name or Directory Path Values

While it is possible to string together multiple FNMATCH expressions into a single conditional statement, you can also build a summary expression as shown in the following example.

The format is fairly straightforward, you only need to edit a couple of options based on what you are trying to accomplish. Lets look at two examples, one for directory paths, another for files names:

```
SUM({||#A=FNMATCH(({"*.txt";"*.aaak";"*apollo*.*";"*.iso";"telemetry.*"})[ROW],NAME)}[5])==1
SUM({||#A=FNMATCH(({"*/*dump2*/*";"*/*apollo*/*"})[ROW],PATH)}[2])==0
```

The expressions should be updated as follows:

- The quoted values inside the inner braces (for example `".txt" or "/dump2/") are the file names or directory paths to match. For paths, you must include wildcards before and after the target path as shown ("/dump/*"`).
- The `NAME` or `PATH` keyword following `[ROW],` indicates whether you are matching a file NAME or a directory PATH.
- The count in square brackets near the end of the expression (`[5]` and `[2]` in these examples) should equal the number of file or directory name examples provided within the expression.
- The final comparison value indicates either a positive (`==1`) or negative (`==0`) match.

```
# Return file name with path and size in bytes for all files that match the 5 names provided
$ hs eval -r -e
'IS_FILE?SUM({||#A=FNMATCH(({"*.txt";"*.aaak";"*apollo*.*";"*.iso";"telemetry.*"})[ROW],NAME)}[5])==1?{PA
TH,SPACE_USED/BYTES}'
{"./log_mercury.aaak", 4096}
```

```
{"./sensor7_apollo.aaaa", 4096}
{"./1char.txt", 4096}
{"./new_output.txt", 4096}
{"./telemetry_dump3/0byte.txt", 0}
{"./newfile2.txt", 4096}
{"./telemetry_dump3/1char.txt", 4096}
{"./telemetry_dump3/log_mercury.aaak", 4096}
{"./telemetry_dump3/newfile2.txt", 4096}
{"./telemetry_dump3/Rocky_v9.iso", 4096}
. . .

# Return file name with path and size in bytes for all files that are not located in the paths provided
$ hs eval -r -e
'SUM({||#A=FNMATCH(({"*/*dump2*/*";"*/*apollo*/*"})[ROW],PATH)}[2])==0?{PATH,SPACE_USED/BYTES}' | more
{"./.Lost+Found", 0}
{"./prj_mercury", 0}
{"./prj_gemini", 0}
{"./telemetry_dump3/data_gemini.aaab", 4096}
{"./telemetry_dump3/data_gemini.aaac", 4096}
{"./mission_cancel/file.aaaj", 4096}
{"./mission_cancel/file.aaak", 4096}
{"./telemetry_dump/data_gemini.aaad", 4096}
{"./telemetry_dump/data_gemini.aaac", 4096}
{"./mission-6/launch_details/sensor7_apollo.aaal", 4096}
{"./mission-6/launch_details/sensor7_apollo.aaam", 4096}
. . .
```

The equivalent **hs sum** command has not been provided, though the format is the same as the examples provided in the previous section.

Find Files or Directories Based on Tag, Label, or Keyword

The following example command will return the path, name, and size of all files that have the **TAG**, **TAG** with value, **LABEL**, or **KEYWORD** indicated. Consult the Metadata Operations section of this document for examples of how to apply and remove these from files and directories.

In the examples provided below, note the first two commands that query for a tag (**HAS_TAG("mission_specs")**), and then for a tag and value (**GET_TAG("mission_specs")=="launch"**). You will see that all files with that tag are returned for the first query, but the second query returns fewer files. This indicates that not all files with the tag have the same tag value.

```
# Return file name with path and size for all files that have the tag mission_specs
$ hs eval -r -e 'IS_FILE?HAS_TAG("mission_specs")?{PATH,SIZE}' .
{"./telemetry_dump/file.acsl", 1.024 KBYTES}
{"./telemetry_dump/file.adrw", 1.024 KBYTES}
{"./telemetry_dump/telemetry.txt", 11 BYTES}
{"./telemetry_dump/qa_summary.txt", 11 BYTES}

# Return file name with path and size for all files that have the tag mission_specs with value launch
```

```
$ hs eval -r -e 'IS_FILE?GET_TAG("mission_specs")=="launch"?{PATH,SIZE}' .
{"./telemetry_dump/file.acsl", 1.024 KBYTES}
{"./telemetry_dump/file.adrw", 1.024 KBYTES}

# Return file name with path and size for all files that have the label launch_report
$ hs eval -r -e 'IS_FILE? HAS_LABEL("launch_report")?{PATH,SIZE}' .
{"./telemetry_dump/file.aalr", 1.024 KBYTES}
{"./telemetry_dump/file.aecq", 1.024 KBYTES}

# Return file name with path and size for all files that have the keyword mission_prep
$ hs eval -r -e 'IS_FILE? HAS_KEYWORD("mission_prep")?{PATH,SIZE}' .
{"./telemetry_dump/file.aeop", 1.024 KBYTES}
{"./telemetry_dump/file.afym", 1.024 KBYTES}
```

- If applied to a directory, tags, keywords, and labels will be inherited by the directory contents and behave as if they were applied directly.
- When querying metadata values such as tags, labels, and keywords it's important to remember that the values are case sensitive.

Find Files Based on Usage

The following example will find all files that have been accessed in the last 7 days, and list the number of those files and space used.

```
# Return the total file count and space used for files used within the last 7 days
$ hs sum -e 'IS_FILE&&ACCESS_AGE<=7DAYS?{1FILE,SPACE_USED}' .
{264 FILES, 56.137 MBYTES}
```

The following example will find all files accessed more than 2 days ago within the **mission-6** folder, list the number of those files, space used, and print the top 10 table, which lists the 10 largest files.

```
# Return the file count and space used for files in the mission-6 folder that were used 2 days or more
ago, include a top 10 table of files by space used
$ hs sum -e 'IS_FILE&&ACCESS_AGE>=2DAYS?{1FILE,SPACE_USED,TOP10_TABLE{{SPACE_USED,DPATH}}}' mission-6
{106 FILES, 425.98 KBYTES, TOP10_TABLE{
|KEY = {4.096 KBYTES, "./mission-6/telemetry.txt"};
|KEY = {4.096 KBYTES, "./mission-6/sensor7_apollo.aaao"};
|KEY = {4.096 KBYTES, "./mission-6/sensor7_apollo.aaan"};
|KEY = {4.096 KBYTES, "./mission-6/sensor7_apollo.aaam"};
```

6.5. Commands - Filesystem Operations

The HSTK provides many useful options for performing offloaded filesystem operations. If you have a need to perform one of the commands that it supports, using it to run the command offloads the operation to the cluster itself, rather than requiring the client to perform the task.By offloading typically resource intensive filesystem operations, you eliminate the client and network overhead they typically require.

6.5.1. CP

The **hs cp** command performs a fast offloaded recursive copy using a clone operation. The following example shows how it is used:

```
# Perform an offloaded clone of file mission-template.db
$ hs cp -a mission-template.db mission.apollo.db
```

The clone operation is only offloaded for online cluster volumes that support offloaded cloning, such as DSX volumes. While Hammerspace clusters can use external NFS storage as an online volume, not all NFS servers support offloaded cloning.

6.5.2. RM

The **hs rm** command is a multithreaded version of the Linux **rm -rf** command. This command performs the operation directly on the Hammerspace cluster, which provides the most efficient way to delete large directories from Hammerspace shares.

The command should be executed using the same syntax as the Linux equivalent (**rm -rf**) as shown in the following example.

```
# Perform a recursive delete of the mission_cancel directory
$ hs rm -rf mission_cancel
{
"status":"PASSED",
"dirents_found":55,
"started":168722881205540,
"finished":168723202884660,
"rate":341.314503
}
```

6.5.3. RSYNC

The **hs rsync** command is an offloaded version of the Linux recursive directory equalizer. Note that like rsync, when updating a directory, files are both added and removed based on the state of the source directory when the command was run. By offloading the operation, this command ensures the best performance when syncing data from within locations on a Hammerspace cluster.

The following is an example of the command syntax, note that the **--delete** and **--archive** options are required:

```
# Rsync the contents of the telemetry_dump directory into the mission_cancel directory
$ hs rsync --delete --archive telemetry_dump/ mission_cancel
{"assim_id":3}
```

The **rsync** command uses the Hammerspace assimilation feature to copy process requests, which explains the status returned of **"assim_id":3**.

6.5.4. RM-ST

The **hs rm-st** is identical to the **hs rm** command, though it is not multi-threaded. The command syntax and options are identical. Refer to the **RM** section of this document for an example of how the command is used.

6.6. Commands - General

The HSTK provides a number of commands that can be used to provide the status of various aspects of the cluster and filesystem. In this section, we will review these various commands.

6.6.1. Status

The **hs status** command provides views into the status of various processes, tasks, and objects in the Hammerspace environment.

```
# Help page for hs status
$ hs status --help
Usage: hs status [OPTIONS] COMMAND [ARGS]...

[sub] System, component, task status

Options:
--help Show this message and exit.

Commands:
assimilation State of current assimilations
csi Details about the kubernetes CSI
collections Collections present in the share
errors Files in the share with errors
open Files open each dir(s)
replication Replication progress for the share(s)
sweeper Progress of sweeper (checks file placement) for each...
volume Health of volumes backing the share(s)
```

These commands generally require no options. Simply execute them from within a Hammerspace share similar to the following example (partial command output shown):

```
# Show the replication status for the current share
$ hs status replication
REPLICATION_DETAILS_TABLE{
|SITE = SITE('LA-HS-A'),
|INTERVAL = 5 SECONDS,
|SEND_TIME = LOCAL_TIME('2024-11-19 14:53:58'),
|RECV_TIME = LOCAL_TIME('2024-11-19 14:53:50'),
SNAPS = REPLICATION_DETAILS_SUB_TABLE{
|SENT_REQUESTS = 15732,
|SENT_BYTES = 0 BYTES,
|SUCCEEDED_REQUESTS = 15732,
```

```
|SUCCEEDED_BYTES = 0 BYTES,
|ERRORED_REQUESTS = 0,
. . .

# Show the status of all cluster volumes
$ hs status volume
{
"NYC-dsx-2.catalyst.local::/hsvol0", STORAGE_VOLUME_STATUS('REMOVED'), OPERATIONAL_STATUS('DOWN');
"NYC-dsx-2.catalyst.local::/hsvol1", STORAGE_VOLUME_STATUS('REMOVED'), OPERATIONAL_STATUS('DOWN');
"NYC-dsx-1.catalyst.local::/hsvol0", STORAGE_VOLUME_STATUS('REMOVED'), OPERATIONAL_STATUS('DOWN');
"NYC-dsx-1.catalyst.local::/hsvol1", STORAGE_VOLUME_STATUS('REMOVED'), OPERATIONAL_STATUS('DOWN');
"Minio-NFS::/vols/NYC1", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
"NYC-dsx-1.catalyst.local::/hsvol0", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
"NYC-dsx-1.catalyst.local::/hsvol1", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
"NYC-dsx-2.catalyst.local::/hsvol0", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
"NYC-dsx-2.catalyst.local::/hsvol1", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
"Bucket-NYC", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
"Bucket-LA", STORAGE_VOLUME_STATUS('OK'), OPERATIONAL_STATUS('UP');
. . .
```

6.6.2. Objective

In this section, we will review the HSTK commands used to list and manage objectives.

Objectives are defined by the administrator using the Hammerspace CLI, GUI, or API. However, they can be applied to files and directories (with inheritance) by users with the HTSK. The HSTK also provides a way to list the inherited and directly applied objectives on files and directories.

```
# Help page for hs objective
$ hs objective --help
Usage: hs objective [OPTIONS] COMMAND [ARGS]...

Options:
--help Show this message and exit.

Commands:
list list all (objective,expression) pairs assigned
has Get/list objective assignments
delete remove (objective,expression) pair from inode(s)
add Add (objective,expression) pair to inode(s)
```

Best Practice:

- If you intend to apply objectives directly to individual files or directories, you should determine if there are any objectives applied directly to the share that you don't want overridden.
- If an upstream objectives expression is set to TRUE, a downstream objective (like one applied directly to a file) could potentially override it. If you want to prevent that from happening, ensure the critical objective expression is set to ALWAYS.

- For more information regarding Hammerspace objectives, including their design and how they are applied, refer to the {kb-objectives-url}[{kb-objectives-title}] {kb-login-note}.

List Inherited or Directly Applied Objectives

The base **hs objective list** command will list objectives that are directly applied or inherited to the specified file or directory. Note that just because an objective is listed, it does not mean it actually applies at the time the command is run.

```
# List all objectives that are applied to the file telemetry.txt
$ hs objective list telemetry.txt
SLOS_TABLE{
|OBJECTIVE = SLO('durability-1-nine'),
|COUNT = EXPRESSION(DATA_ORIGIN_LOCAL?ALWAYS);

|OBJECTIVE = SLO('durability-3-nines'),
|COUNT = EXPRESSION(DATA_ORIGIN_LOCAL AND IS_DURABLE);

|OBJECTIVE = SLO('availability-1-nine'),
|COUNT = EXPRESSION(DATA_ORIGIN_LOCAL?ALWAYS);

|OBJECTIVE = SLO('keep-online'),
|COUNT = EXPRESSION(IS_BEING_CREATED OR HAS_KEEP_ON_THIS_SITE);

|OBJECTIVE = SLO('optimize-for-capacity'),
|COUNT = TRUE;

|OBJECTIVE = SLO('delegate-on-open'),
|COUNT = TRUE;

|OBJECTIVE = SLO('compress-on-object'),
|COUNT = TRUE;

|OBJECTIVE = SLO('content-based-chunk-on-object'),
|COUNT = TRUE;

|OBJECTIVE = SLO('sync-metadata'),
|COUNT = TRUE;

|OBJECTIVE = SLO('keep-online'),
|COUNT = EXPRESSION(IS_BEING_CREATED OR HAS_ONLINE_INSTANCE AND IS_RECENTLY_USED?ALWAYS);

|OBJECTIVE = SLO('layout-get-on-open'),
|COUNT = EXPRESSION(IS_BEING_CREATED OR HAS_ONLINE_INSTANCE)}
```

Use the **--effective** switch to list objectives that (based on the conditions) actually apply to the file, or use **--active** lists those currently affecting the file in some way. Active objectives are subject to change over time as the state of the file changes, which is why the active and effective objectives do not always match.

```
# List all effective objectives for file telemetry.txt
$ hs objective list --effective telemetry.txt
{
SLO('keep-online'), 0;
SLO('durability-1-nine'), ALWAYS;
SLO('compress-on-object'), 1;
SLO('content-based-chunk-on-object'), 1;
SLO('sync-metadata'), 1;
SLO('optimize-for-capacity'), 1;
SLO('place-on-Bucket-NYC'), 1;
SLO('versioning-1-day'), 1;
SLO('undelete-1-day'), 1;
SLO('place-on-DSX'), 1}

# List all active objectives for telemetry.txt
$ hs objective list --active telemetry.txt
{
SLO('durability-1-nine');
SLO('compress-on-object');
SLO('content-based-chunk-on-object');
SLO('sync-metadata');
SLO('optimize-for-capacity');
SLO('place-on-Bucket-NYC');
SLO('versioning-1-day');
SLO('undelete-1-day');
SLO('place-on-DSX')}
```

In the above example, we see that the **keep-online** objective is effective but not active. The file is currently online so no action needs to be taken.

Add or Delete Directory or File Objectives

The **hs objective add** command can be used to add objectives to files or directories, and can be run recursively. Alternatively, the **hs objective delete** command can remove them.

```
# Help page for hs objective add
$ hs objective add --help
Usage: hs objective add [OPTIONS] NAME paths

Add (objective,expression) pair to inode(s)

Options:
-r, --recursive Apply recursively
--nonfiles Apply recursively to non files
-j, --json Expect --exp to be JSON input
-i, --exp-stdin Read expression from stdin
-e, --exp TEXT Value as hammerscript expression
-s, --string TEXT Value as a string
```

```
--help Show this message and exit.
```

The following example shows an objective being added to a file, the objectives checked to verify the objective was added, and then the objective is deleted.

```
# Add the keep-online objective to the telemetry.txt file
$ hs objective add keep-online telemetry.txt

# List the current objectives applied to the telemetry.txt file
$ hs objective list telemetry.txt
SLOS_TABLE{
|OBJECTIVE = SLO('keep-online'),
|COUNT = TRUE;
. . .

# Delete the keep-online objective from the telemetry.txt file
$ hs objective delete keep-online telemetry.txt
```

If you need a list of objectives that you can apply, use the **hs dump objectives** command detailed in the next section.

You can also add objectives with expressions, which act as conditions. This example shows two different commonly used objective expressions, **ALWAYS** and **TRUE**. **ALWAYS** indicates that the objective cannot be overridden by other objectives, while **TRUE** indicates that it can.

That being said, if an objective applied at the share root is marked as **ALWAYS,** it cannot be overridden by ones that are directly applied to files or directories within.

```
# Add the keep-online objective to the telemetry.txt file, effective always
$ hs objective add -e 'always' keep-online telemetry.txt

# List the current objectives applied to the telemetry.txt file
$ hs objective list telemetry.txt
SLOS_TABLE{
|OBJECTIVE = SLO('keep-online'),
|COUNT = EXPRESSION(ALWAYS);
. . .

# Add the keep-online objective to the newout.txt file, effective true
$ hs objective add -e 'true' keep-online newout.txt

# List the current objectives applied to the newout.txt file
$ hs objective list newout.txt
SLOS_TABLE{
|OBJECTIVE = SLO('keep-online'),
|COUNT = TRUE;
. . .
```

6.6.3. Dump

The **hs dump** command provides a number of subcommands used to obtain information about the cluster or the files and directories that it contains. The information obtained from subcommands such as **misaligned**, **threat**, **files_on_volume** can also be obtained using **hs sum** or **hs eval** commands, while others such as **inode** are typically only used by Hammerspace support.

A common use case for the **hs dump** command is to list the objectives whose names you may need to know. The **hs dump objectives** command lists all objectives present on the cluster. The following example is a shortened version of the command output.

```
# List all objectives available on this Hammerspace cluster
$ hs dump objectives
performance-fast
performance-medium
performance-slow
keep-online
do-not-move
optimize-for-capacity
delegate-on-open
```

Remember that objectives are created per-cluster. While some are created by default, all others are created when you add volumes, create volume groups, or create your own custom objectives.

This document focuses primarily on metadata operations and reporting, so we will not go into further detail regarding the **hs dump** command. The following **hs dump** help page provides details about the various subcommands.

```
# Help page for hs dump
$ hs dump --help
Usage: hs dump [OPTIONS] COMMAND [ARGS]...

[sub] Dump info about various items

Options:
--help Show this message and exit.

Commands:
inode inode metadata
iinfo Alternative inode details, always in JSON format
share Full share(s) metadata
misaligned Dump details about misaligned files on the share(s)
threat Dump details about files that are a virus threat on the...
map_file_to_obj For --native object volumes, dump a mapping between file...
files_on_volume List all files that have data on the specified volume
per...

volumes List available volumes in the cluster
```

Chapter 7. Hammerspace and Data Lifecycle Management

There are many definitions of data lifecycle management and what stages are included. Hammerspace provides advanced capabilities for most or all stages including:

- **Ingest** - Assimilation or directly creating data on Hammerspace.
- **Store and Manage** - Different use cases will have different performance, protection including WORM functionality, and access requirements. Objectives and storage tiers in Hammerspace facilitate this stage.
- **Use and Share** - Hammerspace supports multiprotocol including NFS v3, v4.1 and v4.2, all versions of SMB, and S3 so different applications and use cases can access data the way the application was written without having to refactor. You can ingest via one protocol and analyze via another. Global File System is a key feature used in combination with objectives to put the right data close to users at different locations, often before they need it.
- **Archive** - Objectives combined with the use of less expensive tiers such as object or tape make it easy to move unused data to less expensive tiers including cloud and object.
- **Delete** - While Hammerspace does not delete files as a function of the product, queries based on metadata, even going as far as creating a collection, can be used to determine data that has expired and can be deleted.

Note that {kb-assimilation-url}[{kb-assimilation-title}] {kb-login-note} and {kb-objectives-url}[{kb-objectives-title}] {kb-login-note} are covered in other support articles for reference. At the end of the data lifecycle, queries and collections can be used to determine sets of data that may be appropriate for deletion.

This section is broken down into the following categories:

- **Introduction to Hammerspace Collections**
- **Using Collections for Data Lifecycle Management**

7.1. Introduction to Hammerspace Collections

As discussed previously, collections are subsets of files determined by a metadata query that go in attributes evaluated when the file is created or written to and/or on each pass of the sweeper. The hidden collections directory is created in the root of every Hammerspace share and can be accessed by changing into the **/<share>/.collections** directory.

The following default Hammerspace collections are available:

```
# List all the default Hammerspace collections
$ ls .collections
all live open silent-access
assimilation-failed local-objectives permanent-data-loss snapshot
backup misaligned promoted threat
do-not-move not-selected replication-collision undelete
```

```
durable not-selected2 scan volatile
errored offline selected
links online selected2
```

We won't go into all of these in this document. Most of these are determined by predefined queries, but there are two that can be controlled by a Hammerscript expression determined by a user or administrator, **selected** and **selected2**, and their inverses, **not-selected** and **not-selected2**.

An expression that evaluates as true applied to the **selected** or **selected2** attribute for a file will cause that file to appear in the selected or selected2 directory. Something as simple as this will work:

```
# Set selected to TRUE for files that match telemetry_apollo.aag*
$ hs attribute set selected -e 'TRUE' telemetry_apollo.aag*
##### telemetry_apollo.aaga
##### telemetry_apollo.aagb
##### telemetry_apollo.aagc
##### telemetry_apollo.aagd
. . .
```

Double quotes may be required if using the Windows command line.

Now, files that match **telemetry_apollo.aag*** will appear in .collections/selected, within the same directory structure as they did within the share root:

```
# List contents of selected collection for the telemetry_dump3 directory
$ /.collections/selected/telemetry_dump3 # ls
telemetry_apollo.aaga telemetry_apollo.aagg telemetry_apollo.aagm telemetry_apollo.aags
telemetry_apollo.aagy telemetry_apollo.aagb telemetry_apollo.aagh telemetry_apollo.aagn
telemetry_apollo.aagt telemetry_apollo.aagz telemetry_apollo.aagc telemetry_apollo.aagi
```

Setting an individual file with something as simple as "TRUE" works, but isn't terribly useful. More likely, you would use an expression based on some metadata property that can be evaluated and changed over time, and set it recursively on a directory or share.

7.1.1. Expression - Tag with Value

The following example embeds an expression that evaluates as true if the file has a tag named **Apollo** with the value **mission_log** (**EXPRESSION(get_tag("Apollo")=="mission_log")**). The **-r** means the command will execute recursively, so with the current directory (".") as the target, this expression would be set on all files from the current directory down.

```
# Place files that have the Apollo tag with value mission_log in the selected collection
$ hs attribute set selected -r -e 'EXPRESSION(GET_TAG("Apollo")=="mission_log")' telemetry_dump2 .
```

If you set the expression on a directory, its contents will inherit it. While there may be scenarios where you want the attribute applied to each file individually, in most cases, it is

> enough to set it on the parent directories.

To test this workflow, the user (or a script or tool) can set the **Apollo** tag to the value **mission_log** using the **hs tag set** command as shown in the following example:

```
# Apply the Apollo tag with value mission_log to the file telemetry.txt
$ hs tag set Apollo -s 'mission_log' telemetry.txt
```

The file **telemetry.txt** would now appear in the **SELECTED** collection directory.

To remove a tag from a file or directory, use the **hs tag delete** command as shown in the following example. It is not necessary to supply the value.

```
# Delete the tag Apollo from file telemetry.txt
$ hs tag delete Apollo telemetry.txt
```

7.1.2. Expression - Tag with No Specified Value

Alternatively, you could have the expression match files that have just the tag Apollo, ignoring the value:

```
# Place files that have the Apollo tag with any value in the selected collection
$ hs attribute set selected -r -e 'EXPRESSION(has_tag("Apollo"))'
```

Based on the previous expression, applying either of the following tags would cause the files to match and be placed in the **SELECTED** collection.

```
# Apply the Apollo tag with value mission_log to the file telemetry.txt
$ hs tag set Apollo -s 'mission_log' telemetry.txt

# Apply the Apollo tag with the default value to the file telemetry.txt
$ hs tag set Apollo telemetry.txt
```

To remove a tag from a file or directory, use the **hs tag delete** command as shown in the following example. Note that it is not necessary to supply the value.

```
# Delete the tag Apollo from file telemetry.txt
$ hs tag delete Apollo telemetry.txt
```

7.1.3. Determining Where Attributes Were Applied

To check if a file or directory has the attribute set with an expression that determines if a file will be added to the **SELECTED** collection, use the **hs eval** command. Note that the following examples use two similar commands, but one uses **GET_ATTRIBUTE_LOCAL_UNBOUND** and the other **GET_ATTRIBUTE_INHERITED_UNBOUND**.

If the LOCAL version returns #EMPTY, but the INHERITED returns the expression, you know that the file inherited the attribute from a directory above it.

```
# Show the applied expression that determines if a file is added to the SELECTED collection, if the
attribute was inherited then #EMPTY will be returned
$ hs eval -e 'GET_ATTRIBUTE_LOCAL_UNBOUND("SELECTED")' telemetry.txt
#EMPTY

# Show the applied expression that determines if a file is added to the SELECTED collection, if the
attribute was inherited then the expression will be returned
$ hs eval -e 'GET_ATTRIBUTE_INHERITED_UNBOUND("SELECTED")' telemetry.txt
EXPRESSION(HAS_TAG("Apollo"))
```

To determine where the inherited attribute is set, run the **LOCAL** version of the command against the parent directories until you find where it is set. In the following example, we see that the file inherited the attribute from its parent **telemetry_dump** directory:

```
# Same command as the previous example, but run on the directory that contains the telemetry.txt file
$ hs eval -e 'GET_ATTRIBUTE_LOCAL_UNBOUND("SELECTED")' telemetry_dump2
EXPRESSION(HAS_TAG("Apollo"))
```

7.1.4. Clearing File and Directory Attributes

To clear the selected and selected2 expressions, use the **hs attribute set** command:

```
# Set all attributes for the current directory and all contents to empty (removes them)
$ hs attribute set selected -r -e '#empty' .
```

Consult the **Applying Queries using the Hammerspace Metadata Plugin for Windows** section in **Appendix A - General** of this document for information about using that tool to apply query expressions.

7.2. Using Collections for Data Lifecycle Management

In this section, we will look at how to use collections to perform a simple data lifecycle management task. We will identify unused files, place them in a collection, and delete them.

The following **hs eval** command would return a list of files with full path names (**PATH**) within the **telemetry_dump** directory that have not been accessed or modified in 7 years (**LAST_USE_AGE>7YEAR**):

```
# Return names and path for all files last used more than 7 years ago within the telemetry_dump directory
$ hs eval -r -e 'LAST_USE_AGE>7YEAR?PATH' telemetry_dump
"./telemetry_dump/file.aaaa"
"./telemetry_dump/file.aaab"
"./telemetry_dump/file.aaac"
```

```
"./telemetry_dump/file.aaad"
"./telemetry_dump/file.aaae"
"./telemetry_dump/file.aaaf"
"./telemetry_dump/file.aaag"
. . .
```

You could add additional elements to the query to only include certain file types or other metadata. The directory location is already specified as the path in the command itself, but you could go further and specify a file name:

```
# Return names and path for all files matching the name *.ada* last used more than 7 years ago within the
telemetry_dump directory
$ hs eval -r -e 'LAST_USE_AGE>7YEAR&&FNMATCH("*.ada*",NAME)?PATH' telemetry_dump
"./telemetry_dump/file.adaa"
"./telemetry_dump/file.adac"
"./telemetry_dump/file.adab"
"./telemetry_dump/file.adad"
"./telemetry_dump/file.adaf"
"./telemetry_dump/file.adae"
"./telemetry_dump/file.adag"
"./telemetry_dump/file.adai"
. . .
```

The output could be piped to other OS-level commands to take some action, such as deleting the files as shown in the following example:

```
# Return names and path for all files matching the name *.ada* last used more than 7 YEARs ago within the
telemetry_dump directory, then pipe the list to the rm command to delete the files
$ hs eval -r -e 'LAST_USE_AGE>7YEAR&&FNMATCH("*.ada*",NAME)?PATH' telemetry_dump | xargs rm
# Return names and PATH for all files matching the name *.ada* last used more than 7 years ago within the
telemetry_dump directory, and verify that the files are no longer present
$ hs eval -r -e 'LAST_USE_AGE>7YEAR&&FNMATCH("*.ada*",NAME)?PATH' telemetry_dump
```

You should always test your HSTK commands before piping their output into any command that will alter or delete files, directories, or their metadata.

Another safer approach is to add the files to an existing collection (**selected**), which (as long as the attribute remains set) dynamically updates the list of files or directories within the specified target (**telemetry_dump**) that match the expression (**EXPRESSION(LAST_USE_AGE>7YEAR&&FNMATCH(**".afa*",NAME))*). No action is taken on the files. They are merely added to a collection that will only include the files that match the expression you provided.

```
# Place files in the telemetry_dump directory matching *.afa* last used more than 7 years ago in the
selected collection
$ hs attribute set selected -e 'EXPRESSION(LAST_USE_AGE>7YEAR&&FNMATCH("*.afa*",NAME))' telemetry_dump
```

This sets an expression on the **telemetry_dump** directory that its contents will inherit. This expression will be reevaluated as metadata changes and time passes, and makes files appear in **/<share>/.collections/selected/<path to file>** as their last usage goes over 7 years. In other words, the selected collection would contain a dynamically generated tree of files more than 7 years old.

With this attribute in place, you could then simply delete all files in **/<share>/.collections/selected** as the last stage in the data lifecycle, though again, you should verify the collection includes the data that you intend.

The following example builds upon everything we reviewed in this section, starting from setting the attribute through the deletion of the files added to the collection:

```
# Place files located in the telemetry_dump2 directory that both match the name *.acw* and were last used
less than 1 hour ago in the selected collection
$ hs attribute set selected -e 'EXPRESSION(LAST_USE_AGE<1hour&&FNMATCH("*.acw*",NAME))' telemetry_dump2

# List files in the telemetry_dump2 directory that match *.acw*
$ ls telemetry_dump2/*.acw*
telemetry_dump2/file.acwa
telemetry_dump2/file.acwh
telemetry_dump2/file.acwo
telemetry_dump2/file.acwu

# List current members of the selected collection for directory telemetry_dump2 (currently there are
none)
$ ls .collections/selected/telemetry_dump2

# Add data to file.acwu to change the LAST_USE_AGE metadata value, meaning it will now match the
LAST_USE_AGE<1hour condition
$ echo 'Add New Data' >> telemetry_dump2/file.acwu

# List current members of the selected collection for directory telemetry_dump2 (file.acwu that was
edited is now added)
$ ls .collections/selected/telemetry_dump2
file.acwu

# Perform a recursive delete of the .collections/selected/telemetry_dump2 directory, note the Directory
not empty message is expected
$ rm -rf .collections/selected/telemetry_dump2
rm: cannot remove 'telemetry_dump2': Directory not empty

# List current members of the selected collection for directory telemetry_dump2 (file.acwu has been
deleted)
$ ls .collections/selected/telemetry_dump2

# As expected, file.acwu was deleted from the live file tree
$ ls telemetry_dump2/file.acwu
ls: cannot access 'telemetry_dump2/file.acwu': No such file or directory
```

- The **hs attribute set** command may have been executed against a directory (where the

contents inherit it) or against files individually. It's important to know which. Refer to the **Determining Where Attributes Were Applied** section of this document for the procedure used to verify how the attribute was or is being applied.

- Refer to the **Clearing File and Directory Attributes** section of this document for the procedure used to remove attributes from files and directories.
- If the same **hs attribute set** command were run against a different directory, possibly even using a different expression, files that matched that expression would also be placed in the **selected** collection. Before taking actions on a collection, always check to see that it contains the files that you expect.

Appendices

Appendix A: Appendix a - General

This Appendix contains additional information regarding items or concepts referenced in this document. For additional information, consult the Hammerspace product documentation.

A.1. Hammerspace Toolkit

The HSTK is available for both Windows, Mac, and Linux, and can be downloaded from the Hammerspace HSTK GitHub repository.

A.2. Hammerspace Toolkit Sample Scripts

A collection of sample HSTK scripts are available at the Hammerspace Sample Scripts GitHub repository. These scripts are all related to filesystem reporting, some of which were reviewed in this document.

A.3. Hammerspace Metadata Plugin for Windows

You can download the Hammerspace Metadata Plugin for Windows File Explorer from the Hammerspace Licensing Portal. Once you have logged in, look for the following in the Downloads section.

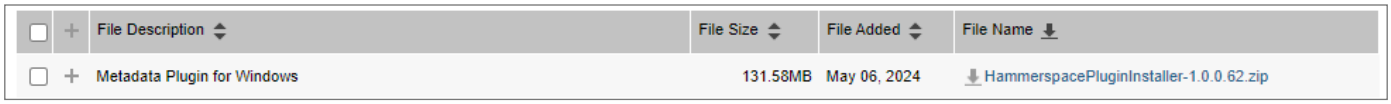

Figure 3. Hammerspace metadata plugin for Windows

Unzip the downloaded package, run (double click) the HammerspaceClassicInstaller.msi file, and follow the prompts. Once the installer completes, open Windows File Explorer and navigate to a Hammerspace share. Then, right-click a file and select **Open Hammerspace**.

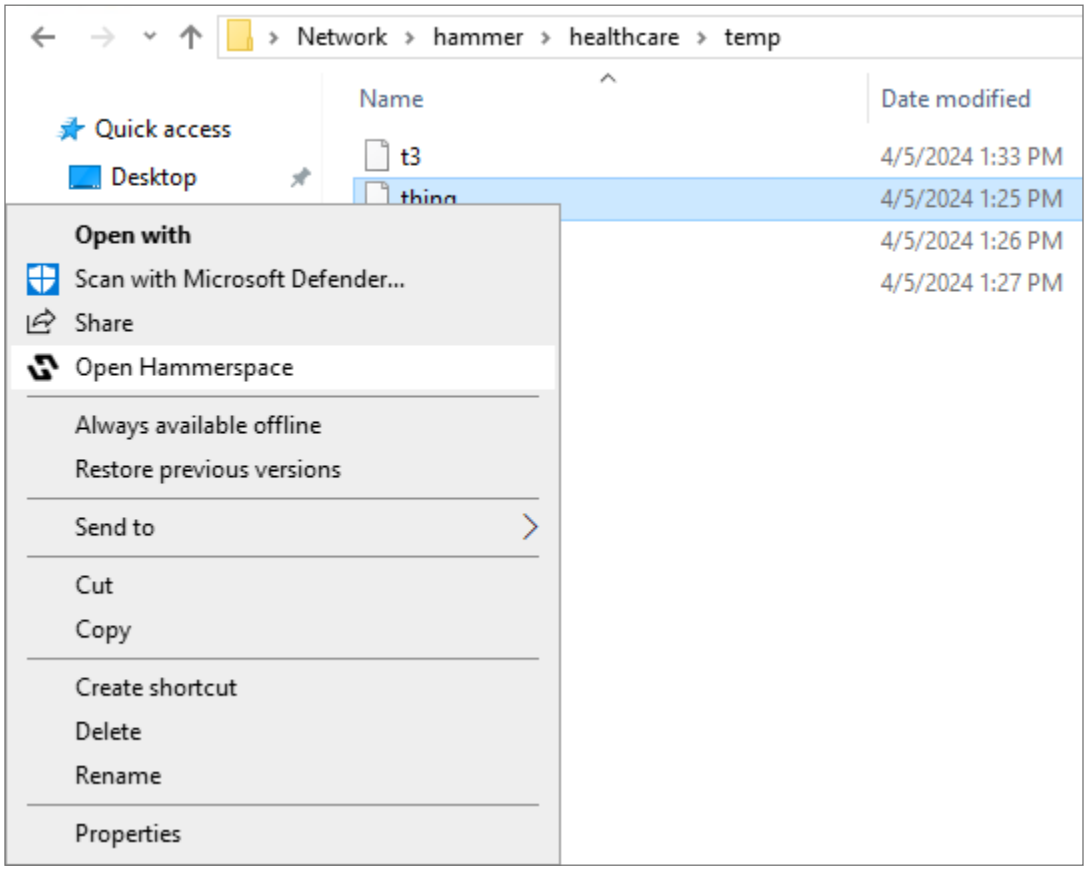

Figure 4. Hammerspace metadata plugin for Windows

You should see the main plugin screen. Note that you can use the plugin to set the most common Hammerspace custom metadata values including keywords, labels, tags, and colors. For users who will be setting these custom metadata values, this option is likely much simpler for everyday use.

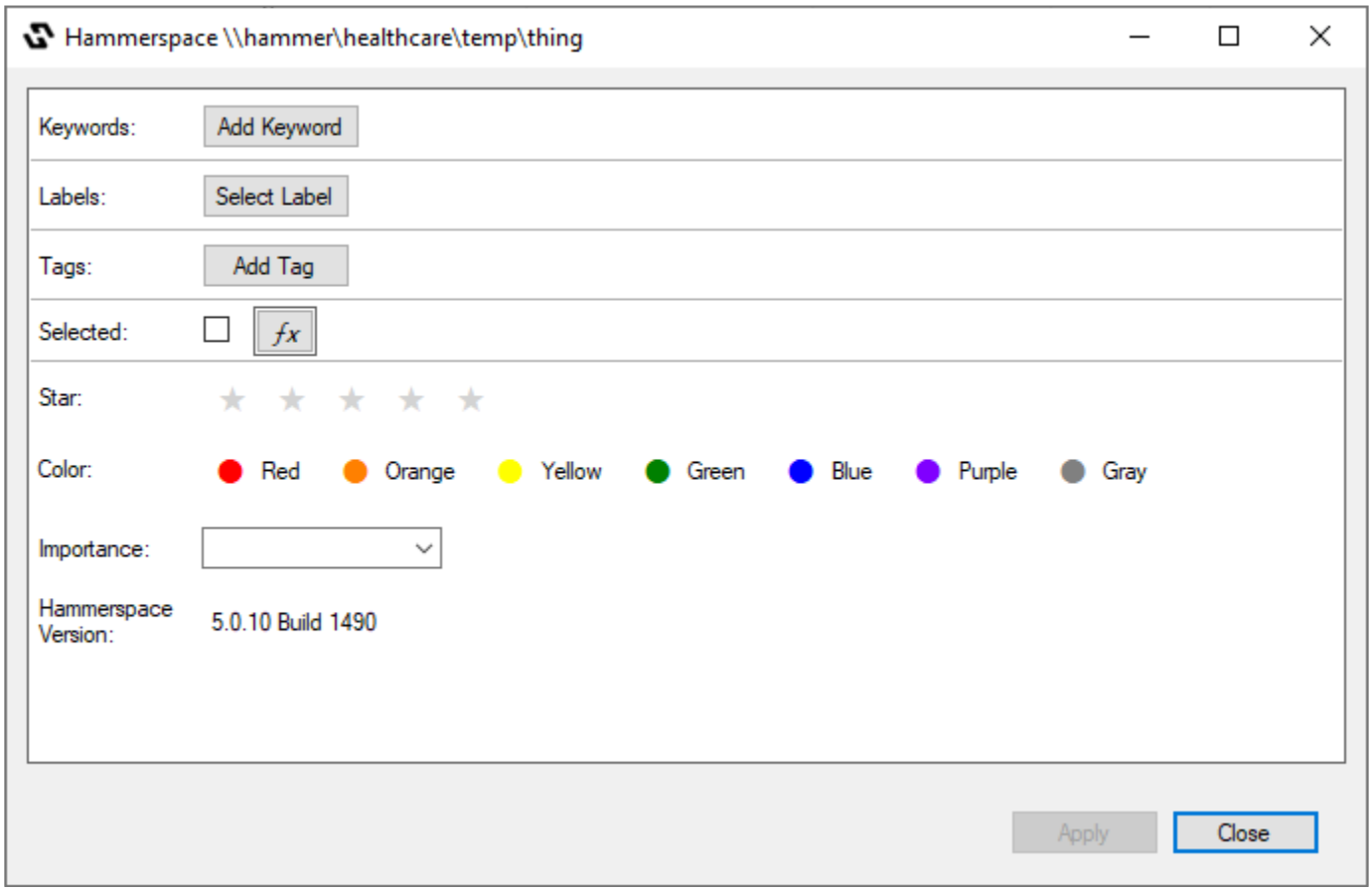

Figure 5. Hammerspace metadata plugin for Windows

A.3.1. Applying Queries Using the Hammerspace Metadata Plugin for Windows

The Hammerspace Metadata Plugin for Windows can also be used to apply queries.

1. In Windows Explorer, right-click on a directory, share, or mapped drive on a Hammerspace share.
2. Click **Open Hammerspace**.
3. Check the **Selected** checkbox and enter the expression for the query you want in the selected collection.

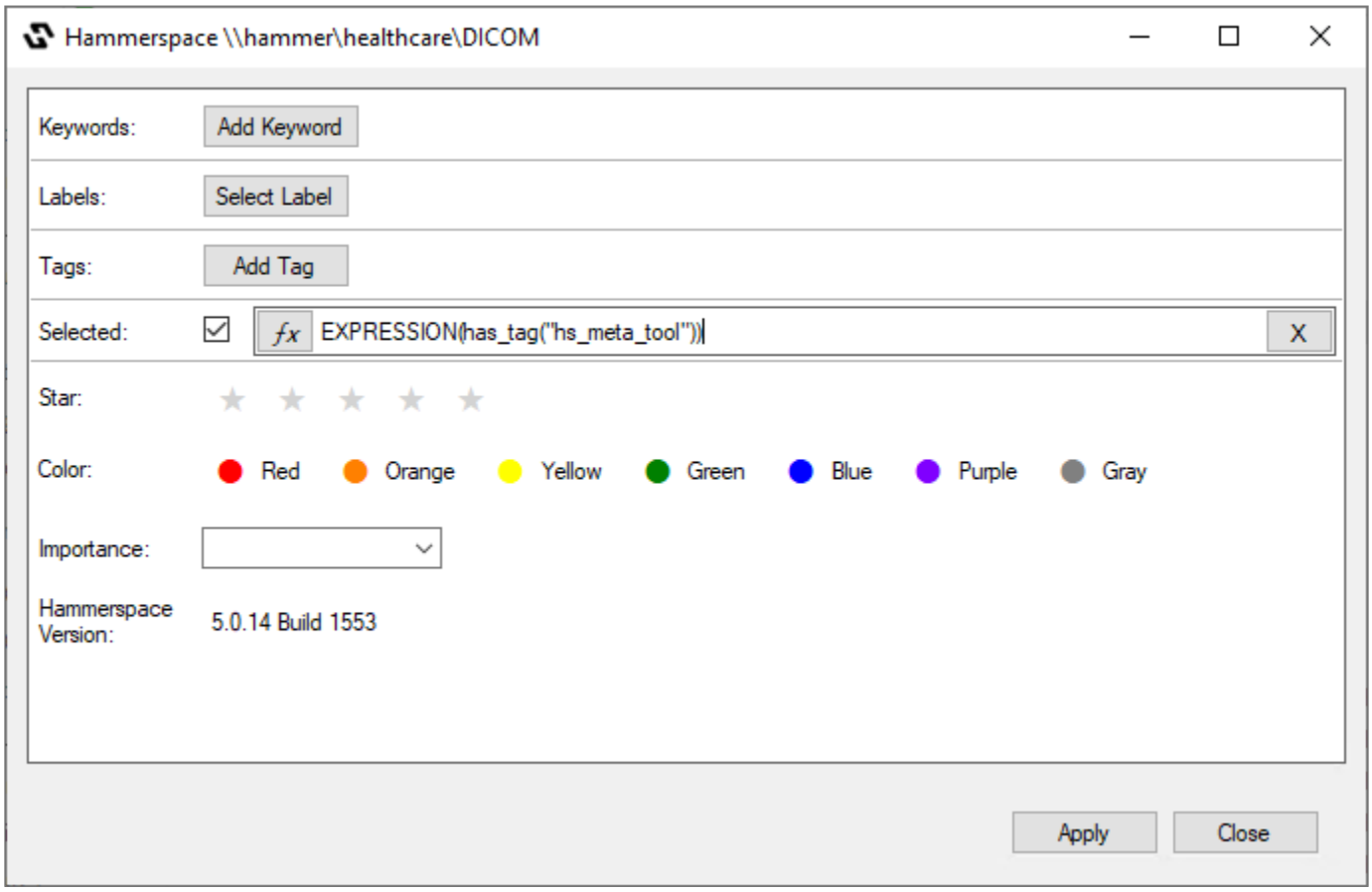

Figure 6. Applying queries using the Hammerspace metadata plugin for Windows

4. Click **Apply**.
5. Navigate within the directory on which you set the selected query and add \.collections\selected to the path in the address bar. From there as you navigate, you will see all directories, but the only files that appear will be the ones that match the expression.

A.4. Common Operators Used with Hammerscript

The following operators can be used within Hammerscript to build conditional statements that can expand or narrow the conditions of the query.

- && (AND, true if all arguments are true)
- || (OR, true if any argument is true)
- ! (NOT, false if the argument is true)
- &&! (AND NOT, true if first argument is true and second is argument false)
- ||! (OR NOT, true if first argument is true or second is argument false)
- == or = (equal to)
- != or <> (not equal to)
- > (greater than)

- >= (greater than or equal to)
- < (less than)
- ⇐ (less than or equal to)
- & (Concatenation)

When creating expressions, the **AND**, **OR**, **NOT**, **AND NOT**, and **OR NOT** operators can also be written in plain text or in the symbol version (**&&**, ||, **!**, **&&!**, and ||**!**).

A.5. Additional Metadata Objects and Functions

The following is a partial list of metadata that may be queried using Hammerscript. This list provides additional examples not previously listed in the **Frequently Used Metadata Values - Reporting** section of this document.

- **ACCESS_TIME:** POSIX atime
- **ACTIVITY:** current average IOPS
- **CHANGE_TIME:** POSIX ctime
- **CREATE_TIME:** Create time. Commonly used with SMB protocol but also set when files are created via NFS.
- **FNMATCH(<TEXT>,NAME|PATH):** Match a string with input such as PATH or NAME
- **MATCH_EXTENSION(<EXT>,NAME):** Match the extension of the filename
- **MODIFY_TIME:** POSIX mtime
- **NOW:** Current date and time
- **PARENT_SHARE:** Share name the file or directory belongs to
- **SIZE:** File size
- **IS_LIVE:** Is the file the live version or not. True indicates the file is in the live tree and not part of a snapshot
- **IS_OPEN:** Does the file have an outstanding layout, i.e. has someone opened the file
- **IS_SNAP:** Is the file in a snapshot?
- **IS_UNDELETE:** Is the file an undelete file? Only true of undelete files in special .snapshot/CURRENT directory
- **VERSION_NEWEST:** Most recent snapshot in which this file exists
- **VERSION_OLDEST:** Oldest snapshot in which this file exists
- **VERSION:** Snapshot version where this file resides
- **VERSION_AGE:** How old is the file compared to the live tree. Age of 0 seconds indicates it is the live version of the file
- **VERSIONS_TOTAL:** Number of snapshots the file is a part of.

Various units can be expressed in easy-to-use English, such as MINUTES, HOURS, WEEKS, and so forth, for time. The same applies for capacity where both GB (gigabytes) and GiB (gibibytes) variants are available.

A.6. File Metadata Example (Full)

The following is the raw output of a file's Hammerspace metadata that contains values for many of the conditions discussed in this document.

Obtain this information for files located on a Hammerspace share using the **hs eval -e this** command as shown in the following example:

```
$ hs eval -e this file.ext
ITEM_TABLE{
|NOW = LOCAL_TIME('2024-09-24 12:53:52'),
|INODE = INODE_NUMBER('961'),
|IS_BEING_CREATED = FALSE,
|NAME = "file-95",
|TYPE = ITEM_TYPE('FILE'),
|MODE = 0664,
|NLINK = 1,
|OWNER = USER('1000'),
|OWNER_GROUP = GROUP('1000'),
|SIZE = 4.096 KBYTES,
|SPACE_USED = 4.096 KBYTES,
|CHANGE_TIME = LOCAL_TIME('2024-09-05 17:15:31'),
|ACCESS_TIME = LOCAL_TIME('2024-09-05 16:58:05'),
|MODIFY_TIME = LOCAL_TIME('2024-09-05 17:15:31'),
|CREATE_TIME = LOCAL_TIME('2024-09-05 16:58:05'),
|LAST_USE_TIME = LOCAL_TIME('2024-09-05 17:15:31'),
|SYMLINK = #EMPTY,
RDEV = RDEV_TABLE{
|MAJOR = 0,
|MINOR = 0},
|ACTIVITY = 0 OPERATIONS / SECOND,
|ERRORS = ERROR_NONE,
|ERRORS_REFRESHED = ERROR_NONE,
|IS_MODIFIED_AFTER_SNAP = FALSE,
|VERSION = 1,
|VERSION_OLDEST = 2,
|VERSION_NEWEST = 1,
ALIGNMENT_DETAILS = ALIGNMENT_DETAILS_TABLE{
|CHANGE_TIME = LOCAL_TIME('1969-12-31 19:00:00'),
|LAST_CHECKED_TIME = LOCAL_TIME('1969-12-31 19:00:00'),
|OVERALL = ALIGNMENT('ALIGNED'),
|PLACE_ON = ALIGNMENT('ALIGNED'),
|EXCLUDE_FROM = ALIGNMENT('ALIGNED'),
|CONFINE_TO = ALIGNMENT('ALIGNED'),
|DO_NOT_MOVE = ALIGNMENT('UNKNOWN'),
|DURABILITY = ALIGNMENT('ALIGNED'),
|ONLINE_DELAY = ALIGNMENT('ALIGNED'),
|AVAILABILITY = ALIGNMENT('ALIGNED'),
|READ_BANDWIDTH = ALIGNMENT('ALIGNED'),
|READ_IOPS = ALIGNMENT('ALIGNED'),
```

```
|READ_LATENCY = ALIGNMENT('ALIGNED'),
|WRITE_BANDWIDTH = ALIGNMENT('ALIGNED'),
|WRITE_IOPS = ALIGNMENT('ALIGNED'),
|WRITE_LATENCY = ALIGNMENT('ALIGNED'),
|STATE = ALIGNMENT_STATE('WAITING_FOR_TARGET_WITH_SPACE'),
#R = 2},
ALIGNMENT_ASPECTS = {
ASPECT('OVERALL'), ALIGNMENT('ALIGNED');
ASPECT('PLACE ON'), ALIGNMENT('ALIGNED');
ASPECT('EXCLUDE FROM'), ALIGNMENT('ALIGNED');
ASPECT('CONFINE TO'), ALIGNMENT('ALIGNED');
ASPECT('DURABILITY'), ALIGNMENT('ALIGNED');
ASPECT('ONLINE DELAY'), ALIGNMENT('ALIGNED');
ASPECT('AVAILABILITY'), ALIGNMENT('ALIGNED');
ASPECT('READ BANDWIDTH'), ALIGNMENT('ALIGNED');
ASPECT('READ IOPS'), ALIGNMENT('ALIGNED');
ASPECT('READ LATENCY'), ALIGNMENT('ALIGNED');
ASPECT('WRITE BANDWIDTH'), ALIGNMENT('ALIGNED');
ASPECT('WRITE IOPS'), ALIGNMENT('ALIGNED');
ASPECT('WRITE LATENCY'), ALIGNMENT('ALIGNED')},
INSTANCES = INSTANCES_TABLE{
|CREATE_TIME = LOCAL_TIME('2024-03-20 08:37:56'),
|START_TIME = LOCAL_TIME('2024-03-20 08:37:56'),
|END_TIME = LOCAL_TIME('2024-09-24 12:53:52'),
|VOLUME = STORAGE_VOLUME('AZ2:VOL2-2'),
|START_OFFSET = 0 BYTES,
|END_OFFSET = 4.096 KBYTES,
|SPACE_USED = 4.096 KBYTES,
|ID = 9223372036854817e3,
|NSHARED = 1,
|READABLE = 1,
|WRITEABLE = 1,
|UID = 10,
|GID = 10,
|MTIME = LOCAL_TIME('2024-09-05 17:18:19'),
|ATIME = LOCAL_TIME('2024-03-20 08:37:56'),
|ERROR_COUNT = 0,
|PATH = "/PrimaryData/Instances/1/0/7624/v40-f1-8000000000009dc6",
|SIGNATURE = "000293DCC16BDFA6C58A1ECA946B140D813A4EF1F843F0FCFE6ECA0D1BBDB8D9CE5D0000000000001000",
|SHARE_PERCENTAGE = 100%,
|LENGTH = 4.096 KBYTES,
|SPARSENESS = 100%,
|ONLINE_DELAY = 0 SECONDS,
|IS_ONLINE = TRUE,
|IS_CURRENT = TRUE,
|IS_BEING_CREATED = FALSE,
|IS_BEING_MOVED = FALSE,
|PERCENTAGE_COMPLETE = 100%},
|HASH = "000293DCC16BDFA6C58A1ECA946B140D813A4EF1F843F0FCFE6ECA0D1BBDB8D9CE5D0000000000001000",
|ASSIMILATION_STATUS = #EMPTY,
```

```
SWEEP_DETAILS = SWEEP_DETAILS_TABLE{
|NUMBER = 166821,
|START_TIME = LOCAL_TIME('2024-09-24 12:53:10'),
|END_TIME = LOCAL_TIME('2024-09-24 12:53:48'),
|COUNT = 1.002 KFILES,
ERROR_COUNTS = ERROR_STATS_TABLE{
|ERROR = ERROR_NUMBER('1'),
|COUNT = 6},
|TOTAL_COUNT = 1.002 KFILES,
|RATE = 26.243 FILES / SECOND,
|PROGRESS = 100%,
|ELAPSED_TIME = 38.182 SECONDS,
|REMAINING_TIME = 0 SECONDS,
|COMPLETION_TIME = LOCAL_TIME('2024-09-24 12:53:48');

|NUMBER = 166822,
|START_TIME = LOCAL_TIME('2024-09-24 12:53:48'),
|END_TIME = LOCAL_TIME('2024-09-24 12:53:52'),
|COUNT = 79 FILES,
ERROR_COUNTS = ERROR_STATS_TABLE{
|ERROR = ERROR_NUMBER('1'),
|COUNT = 1},
|TOTAL_COUNT = 1.002 KFILES,
|RATE = 21.02 FILES / SECOND,
|PROGRESS = 7.8842%,
|ELAPSED_TIME = 3.75837 SECONDS,
|REMAINING_TIME = 43.911 SECONDS,
|COMPLETION_TIME = LOCAL_TIME('2024-09-24 12:54:36')},
|BOOT_EPOCH = 4,
REPLICATION_DETAILS = REPLICATION_DETAILS_TABLE{
|SITE = SITE('NYC-HS-A'),
|INTERVAL = 5 SECONDS,
|SEND_TIME = LOCAL_TIME('2024-09-24 12:53:44'),
|RECV_TIME = LOCAL_TIME('2024-09-24 12:53:37'),
SNAPS = REPLICATION_DETAILS_SUB_TABLE{
|SENT_REQUESTS = 76557,
|SENT_BYTES = 0 BYTES,
|SUCCEEDED_REQUESTS = 76556,
|SUCCEEDED_BYTES = 0 BYTES,
|ERRORED_REQUESTS = 1,
|ERRORED_BYTES = 0 BYTES,
|COMPLETED_REQUESTS = 76557,
|COMPLETED_BYTES = 0 BYTES,
|PENDING_REQUESTS = 0,
|PENDING_BYTES = 0 BYTES},
DIRENTS = REPLICATION_DETAILS_SUB_TABLE{
|SENT_REQUESTS = 0,
|SENT_BYTES = 0 BYTES,
|SUCCEEDED_REQUESTS = 0,
|SUCCEEDED_BYTES = 0 BYTES,
```

```
|ERRORED_REQUESTS = 0,
|ERRORED_BYTES = 0 BYTES,
|COMPLETED_REQUESTS = 0,
|COMPLETED_BYTES = 0 BYTES,
|PENDING_REQUESTS = 0,
|PENDING_BYTES = 0 BYTES},
INODES = REPLICATION_DETAILS_SUB_TABLE{
|SENT_REQUESTS = 0,
|SENT_BYTES = 0 BYTES,
|SUCCEEDED_REQUESTS = 0,
|SUCCEEDED_BYTES = 0 BYTES,
|ERRORED_REQUESTS = 0,
|ERRORED_BYTES = 0 BYTES,
|COMPLETED_REQUESTS = 0,
|COMPLETED_BYTES = 0 BYTES,
|PENDING_REQUESTS = 0,
|PENDING_BYTES = 0 BYTES},
TOTAL = REPLICATION_DETAILS_SUB_TABLE{
|SENT_REQUESTS = 76557,
|SENT_BYTES = 0 BYTES,
|SUCCEEDED_REQUESTS = 76556,
|SUCCEEDED_BYTES = 0 BYTES,
|ERRORED_REQUESTS = 1,
|ERRORED_BYTES = 0 BYTES,
|COMPLETED_REQUESTS = 76557,
|COMPLETED_BYTES = 0 BYTES,
|PENDING_REQUESTS = 0,
|PENDING_BYTES = 0 BYTES},
|SEND_AGE = 8.0824 SECONDS,
|RECV_AGE = 14.333 SECONDS},
ASSIMILATION_DETAILS = ASSIMILATION_DETAILS_TABLE{},
|HAS_PERMANENT_DATA_LOSS = FALSE,
|HAS_PERMANENT_DATA_LOSS_REFRESHED = FALSE,
|HAS_ERROR = FALSE,
|ASSIMILATION_ERROR = ERROR_NONE,
|REPLICATION_ERROR = ERROR_NONE,
VERSION_DETAILS = VERSION_DETAILS_TABLE{
|TIME = LOCAL_TIME('2024-09-24 12:53:52'),
|PATH = "./file-95";

|TIME = LOCAL_TIME('2024-09-24 12:53:52'),
|PATH = "./.snapshot/current/file-95"},
|VERSIONS_TOTAL = 2,
|VERSION_TIME = LOCAL_TIME('2024-09-24 12:53:52'),
|LAST_EXISTED_TIME = LOCAL_TIME('2024-09-24 12:53:52'),
|SPARSENESS = 100%,
|OVERALL_ALIGNMENT = ALIGNMENT('ALIGNED'),
|OVERALL_ALIGNMENT_STATE = ALIGNMENT_STATE('WAITING_FOR_TARGET_WITH_SPACE'),
|ALIGNMENT_CHANGE_TIME = LOCAL_TIME('1969-12-31 19:00:00'),
|DATA_ORIGIN_LOCAL = FALSE,
```

```
|DATA_ORIGIN_REMOTE = TRUE,
DATA_ORIGIN_SITE = PARTICIPANTS_TABLE{
|ID = 1,
|SHARE_ID = 4,
|ADDRESS = "192.200.10.175",
|PORT = 9097,
|INTERVAL = 5 SECONDS,
|SITE_PARTICIPANT_ID = 2,
|ADMIN_STATUS = ADMINISTRATIVE_STATUS('ENABLED'),
|OPER_STATUS = OPERATIONAL_STATUS('UP'),
|SITE_NAME = "NYC-HS-A"},
|IS_OPEN = FALSE,
|HAS_ONLINE_INSTANCE = TRUE,
|IS_ONLINE = TRUE,
|IS_BEING_MOVED = FALSE,
|IS_OFFLINE = FALSE,
|IS_VOLATILE = FALSE,
|IS_DURABLE = TRUE,
|IS_UNAVAILABLE = FALSE,
|IS_AVAILABLE = TRUE,
|IS_ACTIVE = FALSE,
|IS_INACTIVE = TRUE,
|IS_LIVE = TRUE,
|IS_SNAP = FALSE,
|IS_UNDELETE = 0,
|IS_DELETED = FALSE,
|IS_FILE = TRUE,
|IS_SYMLINK = FALSE,
|IS_DIRECTORY = FALSE,
|HAS_DIRECTORY_ENTRIES = FALSE,
|IS_DEVICE = FALSE,
|IS_THREAT = FALSE,
|IS_NON_THREAT = FALSE,
|IS_THREAT_UNKNOWN = TRUE,
|MAY_BE_THREAT = TRUE,
|CREATE_AGE = (18 DAYS+19:55:51),
|CHANGE_AGE = (18 DAYS+19:38:21),
|ACCESS_AGE = (18 DAYS+19:55:51),
|MODIFY_AGE = (18 DAYS+19:38:21),
|LAST_USE_AGE = (18 DAYS+19:38:21),
|ACTUAL_CREATE_AGE = (18 DAYS+19:55:51),
|ACTUAL_ACCESS_AGE = (18 DAYS+19:55:51),
|ACTUAL_MODIFY_AGE = (18 DAYS+19:38:21),
|ACTUAL_LAST_USE_AGE = (18 DAYS+19:38:21),
|VERSION_AGE = 0 SECONDS,
|DELETED_AGE = 0 SECONDS,
|ALIGNMENT_CHANGE_AGE = (54 YEARS+280 DAYS),
|ONLINE_DELAY = 0 SECONDS,
|ORIGIN_SITE = SITE('LA-HS-B'),
ATTRIBUTES = ITEM_ATTRIBUTES_TYPED_TABLE{
```

```
|VIRUS_SCAN = VIRUS_SCAN_STATE('UNSCANNED'),
MIME = MIME_TYPE_TABLE{
|STRING = "/"},
|DO_NOT_MOVE = FALSE,
|CONSIDER_VOLATILE = FALSE,
|RECENTLY_USED_DURATION = 00:05:00,
|SELECTED = FALSE,
|SELECTED2 = FALSE,
|PROMOTE = FALSE,
|DEMOTE = FALSE,
COLORS = COLORS_TABLE{},
CSI_DETAILS = CSI_DETAILS_TABLE{}},
|OBJECT = OBJECT_NUMBER('1125899906842719'),
|DEPTH = 2,
|PATH = "./file-95",
DIRECTORY_STATS = DIRECTORY_STATS_TABLE{
|FILE_COUNT = 1 FILE,
|SPACE_USED = 1.234 MBYTES,
|ACTIVITY = 0.011574 OPERATIONS / SECOND,
|LAST_USE_TIME = LOCAL_TIME('1901-05-15 20:00:00'),
|LAST_USE_AGE = (123 YEARS+162 DAYS),
|ACTIVITY_LEVEL = IOPS_LEVEL('UNDER 10 IOPS'),
|HEAT = TEMPERATURE(|ACTIVITY=0.011574 OPERATIONS / SECOND),
|HEAT_LEVEL = TEMPERATURE_LEVEL('UNDER 10 IOPS')},
|DPATH = "./file-95",
|IS_ROOT = FALSE,
ALL_LABELS = LABELS_TABLE{},
|ACTUAL_CREATE_TIME = LOCAL_TIME('2024-09-05 16:58:05'),
|ACTUAL_MODIFY_TIME = LOCAL_TIME('2024-09-05 17:15:31'),
|ACTUAL_ACCESS_TIME = LOCAL_TIME('2024-09-05 16:58:05'),
|ACTUAL_LAST_USE_TIME = LOCAL_TIME('2024-09-05 17:15:31'),
|READONLY = FALSE,
|HIDDEN = FALSE,
|SYSTEM = FALSE,
|OFFLINE = FALSE,
|DIRECTORY = FALSE,
|ARCHIVE = FALSE,
|NORMAL = TRUE,
|REPARSEPOINT = FALSE,
|NOSCRUBDATA = FALSE,
|IS_NEWBORN = FALSE,
|IS_RECENTLY_USED = FALSE,
|IS_UNUSED_SINCE_CREATION = FALSE,
|ACTIVITY_LEVEL = IOPS_LEVEL('UNDER 10 IOPS'),
|HEAT = TEMPERATURE(|LAST_USE_AGE=(18 DAYS+19:38:21)),
|HEAT_LEVEL = TEMPERATURE_LEVEL('2 TO 3 WEEKS OLD'),
|SPACE_USED_LEVEL = SIZE_LEVEL('4 TO 32 KBYTES'),
|ACCESS_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|MODIFY_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|CHANGE_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
```

```
|CREATE_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|LAST_USE_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|ACTUAL_ACCESS_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|ACTUAL_MODIFY_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|ACTUAL_CREATE_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|ACTUAL_LAST_USE_AGE_LEVEL = TIMESPAN_LEVEL('2 TO 3 WEEKS OLD'),
|PARENT_SHARE = SHARE('test_kent'),
PARTICIPANTS = PARTICIPANTS_TABLE{
|ID = 0,
|SHARE_ID = 42,
|ADDRESS = "192.200.10.160",
|PORT = 9097,
|INTERVAL = 0 SECONDS,
|SITE_PARTICIPANT_ID = 0,
|ADMIN_STATUS = ADMINISTRATIVE_STATUS('ENABLED'),
|OPER_STATUS = OPERATIONAL_STATUS('UP'),
|SITE_NAME = "LA-HS-A";

|ID = 1,
|SHARE_ID = 4,
|ADDRESS = "192.200.10.175",
|PORT = 9097,
|INTERVAL = 5 SECONDS,
|SITE_PARTICIPANT_ID = 2,
|ADMIN_STATUS = ADMINISTRATIVE_STATUS('ENABLED'),
|OPER_STATUS = OPERATIONAL_STATUS('UP'),
|SITE_NAME = "NYC-HS-A"},
DIRECTORY_ENTRIES = ITEM_TABLE{}}
```

Appendix B: Appendix B - Leveraging HSTK with PowerShell

Once you have developed a list of Hammerspace commands that you intend to frequently use, you can develop PowerShell scripts that guide you through the command options. You may find this easier than maintaining a list of frequently used commands and trying to remember their syntax.

Similar to some techniques that were shown previously in this document, PowerShell enables you to do things like run commands repeatedly with only a single action, rather than having to repeat them over and over again.

A collection of sample PowerShell scripts is available on the sample script repository on Github.