XC™ Series DataWarp™ User Guide (CLE 6.0.UP02) S-2558

Contents

1 About the DataWarp User Guide....3
2 Quick Start to Using DataWarp....4
2.1 Use DataWarp as Application Scratch....4
3 About DataWarp....7
3.1 Overview of the DataWarp Process....8
3.2 DataWarp Concepts....10
4 Check the Status of DataWarp Resources....14
5 DataWarp Job Script Commands....16
5.1 #DW jobdw - Job Script Command....16
5.2 #DW persistentdw - Job Script Command....20
5.3 #DW stage_in - DataWarp Job Script Command....22
5.4 #DW stage_out - Job Script Command....23
5.5 #DW swap - Job Script Command....25
5.6 DataWarp Job Script Command Examples....25
5.7 Diagrammatic View of Batch Jobs....28
6 Additional Considerations when Using DataWarp....32
6.1 DVS Client-side Caching can Improve DataWarp Performance....32
6.2 Use SSD Protection Settings....32
7 libdatawarp - the DataWarp API....34
8 Troubleshooting....37
8.1 Why Do `dwcli` and `dwstat` Fail?....37
9 Terminology....39
10 Prefixes for Binary and Decimal Multiples....41

1 About the DataWarp User Guide

Scope and Audience

XC™ Series DataWarp User Guide covers DataWarp concepts, commands, and the API. It does not cover specific commands of the supported workload managers. This publication is intended for users of Cray XC™ series systems installed with DataWarp SSD cards.

Release CLE 6.0

This publication supports the CLE 6.0.UP02 release of the Cray Linux Environment (CLE).

Revision Information

November 3, 2016: initial release

Typographic Conventions

`Monospace`	Indicates program code, reserved words, library functions, command-line prompts, screen output, file/path names, key strokes (e.g., `Enter` and `Alt-Ctrl-F`), and other software constructs.
`Monospaced Bold`	Indicates commands that must be entered on a command line or in response to an interactive prompt.
`Oblique` or *`Italics`*	Indicates user-supplied values in commands or syntax definitions.
Proportional Bold	Indicates a graphical user interface window or element.
\ (backslash)	At the end of a command line, indicates the Linux® shell line continuation character (lines joined by a backslash are parsed as a single line). Do not type anything after the backslash or the continuation feature will not work correctly.

Feedback

Please provide feedback by visiting *http://pubs.cray.com* and clicking the *Contact Us* button in the upper-right corner, or by sending email to *pubs@cray.com*.

2 Quick Start to Using DataWarp

DataWarp storage is accessed through a site's workload manager (WLM) such as PBS, Moab, and SLURM. DataWarp job script commands are added to a batch script to indicate the amount of DataWarp storage required, how the storage is to be configured, and whether files are to be staged from the parallel file system (PFS) to DataWarp or from DataWarp to the PFS.

How the DataWarp storage is to be used determines how it needs to be configured. The most common use cases are:

- application scratch
- shared storage
- data cache between an application and the PFS

Examples of these configurations are in the process of being developed and will be added in future revisions of this document. Note that these examples include WLM commands, and that each WLM has its own syntax for interacting with DataWarp. It is beyond the scope of this guide to detail the various methods. Examples are provided with the caveat that they may be out of sync with changes made by the WLM vendors. For details, see the appropriate WLM documentation.

2.1 Use DataWarp as Application Scratch

Prerequisites

This procedure assumes the existence of a successfully runnable job script.

About this task

I/O intensive applications can benefit from the higher bandwidth available to DataWarp storage than to a PFS by using DataWarp like a /tmp file system.

Procedure

1. Add a `#DW jobdw` command to the job script to define the scratch instance and how it will be accessed.

   ```
   #DW jobdw type=scratch capacity=n access_mode=mode
   ```

   Where:

   capacity

   Specifies the amount of DataWarp storage

   access_mode

Defines how the storage looks to the compute nodes. It can be either or both of the following:

striped Data is striped across multiple DataWarp nodes, and the compute node path to the storage is $DW_JOB_STRIPE.

private Each of the job's compute nodes has its own, private storage, and the compute node path to the storage is $DW_JOB_PRIVATE.

```
#DW jobdw type=scratch access_mode=striped capacity=100TiB
```

Each compute node has striped/shared access to DataWarp via $DW_JOB_STRIPED.

2. (Optional) Add a #DW stage_in command to the job script to stage data from the PFS into DataWarp storage as input to the application.

```
#DW stage_in type=type source=pfs_path destination=dws_path
```

Where:

type=directory|file|list

Specifies the type of entity for staging; a single directory, including all files and sub-directories; a single file; or a file containing a list of source-file/destination pairs

source

Specifies a PFS path to which the user has read privileges.

Specifies the path to the directory|file|list within the DataWarp instance. `source` must start with $DW_JOB_STRIPE.

destination

Specifies the path to the location within the DataWarp instance where the data is to be staged. *`dws_path`* must start with $DW_JOB_STRIPE.

The following stages data from `/pfs/mystuff/data` on the PFS to the `input` directory of the job's instance pointed to by `$DW_JOB_STRIPED`.

```
#DW stage_in type=directory source=/pfs/mystuff/data \
#DW          destination=$DW_JOB_STRIPED/input
```

3. (Optional) Add a #DW stage_out command to the job script to stage data out to the parallel file system (PFS) for retention.

At the end of a job, the WLM runs a series of commands that, among other things, cleans up any usage of the DataWarp storage. Therefore, to retain any of the data, it must be *staged out* to the PFS.

```
#DW stage_out type=type source=$DW_JOB_STRIPED/path destination=pfs_path
```

Where:

type=directory|file|list

Specifies the type of entity for staging; a single directory, including all files and sub-directories; a single file; or a file containing a list of source-file/destination pairs

source

Specifies the path to the directory|file|list within the DataWarp instance. `source` must start with $DW_JOB_STRIPE.

destination

Specifies a PFS path to which the user has write privileges.

```
#DW stage_out type=directory source=$DW_JOB_STRIPED/results \
#DW           destination=/pfs/mystuff/runresults1
```

The `results` directory within the DataWarp instance is staged to the PFS.

4. Provide DataWarp storage access information to the application. Without this information, the application will not find the storage.

   This assumes that the application requires arguments specifying input and/or output paths.

   ```
   srun app.out app_args_here
   ```

   A simple SLURM example:

   ```
   #!/bin/bash
   #SBATCH -p regular
   #SBATCH -N 4
   #SBATCH -t 01:00:00
   #DW jobdw type=scratch access_mode=striped capacity=100TiB
   #DW stage_in type=directory source=/pfs/mystuff/data destination=$DW_JOB_STRIPED/input
   srun app.out $DW_JOB_STRIPED/input
   ```

3 About DataWarp

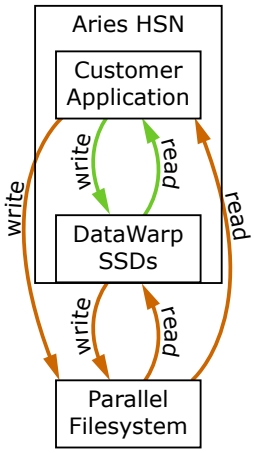
Cray DataWarp provides an intermediate layer of high bandwidth, file-based storage to applications running on compute nodes. It is comprised of commercial SSD hardware and software, Linux community software, and Cray system hardware and software. DataWarp storage is located on server nodes connected to the Cray system's high speed network (HSN). I/O operations to this storage completes faster than I/O to the attached parallel file system (PFS), allowing the application to resume computation more quickly and resulting in improved application performance. DataWarp storage is transparently available to applications via standard POSIX I/O operations and can be configured in multiple ways for different purposes. DataWarp capacity and bandwidth are dynamically allocated to jobs on request and can be scaled up by adding DataWarp server nodes to the system.

Each DataWarp server node can be configured either for use by the DataWarp infrastructure or for a site specific purpose such as a Hadoop Distributed File System (HDFS).

> **IMPORTANT:** Keep in mind that DataWarp is focused on performance and not long-term storage. SSDs can and do fail.

The following diagram is a high level view of how applications interact with DataWarp. SSDs on the Cray high-speed network enable compute node applications to quickly read and write data to the SSDs, and the DataWarp file system handles staging data to and from a parallel filesystem.

Figure 1. DataWarp Overview



DataWarp Use Cases

There are four basic use cases for DataWarp:

Parallel File System (PFS) cache	DataWarp can be used to cache data between an application and the PFS. This allows PFS I/O to be overlapped with an application's computation. In this release there are two ways to use DataWarp to influence data movement (staging) between DataWarp and the PFS. The first requires a job and/or application to explicitly make a request and have the DataWarp Service (DWS) carry out the operation. In the second way, data movement occurs implicitly (i.e., read-

ahead and write-behind) and no explicit requests are required. Examples of PFS cache use cases include:

- **Checkpoint/Restart**: Writing periodic checkpoint files is a common fault tolerance practice for long running applications. Checkpoint files written to DataWarp benefit from the high bandwidth. These checkpoints either reside in DataWarp for fast restart in the event of a compute node failure or are copied to the PFS to support restart in the event of a system failure.
- **Periodic output**: Output produced periodically by an application (e.g., time series data) is written to DataWarp faster than to the PFS. Then as the application resumes computation, the data is copied from DataWarp to the PFS asynchronously.
- **Application libraries**: Some applications reference a large number of libraries from every rank (e.g., Python applications). Those libraries are copied from the PFS to DataWarp once and then directly accessed by all ranks of the application.

Application scratch

DataWarp can provide storage that functions like a `/tmp` file system for each compute node in a job. This data typically does not touch the PFS, but it can also be configured as PFS cache. Applications that use out-of-core algorithms, such as geographic information systems, can use DataWarp scratch storage to improve performance.

Shared storage

DataWarp storage can be shared by multiple jobs over a configurable period of time. The jobs may or may not be related and may run concurrently or serially. The shared data may be available before a job begins, extend after a job completes, and encompass multiple jobs. Shared data use cases include:

- **Shared input**: A read-only file or database (e.g., a bioinformatics database) used as input by multiple analysis jobs is copied from PFS to DataWarp and shared.
- **Ensemble analysis**: This is often a special case of the above **shared input** for a set of similar runs with different parameters on the same inputs, but can also allow for some minor modification of the input data across the runs in a set. Many simulation stategies use ensembles.
- **In-transit analysis**: This is when the results of one job are passed as the input of a subsequent job (typically using job dependencies). The data can reside only on DataWarp storage and may never touch the PFS. This includes various types of workflows that go through a sequence of processing steps, transforming the input data along the way for each step. This can also be used for processing of intermediate results while an application is running; for example, visualization or analysis of partial results.

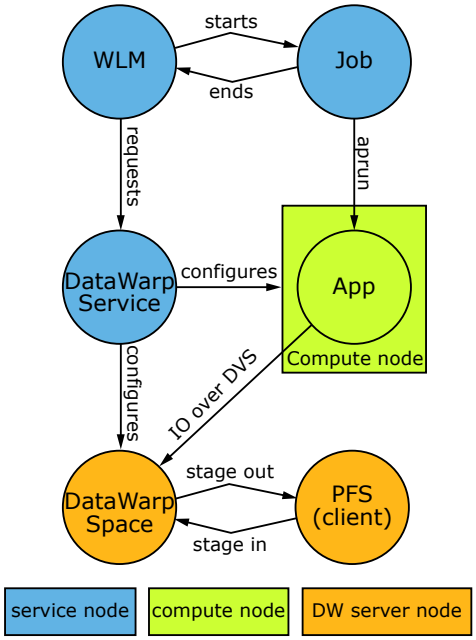
Compute node swap

When configured as swap space, DataWarp allows applications to over-commit compute node memory. This is often needed by pre- and post-processing jobs with large memory requirements that would otherwise be killed.

3.1 Overview of the DataWarp Process

The following figure provides visual representation of the DataWarp process.

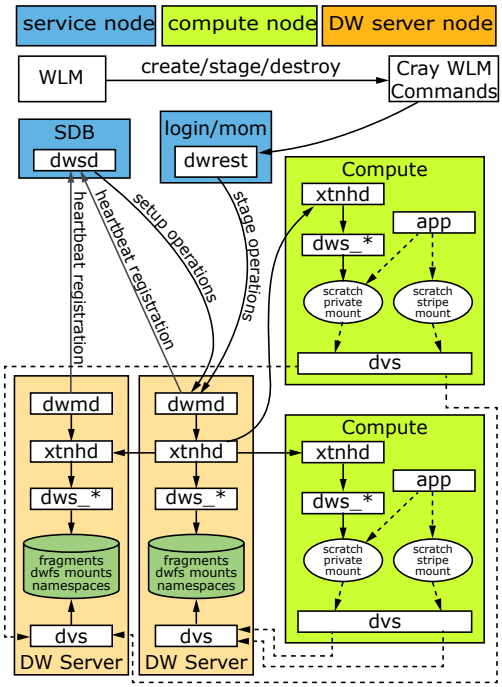
Figure 2. DataWarp Component Interaction - bird's eye view



1. A user submits a job to a workload manager. Within the job submission, the user must specify: the amount of DataWarp storage required, how the storage is to be configured, and whether files are to be staged from the parallel file system (PFS) to DataWarp or from DataWarp to the PFS.
2. The workload manager (WLM) provides queued access to DataWarp by first querying the DataWarp service for the total aggregate capacity. The requested capacity is used as a job scheduling constraint. When sufficient DataWarp capacity is available and other WLM requirements are satisfied, the workload manager requests the needed capacity and passes along other user-supplied configuration and staging requests.
3. The DataWarp service dynamically assigns the storage and initiates the stage in process.
4. After this completes, the workload manager acquires other resources needed for the batch job, such as compute nodes.
5. After the compute nodes are assigned, the workload manager and DataWarp service work together to make the configured DataWarp accessible to the job's compute nodes. This occurs prior to execution of the batch job script.
6. The batch job runs and any subsequent applications can interact with DataWarp as needed (e.g., stage additional files, read/write data).
7. When the batch job ends, the workload manager stages out files, if requested, and performs cleanup. First, the workload manager releases the compute resources and requests that the DataWarp service (DWS) make the previously accessible DataWarp configuration inaccessible to the compute nodes. Next, the workload manager requests that additional files, if any, are staged out. When this completes, the workload manager tells the DataWarp service that the DataWarp storage is no longer needed.

The following diagram includes extra details regarding the interaction between a WLM and the DWS as well as the location of the various DWS daemons.

Figure 3. DataWarp Component Interaction - detailed view



3.2 DataWarp Concepts

For basic definitions, refer to *Terminology* on page 39.

Instances

DataWarp storage is assigned dynamically when requested, and that storage is referred to as an *instance*. The space is allocated on one or more DataWarp server nodes and is dedicated to the instance for the lifetime of the instance. A DataWarp instance has a lifetime that is specified when the instance is created, either *job instance* or *persistent instance*. A job instance is relevant to all previously described use cases except the shared data use case.

- **Job instance**: The lifetime of a job instance, as it sounds, is the lifetime of the job that created it, and is accessible only by the job that created it.
- **Persistent instance**: The lifetime of a persistent instance is not tied to the lifetime of any single job and is terminated by command. Access can be requested by any job, but file access is authenticated and authorized based on the POSIX file permissions of the individual files. Jobs request access to an existing persistent instance using a persistent instance name. A persistent instance is relevant only to the shared data use case.

  IMPORTANT: New DataWarp software releases may require the re-creation of persistent instances.

When either type of instance is destroyed, DataWarp ensures that data needing to be written to the parallel file system (PFS) is written before releasing the space for reuse. In the case of a job instance, this can delay the completion of the job.

Application I/O

The DataWarp service (DWS) dynamically configures access to a DataWarp instance for all compute nodes assigned to a job using the instance. Application I/O is forwarded from compute nodes to the instance's DataWarp server nodes using the Cray Data Virtualization Service (DVS), which provides POSIX based file system access to the DataWarp storage.

A DataWarp instance is configured as scratch, cache, or swap. For scratch instances, all data staging between the instance and the PFS is explicitly requested using the DataWarp job script staging commands or the application C library API (`libdatawarp`). For cache instances, all data staging between the cache instance and the PFS occurs implicitly. For swap instances, each compute node has access to a unique swap instance that is distributed across all server nodes.

Scratch Configuration I/O

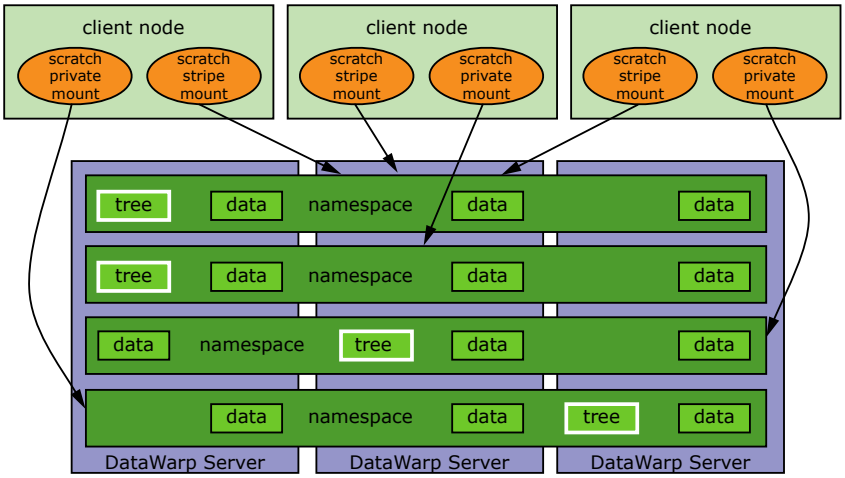
A scratch configuration is accessed in one or more of the following ways:

- **Striped**: In striped access mode individual files are striped across multiple DataWarp server nodes (aggregating both capacity and bandwidth *per file*) and are accessible by all compute nodes using the instance.
- **Private**: In private access mode individual files are also striped across multiple DataWarp server nodes (also aggregating both capacity and bandwidth *per file*), but the files are accessible only to the compute node that created them (e.g., /tmp). Private access is not supported for persistent instances, because a persistent instance is usable by multiple jobs with different numbers of compute nodes.
- **Load balanced**: (deferred implementation) In load balanced access mode individual files are replicated (read only) on multiple DataWarp server nodes (aggregating bandwidth but not capacity *per instance*) and compute nodes choose one of the replicas to use. Load balanced mode is useful when the files are not large enough to stripe across a sufficient number of nodes.

There is a separate file namespace for every scratch instance (job and persistent) and access mode (striped, private, loadbalanced) except persistent/private is not supported. The file path prefix for each is provided to the job via environment variables; see the .

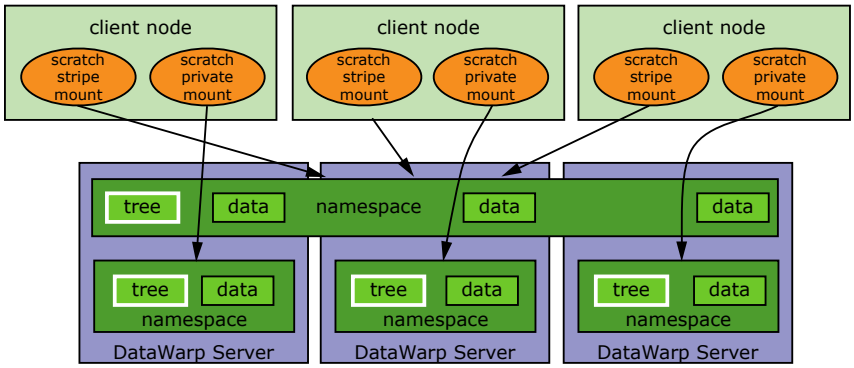
The following diagram shows a scratch private and scratch stripe mount point on each of three compute (client) nodes in a DataWarp installation configured with default settings for CLE 6.0.UP01; where `tree` represents which node manages metadata for the namespace, and `data` represents where file data may be stored. For scratch private, each compute node reads and writes to its own namespace that spans all allocated DataWarp server nodes, giving any one private namespace access to all space in an instance. For scratch stripe, each compute node reads and writes to a common namespace, and that namespace spans all three DataWarp nodes.

Figure 4. Scratch Configuration Access Modes (with Default Settings)



The following diagram shows a scratch private and scratch stripe mount point on each of three compute (client) nodes in a DataWarp installation where the scratch private access type is configured to not behave in a striped manner (`scratch_private_stripe=no` in the `dwsd.yaml` configuration file). That is, every client node that activates a scratch private configuration has its own unique namespace on only one server, which is restricted to one fragment's worth of space. This is the default for CLE 5.2.UP04 and CLE 6.0.UP00 DataWarp. For scratch stripe, each compute node reads and writes to a common namespace, and that namespace spans all three DataWarp nodes. As in the previous diagram, `tree` represents which node manages metadata for the namespace, and `data` represents where file data may be stored.

Figure 5. Scratch Configuration Access Modes (with `scratch_private_stripe=no`)



Cache Configuration I/O

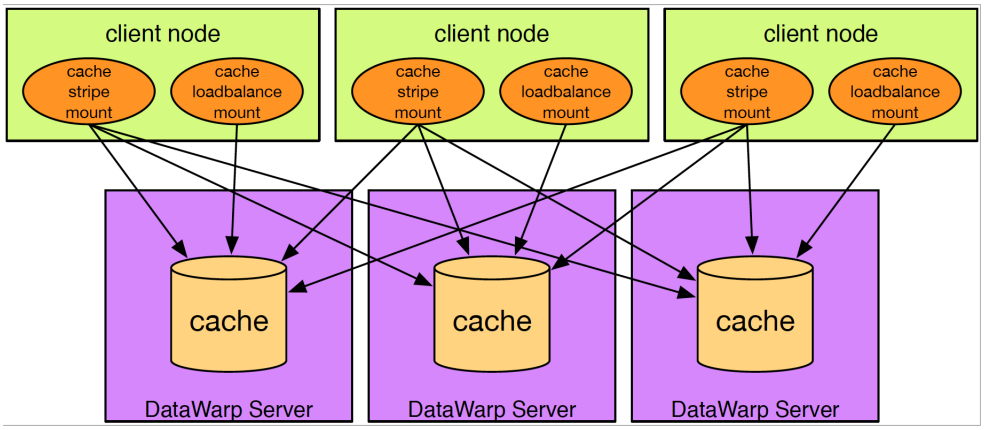
A cache configuration is accessed in one or more of the following ways:

- **Striped**: in striped access mode all read/write activity performed by all compute nodes is striped over all DataWarp server nodes.
- **Load balanced** (read only): in load balanced access mode, individual files are replicated on multiple DataWarp server nodes (aggregating bandwidth but not capacity *per instance*), and compute nodes choose one of the replicas to use. Load balanced mode is useful when the files are not large enough to stripe across a sufficient number of nodes or when data is only read, not written.

There is only one namespace within a cache configuration; that namespace is essentially the user-provided PFS path. Private access it is not supported for cached instances because all files are visible in the PFS.

The following diagram shows a cache stripe and cache loadbalance mount point on each of three compute (client) nodes.

Figure 6. Cache Configuration Access Modes



4 Check the Status of DataWarp Resources

Prerequisites

The `dws` module must be loaded:

```
$ module load dws
```

> **TIP:** On external login nodes (eLogin), the `eswrap` service may be configured for `dwstat`, in which case, the `dws` module should not be loaded. The following message is displayed if this command collision occurs:
>
> ```
> Cannot determine gateway via libdws_thin
> fatal: Cannot find a valid api host to connect to or no config file found.
> ```
>
> This is fixed by removing the `dws` module from the shell environment:
>
> ```
> elogin> module unload dws
> ```

The `dwstat` command

To check the status of various DataWarp resources, invoke the `dwstat` command, which has the following format:

```
dwstat [-h] [unit_options] [RESOURCE [RESOURCE]...]
```

Where:

`unit_options`

Includes a number of options that determine the SI or IEC units with which output is displayed. See the `dwstat(1)` man page for details.

`RESOURCE`

May be: `activations`, `all`, `configurations`, `fragments`, `instances`, `most`, `namespaces`, `nodes`, `pools`, `registrations`, or `sessions`.

By default, `dwstat` displays the status of pools:

```
$ dwstat
    pool units quantity    free   gran
wlm_pool bytes        0       0   1GiB
 scratch bytes  7.12TiB 2.88TiB 128GiB
  mypool bytes        0       0 1 6MiB
```

In contrast, `dwstat all` reports on all resources for which it finds data:

```
    pool units quantity     free   gran
wlm_pool bytes 14.38TiB 13.88TiB 128GiB
```

```
  sess state token     creator owner             created expiration nodes
     4 CA---  1527 MOAB-TORQUE  1226 2016-09-19T21:16:12      never     0
     7 CA---  1534 MOAB-TORQUE  1226 2016-09-19T23:53:17      never     0
   138 CA---  1757 MOAB-TORQUE   827 2016-09-29T14:46:09      never     0
   139 CA---  1759 MOAB-TORQUE 10633 2016-09-29T16:06:26      never    32

  inst state sess  bytes nodes             created expiration intact  label public confs
     4 CA---    4 128GiB     1 2016-09-19T21:16:12      never   true   I4-0  false     1
     7 CA---    7 128GiB     1 2016-09-19T23:53:17      never   true   I7-0  false     1
   138 CA---  138 128GiB     1 2016-09-29T14:46:09      never   true I138-0  false     1
   139 CA---  139 128GiB     1 2016-09-29T16:06:26      never   true I139-0  false     1

  conf state inst    type activs
     4 CA---    4 scratch      0
     7 CA---    7 scratch      0
   138 CA---  138 scratch      0
   139 CA---  139 scratch      0

  reg state sess conf wait
    4 CA---    4    4 true
    7 CA---    7    7 true
  137 CA---  139  139 true

  frag state  nst capacity     node
    10  CA--    4   128GiB nid00350
    15  CA--    7   128GiB nid00350
   180  CA--  138   128GiB nid00350
   181  CA--  139   128GiB nid00350

  nss state conf frag span
    4  CA--    4   10    1
    7  CA--    7   15    1
  138  CA--  138  180    1
  139  CA--  139  181    1

    node     pool online drain  gran capacity insts activs
nid00322 wlm_pool   true false  8MiB  5.82TiB     0      0
nid00349 wlm_pool   true false  4MiB  1.46TiB     0      0
nid00350 wlm_pool   true false 16MiB  7.28TiB     4      0

did not find any cache configurations, swap configurations, activations
```

For further information, see the `dwstat(1)` man page.

5 DataWarp Job Script Commands

In addition to workload manager (WLM) commands, the job script file passed to the WLM submission command (e.g., `qsub`, `msub`) can include DataWarp commands that are treated as comments by the WLM and passed to the DataWarp infrastructure. They provide the DataWarp Service (DWS) with information about the DataWarp resources a job requires. The DataWarp job script commands start with the characters `#DW` and include:

- `#DW jobdw` - Create and configure access to a DataWarp job instance
- `#DW persistentdw` - Configure access to an existing persistent DataWarp instance
- `#DW stage_in` - Stage files into the DataWarp instance at job start
- `#DW stage_out` - Stage files from the DataWarp instance at job end
- `#DW swap` - Create swap space for each compute node in a job

Each `#DW` job script command can span multiple lines by placing a backslash (`\`) at the end of one line and `#DW` at the beginning of the next. For example:

```
#DW jobdw type=scratch \
#DW       access_mode=striped
```

5.1 #DW jobdw - Job Script Command

NAME

`#DW jobdw` - Create and configure a DataWarp job instance

SYNOPSIS

```
#DW jobdw access_mode=mode[(MODIFIERS)] capacity=n type=scratch|cache
          [max_mds=yes|no]
          [modified_threshold=N]
          [optimization_strategy=strategy]
          [pfs=path]
          [pool=poolname]
          [read_ahead=N:rasize]
          [sync_on_close=yes|no]
          [sync_to_pfs=yes|no]
          [write_window_multiplier=mult]
          [write_window_length=numsecs]
```

DESCRIPTION

Optional command to create and configure access to a DataWarp job instance with the specified parameters; it can appear only once in a job script.

IMPORTANT:

The possibility exists for a user program to unintentionally cause excessive activity to SSDs, which can diminish the lifetime of the devices. To mitigate this issue, the `#DW jobdw` command includes options that help the DataWarp service (DWS) detect when a program's behavior is anomalous and then react based on configuration settings.

Cray encourages users to implement SSD protection options to prevent unintentional activity that over utilizes the SSDs through excessive activity. Use of these options can prolong the lifetime of these devices. For further information, see *Use SSD Protection Settings* on page 32.

#DW jobdw `type` Argument

The `type` argument specifies how the DataWarp instance will function. Options are:

`scratch`

All data staging between a scratch instance and the parallel file system (PFS) is explicitly requested using DataWarp job script staging commands.

`cache`

All data staging between a cache instance and the PFS occurs implicitly.

Command Arguments and Options for Scratch Configurations

When `type` = `scratch`, the `#DW jobdw` command requires the following arguments:

`access_mode=striped|private[(MODIFIERS)]`

The compute node path to the instance storage is communicated via the following automatically-created environment variables:

- scratch striped access mode: `$DW_JOB_STRIPED`
- scratch private access mode: `$DW_JOB_PRIVATE`

Additionally, the `access_mode` option accepts the following modifiers:

`client_cache=yes|no`	Enable or disable client-side caching. Although many workloads can benefit from client-side caching because it can reduce the frequency and necessity of network operations, others will be negatively affected. In some cases (e.g., many compute nodes modifying a specific file simultaneously with this access mode) data corruption can occur. It is important to understand how client-side caching works before invoking this option.
`MFS=`*`mfs`*	For SSD protection: maximum size of any file in the access mode
`MFC=`*`mfc`*	For SSD protection: maximum number of files created in the access mode. For private access mode, each compute node can create up to that many files. Valid for `type` = `scratch` only.

capacity=*n*

Requested amount of space for the instance (MiB|GiB|TiB|PiB). The DataWarp Service (DWS) may round this value up to the nearest DataWarp allocation unit or higher to improve performance. Note that `optimization_strategy` influences how capacity is selected.

When `type` = `scratch`, the `#DW jobdw` command also accepts the following options:

max_mds=yes|no

Controls whether or not multiple MDS servers (up to the number of DataWarp servers assigned to the instance) are used in order to improve the metadata transaction rate. When enabled, a mount point is created for each metadata server. This is only effective if the application is written to make use of it by calling the `dw_get_mds_path` library function to decode which paths to use on the compute nodes. If not, `max_mds` creates the multiple mount points, but only one is used.

For further information, see the `dw_get_mds_path(3)` man page.

optimization_strategy=*strategy*

Specifies a preference for how space is chosen on server nodes. The chosen strategy is best effort; it is not guaranteed. The default is controlled by the `instance_optimization_default` parameter in `dwsd.yaml` and is modifiable by an administrator.

Strategy options are:

bandwidth (default)	Assign as many servers as possible (as determined by the capacity request, pool granularity and available space) to maximize bandwidth
interference	Assign as few servers as possible to minimize interference (e.g., sharing servers) from other jobs
wear	Assign servers with the least wear (i.e., most remaining endurance/ lifetime)

pool=*poolname*

Suggests which storage pool to use. This option is only supported by SLURM.

write_window_multiplier=*mult*

Number of times `capacity` number of bytes may be written in a period defined by `write_window_length`; default = `10`.

write_window_length=*numsecs*

Number of seconds to use when calculating the moving average of bytes written; default = `86,400` (24 hours).

Command Arguments and Options for Cache Configurations

When `type` = `cache`, the `#DW jobdw` command requires the following arguments:

access_mode=striped|ldbalance[(MODIFIERS)]

The compute node path to the instance storage is communicated via the following automatically-created environment variables:

- cache striped access mode: `$DW_JOB_STRIPED_CACHE`
- cache ldbalance access mode: `$DW_JOB_LDBAL_CACHE`

Additionally, the `access_mode` option accepts the following modifiers:

client_cache=yes|no	Enable or disable client-side caching. Although many workloads can benefit from client-side caching because it can reduce the frequency and necessity of network operations, others will be negatively affected. In some cases (e.g., many compute nodes modifying a specific file simultaneously with this access mode) data corruption can occur. It is important to understand how client-side caching works before invoking this option.
MFS=*mfs*	For SSD protection: maximum size of any file in the access mode

When `type` = `cache`, the `#DW jobdw` command also accepts the following options:

modified_threshold=*N*

Maximum amount of modified data (in bytes or MiB|GiB|TiB) cached per file before write back to PFS starts

- If `modified_threshold=0`, no maximum is set and modified data can be written back at any time; default = `256MiB`.
- If `modified_threshold=-1`, an infinite maximum is set and modified data will not be written back until a `close` or `sync` occurs or the cache is full.

optimization_strategy=*strategy*

Specifies a preference for how space is chosen on server nodes. The strategy chosen is best effort; it is not guaranteed. The default is controlled by the `instance_optimization_default` parameter in `dwsd.yaml` and is modifiable by an administrator.

Strategy options are:

bandwidth (default)	Assign as many servers as possible (as determined by the capacity request, pool granularity and available space) to maximize bandwidth
interference	Assign as few servers as possible to minimize interference (e.g., sharing servers) from other jobs
wear	Assign servers with the least wear (i.e., most remaining endurance/ lifetime)

pfs=*path*

Path to a directory on the parallel file system

pool=*poolname*

Suggests which pool to use. This option is only supported by Slurm.

read_ahead=*N*:*rasize*

`N` specifies the minimum amount of data (in bytes or MiB|GiB|TiB) read sequentially per stripe before read ahead starts; *`rasize`* specifies the amount (in bytes or MiB|GiB|TiB) to read ahead. Default is no read ahead.

sync_on_close=yes|no

Controls whether modified data should be flushed to the PFS on close; default = `no`.

sync_to_pfs=yes|no

Controls whether a POSIX `sync` or `fsync` request flushes to the PFS or just to DataWarp storage; default = `no`.

write_window_multiplier=*mult*

Number of times `capacity` number of bytes may be written in a period defined by `write_window_length`; default = `10`.

write_window_length=*numsecs*

Number of seconds to use when calculating the moving average of bytes written; default = `86,400` (24 hours).

NOTES

The `#DW jobdw` command can span multiple lines by placing a backslash (`\`) at the end of one line and `#DW` at the beginning of the next. For example:

```
#DW jobdw type=scratch \
#DW       access_mode=striped
```

5.2 #DW persistentdw - Job Script Command

NAME

`#DW persistentdw` - Configure access to an existing persistent DataWarp instance

SYNOPSIS

```
#DW persistentdw name=resname [client_cache=yes|no]
#DW persistentdw name=resname [type=type access_mode=mode[(MODIFIERS)]]
```

DESCRIPTION

Optional command to configure access to an existing persistent DataWarp instance (created through the WLM) with the specified parameters; it can appear multiple times in a job script.

The `#DW persistentdw` command requires the following argument:

name=*resname*

The name given when the persistent instance was created; valid values are anything in the `label` column of the `dwstat instances` command where the `public` value is also `true`.

Common Command Option

The `#DW persistentdw` command accepts the following option:

client_cache=yes|no

Enable or disable client-side caching. Although many workloads can benefit from client-side caching because it can reduce the frequency and necessity of network operations, others

can be negatively affected. It is important to understand how client-side caching works before invoking this option. Not valid with options `type` and `access_mode`.

Command Options for Persistent Scratch Configurations

When `type` = `scratch`, the following option must also be set:

`access_mode=striped[(MODIFIER)]`

Currently only striped access mode (files are striped across multiple DataWarp nodes) is valid for scratch configurations.

The compute node path to the instance storage is:

- scratch stripe access mode: `$DW_PERSISTENT_STRIPED_`*`resname`*

where *`resname`* is the name of the persistent instance.

Additionally, the `access_mode` option accepts the following optional modifier:

`client_cache=yes|no`	Enable or disable client-side caching. Although many workloads can benefit from client-side caching because it can reduce the frequency and necessity of network operations, others will be negatively affected. In some cases (e.g., many compute nodes modifying a specific file simultaneously with this access mode) data corruption can occur. It is important to understand how client-side caching works before invoking this option.

Command Options for Persistent Cache Configurations

When `type=cache`, the following option must also be set:

`access_mode=striped|ldbalance[(MODIFIER)]`

Valid access modes are:

`striped`	Files are striped across multiple DataWarp nodes.
`ldbalance`	Files are replicated on multiple DataWarp nodes; valid only for cache configurations.

The compute node path to the instance storage is as follows, where *`resname`* is the name of the persistent instance:

- cache striped access mode: `$DW_PERSISTENT_STRIPED_CACHE_`*`resname`*
- cache ldbalance access mode: `$DW_PERSISTENT_LDBAL_CACHE_`*`resname`*

Additionally, the `access_mode` option accepts the following optional modifier:

`client_cache=yes|no`	Enable or disable client-side caching. Although many workloads can benefit from client-side caching because it can reduce the frequency and necessity of network operations, others will be negatively affected. In some cases (e.g., many compute nodes modifying a specific file simultaneously with this access mode) data corruption can occur. It is important to understand how client-side caching works before invoking this option.

NOTES

The `#DW persistentdw` command can span multiple lines by placing a backslash (`\`) at the end of one line and `#DW` at the beginning of the next. For example:

```
#DW persistentdw type=scratch \
#DW              access_mode=striped
```

5.3 #DW stage_in - DataWarp Job Script Command

NAME

`#DW stage_in` - Stage files into a DataWarp scratch instance

SYNOPSIS

```
#DW stage_in destination=dpath source=spath type=type
             [tolerate_errors=yes|no|nerror]
```

DESCRIPTION

Optional command, currently valid for scratch configurations only, to stage files from a parallel file system (PFS) into an existing DataWarp instance at job start; it can appear multiple times in a job script. Missing files cause the job to fail.

The `#DW stage_in` command requires the following arguments:

destination=*dpath* Path of the DataWarp instance; `destination` must start with the exact string `$DW_JOB_STRIPED`, or `$DW_PERSISTENT_STRIPED_`*`resname`* if staging in to a persistent instance. Not valid when `type=list`.

source=*spath* The PFS path; it must be readable by the user.

type=*type* The type of entity for staging; options are:

- `directory` - `source` is a single directory to stage, including all files and sub-directories. All symlinks, other non-regular files, and hard linked files are ignored.
- `file` - `source` is a single file to stage. If the specified file is a directory, other non-regular file, or has hard links, the stage in fails.
- `list` - `source` is a file containing a list of files to stage (one file/destination pair per line); the `destination` parameter is not used. If a specified file is a directory, other non-regular file, or has hard links, the stage out fails.

  Additionally, the `list` file path must be accessible to the workload manager, wherever it runs. Valid locations are site dependent and certain workload manager configurations may be incompatible with the `list` option.

The `#DW stage_in` command also accepts the following option:

tolerate_errors=yes|no|*nerror*

Determines behavior if stage in operations fail. By default, stage in errors are not tolerated, causing the job to fail. Valid values for `tolerate_errors` are:

yes

Allow the job to continue although there are stage in failures. In this case, the job fails if the default maximum number of failures allowed (set by the administrator) is reached.

no

Stage in errors are not tolerated; the job fails (default).

nerror

Number of errors to tolerate (implicitly sets `tolerate_errors=yes`).

- If *nerror*=0, tolerate all stage in errors.
- If *nerror*>0, tolerate a maximum of *nerror* stage in errors before the job fails.

Note that an application can detect a stage in failure using one of the `libdatawarp` query stage functions.

NOTES

Each `#DW stage_in` command can span multiple lines by placing a backslash (\) at the end of one line and #DW at the beginning of the next. For example:

```
#DW stage_in destination=dpath \
#DW          source=spath \
#DW          type=type
```

5.4 #DW stage_out - Job Script Command

NAME

`#DW stage_out` - Stage files from a DataWarp instance

SYNOPSIS

```
#DW stage_out destination=dpath source=spath type=type
              [tolerate_errors=yes|no|nerror]
```

DESCRIPTION

Optional command to stage files from a DataWarp instance to the PFS at job end; can appear multiple times in a job script. Valid for scratch configurations only.

The `#DW stage_out` command requires the following arguments:

destination=*dpath* Path within the PFS; it must be writable by the user. Not valid with `type=list`.

source=*spath* Path within the DataWarp instance; `source` must start with the exact string `$DW_JOB_STRIPED`, or `$DW_PERSISTENT_STRIPED_`*`resname`* if staging out from a persistent instance.

type=*type* Specifies the type of entity for staging. Options are:

- **directory** `source` is a single directory to stage, including all files and sub-directories. All symlinks, other non-regular files, and hard linked files are ignored.
- **file** `source` is a single file to stage. If the specified file is a directory, other non-regular file, or has hard links, the stage out fails.
- **list** `source` is a file containing a list of files to stage (one file/destination pair per line); the `destination` parameter is not used. If a specified file is a directory, other non-regular file, or has hard links, the stage out fails.

  Additionally, the `list` file path must be accessible to the workload manager, wherever it runs. Valid locations are site dependent and certain workload manager configurations may be incompatible with the `list` parameter.

The `#DW stage_out` command also accepts the following option:

tolerate_errors=yes|no|*nerror* Determines behavior if stage out operations fail. By default, stage out errors are not tolerated, causing the job to fail. Valid values for `tolerate_errors` are:

- **yes**

  Allow the job to continue although there are stage out failures. In this case, the job fails if the default maximum number of failures allowed (set by the administrator) is reached.
- **no**

  Stage out errors are not tolerated; the job fails (default).
- ***nerror***

  Number of errors to tolerate (implicitly sets `tolerate_errors=yes`).

  - If *`nerror`*`=0`, tolerate all stage out errors.
  - If *`nerror`*`>0`, tolerate a maximum of *`nerror`* stage out errors before the job fails.

Note that an application can detect a stage out failure using one of the `libdatawarp` query stage functions.

NOTES

Each `#DW stage_out` command can span multiple lines by placing a backslash (\) at the end of one line and #DW at the beginning of the next. For example:

```
#DW stage_out destination=dpath \
#DW           source=spath \
#DW           type=type
```

5.5 #DW swap - Job Script Command

NAME

`swap` - Configure swap space per compute node

SYNOPSIS

```
#DW swap n
```

DESCRIPTION

Optional command to configure *n* GiB of swap space per compute node assigned to the job; can appear only once in the job script. The job instance capacity must be large enough to provide *N* GiB of space to each node in the node list, or the job will fail.

`#DW swap` is only valid with `type` = `scratch`, and the swap space is shared with any other use of a scratch instance.

5.6 DataWarp Job Script Command Examples

For examples using DataWarp with Slurm, see *http://www.slurm.schedmd.com/burst_buffer.html*.

EXAMPLE: Job instance (type=scratch), no staging

Batch command:

```
% qsub -lmppwidth=3,mppnppn=1 job.sh
```

Job script `job.sh`:

```
#DW jobdw type=scratch access_mode=striped,private capacity=100TiB
aprun -n 3 -N 1 my_app $DW_JOB_STRIPED/sharedfile $DW_JOB_PRIVATE/scratchfile
```

Each compute node has striped/shared access to DataWarp via `$DW_JOB_STRIPED` and access to a per-compute node scratch area via `$DW_JOB_PRIVATE`. At the end of the job, the WLM runs a series of commands to initiate and wait for data staged out as well as to clean up any usage of the DataWarp resource.

EXAMPLE: Job instance (type=scratch), uses SSD write protection, no staging

Job script `job.sh`:

```
#DW jobdw type=scratch access_mode=striped(MFC=1000),private capacity=100TiB \
#DW       write_window_length=86400 write_window_multiplier=10
aprun -n 3 -N 1 $DW_JOB_STRIPED/sharedfile $DW_JOB_PRIVATE/scratchfile
```

This is the previous example with SSD write protection (see *Use SSD Protection Settings* on page 32) added. It specifies that the job may write 10 * 100TiB = 1PiB of data in any window of 86400 seconds (1 day). Over the entire batch job, only 1000 files can be re-created within the striped access mode. When either threshold is hit, continued violations result in either a log message to the system console, an IO error to the application process, or both. The error action is determined by a DataWarp configuration option.

EXAMPLE: Job instance (type=cache)

Job script `job.sh`

```
#DW jobdw type=cache access_mode=striped pfs=/lus/users/seymour \
#DW       modified_threshold=500MiB read_ahead=8MiB:2MiB sync_on_close=yes \
#DW       sync_to_pfs=yes capacity=100TiB
aprun -n 3 -N 1 ./a.out $DW_JOB_STRIPED_CACHE
```

DWS implicitly caches reads and writes to any files in `/lus/users/seymour` via the `$DW_JOB_STRIPE_CACHE` mount on computes. Write back starts when a file has at least 500MiB of modified data in the cache, or sooner if the cache fills up. Read ahead (in 2MiB chunks) starts after 8MiB of contiguous reads. The file is sync'd to the PFS on the last `close` and every `fsync` request.

EXAMPLE: Persistent instance

Creating persistent instances is done via the site-specific WLM. Each WLM has its own syntax for this, and it is beyond the scope of this guide to detail the various methods. The following examples are provided with the caveat that they may be out of sync with changes made by the WLM vendors. For details, see the appropriate WLM documentation.

Slurm: This example creates a persistent instance `persist1`.

```
#!/bin/bash
#SBATCH -n 1 -t 1
#BB create_persistent name=persist1 capacity=700GB access=striped type=scratch
```

Which results in:

```
$ dwstat most
    pool units quantity     free      gran
  kiddie bytes  5.82TiB  4.66TiB 397.44GiB
wlm_pool bytes 17.47TiB 16.69TiB 397.44GiB

sess state    token creator owner             created expiration nodes
9924 CA--- persist1     CLI 29993 2016-02-25T23:04:04      never     0

inst state sess     bytes nodes   created expiration intact    label public confs
3234 CA--- 9924 794.88GiB     2  23:04:04      never   true persist1   true     1
```

Each compute node has shared access to DataWarp via `$DW_PERSISTENT_STRIPED_`*`piname`* (scratch instances), `$DW_PERSISTENT_STRIPED_CACHE_`*`piname`* (cache instances), or `$DW_PERSISTENT_LDBAL_CACHE_`*`piname`* (cache instances) as described in *#DW persistentdw - Job Script Command* on page 20.

To remove the persistent instance (with or without the `hurry` option):

```
#!/bin/bash
#SBATCH -n 1 -t 1
#BB destroy_persistent name=persist1 hurry
```

See *http://www.slurm.schedmd.com/burst_buffer.html* for more Slurm examples.

Moab: The `ac_dw_admin_cli` command creates a persistent instance and has the following syntax:

```
$ ac_dw_admin_cli -h

Options:
  -c: Create a DW persistent instance
  -d: Diagnose user job requesting DW storage

Additional params for (-c) Create:
 Params: -n name, -u user, -S size, -p pool-name, -s start-time, -d duration
 Params from dw_create_persistent_instance: --type, --access_mode, --pfs,
 --modified_threshold, --read_ahead, --sync_on_close, --sync_to_pfs

Additional params for (-d) Diagnose:
 Params: -j jobid, --logs-stagein, --logs-stageout, --logs-teardown
```

For example:

```
$ ac_dw_admin_cli -c -n dwname -u username -S 256GiB -p poolname -s +0:00:00:00 \
-d +1:00:00:00 --type scratch --access_mode striped
```

Each compute node has shared access to DataWarp via `$DW_PERSISTENT_STRIPED_`*`piname`* (scratch instances), `$DW_PERSISTENT_STRIPED_CACHE_`*`piname`* (cache instances), or `$DW_PERSISTENT_LDBAL_CACHE_`*`piname`* (cache instances) as described in *#DW persistentdw - Job Script Command* on page 20.

EXAMPLE: Staging

```
qsub -lmppwidth=128,mppnppn=32 job.sh
```

Job script `job.sh`

```
#DW jobdw type=scratch access_mode=striped capacity=100TiB
#DW stage_in type=directory source=/pfs/dir1 destination=$DW_JOB_STRIPED/dir1
#DW stage_in type=list source=/pfs/inlist
#DW stage_in type=file source=/pfs/file1 destination=$DW_JOB_STRIPED/file1
#DW stage_out type=directory destination=/pfs/dir1 source=$DW_JOB_STRIPED/dir1
#DW stage_out type=list source=/pfs/inlist
#DW stage_out type=file destination=/pfs/file1 source=$DW_JOB_STRIPED/file1

aprun -n 128 -N 32 my_app $DW_JOB_STRIPED/file1
```

EXAMPLE: Compute node swap

Job script `job.sh`:

```
#DW jobdw type=scratch access_mode=striped capacity=100GiB
#DW swap 10GiB
#Supports up to 10 compute nodes in this case
aprun -n 10 -N 1 big_memory_application
```

Each compute node has striped access to a unique swap instance (10GiB) via `$DW_JOB_STRIPED`.

EXAMPLE: Interactive PBS job with DataWarp job instance

```
qsub -I -lmppwidth=3,mppnppn=1 job.sh
```

Job script `job.sh`

```
#DW jobdw type=scratch access_mode=striped,private capacity=100TiB
```

For the interactive PBS job case, the job script file is only used to specify the DataWarp configuration - all other commands in the job script are ignored and job commands are taken from the interactive session same as for any interactive job. This allows the same job script to be used to configure DataWarp instances for both a batch and interactive job.

5.7 Diagrammatic View of Batch Jobs

These diagrams are graphs of how these batch jobs look and how the objects are linked with each other, as seen in `dwstat` output.

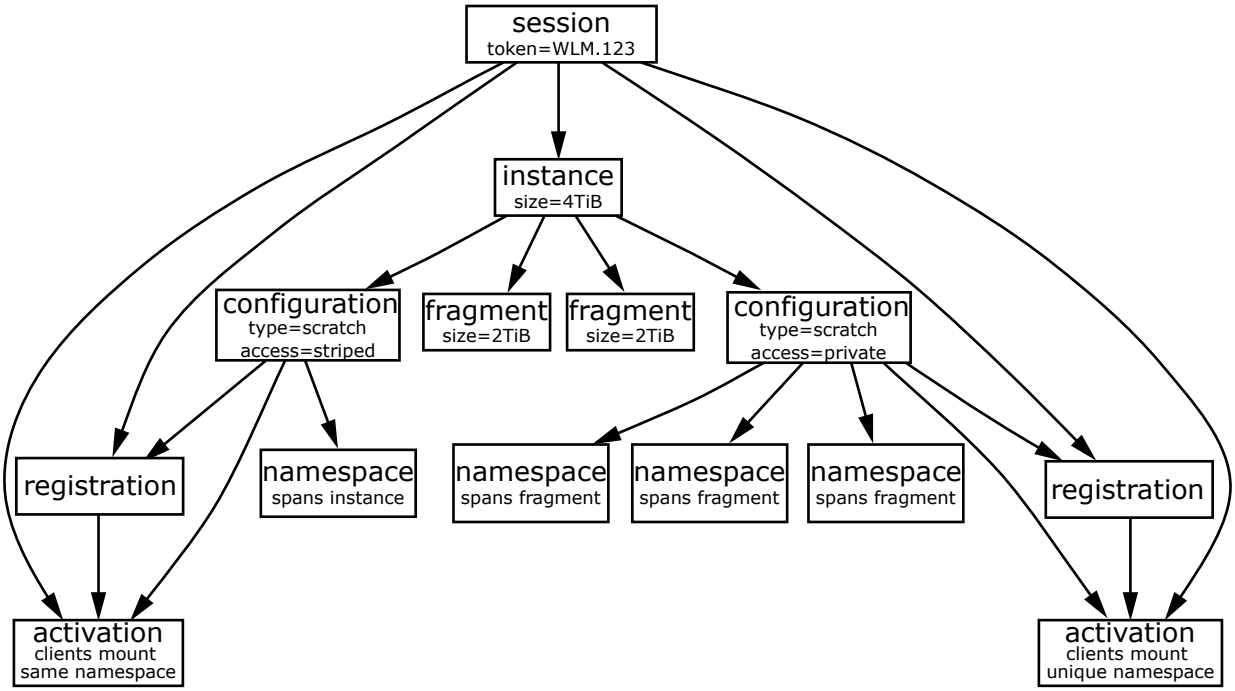
EXAMPLE: DataWarp job instance (type = scratch), no staging

The following diagram shows how the `#DW jobdw` request is represented in the DWS for a batch job in which a job instance is created, but no staging occurs. For this example, assume that the job gets three compute nodes and the batch job name is `WLM.123`.

```
#DW jobdw type=scratch access_mode=striped,private capacity=4TiB
```

If any of the referenced boxes are removed (e.g., `dwcli rm session --id` *id*), then all boxes that it points to, recursively, are removed. In this example, the scratch stripe configuration gets one namespace and the scratch private configuration gets three namespaces, one for each compute node. The 4TiB capacity request is satisfied by having an instance of size 4TiB, which in turn consists of two 2TiB fragments that exist on two separate DW servers.

Figure 7. Job Instance (type = scratch) with No Staging

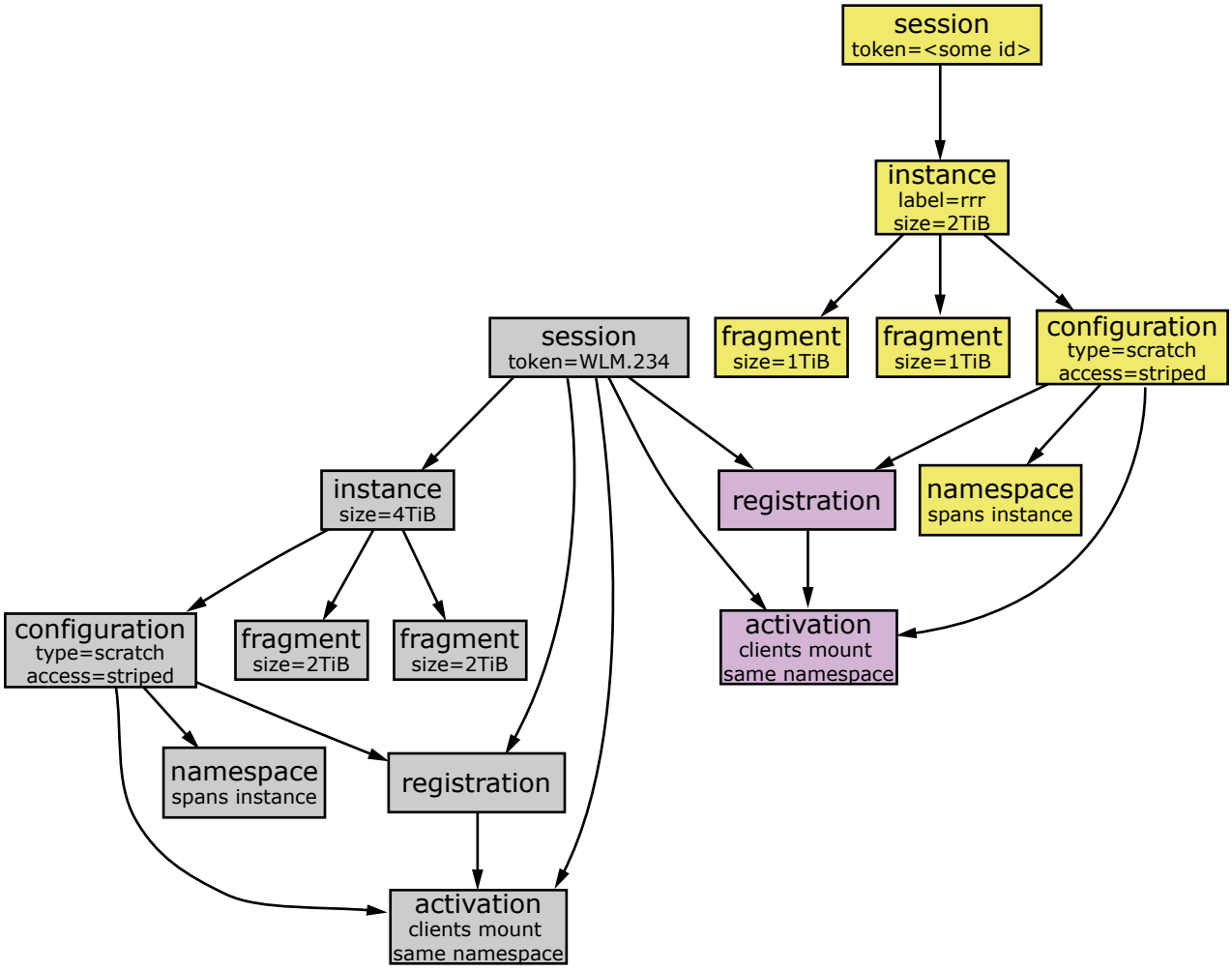


EXAMPLE: Use both job and persistent instances

The following diagram shows how the `#DW jobdw` request is represented in the DWS for a batch job in which both a job and persistent instance are created. For this example, assume that the existing persistent DataWarp instance `rrr` has a stripe configuration of 2TiB capacity and the batch job name is `WLM.234`.

```
#DW jobdw type=scratch access_mode=striped,private capacity=4TiB
#DW persistentdw name=rrr
```

Figure 8. Job and Persistent Instances (type = scratch)



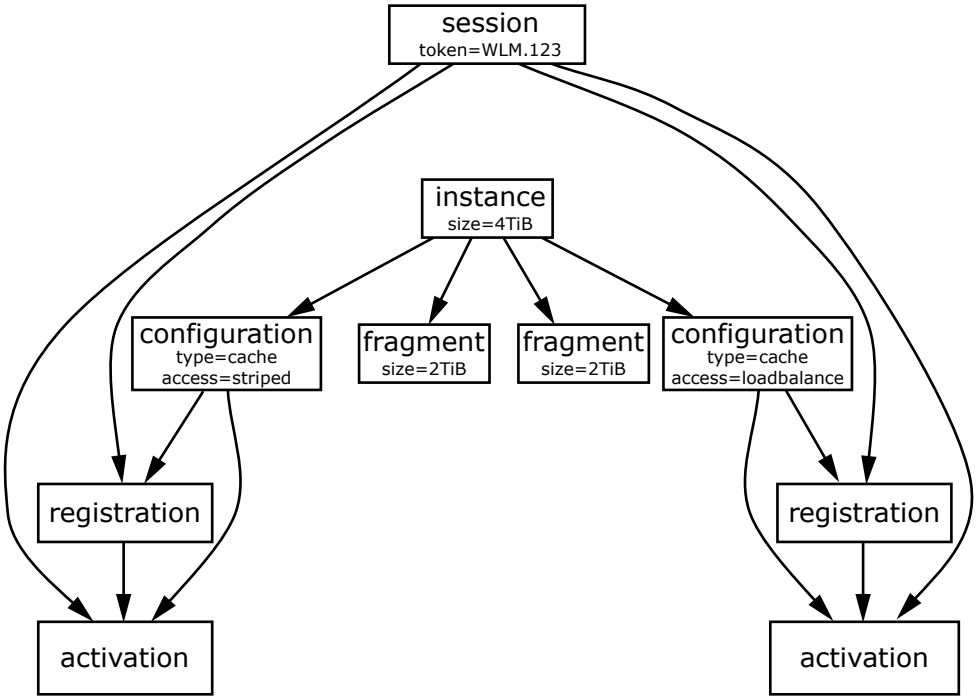
EXAMPLE: Job Instance for Cache Configuration

The following diagram shows how the `#DW jobdw` command is represented in the DWS for a batch job for a cache configuration.

```
#DW jobdw type=cache access_mode=stripe,ldbalance capacity=4TiB pfs=/lus/peel/
users/seymour
```

In this example, the cache stripe configuration and cache loadbalance configuration read and/or write to the files in the PFS at the `/lus/peel/users/seymour` path. The 4TiB capacity request is satisfied by having an instance of size 4TiB, which in turn consists of two 2TiB fragments that exist on two separate DataWarp servers.

Figure 9. Job Instance (type = cache)



6 Additional Considerations when Using DataWarp

6.1 DVS Client-side Caching can Improve DataWarp Performance

With the advent of DataWarp and faster backing storage, the overhead of network operations has become an increasingly large portion of overall file system operation latency. In this release, DVS provides the ability to cache both read and write data on a client node while preserving close-to-open coherency and without contributing to out-of-memory issues on compute nodes. Instead of using network communication for all read/write operations, DVS can aggregate those operations and reuse data already read by or written from a client. This can provide a substantial performance benefit for these I/O patterns, which typically bear the additional cost of network latency:

- small reads and writes
- reads following writes
- multiple reads of the same data

Client-side Write-back Caching may not be Suitable for all Applications

CAUTION: Possible data corruption or performance penalty!

Using the page cache may not provide a benefit for all applications. Applications that require very large reads or writes may find that introducing the overhead of managing the page cache slows down I/O handling. Benefit can also depend on write access patterns: small, random writes may not perform as well as sequential writes. This is due to pages being aggregated for write-back. If random writes do not access sequential pages, then less-than-optimal-sized write-backs may have to be issued when a break in contiguous cache pages is encountered.

More important, successful use of write-back caching on client nodes requires a clear understanding and acceptance of the limitations of close-to-open coherency. It is important for site system administrators to ensure that users at their site understand how client-side write-back caching works before enabling it. Without that understanding, users could experience data corruption issues.

For detailed information about DVS client-side caching, see *XC™ Series DVS Administration Guide* (S-0005).

6.2 Use SSD Protection Settings

The possibility exists for a user program to unintentionally cause excessive activity to SSDs, and thereby diminish the lifetime of the devices. To mitigate this issue, DataWarp includes both administrator-defined configuration options and user-specified job script command options that help the DataWarp service (DWS) detect when a program's behavior is anomalous and then react based on configuration settings.

Job Script Command Options

The `#DW jobdw` job script command provides users with options for the following DataWarp SSD protection features:

- write tracking
- File creation limits
- File size limits

Users are encouraged to implement the following options to prevent unintentional activity that over utilizes the SSDs through excessive writes. Use of these options can prolong the lifetime of these devices. The `#DW jobdw` SSD protection options are:

`write_window_multiplier=`*`mult`*

Number of times `capacity` number of bytes may be written in a period defined by `write_window_length`; default = `10`.

`write_window_length=`*`numsecs`*

Number of seconds to use when calculating the moving average of bytes written; default = `86,400` (24 hours).

Example 1: This `#DW jobdw` command indicates that the user may write up to 10 * 222GiB in any 10 second rolling window:

```
#DW jobdw type=scratch access_mode=striped capacity=222GiB \
#DW       write_window_length=10 write_window_multiplier=10
```

Example 2: This `#DW jobdw` command indicates that the user does not require files greater than 16777216 bytes, and does not intend to create more than 12 files:

```
#DW jobdw type=scratch access_mode=striped(MFS=16777216,MFC=12) capacity=222GiB
```

For further information regarding the `#DW jobdw` command and the SSD protection options, see *#DW jobdw - Job Script Command* on page 16 and *DataWarp Job Script Command Examples* on page 25.

7 libdatawarp - the DataWarp API

`libdatawarp` is a C library API for use by applications to control the staging of data to/from a DataWarp configuration, and to query staging and configuration data.

The behavior of the explicit staging APIs is affected by the DataWarp access mode. For this release, `libdatawarp` supports explicit staging in and out only on DataWarp configurations of type scratch for striped or private access modes. Batch jobs, however, only support staging in and out for striped access mode.

- For striped access mode, any rank can call the APIs and all ranks see the effects of the API call. If multiple ranks on any node stage the same file concurrently, all but the first will get an error indicating a stage is already in progress. The actual stage will run in parallel on one or more DW nodes depending on the size of the file and number of DW nodes assigned.

  IMPORTANT: Before compiling programs that use `libdatawarp`, load the `datawarp` module.

  ```
  $ module load datawarp
  ```

API Routines

The `libdatawarp` routines and a brief description of their functionality are listed in the following table. For complete details of a specific routine, see its man page (e.g., `dw_stage_file_in(3)`).

Table 1. libdatawarp Routines

Routine	Function
`dw_get_mds_path`	Returns the MDS path
`dw_get_stripe_configuration`	Returns the current stripe configuration for a file
`dw_query_directory_stage`	Queries all files within a directory and all subdirectories
`dw_query_file_stage`	Queries stage operations for a DataWarp file
`dw_query_list_stage`	Queries stage operations for all files within a list
`dw_set_stage_concurrency`	Sets the maximum number of concurrent stage operations
`dw_stage_directory_in`	Stages all regular files from a PFS directory into a DataWarp directory
`dw_stage_directory_out`	Stages all regular files in a DataWarp directory to a PFS directory
`dw_stage_file_in`	Stage a PFS file into a DataWarp file
`dw_stage_file_out`	Stages from a DataWarp file into a PFS file

Routine	Function
dw_stage_list_in	Stages all regular PFS files within a list into a DataWarp directory
dw_stage_list_out	Stages all DataWarp files within a list into a PFS directory
dw_terminate_directory_stage	Terminates one or more in-progress or waiting stage operations
dw_terminate_file_stage	Terminates an in-progress or waiting stage operation
dw_terminate_list_stage	Terminates one or more in-progress or waiting stage operations (within a list)
dw_wait_directory_stage	Waits for one or all stage operations to complete
dw_wait_file_stage	Waits for a stage operation to complete for a target file
dw_wait_list_stage	Waits for one or all stage operations within a list to complete
dw_open_failed_stage, dw_read_failed_stage, dw_close_failed_stage	Used in combination to identify failed stages

Example

The following C program uses several of the API routines found in `libdatawarp`.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/limits.h>

#include <datawarp.h>

/* build with:
 *   gcc dirstageandwait.c -o dirstageandwait `pkg-config --cflags \
 *     --libs cray-datawarp`
 */

int main(int argc, char **argv)
{
   int ret;
   int comp, pend, defer, fail;

   if (argc != 4) {
      printf("Error: Expected usage:  \n"
         "%s [in | out | defer | revoke | terminate] [dw dir] [PFS dir]\n",
            argv[0]);
```

```
      return 0;
   }

   /* perform stage in */
   if (strcmp(argv[1], "in") == 0) {
      ret = dw_stage_directory_in(argv[2], argv[3]);a
   /* perform stage out */
   } else if (strcmp(argv[1], "out") == 0) {
        ret = dw_stage_directory_out(argv[2], argv[3], DW_STAGE_IMMEDIATE);
   /* mark files as deferred stage */
   } else if (strcmp(argv[1], "defer") == 0) {
        ret = dw_stage_directory_out(argv[2], argv[3], DW_STAGE_AT_JOB_END);
   /* revoke deferred stage tag */
   } else if (strcmp(argv[1], "revoke") == 0) {
        ret = dw_stage_directory_out(argv[2], NULL, DW_REVOKE_STAGE_AT_JOB_END);
   /* cancel an in progress or deferred stage */
   } else if (strcmp(argv[1], "terminate") == 0) {
        ret = dw_terminate_directory_stage(argv[2]);
   } else {
        printf("%s:  invalid option - %s\n", argv[0], argv[1]);
        return 0;
   }

   if (ret != 0) {
      printf("%s: dw_stage_file error - %d %s\n", argv[0], ret,
        strerror(-ret));
      return ret;
   }

   printf("%s:  STAGE SUCCESS!\n", argv[0]);

   /* wait for stage request to complete */
   ret = dw_wait_directory_stage(argv[2]);
   if (ret != 0) {
      printf("%s: dw_wait_dir_stage error %d %s\n", argv[0], ret,
        strerror(-ret));
      return ret;
   }

   /* query final stage state of dw target */
   ret = dw_query_directory_stage(argv[2], &comp, &pend, &defer, &fail);
   if (ret != 0) {
      printf("%s: query_file_stage error %d %s\n", argv[0], ret, strerror(-ret));
      return ret;
   }

   printf("%s: Wait and query complete:  complete %d pending %d defer %d
     failed %d\n", argv[0], comp, pend, defer, fail);

   return 0;
}
```

8 Troubleshooting

8.1 Why Do `dwcli` and `dwstat` Fail?

The `dwcli` and `dwstat` commands fail for a variety of reasons, some of which are described here.

1. Both commands fail if the DataWarp service is not configured or not up and running.

```
> dwstat
Cannot determine gateway via libdws_thin
fatal: Cannot find a valid api host to connect to or no config file found.
```

Fix: Contact site support personnel.

2. Both commands fail if the `dws` module is not loaded. See item *4* on page 37 if executing on an external login node (eLogin).

```
> dwstat
If 'dwstat' is not a typo you can use command-not-found to lookup the package
that contains it, like this:
cnf dwstat
```

Fix: load the module and try again.

```
> module load dws
> dwstat
      pool units quantity     free   gran
   wlm_pool bytes 53.12TiB 16.74TiB   1GiB
```

3. Both commands fail if the DataWarp scheduler daemon goes offline.

```
> dwstat
cannot communicate with dwsd daemon at sdb-hostname port 2015
[Errno 111] Connection refused
```

Fix: Contact site support personnel.

4. Both commands fail when executed by a user on an external login node (eLogin) on which the `eswrap` service has been configured for `dwcli` and `dwstat` after loading the `dws` module.

```
elogin> module load dws
dwstat
Cannot determine gateway via libdws_thin
fatal: Cannot find a valid api host to connect to or no config file found.
```

Fix: Determine if `dwstat`/`dwcli` are among the available wrapped commands, and if so, remove the `dws` module from the shell environment.

```
elogin> eswrap
eswrap version 2.0.3
...
     Valid commands:
...
         dwstat
         dwcli
...
elogin> module unload dws
dwstat
       pool units quantity     free   gran
   wlm_pool bytes 53.12TiB 16.74TiB   1GiB
```

5. Both commands fail if the DataWarp configuration option `allow_dws_cli_from_computes` is set to `false` and one of the following is true:

   - the command is executed from a batch script
   - the command is executed from a compute node

   Both commands output an error message similar to the following:

   ```
   Connecting to https://dwrest-nodename yielded fatal error:
   [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:581)
   ```

   Fix: To have this functionality, the system administrator must change the configuration setting and restart DataWarp.

6. Depending on the options and actions invoked, `dwcli` can fail when `dwmd` is not functional.

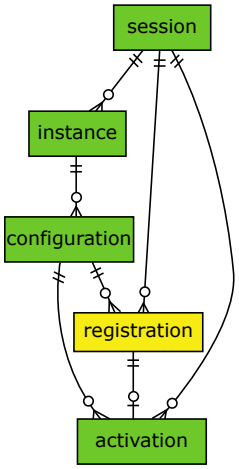
   ```
   > dwcli stage in -c 1 -s 1 --backing-path /etc/lvm/ --dir /test
   cannot communicate with backend dwmd daemon at datawarp port 49214
   [Errno 111] Connection refused
   ```

   Fix: Contact site support personnel.

9 Terminology

The following diagram shows the relationship between the majority of the DataWarp service terminology using Crow's foot notation. A session can have 0 or more instances, and an instance must belong to only one session. An instance can have 0 or more configurations, but a configuration must belong to only one instance. A registration belongs to only one configuration and only one session. Sessions and configurations can have 0 or more registrations. An activation must belong to only one configuration, registration and session. A configuration can have 0 or more activations. A registration is used by 0 or no activations. A session can have 0 or more activations.

Figure 10. DataWarp Component Relationships



Activation An object that represents making a DataWarp configuration available to one or more client nodes, e.g., creating a mount point.

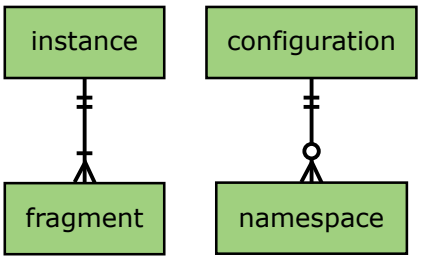
Client Node A compute node on which a configuration is activated; that is, where a DVS client mount point is created. Client nodes have direct network connectivity to all DataWarp server nodes. At least one parallel file system (PFS) is mounted on a client node.

Configuration A configuration represents a way to use the DataWarp space.

Fragment A piece of an instance as it exists on a DataWarp service node.

The following diagram uses Crow's foot notation to illustrate the relationship between an instance-fragment and a configuration-namespace. One instance has one or more fragments; a fragment can belong to only one instance. A configuration has 0 or more namespaces; a namespace can belong to only one configuration.

Figure 11. Instance/Fragment ↔ Configuration/Namespace Relationship



Instance — A specific subset of the storage space comprised of DataWarp fragments, where no two fragments exist on the same node. An instance is essentially raw space until there exists at least one DataWarp instance configuration that specifies how the space is to be used and accessed.

DataWarp Service — The DataWarp Service (DWS) manages access and configuration of DataWarp instances in response to requests from a workload manager (WLM) or a user.

Fragment — A piece of an instance as it exists on a DataWarp service node

Job Instance — A DataWarp instance whose lifetime matches that of a batch job and is only accessible to the batch job because the `public` attribute is not set.

Namespace — A piece of a scratch configuration; think of it as a folder on a file system.

Node — A DataWarp service node (with SSDs) or a compute node (without SSDs). Nodes with space are server nodes; nodes without space are client nodes.

Persistent Instance — A DataWarp instance whose lifetime matches that of possibly multiple batch jobs and may be accessed by multiple user simultaneously because the `public` attribute is set.

Pool — Groups server nodes together so that requests for capacity (instance requests) refer to a pool rather than a bunch of nodes. Each pool has an overall quantity (maximum configured space), a granularity of allocation, and a unit type. The units are either bytes or nodes (currently only bytes are supported). Nodes that host storage capacity belong to at most one pool.

Registration — A known usage of a configuration by a session.

Server Node — An IO service blade that contains two SSDs and has network connectivity to the PFS.

Session — An intagible object (i.e., not visible to the application, job, or user) used to track interactions with the DWS; typically maps to a batch job.

10 Prefixes for Binary and Decimal Multiples

Multiples of bytes						
SI decimal prefixes				IEC binary prefixes		
Name	**Symbol**	**Standard SI**	**Binary Usage**	**Name**	**Symbol**	**Value**
kilobyte	kB	10^3	2^{10}	kibibyte	KiB	2^{10}
megabyte	MB	10^6	2^{20}	mebibyte	MiB	2^{20}
gigabyte	GB	10^9	2^{30}	gibibyte	GiB	2^{30}
terabyte	TB	10^{12}	2^{40}	tebibyte	TiB	2^{40}
petabyte	PB	10^{15}	2^{50}	pebibyte	PiB	2^{50}
exabyte	EB	10^{18}	2^{60}	exbibyte	EiB	2^{60}
zettabyte	ZB	10^{21}	2^{70}	zebibyte	ZiB	2^{70}
yottabyte	YB	10^{24}	2^{80}	yobibyte	YiB	2^{80}

For a detailed explanation, including a historical perspective, see *http://physics.nist.gov/cuu/Units/binary.html*.