XC™ Series DataWarp™ User Guide

(CLE 6.0.UP06)

S-2558

Contents

1 About the DataWarp User Guide

Scope and Audience

XC™ Series DataWarp™ User Guide (S-2558) covers DataWarp concepts, commands, and the API. Note that it also includes examples with 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.

This publication is intended for users of Cray XC™ series systems installed with DataWarp SSD cards.

Release Information

Publication Title	Date	Release
XC™ Series DataWarp™ User Guide	March 2018	CLE 6.0.UP06

Changes to this document since CLE 6.0.UP05 are primarily due to the significant changes to the transparent cache functionality of DataWarp. These changes include the following new and updated topics:

- *Overview of the DataWarp Process for Cache Configurations* on page 13 is added, providing a birdseye view and explanation of a cache configuration.
- New topics added for the management of persistent instances: *#DW create_persistent Job Script Command* on page 26 and *#DW destroy_persistent Job Script Command* on page 31
- *Use DataWarp as Shared Storage* on page 7 is modified to use new #DW commands for persistent instances.
- Updates to *DataWarp Job Script Command Examples* on page 37 support the new cache file system.

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.

2 Quick Start to Using 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 Aries high speed network (HSN). I/O operations to this storage complete 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.

DataWarp storage is accessed through a site's workload manager (WLM) such as PBS, Moab, or 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 must be configured. Examples are provided for these common use cases:

- application scratch - DataWarp can provide storage that functions like a `/tmp` file system for each compute node in a job.
- 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.

Each WLM has its own job submission mechanism and syntax for requesting resources such as compute nodes. It is beyond the scope of this guide to detail all 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 that is to be modified to utilize DataWarp storage.

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.

Each WLM has its own job submission mechanism and syntax for requesting resources such as compute nodes. It is beyond the scope of this guide to detail all 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.

Procedure

1. Add a `#DW jobdw` command to the job script to define a scratch job instance (an occurrence of DataWarp storage available for the duration of the job) and how it will be accessed.

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

   Where:

   n

   Specifies the amount of DataWarp storage (MiB|GiB|TiB|PiB).

   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_STRIPED`.

   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`.

   In the following example, each compute node has striped/shared access to DataWarp via `$DW_JOB_STRIPED`.

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

2. (Optional) Add a `#DW stage_in` command to the job script to stage data from the PFS into DataWarp storage. This is only available when `access_mode=striped`.

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

   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.

   spath

   Specifies the PFS path to the directory|file|list. *spath* must be readable by the user.

   dpath

   Specifies the path to the location within the DataWarp instance where the data is to be staged. *dpath* must start with the exact string `$DW_JOB_STRIPED` or `$DW_JOB_PRIVATE`, and must be followed by a file name, a subdirectory, or a subdirectory and file name.

   This example stages data from the PFS (`/pfs/mystuff/data`) to the file `input`, within the DataWarp instance.

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

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

At the end of the 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=spath destination=dpath
```

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.

spath

Specifies the path to the directory|file|list within the DataWarp instance. *spath* must start with the exact string `$DW_JOB_STRIPED` or `$DW_JOB_PRIVATE`.

dpath

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

In this example, the directory, `output`, within the DataWarp instance is staged to the PFS directory `/pfs/mystuff/runresults1`.

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

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

   The application must be written to accept such options.

   ```
   wlm_run_command app.out app_args_here
   ```

   Where `wlm_run_command` is the WLM-specific command for running a job, e.g., `srun` for SLURM.

   This SLURM example creates and accesses scratch space:

   ```
   #!/bin/bash
   #SBATCH -p regular
   #SBATCH -N 4
   #SBATCH -t 01:00:00
   #DW jobdw type=scratch access_mode=striped capacity=100TiB
   srun app.out $DW_JOB_STRIPED/tmp
   ```

It is not necessary to delete unwanted data left in a DataWarp job instance as it is automatically removed by the DataWarp service after the job completes.

2.2 Use DataWarp as Shared Storage

Prerequisites

This procedure assumes the existence of a successfully runnable job script that is to be modified to utilize DataWarp storage.

About this task

Multiple jobs can access the same files through a persistent DataWarp instance that persists after the end of the job that created it. File access is authenticated and authorized based on the POSIX file permissions of the individual files.

> **TIP:** With CLE 6.0.UP06, Cray introduces the `#DW create_persistent` and `#DW destroy_persistent` job script commands to enable users to control persistent instances. Because CLE and the various WLMs are released asynchronously, Cray cannot predict when all WLMs will fully support these two new commands. WLM-controlled persistent instance creation and destruction will continue to be supported at the discretion of each vendor. See the WLM-specific documentation for further information.

Each WLM has its own job submission mechanism and syntax for requesting resources such as compute nodes. It is beyond the scope of this guide to detail all 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.

Procedure

1. Create a second job script (`job2.sh`) to support the prerequisite successfully runnable job script (`job1.sh`). Add a `#DW create_persistent` command to `job2.sh` to define a persistent instance. In this release a persistent instance cannot be created and accessed through the same job.

   ```
   #DW create_persistent name=resname type=scratch|cache capacity=n
   ```

   Where
   n

   Specifies the amount of DataWarp storage (MiB|GiB|TiB|PiB)

   resname

   Specifies a unique name for the persistent instance

   type

   Specifies whether the instance is configured as scratch (all data movement between DataWarp and the parallel file system (PFS) is explicitly requested), or as cache configuration (all data movement between DataWarp and the PFS occurs implicitly).

2. Add a `#DW persistentdw` command to the original job script (`job1.sh`) to configure access to the persistent DataWarp instance.

   ```
   #DW persistentdw name=resname
   ```

   Where:
   resname

   The reservation name given when the persistent instance was created through the WLM.

   Access is granted if the user meets standard POSIX file permission requirements for the instance. All compute nodes have access to the persistent instance via the `$DW_JOB_STRIPED_CACHE_`*resname* environment variable.

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

The application must be written to accept such options.

```
wlm_run_command app.out app_args_here
```

Where `wlm_run_command` is the WLM-specific command for running a job, e.g., `srun` for SLURM.

4. Remember to remove the persistent instance when it is no longer needed in order to free up DataWarp space for other users. The persistent instance is deleted by its creator using the `#DW destroy_persistent` command, typically within a standalone job.

```
#DW destroy_persistent name=resname
```

Where:

resname

The reservation name given when the persistent instance was created through the WLM.

TIP: The persistent instance is destroyed as soon as the scheduler reads the batch job; therefore, use caution to ensure that it is not destroyed before all related jobs have completed.

This job script requests that persistent instance `shared_data` is created as a cache configuration:

```
job2.sh
#!/bin/sh
#DW create_persistent name=shared_data type=cache capacity=100TiB
...
```

This job script requests access to the persistent instance and accesses it via the application `my_app`.

```
job1.sh
#!/bin/sh
#WLM JOB DIRECTIVES HERE
...
#DW persistentdw name=shared_data
wlm_run_command my_app $DW_JOB_STRIPED_CACHE_share_data
```

This job script renoves the persistent instance when it is no longer needed:

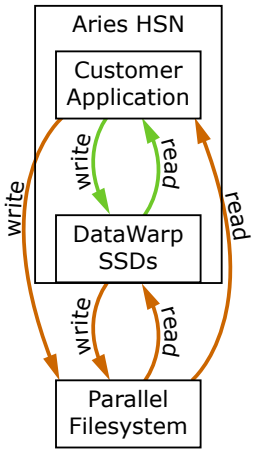
```
job3.sh
#!/bin/sh
#DW destroy_persistent resname=shared_data
```

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 Aries high speed network (HSN). I/O operations to this storage complete 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.

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 data transfer to and from a parallel file system.

Figure 1. DataWarp Overview



3.1 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 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:

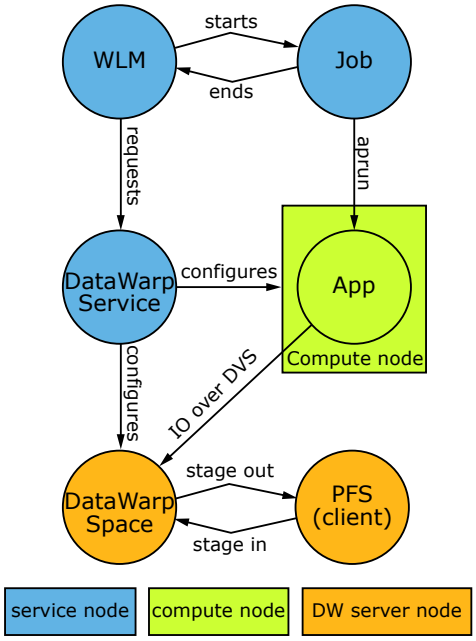
- **Shared input**: A read-only file or database (e.g., a bioinformatics database) used as input by multiple analysis jobs is copied from PFS to DataWarp and shared.
- **Ensemble analysis**: This is often a special case of the above **shared input** for a set of similar runs with different parameters 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.

3.2 Overview of the DataWarp Process for Scratch Configurations

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

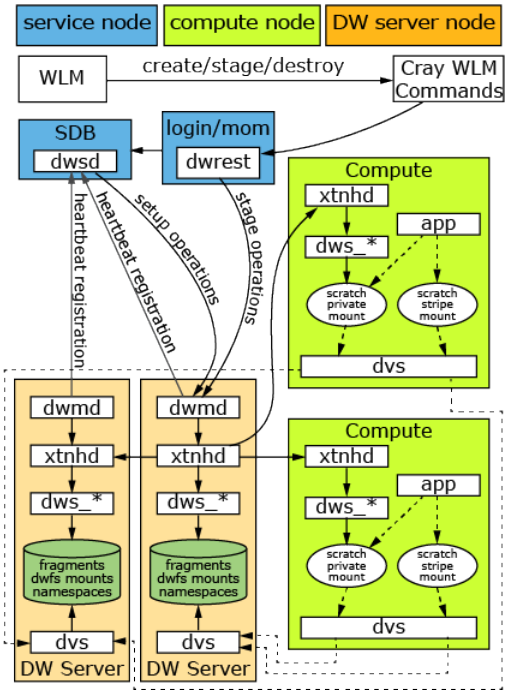


The diagram above provides a high level visual representation of the DataWarp process for scratch configurations, as described here.

1. A user submits a job to a workload manager (WLM). 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 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 WLM requests the needed capacity and passes along other user-supplied configuration and staging requests.
3. The DataWarp service (DWS) dynamically assigns the storage and initiates the stage in process.
4. After this completes, the WLM acquires other resources needed for the batch job, such as compute nodes.
5. After the compute nodes are assigned, the WLM and DWS 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 WLM stages out files, if requested, and performs cleanup. First, the WLM releases the compute resources and requests that the DWS make the previously accessible DataWarp configuration inaccessible to the compute nodes. Next, the WLM requests that additional files, if any, are staged out. When this completes, the WLM tells the DWS that the DataWarp storage is no longer needed.

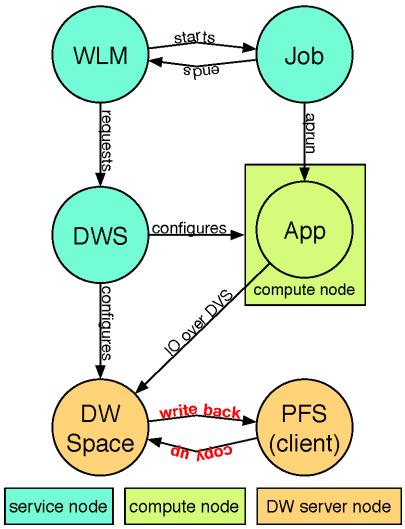
The following diagram includes extra details regarding the interaction between a WLM and the DWS as well as the location of the various DWS daemons.

Figure 3. DataWarp Component Interaction - detailed view



3.3 Overview of the DataWarp Process for Cache Configurations

Figure 4. DataWarp Component Interaction (Cache) - bird's eye view



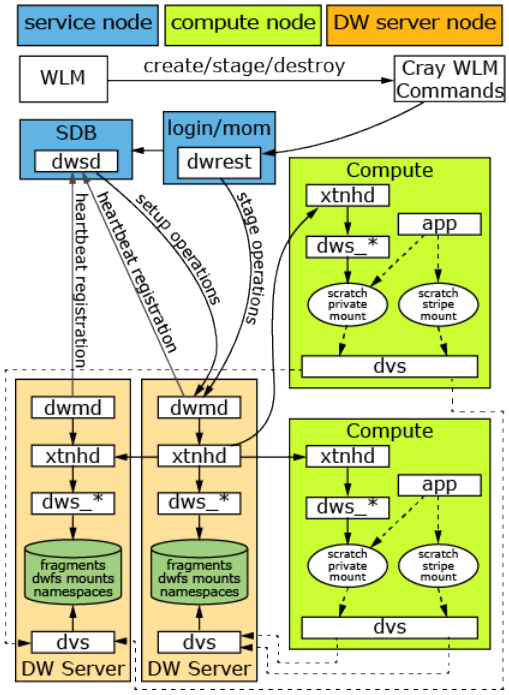
The diagram above provides a high level visual representation of the DataWarp process for cache configurations, as described here.

1. A user submits a job to a workload manager (WLM). Within the job submission, the user must specify the amount of DataWarp storage required and how the storage is to be configured.

2. The 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 WLM requests the needed capacity and passes along other user-supplied configuration requests.
3. The DataWarp service (DWS) dynamically assigns the storage.
4. After this completes, the WLM acquires other resources needed for the batch job, such as compute nodes.
5. After the compute nodes are assigned, the WLM and DWS 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 to read/write data as needed. Data movement occurs implicitly (i.e., copy up and write back) and no explicit requests are required.
7. When the batch job ends, the WLM releases the compute resources and requests that the DWS make the previously accessible DataWarp configuration inaccessible to the compute nodes. Dirty data is flushed to the PFS, after which the WLM performs cleanup. Next, the WLM requests that additional files, if any, are staged out. When this completes, the WLM tells the DWS 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 5. DataWarp Component Interaction - detailed view



3.4 DataWarp Concepts

For basic definitions, refer to *Terminology* on page 58.

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 and accessibility managed by whatever creates it. 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 transfer 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 transfer between the cache instance and the PFS occurs implicitly. For swap instances, each compute node has access to a unique swap file and swap files are distributed across all server nodes.

Scratch Configuration I/O

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

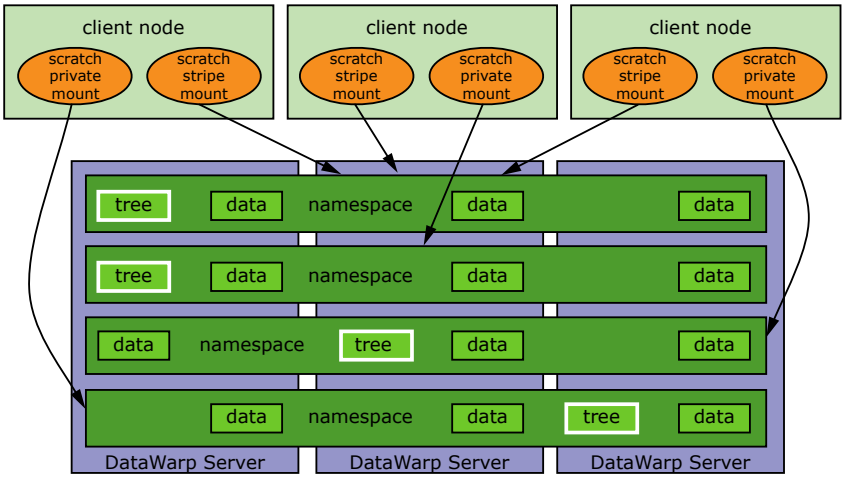
- **Striped**: In striped access mode individual files are striped across multiple DataWarp server nodes (aggregating both capacity and bandwidth *per file*) and are accessible by all compute nodes using the instance.
- **Private**: In private access mode individual files are also striped across multiple 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 *XC™ Series DataWarp™ User Guide*.

TIP: The default settings for scratch configuration access modes changed beginning with the CLE 6.0.UP01 release. This affects users whose systems have recently been upgraded from CLE 5.2.UP04 or CLE 6.0.UP00 to this release. The differences are pointed out in the following information and diagrams.

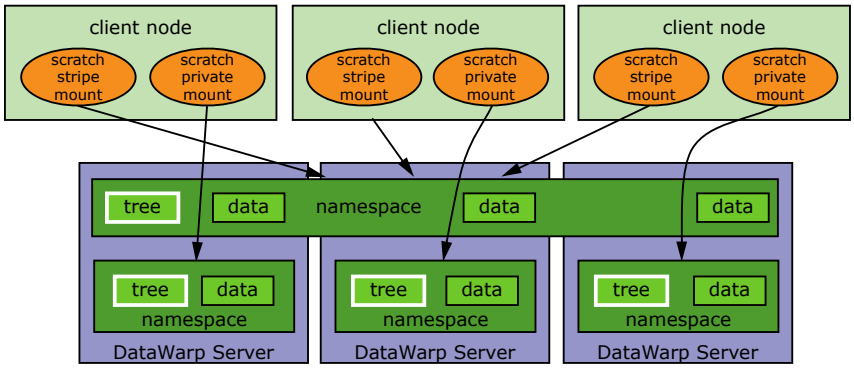
The following diagram shows a scratch private and scratch stripe mount point on each of three compute (client) nodes in a DataWarp installation configured with default settings; where `tree` represents which node manages metadata for the namespace, and `data` represents where file data may be stored. 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 6. 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 7. Scratch Configuration Access Modes (with `scratch_private_stripe=no`)



Cache Configuration I/O

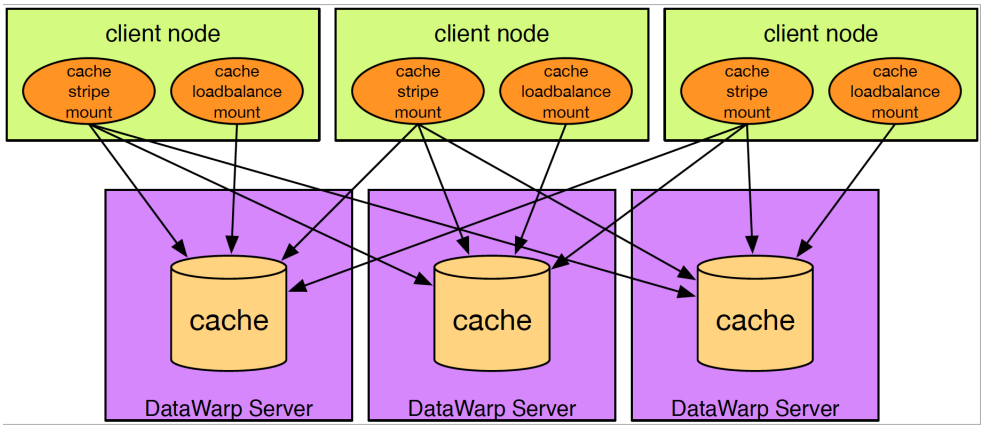
A cache configuration is accessed in one or more of the following ways:

- **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)**: (deferred implementation) 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 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 8. Cache Configuration Access Modes



3.5 DataWarp Limitations

For optimal use, it is important to understand DataWarp's limitations, including bugs, clarification on functionality, or warnings about the state of components.

General Limitations

1. The DataWarp service intentionally panics DataWarp server nodes to avoid returning corrupted data to a user.
2. SSD Write Protection
   a. I/O to swap files does not count against the SSD write protection policies.
   b. Violation of the write window policy is calculated on a per server basis. This can result in some processes seeing `EROFS` (read-only file system) errors while others see no errors when interacting with the DataWarp file system.

Scratch

The currently known caveats for scratch DataWarp are:

1. Scratch file system recovery
   - When a DataWarp server crashes, recovery is reliable (but not guaranteed) when all file system servers are rebooted. If only the crashed DataWarp server is rebooted, the file system may not function properly.
2. Load balance
   - Scratch load balance functionality is not yet implemented.
3. Staging
   - Staging through the WLM interface is susceptible to timeouts. See *Stage In or Out Fails When Transferring a Large Number of Files* for information on extending the timeout.
   - The explanations provided when stage failures occur are inadequate. Jobs with typos in the `#DW stage_in` and `#DW stage_out` syntax fail in the stage in or stage out phases, respectively, with no further explanation. Other staging failures may result in a message regarding `namespaces offline`.
4. Striping
   - The libdatawarp `dw_set_stripe_configuration()` functionality is not implemented and returns `ENOSYS`.
5. Limits
   - Scratch file systems may have a directory depth up to 1000 directories deep. DataWarp service management tasks are subject to various resource limits such as maximum open file descriptors. By default, the maximum number of open file descriptors per process is 1024. If a user creates a directory structure larger than 1000 directories deep, stage activity and tear down activity may fail. If deeper than 1000 directories deep is needed, the maximum number of open file descriptors per process limit can be increased on the DataWarp servers.

Cache

The currently known caveats for cache DataWarp are:

1. No SSD write protection support
2. No support for influencing `fsync()` and `close()` behavior
   a. Currently `fsync()` syncs to the SSDs
   b. Currently `close()` does not sync to the SSDs
3. No resiliency/recovery support - data is lost if a DataWarp server crashes or reboots.
   a. Impacted configurations display a blown fuse in `dwstat` output.
4. Transparent caching is only supported with Lustre at this time
5. Scratch-specific libdatawarp APIs have undefined behavior when used with transparent caching functionality and are not supported.
6. Only striped access mode is supported.
7. Users should not interact with a PFS file through multiple DataWarp cache instances.
   a. Transparent caching instances are not cache coherent with each other.

   b. Corruption may occur.

8. I/O to/from open unlinked files may fail. This is due to a Lustre behavior issue that our Lustre team is working on.

   a. Sometimes a server panics.

   b. Sometimes the user receives an error.

9. Opening a file with `O_DIRECT` does not work and results in an `-ENOTSUPP` error.

   a. It causes an `-ENOTSUPP` error during the first read/write.

10. Unexpected errors (e.g., `-ENOSPC`) when trying to write data back to the PFS may result in a panic to prevent returning corrupted data to a user.

11. Each server only uses a hard-coded 90% of requested capacity for caching.

   a. XFS performs poorly when free space is exhausted.

12. Only one cache configuration per instance lifetime is allowed.

   a. Multiple concurrent cache configurations on a single instance do not work and may cause DataWarp servers to crash.

   b. Creating a cache configuration on an instance, removing the configuration, and then creating a new one, fails. The configuration state displays a blown fuse.

13. The dwsd `equalize_fragments` option must be enabled. By default, `equalize_fragments` is enabled, and Cray recommends keeping it enabled.

14. Performance

   a. Transparent caching is not yet tuned for performance. Significant enhancements are forthcoming in patches and future releases.

4 Check the Status of DataWarp Resources

Prerequisites

The `dws` module must be loaded:

```
> module load 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
```

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

```
> dwstat all
    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 2017-05-19T21:16:12      never     0
     7 CA---  1534 MOAB-TORQUE  1226 2017-05-19T23:53:17      never     0
   138 CA---  1757 MOAB-TORQUE   827 2017-05-29T14:46:09      never     0
   139 CA---  1759 MOAB-TORQUE 10633 2017-05-29T16:06:26      never    32

  inst state sess  bytes nodes             created expiration intact  label  public confs
     4 CA---    4 128GiB     1 2017-05-19T21:16:12      never intact   I4-0 private     1
     7 CA---    7 128GiB     1 2017-05-19T23:53:17      never intact   I7-0 private     1
   138 CA---  138 128GiB     1 2017-05-29T14:46:09      never intact I138-0 private     1
   139 CA---  139 128GiB     1 2017-05-29T16:06:26      never intact I139-0 private     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 wait
    7 CA---    7    7 wait
  137 CA---  139  139 wait

  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

   ns 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 online  fill  8MiB  5.82TiB     0      0
nid00349 wlm_pool online  fill  4MiB  1.46TiB     0      0
nid00350 wlm_pool online  fill 16MiB  7.28TiB     4      0

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

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

4.1 Why Does the Free Capacity Displayed by `dwstat pools` not Match the Quantity Capacity?

There are several reasons the free capacity displayed by `dwstat pools` may not match the quantity capacity:

- One or more nodes in the pool is offline
- One or more nodes in the pool is marked drain
- The dwsd.yaml `equalize_fragments` and `equalize_fragments_guarantee` settings are both enabled

In earlier versions, free capacity indicated how much storage capacity was available for use in that pool at that moment in time. However, when the dwsd.yaml `equalize_fragments` and `equalize_fragments_guarantee` settings are both enabled (default settings beginning in CLE 6.0.UP05), free capacity takes on a new meaning, which is the largest capacity instance that can be requested using that pool at that moment in time. This difference leads to the following non-intuitive behavior when both settings are enabled.

- The free capacity may never equal the quantity capacity. For example, this happens if a pool has two nodes but the contributing capacity of each node is unequal.
- When an instance is created, the free capacity is reduced by up to the instance capacity size. In other words, after creating an instance, the free capacity may remain the same.
- When an instance is removed, the free capacity is increased by up to the instance capacity size. In other words, after removing an instance, the free capacity may remain the same.

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 create_persistent` - Create a persistent instance
- `#DW destroy_persistent` - Remove an existing persistent 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

(PBS Pro and Slurm only) 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.

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

IMPORTANT: For `type=cache`, ldbalance access mode is not available with this release.

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. The `#DW jobdw` job script command can create scratch or cache configurations.

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 50.

Required `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 following arguments are required:

`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.

Scratch Configuration Options

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)`]

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. As noted in the description, ldbalance access mode is not available with this release for `type=cache`.

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

Cache Configuration Options

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

modified_threshold=*N* (deferred implementation)

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* (deferred implementation)

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.

sync_on_close=yes|no (deferred implementation)

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

sync_to_pfs=yes|no (deferred implementation)

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

write_window_multiplier=*mult* (deferred implementation)

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

write_window_length=*numsecs* (deferred implementation)

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

NOTES

(PBS Pro and Slurm only) 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` spanning multiple lines:

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

5.2 #DW create_persistent Job Script Command

NAME

`#DW create_persistent` - Create a new persistent instance

SYNOPSIS

```
#DW create_persistent access_mode=mode][(MODIFIERS) capacity=n name=resname
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

IMPORTANT: For `type=cache`, ldbalance access mode is not available with this release.

Optional command to create a persistent DataWarp instance, that is, an instance with a lifetime not tied to the lifetime of any single job, and that can be accessed by any job satisfying POSIX file permissions. The `#DW create_persistent` command can only occur once within a job script and not in conjunction with the `#DW jobdw` command. A persistent instance can be configured as scratch or cache.

TIP: With CLE 6.0.UP06, Cray introduces the `#DW create_persistent` and `#DW destroy_persistent` job script commands to enable users to control persistent instances. Because CLE and the various WLMs are released asynchronously, Cray cannot predict when all WLMs will fully support these two new commands. WLM-controlled persistent instance creation and destruction will continue to be supported at the discretion of each vendor. See the WLM-specific documentation for further information.

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` and `#DW create_persistent` commands include 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 50.

Required `name` and `type` Arguments

All persistent instances must have unique names, specified using the `name` argument:

`name=resname`

This is the unique name used to create the persistent instance. `name` must be of length and characters that are compatible with environment variables, e.g., no spaces.

Use the `dwstat instances` command to view all instances. Existing persistent instance names are in the `label` column of public instances.

```
login> dwstat instances
  inst state sess  bytes nodes created expiration  intact   label  public confs
   753 CA---  832 128GiB     1     …          never  intact  mydata private     1
   807 CA---  903 256GiB     1     …          never  intact bigdata  public     1
```

In this example, `bigdata` is the name of the only persistent instance.

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 following arguments are required:

`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.

Scratch Configuration Options

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)]

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. As noted in the description, ldbalance access mode is not available with this release for `type=cache`.

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

Cache Configuration Options

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

modified_threshold=*N* (deferred implementation)

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* (deferred implementation)

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.

sync_on_close=yes|no (deferred implementation)

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

sync_to_pfs=yes|no (deferred implementation)

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

write_window_multiplier=*mult* (deferred implementation)

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

write_window_length=*numsecs* (deferred implementation)

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

NOTES

(PBS Pro and Slurm only) 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` spanning multiple lines:

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

5.3 #DW destroy_persistent Job Script Command

NAME

`#DW destroy_persistent` - Remove a persistent instance

SYNOPSIS

```
#DW destroy_persistent name=resname
```

DESCRIPTION

> **TIP:** With CLE 6.0.UP06, Cray introduces the `#DW create_persistent` and `#DW destroy_persistent` job script commands to enable users to control persistent instances. Because CLE and the various WLMs are released asynchronously, Cray cannot predict when all WLMs will fully support these two new commands. WLM-controlled persistent instance creation and destruction will continue to be supported at the discretion of each vendor. See the WLM-specific documentation for further information.

Optional command to remove an existing persistent instance. Only the creator of a persistent instance or a DataWarp administrator can remove it.

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

name=*resname*

This is the unique name used to create the persistent instance. `name` must be of length and characters that are compatible with environment variables, e.g., no spaces.

Use the `dwstat instances` command to view all instances. Existing persistent instance names are in the `label` column of public instances.

```
login> dwstat instances
  inst state sess  bytes nodes created expiration  intact   label  public confs
   753 CA---  832 128GiB     1    …          never  intact  mydata private     1
   807 CA---  903 256GiB     1    …          never  intact bigdata  public     1
```

In this example, `bigdata` is the name of the only persistent instance.

5.4 #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]
```

DESCRIPTION

WARNING: For `type=cache`, ldbalance access mode is not ready for use in production.

Optional command to request access to an existing persistent DataWarp instance; it can appear multiple times in a job script. Access is granted if the user meets standard POSIX file permission requirements for the instance. Each compute node has shared access to the persistent instance via `$DW_PERSISTENT_STRIPED_`*`resname`* or `$DW_PERSISTENT_STRIPED_CACHE_`*`resname`*, depending on whether the instance was created as scratch or cache, respectively.

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

name=*resname*

This is the unique name used to create the persistent instance. `name` must be of length and characters that are compatible with environment variables, e.g., no spaces.

Use the `dwstat instances` command to view all instances. Existing persistent instance names are in the `label` column of public instances.

```
login> dwstat instances
  inst state sess  bytes nodes created expiration  intact   label  public confs
   753 CA---  832 128GiB     1     …          never  intact  mydata private     1
   807 CA---  903 256GiB     1     …          never  intact bigdata  public     1
```

In this example, `bigdata` is the name of the only persistent instance.

Command Options

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`.

NOTES

(PBS Pro and Slurm only) 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` spanning multiple lines:

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

5.5 #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, only valid when `access_mode=striped`, 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. It must be followed by a file name, a subdirectory, or a subdirectory and file name. 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** When `type=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** When `type=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. `destination` must include a file name or subdirectory and file name. For example (command lines may wrap due to space limitations):

  ```
  #DW stage_in type=file source=spath destination=
  $DW_JOB_STRIPED/mydata
  ```

copies the specified source file into file `mydata` in the root of the DataWarp allocation, and

```
#DW stage_in type=file source=spath destination=
$DW_JOB_STRIPED/mydir/mydata
```

creates subdirectory `mydir` in the root of the DataWarp allocation and copies the specified source into file `mydata` in that subdirectory.

list

When `type=list`, `source` is a file containing a source and destination pair on each line. `destination` must include a file name or subdirectory and file name. For example:

```
spath $DW_JOB_STRIPED/mydata
```

copies the specified source file into the file `mydata` in the root of the DataWarp allocation, and

```
spath $DW_JOB_STRIPED/mydir/mydata
```

creates subdirectory `mydir` in the root of the DataWarp allocation and copies the specified source into file `mydata` in that subdirectory.

Additionally, the `list` file path must be accessible to the workload manager (WLM), wherever it runs. Valid locations are site dependent and certain WLM configurations may be incompatible with the `list` option. The command line `destination` parameter is not used. If a specified file is a directory, other non-regular file, or has hard links, the stage out fails.

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 following `libdatawarp` functions:

- `dw_query_file_stage()`

- `dw_query_directory_stage()`
- `dw_query_list_stage()`

Additionally, `libdatawarp` includes three `dw_failed_stage` functions that can be used in combination to identify failed stages:

- `dw_open_failed_stage()`
- `dw_close_failed_stage()`
- `dw_read_failed_stage()`

NOTES

(PBS Pro and Slurm only) 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` spanning multiple lines:

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

5.6 #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 in failure using one of the following `libdatawarp` functions:

- `dw_query_file_stage()`
- `dw_query_directory_stage()`
- `dw_query_list_stage()`

Additionally, `libdatawarp` includes three `dw_failed_stage` functions that can be used in combination to identify failed stages:

- `dw_open_failed_stage()`
- `dw_close_failed_stage()`
- `dw_read_failed_stage()`

NOTES

(PBS Pro and Slurm only) 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` spanning multiple lines:

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

5.7 #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.

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.

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

5.8 DataWarp Job Script Command Examples

TIP: In PDF format some of the lengthy `#DW` *command* lines wrap to the next line.

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`. The stripe configuration and the private configuration share

the full capacity of the instance. No other limits on block usage are imposed on either of them; either one can fill up the full 100TiB. 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
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 50) 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 capacity=100TiB
aprun -n 3 -N 1 ./a.out $DW_JOB_STRIPED_CACHE
```

DWS implicitly caches reads and writes to/from any files in `/lus/users/seymour` via the `$DW_JOB_STRIPED_CACHE` mount on compute nodes.

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=$DW_JOB_STRIPED/mydata/
dir1
#DW stage_in type=file source=/pfs/file1 destination=$DW_JOB_STRIPED/mydata/file1
#DW stage_in type=list source=/pfs/inlist
#DW stage_out type=directory source=$DW_JOB_STRIPED/mydir/dir1 destination=/pfs/
dir2
#DW stage_out type=file source=$DW_JOB_STRIPED/file1 destination=/pfs/file2
#DW stage_out type=list source=$DW_JOB_STRIPED/outlist

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

DWS stages:

- the PFS directory `/pfs/dir1`, including all of its files and sub-directories, in to the `/mydata/file1` directory with the DataWarp allocation
- the PFS file `/pfs/file1` in to the `/mydata/file1` file within the DataWarp allocation
- the PFS files defined in the list `/pfs/inlist` in to their designated destinations (as defined in the list) within the DataWarp instance

Each compute node has striped/shared access via `$DW_JOB_STRIPED`. When the job completes, DWS stages:

- the directory `/mydata/dir1` in the DataWarp allocation, including all of its files and sub-directories, out to the PFS directory `/pfs/dir2`
- the file `/mydata/file1` in the DataWarp allocation out to the PFS file `/pfs/file2`
- the files within the DataWarp allocation defined in the list `$DW_JOB_STRIPED/outlist` out to their designated destinations (as defined in the list) on the PFS

EXAMPLE: Persistent instance created through DataWarp

> **TIP:** With CLE 6.0.UP06, Cray introduces the `#DW create_persistent` and `#DW destroy_persistent` job script commands to enable users to control persistent instances. Because CLE and the various WLMs are released asynchronously, Cray cannot predict when all WLMs will fully support these two new commands. WLM-controlled persistent instance creation and destruction will continue to be supported at the discretion of each vendor. See the WLM-specific documentation for further information.

Job script `job1.sh`

```
#!/bin/sh
#DW create_persistent name=bigdata type=scratch access_mode=striped
capacity=100TiB
```

This job creates the persistent instance `bigdata`. In this release, a user cannot create and access a persistent instance in the same job.

Job script `job2.sh`

```
#!/bin/sh
#DW persistentdw name=bigdata
#DW stage_in type=file source=/path/to/pfs/datafile destination=
$DW_PERSISTENT_STRIPED_bigdata/some-input
```

This job requests access to the persistent instance and stages a file to it. Each compute node has shared access to the persistent instance via `$DW_PERSISTENT_STRIPED_bigdata`.

EXAMPLE: Access persistent instance created by another user

Job script `job3.sh`

```
#!/bin/sh
#DW persistentdw name=bigdata
#DW stage_out type=directory source=$DW_PERSISTENT_STRIPED_bigdata/my_analysis
destination=/pfs/my_data

aprun -n 3 -N 1 ./analyze_data $DW_PERSISTENT_STRIPED_bigdata/some-input
```

This example requests access to the persistent instance `bigdata` created by a coworker. Access is granted if the user meets standard POSIX file permission requirements for the instance. Each compute node has shared access to the persistent instance via `$DW_PERSISTENT_STRIPED_bigdata`. When the job completes, DWS stages the directory `/my_analysis` in the persistent instance, including all of its files and sub-directories, out to the PFS directory `/pfs/my_data`.

EXAMPLE: Persistent instance created through WLMs

Creating persistent instances can also be 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 2017-02-25T23:04:04      never     0

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

Each compute node has shared access to DataWarp via `$DW_PERSISTENT_STRIPED_persist1` as described in *#DW persistentdw - Job Script Command* on page 31.

Slurm: 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: Only privileged users are able to create persistent instances via Moab job commands. Moab users without administrator privileges must contact an administrator for assistance with creating a persistent instance. After a persistent instance is created, each compute node has shared access to DataWarp via `$DW_PERSISTENT_STRIPED_`*`resname`* as described in *#DW persistentdw - Job Script Command* on page 31.

EXAMPLE: 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.

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 access to a unique swap instance (10GiB). Each compute node also has striped/shared access to a 100GiB instance via `$DW_JOB_STRIPED`. The stripe configuration and the swap configuration share the full capacity of the instance.

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.

5.9 Example Batch Jobs that Use DataWarp Scratch

IMPORTANT: Each workload manager (WLM) has its own batch syntax, but they all share a common DataWarp request syntax. It is beyond the scope of this guide to completely detail the various WLM commands. Therefore, WLM examples are provided with the caveat that they may be out of sync with changes made by the WLM vendors, although it is Cray's intent to keep them up-to-date. For details, see the appropriate WLM documentation.

A scratch file system is requested by specifying its size and access mode in the DataWarp job commands added to a batch script. An application accesses the file system through a DataWarp environment variable based on the access mode (striped, private, or loadbalance) and instance type (job or persistent). DataWarp, through the batch scheduler, sets this environment variable to point to the directory on the compute nodes where the DataWarp file system is accessible. An application or job script may need to be modified to use this directory.

The following examples demonstrate how to use DataWarp job commands to configure and access a DataWarp scratch file system for use with applications. Some examples use IOR, a parallel file system benchmark application, as the example application.

Striped Access Mode

Example 1: writes a file (`ior_example1`) to DataWarp storage, and then checks it. It requests 2TiB of per-job striped DataWarp space, which is accessed through `$DW_JOB_STRIPED`.

Moab:

```
#!/bin/bash
#MSUB -l nodes=16:ppn=4
#MSUB -l walltime=1:00:00
#DW jobdw type=scratch access_mode=striped capacity=2TiB

aprun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example1 \
-G 1234567890 -w -k

aprun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example1 \
-G 1234567890 -W
```

SLURM:

```
#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example1
#DW jobdw type=scratch access_mode=striped capacity=2TiB

srun -n 64 -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example1 \
-G 1234567890 -w -k

srun -n 64 -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example1 \
-G 1234567890 -W
```

Example 2: extends the previous example by staging out the generated data file to the parallel file system (PFS) after the job completes. If the job is sized to use many DataWarp service nodes, all of them participate in copying the data. Note that the `#DW stage_out source` argument must begin with `$DW_JOB_STRIPED` in order to access the requested DataWarp scratch storage.

Moab:

```
#!/bin/bash
#MSUB -l nodes=16:ppn=4
#MSUB -l walltime=1:00:00
#DW jobdw type=scratch access_mode=striped capacity=2TiB
#DW stage_out type=file destination=/pfs/doc/example_1 source=$DW_JOB_STRIPED/ior_example2

aprun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example2 \
-G 1234567890 -w -k

aprun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example2 \
-G 1234567890 -W
```

SLURM:

```
#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example2
#DW jobdw type=scratch access_mode=striped capacity=2TiB
#DW stage_out type=file destination=/pfs/doc/example2 source=$DW_JOB_STRIPED/ior_example2

srun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o $DW_JOB_STRIPED/ior_example2 \
-G 1234567890 -w -k
```

Example 3: stages in a data file from the PFS to DataWarp storage, processes the data, and then stages out a directory of results back to the PFS. Compute node time is reduced by staging in data at job start, prior to launching the application, as well as staging data out at job end, after the application has completed. The application also runs faster reading/writing data to/from the SSDs rather than the PFS.

The data file `example3_data` was previously created by running IOR with the following options:

```
IOR -o /pfs/doc/example3_data -k -v -b 64m -t 1m -E -C -w -G 1248
```

Note that the `#DW stage_in destination` argument must begin with `$DW_JOB_STRIPED` in order to access the requested DataWarp scratch storage.

Moab:

```
#!/bin/bash
#MSUB -l nodes=16:ppn=4
#MSUB -l walltime=1:00:00
#DW jobdw type=scratch access_mode=striped capacity=128GiB
#DW stage_in type=file source=/pfs/doc/example3_data destination=$DW_JOB_STRIPED/input
#DW stage_out type=directory destination=/pfs/doc/results3 source=$DW_JOB_STRIPED/output

echo DW_JOB_STRIPED $DW_JOB_STRIPED

aprun -n 16 IOR -o $DW_JOB_STRIPED/input -k -v -b 64m -t 1m -E -C -W -r -G 1248
aprun -n 1 mkdir $DW_JOB_STRIPED/output
aprun -n 16 IOR -o $DW_JOB_STRIPED/output/res -k -v -b 64m -t 1m -E -C -w -G 163264 -F
```

SLURM:

```
#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example3
#DW jobdw type=scratch access_mode=striped capacity=128GiB
#DW stage_in type=file source=/pfs/doc/example3_data destination=$DW_JOB_STRIPED/input
#DW stage_out type=directory destination=/pfs/doc/results3 source=$DW_JOB_STRIPED/output

echo DW_JOB_STRIPED $DW_JOB_STRIPED

srun -n 16 IOR -o $DW_JOB_STRIPED/input -k -v -b 64m -t 1m -E -C -W -r -G 1248
srun -n 1 mkdir $DW_JOB_STRIPED/output
srun -n 16 IOR -o $DW_JOB_STRIPED/output/res -k -v -b 64m -t 1m -E -C -w -G 163264 -F
```

Persistent Instances

> **TIP:** With CLE 6.0.UP06, Cray introduces the `#DW create_persistent` and `#DW destroy_persistent` job script commands to enable users to control persistent instances. Because CLE and the various WLMs are released asynchronously, Cray cannot predict when all WLMs will fully support these two new commands. WLM-controlled persistent instance creation and destruction will continue to be supported at the discretion of each vendor. See the WLM-specific documentation for further information.

Persistent instances provide storage that is usable across jobs, including jobs run by multiple users. The storage remains until it is deleted, or optionally its lifetime expires. Any user can request access to a persistent instance subject to the WLM configuration, but standard POSIX file permissions still apply to the files.

Each WLM has a different way of creating persistent instances.

- As of release time, when using Moab/Torque, only an administrator can create a persistent instance. Contact site support services for more information.
- When using SLURM, a persistent instance is requested by submitting a job with the following request:

```
#BB create_persistent name=DataRun1 capacity=1GiB access=striped type=scratch
#BB create_persistent name=DataRun2 capacity=1GiB access=striped type=scratch
```

Example 4: requests access to two persistent instances. The persistent instance is named via `#DW persistentdw` job script command, and the environment variable set by DataWarp is `$DW_PERSISTENT_STRIPED_`*`name`* where *`name`* is the name of the persistent instance.

Moab:

```
#!/bin/bash
#MSUB -l nodes=16:ppn=4
#MSUB -l walltime=1:00:00
#DW persistentdw name=DataRun1
#DW persistentdw name=DataRun2

echo DW_PERSISTENT_STRIPED_DataRun1 $DW_PERSISTENT_STRIPED_DataRun1
echo DW_PERSISTENT_STRIPED_DataRun2 $DW_PERSISTENT_STRIPED_DataRun2

aprun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o \
$DW_PERSISTENT_STRIPED_DataRun1/input1 -G 1234567890 -w -W -r
aprun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o \
$DW_PERSISTENT_STRIPED_DataRun2/input2 -G 1248163264 -w -W -r
```

SLURM:

```
#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example4
#DW persistentdw name=DataRun1
#DW persistentdw name=DataRun2

echo DW_PERSISTENT_STRIPED_DataRun1 $DW_PERSISTENT_STRIPED_DataRun1
echo DW_PERSISTENT_STRIPED_DataRun2 $DW_PERSISTENT_STRIPED_DataRun2

srun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o \
$DW_PERSISTENT_STRIPED_DataRun1/input1 -G 1234567890 -w -W -r
srun -n 64 IOR -a POSIX -g -b 8G -t 1M -e -o \
$DW_PERSISTENT_STRIPED_DataRun2/input2 -G 1248163264 -w -W -r
```

Example 5: staging is used to copy data into or out of a persistent instance by specifying the persistent instance's reference variable in the source or destination. Staging out is a useful way to make a backup of a persistent instance.

Moab:

```
#!/bin/bash
#MSUB -l walltime=1:00:00
#MSUB -l nodes=1
#DW persistentdw name=DataRun1
#DW persistentdw name=DataRun2
#DW stage_in type=file source=/pfs/doc/example5_in destination=
$DW_PERSISTENT_STRIPED_DataRun1/example5_on_dw
#DW stage_in type=file source=/pfs/doc/example5_in destination=
$DW_PERSISTENT_STRIPED_DataRun2/example5_on_dw
#DW stage_out type=file destination=/pfs/doc/example5_out source=
$DW_PERSISTENT_STRIPED_DataRun1/example5_on_dw
```

SLURM:

```
#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example5
#DW persistentdw name=DataRun1
#DW persistentdw name=DataRun2
#DW stage_in type=file source=/pfs/doc/example5_in destination=
$DW_PERSISTENT_STRIPED_DataRun1/example5_on_dw
#DW stage_in type=file source=/pfs/doc/example5_in destination=
$DW_PERSISTENT_STRIPED_DataRun2/example5_on_dw
#DW stage_out type=file destination=/pfs/doc/example5_out source=
$DW_PERSISTENT_STRIPED_DataRun1/example5_on_dw
```

Private Access Mode

An alternative to striped access mode is private. A private instance has a directory hierarchy for each compute node, but the SSD space is shared. Data can be restricted to one DataWarp service node, or striped across all service nodes. Each namespace has its own metadata server (MDS), and the MDSs are spread across the service nodes that are used for the file system. This is the primary difference between striped and private.

A use case for this is an application that uses temporary files, but requires more temporary space than `/tmp`, a ram disk on the compute node, provides. With a private access mode the application on each node can use the file system without concern for having filename collisions.

Example 6: requests 100GiB of scratch file system such that each of the job's compute nodes has its own private storage, accessible through `$DW_JOB_PRIVATE`.

Moab:

```

#!/bin/bash
#MSUB -l nodes=16:ppn=4
#MSUB -l walltime=1:00:00
#DW jobdw type=scratch access_mode=private capacity=100GiB

echo DW_JOB_PRIVATE $DW_JOB_PRIVATE

aprun -n 1 df -h $DW_JOB_PRIVATE
```

SLURM:

```

#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example6

#DW jobdw type=scratch access_mode=private capacity=100GiB

echo DW_JOB_PRIVATE $DW_JOB_PRIVATE
srun -n 1 df $DW_JOB_PRIVATE
```

Example 7: requests 100GiB of DataWarp scratch that is accessible using both private and striped access mode via `$DW_JOB_STRIPED` and `$DW_JOB_PRIVATE`, respectively. Using both striped and private access mode is

useful when an application does a lot of metadata operations such as creates and deletes that it can keep private to the node, but also needs to share data with all other ranks.

Moab:

```
#!/bin/bash
#MSUB -l nodes=16:ppn=4
#MSUB -l walltime=1:00:00
#DW jobdw type=scratch access_mode=striped,private capacity=100GiB

echo DW_JOB_STRIPED $DW_JOB_STRIPED
echo DW_JOB_PRIVATE $DW_JOB_PRIVATE

aprun -n 1 df -h $DW_JOB_STRIPED $DW_JOB_PRIVATE
```

SLURM

```
#!/bin/bash
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --job-name=example7

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

echo DW_JOB_STRIPED $DW_JOB_STRIPED
echo DW_JOB_PRIVATE $DW_JOB_PRIVATE

srun -n 1 df $DW_JOB_STRIPED $DW_JOB_PRIVATE
```

5.10 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.`

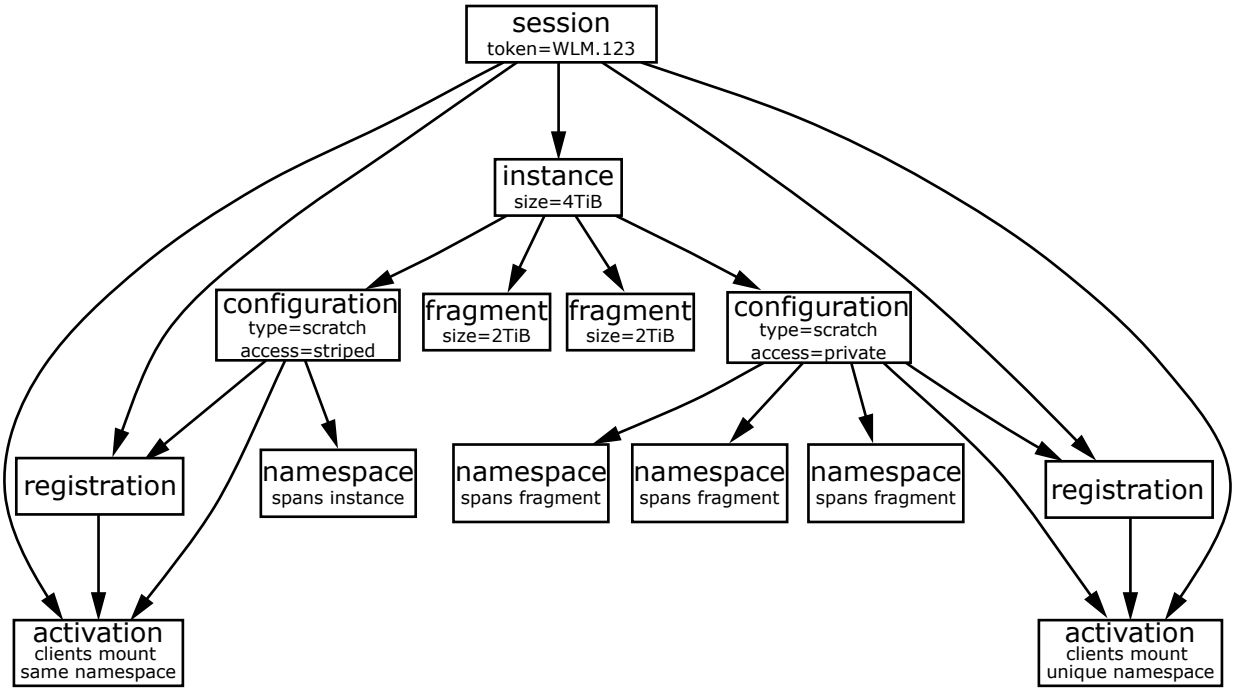
```
#DW jobdw type=scratch access_mode=striped,private capacity=4TiB
```

In this example, 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. The stripe configuration and the private configuration share the full capacity of the instance. No other limits on block usage are imposed on either of them; either one can fill of the full 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 9. Job Instance (type = scratch) with No Staging

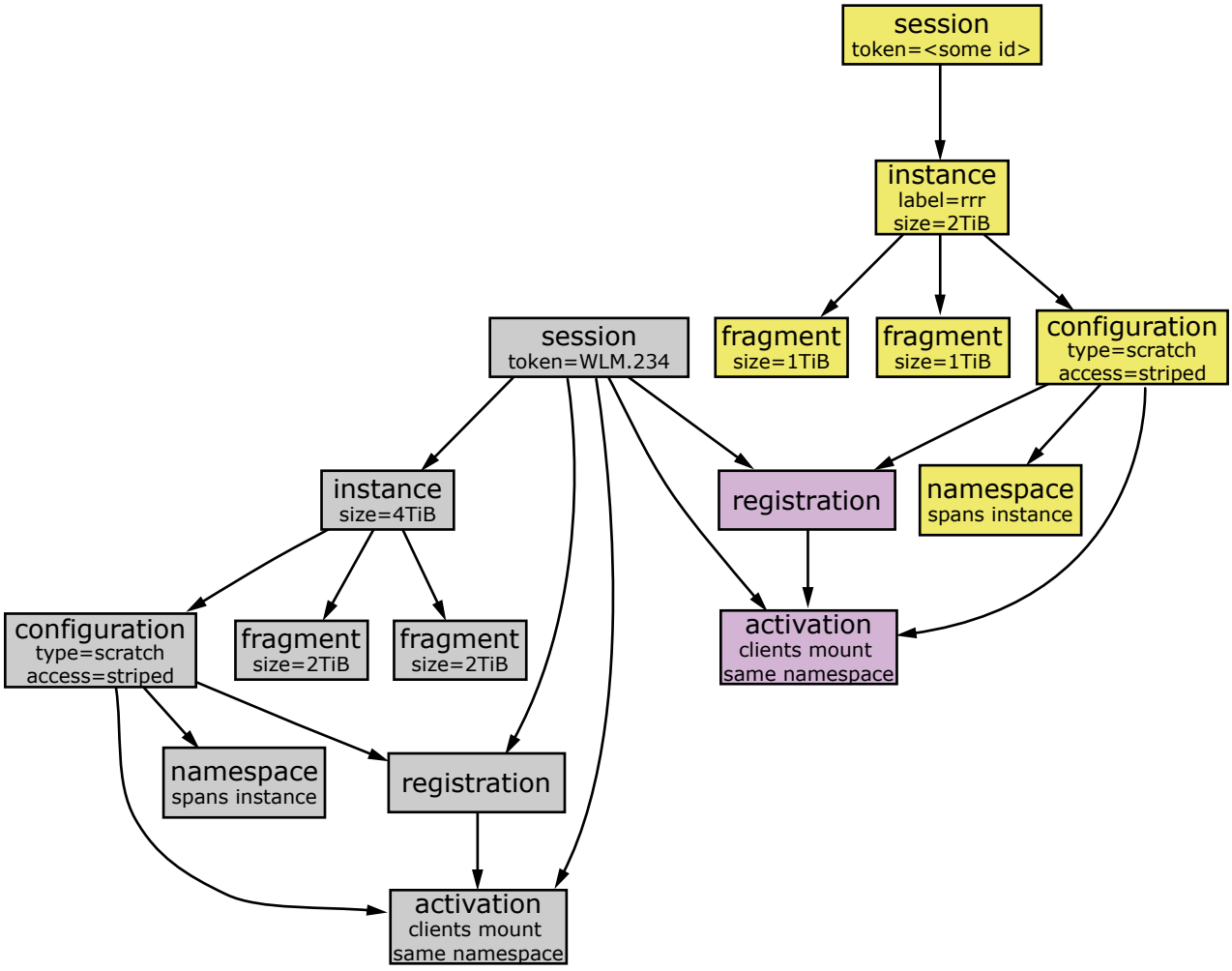


EXAMPLE: Use both job and persistent instances

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 10. Job and Persistent Instances (type = scratch)



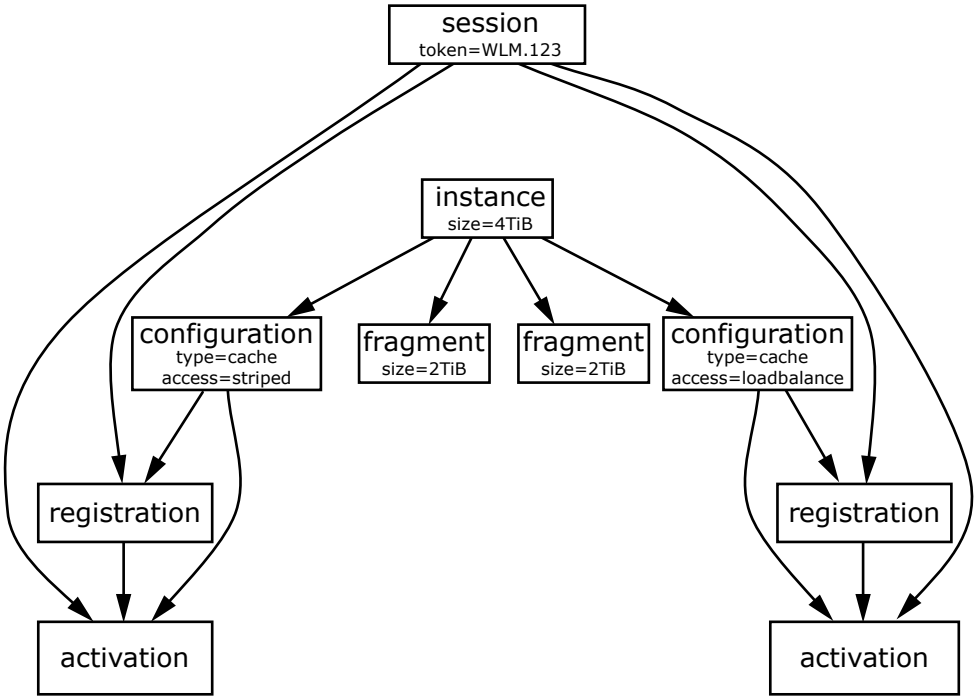
EXAMPLE: Job Instance for Cache Configuration

The following diagram shows how the `#DW jobdw` command is represented in the DWS for a batch job for a cache configuration. Note that ldbalance access mode is not available for cache in this release.

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

In this example, the cache stripe configuration and cache ldbalance 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 11. Job Instance (type = cache)



6 Additional Considerations when Using DataWarp

6.1 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.

Error messages generated when a protection limit is exceeded indicate the offense:

EROFS	Write window exceeded
EMFILE	Maximum files created exceeded
EFBIG	Maximum file size exceeded

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 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 22 and *DataWarp Job Script Command Examples* on page 37.

6.2 Memory Swapping Caveats

When swapping is enabled, CLE uses disk space to augment RAM. This may allow programs that would otherwise not fit within RAM, or even those that would fit, to run more efficiently by allowing CLE to keep the most frequently used memory contents in RAM. Swapping is used for memory that is not backed by a file, for example, heap allocations, program data and bss segments, or anonymous memory mapped by `mmap()`.

However, not all programs reap the same benefits when swapping is enabled. Swapping primarily benefits applications that do not use the communication runtime. Some operations and performance optimizations prevent memory from being swapped. For example, I/O operations usually prevent the memory that contains the I/O buffer from being swapped while the I/O is being performed. This includes file I/O and network I/O. When the I/O operation is complete, the memory is generally free to be swapped.

Additionally, hugepages cannot be swapped. Hugepages are used to improve the performance of network codes. They are automatically used by the programming environment runtime for some data structures described in the table. They are also used for application memory when one of the Cray hugepages modules is loaded (see the `intro_hugepages(1)` man page).

Many parallel programming models also prevent some amount of memory from being swapped. The restrictions frequently stem from performance optimizations and the use of network I/O. The table *Swapping restrictions by parallel programming model* on page 51 summarizes the programming models and their operations, using default settings, that prevent swapping.

Table 1. Swapping restrictions by parallel programming model

Parallel Programming Model	Operations that Restrict Swapping
MPI	• Internal MPI buffers used for eager messages and small message mailboxes. Allocated during `MPI_Init()` or program execution and released during `MPI_Finalize()`. • User buffers for large off-node transfers. During the transfer, and while in-use by the MPI buffer cache. • User buffers for large on-node transfers. During the transfer, and while in-use by the MPI buffer cache.
DMAPP and SHMEM	• Program data, bss and symmetric heap: during the entire program execution • Buffers for transfers: during the transfer
PGAS	Program data, bss, often the stack, private and shared heap: during the entire program execution

6.3 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. 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).

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, query staging and configuration data, and access accounting data. It allows greater control for what a user can do with staging than is available via the job (#DW) commands. Batch jobs only support staging in and out for striped access mode, and the commands are implemented before compute node allocation. The `libdatawarp` functions are used after the compute nodes have been allocated and support both striped and private access modes.

- 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 2. libdatawarp Routines

Routine	Function
`dw_failed_stage`	Trio of functions (`dw_open_failed_stage`, `dw_read_failed_stage`, and `dw_close_failed_stage`) used in combination to identify failed stages
`dw_get_accounting_data_json`	Converts raw scratch accounting data to JSON format
`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

Routine	Function
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
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

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.

```
user@login> 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.

```
user@login> 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.

```
user@login> 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.

```
user@login> 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 SSL certificate verification fails.

```
user@login> dwstat all
Connecting to https://c1-0c0s0n2:81 yielded fatal error:
[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:581)
```

   Fix: contact site support personnel.

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. Both commands fail when there is a MUNGE authentication issue.

```
user@login> dwstat
You must be authenticated to request this resource.
```

Fix: contact site support personnel.

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

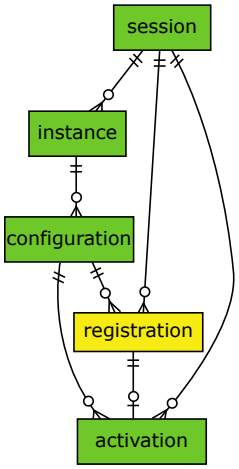
```
user@login> dwcli stage in -c 1 -s 1 --backing-path /etc/lvm/ --dir /test
cannot communicate with backend dwmd daemon at datawarp port 49214
[Errno 111] Connection refused
```

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 12. 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.

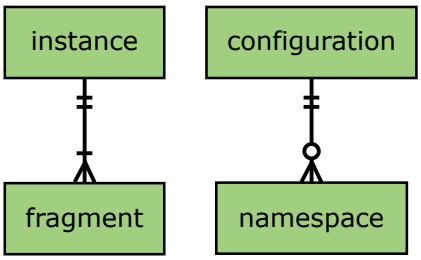
Client Node A compute node on which a configuration is activated; that is, where a DVS client mount point is created. Client nodes have direct network connectivity to all DataWarp server nodes. At least one parallel file system (PFS) is mounted on a client node.

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 13. 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 directory 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

The International System of Units (SI) prefixes and symbols (e.g., **k**ilo-, **M**ega-, **G**iga-) are often used interchangeably (and incorrectly) for decimal and binary values. This misuse not only causes confusion and errors, but the errors compound as the numbers increase. In terms of storage, this can cause significant problems. For example, consider that a kilobyte (10^3) of data is only 24 bytes less than 2^{10} bytes of data. Although this difference may be of little consequence, the table below demonstrates how the differences increase and become significant.

To alleviate the confusion, the International Electrotechnical Commission (IEC) adopted a standard of prefixes for binary multiples for use in information technology. The table below compares the SI and IEC prefixes, symbols, and values.

SI decimal vs IEC binary prefixes for multiples					
SI decimal standard			IEC binary standard		
Prefix (Symbol)	**Power**	**Value**	**Value**	**Power**	**Prefix (Symbol)**
kilo- (kB)	10^3	1000	1024	2^{10}	kibi- (KiB)
mega- (MB)	10^6	1000000	1048576	2^{20}	mebi- (MiB)
giga- (GB)	10^9	1000000000	1073741824	2^{30}	gibi- (GiB)
tera- (TB)	10^{12}	1000000000000	1099511627776	2^{40}	tebi- (TiB)
peta- (PB)	10^{15}	1000000000000000	1125899906842624	2^{50}	pebi- (PiB)
exa- (EB)	10^{18}	1000000000000000000	1152921504606846976	2^{60}	exbi- (EiB)
zetta- (ZB)	10^{21}	1000000000000000000000	1180591620717411303424	2^{70}	zebi- (ZiB)
yotta- (YB)	10^{24}	1000000000000000000000000	1208925819614629174706176	2^{80}	yobi- (YiB)

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