DataWarp User Guide S-2558-5204

Contents

About the DataWarp User Guide....3
About DataWarp....4
Overview of the DataWarp Process....5
DataWarp Concepts....7
dwstat(1)....10
dwcli(8)....17
DataWarp Job Script Commands....25
#DW jobdw - Job Script Command....25
#DW persistentdw - Job Script Command....26
#DW stage_in - DataWarp Job Script Command....26
#DW stage_out - Job Script Command....27
DataWarp Job Script Command Examples....27
Diagrammatic View of Batch Jobs....29
libdatawarp - the DataWarp API....32
dw_get_stripe_configuration....32
dw_query_directory_stage....33
dw_query_file_stage....33
dw_query_list_stage....34
dw_set_stage_concurrency....35
dw_stage_directory_in....35
dw_stage_directory_out....36
dw_stage_file_in....37
dw_stage_file_out....38
dw_stage_list_in....39
dw_stage_list_out....39
dw_terminate_directory_stage....40
dw_terminate_file_stage....41
dw_terminate_list_stage....41
dw_wait_directory_stage....42
dw_wait_file_stage....42
dw_wait_list_stage....43
Failed Stage Identification....44
Terminology....46
Prefixes for Binary and Decimal Multiples....48

About the DataWarp User Guide

Scope and Audience

This publication covers DataWarp commands, DataWarp job script commands, and the DataWarp API and is intended for users of Cray XC™ series systems with DataWarp SSD cards.

Release Information

This is the initial release of this publication; it supports DataWarp with the DataWarp Service on Cray XC™ series systems running Cray software release CLE5.2.UP04. It does not support Static DataWarp, the previously released statically configured implementation with no user interface.

Typographic Conventions

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

Feedback

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

About DataWarp

> **TIP:** All DataWarp documentation describes units of bytes using the binary prefixes defined by the International Electrotechnical Commission (IEC), e.g., MiB, GiB, TiB. For further information, see *Prefixes for Binary and Decimal Multiples* on page 48.

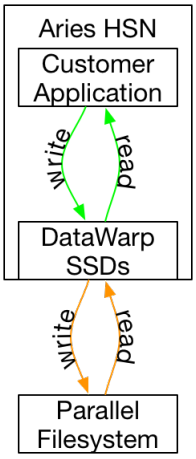
Cray DataWarp provides an intermediate layer of high bandwidth, file-based storage to applications running on compute nodes. It is comprised of commercial SSD hardware and software, Linux community software, and Cray system hardware and software. DataWarp storage is located on server nodes connected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

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

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

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

Figure 1. DataWarp Overview



DataWarp Use Cases

There are four basic use cases for DataWarp:

Parallel file	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. Initially, data movement (staging) between

system (PFS) cache

DataWarp and the PFS must be explicitly requested by a job and/or application and then performed by the DataWarp service. In a future release, data staging between DataWarp and the PFS can also be done implicitly (i.e., read ahead and write behind) by the DataWarp service without application intervention. 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.

Scratch storage

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

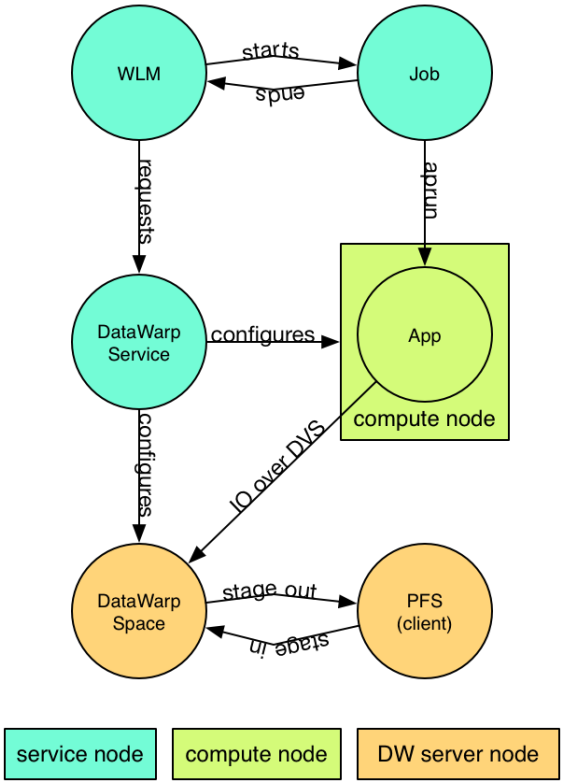
DataWarp storage can be shared by multiple jobs over a configurable period of time. The jobs may or may not be related and may run concurrently or serially. The shared data may be available before a job begins, extend after a job completes, and encompass multiple jobs. Shared data use cases include:

- **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.

Overview of the DataWarp Process

Refer to Figures *DataWarp Component Interaction - bird's eye view* on page 6 and *DataWarp Component Interaction - detailed view* on page 7 for visual representation of the process.

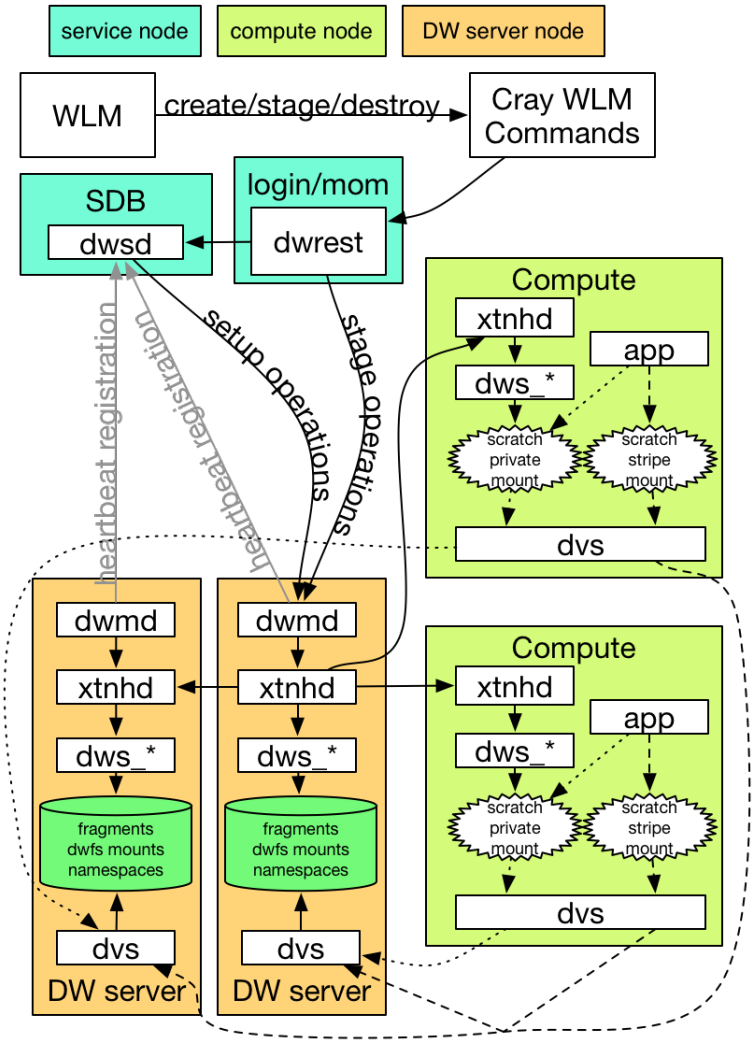
Figure 2. DataWarp Component Interaction - bird's eye view



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

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

Figure 3. DataWarp Component Interaction - detailed view



DataWarp Concepts

For basic definitions, refer to *Terminology* on page 46.

Instances

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

- **Job instance**: The lifetime of a job instance, as it sounds, is the lifetime of the job that created it, and is accessible only by the job that created it.

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

When either type of instance is destroyed, DataWarp ensures that data needing to be written to the 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 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.

For this release, a DataWarp instance can be configured as scratch. Additionally, all data staging between either type of instance and the PFS must be explicitly requested using the DataWarp job script staging commands or the application C library API (`libdatawarp`). In a future release, an instance will be configurable as cache, and all data staging between the cache instance and the PFS will occur implicitly.

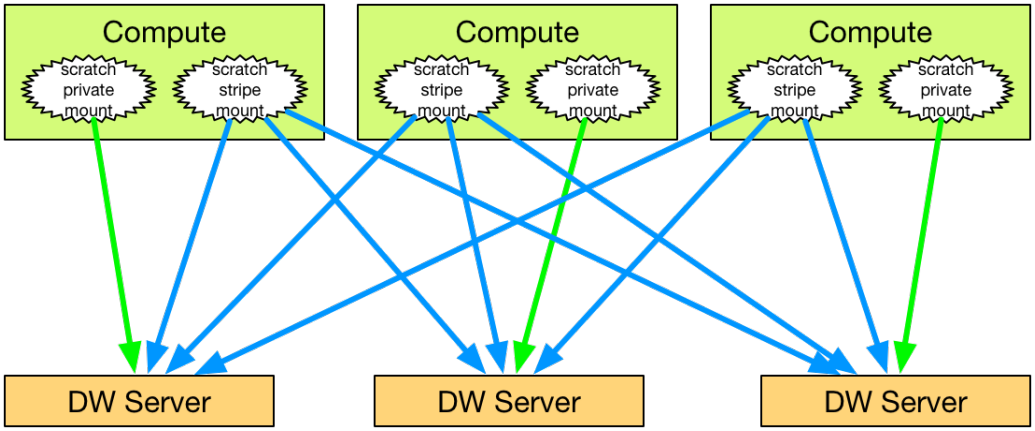
A scratch configuration can be accessed in one or more of the following ways:

- **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 reside on one DataWarp server node. For scratch instances the files are only accessible to the compute node that created them (e.g., /tmp). Private access is not supported for persistent instances, because a persistent instance can be used by multiple jobs with different numbers of compute nodes.

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

- **Striped**: All compute nodes share one namespace; files stripe across all servers.
- **Private**: Each compute node gets its own namespace. Each namespace maps to one server node, therefore, files in a namespace are only on one server node.

The following diagram shows a scratch private and scratch stripe mount point on each of three compute (client) nodes. For scratch private, each compute node reads and writes to its own namespace that exists on one of the DataWarp server nodes. For scratch stripe, each compute node reads and writes to a common namespace, and that namespace spans all three DataWarp nodes.


Compute
scratch private mount
scratch stripe mount
Compute
scratch stripe mount
scratch private mount
Compute
scratch stripe mount
scratch private mount
DW Server
DW Server
DW Server

dwstat(1)

NAME

`dwstat` - Provides status information about DataWarp resources

SYNOPSIS

```
dwstat [-h]
dwstat [--all] [-b | -e | -E | -g | -G | -H | -k | -K | -m | -M | -p | -P | -t | -T | -
y | -Y | -z | -Z] [--role role] [resource [resource]...]
```

DESCRIPTION

The `dwstat` command provides status information about DataWarp resources in tabular format.

IMPORTANT: The `dws` module must be loaded to use this command.

```
$ module load dws
```

`dwstat` accepts the following options:

--all	Used with `nodes` resource; displays all nodes. Default display includes nodes with `capacity>0` only.
-b	Display output in bytes
-e	Display output in IEC Exbibyte (EiB) units; for further information, see *Prefixes for Binary and Decimal Multiples* on page 48
-E	Display output in SI Exabyte (EB) units
-g	Display output in IEC gibibyte (GiB) units
-G	Display output in SI kigabyte (GB) units
-h | --help	Display usage information.
-H	Display output in SI units (IEC is default)
-k	Display output in IEC kibibyte (KiB) units
-K	Display output in SI kilobyte (KB) units
-m	Display output in IEC mebibyte (MiB) units
-M	Display output in SI megabyte (MB) units
-p	Display output in IEC Pebibyte (PiB) units
-P	Display output in SI Petabyte (PB) units
--role *ROLE*	Request a role outside of user's level

-t	Display output in IEC Tebibyte (TiB) units
-T	Display output in SI Terabyte (TB) units
-y	Display output in IEC Yobibyte (YiB) units
-Y	Display output in SI Yottabyte (YB) units
-z	Display output in IEC Zebibyte (ZiB) units
-Z	Display output in SI Zettabyte (ZB) units

Resources

The `dwstat` command accepts the following resources:

activations	Displays a table of current activations; a DataWarp activation is an object that represents an available instance configuration on a set of nodes.
all	Displays the tables for all resource types.
configurations	Displays a table of current configurations; a DataWarp configuration represents a specific way in which a DataWarp instance will be used.
fragments	Displays a table of current fragments; a DataWarp fragment is a subset of managed space on a DataWarp node.
instances	Displays a table of current instances; a DataWarp instance is a collection of DataWarp fragments, where no two fragments in the instance exist on the same node.
most	Displays tables for pools, sessions, instances, configurations, registrations, and activations
namespaces	Displays a table of current namespaces; a DataWarp namespace represents a partitioning of a DataWarp scratch configuration.
nodes	Displays a table of current nodes; a DataWarp node can host DataWarp capacity, have DataWarp configurations activated on it, or both. By default, displays nodes with `capacity>0` only.
pools	Displays a table of current pools; a DataWarp pool represents an aggregate DataWarp capacity. (Default output)
registrations	Displays a table of current registrations; a DataWarp registration represents a known use of a configuration by a session.
sessions	Displays a table of current sessions; a DataWarp session is an object used to map events between a client context and a DataWarp service context. A WLM typically creates a DataWarp session for each batch job that uses the DataWarp service.

EXAMPLE: `dwstat pools`

```
$ dwstat pools
    pool units quantity    free   gran
wlm_pool bytes        0       0   1GiB
   space bytes  7.12TiB 2.88TiB 128GiB
testpool bytes        0       0  16MiB
```

The column headings are defined as:

pool	Pool name
units	Pool units; currently only bytes are supported

quantity	Maximum configured space
free	Currently available space
gran	Granularity - pool stripe size

EXAMPLE: dwstat sessions

```
$ dwstat sessions
 sess state      token creator owner             created expiration nodes
  832 CA---  783000000  tester 12345 2015-09-08T16:20:36      never    20
  833 CA---  784100000  tester 12345 2015-09-08T16:21:36      never     1
  903 D---- 1875700000  tester 12345 2015-09-08T17:26:05      never     0
```

The column headings are defined as:

sess	Numeric session ID
state	Five-character code representing a session's state as follows (left to right): **1.** Goal: `C` = Create; `D` = Destroy **2.** Setup: `A` = Actualized; `-` = non-actualized **3.** Condition: `F` = Fuse blown (an error exists); `-` = fuse intact **4.** Status: `T` = Transitioning; `-` = inert **5.** Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks
token	Unique identifier typically created by WLM
creator	Typically WLM
owner	UID of job
created	Creation timestamp
expiration	*`date`* = expiration date; `never` = no expiration date
nodes	Number of nodes at session set up

EXAMPLE: dwstat instances

```
$ dwstat instances
 inst state sess  bytes nodes            created  expiration intact  label public confs
  753 CA---  832 128GiB     1 2015-09-08T16:20:36       never   true I832-0  false     1
  754 CA---  833 128GiB     1 2015-09-08T16:21:36       never   true I833-0  false     1
  807 D----  903 128GiB     1 2015-09-08T17:26:05       never  false I903-0  false     1
  808 CA---  904 128GiB     1 2015-09-08T17:26:08       never   true I904-0  false     1
  810 CA---  906 128GiB     1 2015-09-08T17:26:10       never   true I906-0  false     1
```

The column headings are defined as:

inst	Numeric instance ID
state	Five-character code representing a instance's state as follows (left-to-right): **1.** Goal: `C` = Create; `D` = Destroy **2.** Setup: `A` = Actualized; `-` = non-actualized **3.** Condition: `F` = Fuse blown (an error exists); `-` = fuse intact

4. Status: `T` = Transitioning; `-` = inert
5. Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks

sess	Numeric session ID
bytes	Instance size
nodes	Number of nodes on which this instance is active
created	Creation timestamp
expiration	*`date`* = expiration date; `never` = no expiration date
intact	True, if `Goal=C` (create), and all fragments associated with this instance are themselves associated with a node
label	User-defined label (name)
public	`true` = shared resource (visible to all users)
confs	Number of configurations to which an instance belongs

EXAMPLE: `dwstat configurations`

```
$ dwstat configurations
 conf state inst    type access_type activs
  715 CA---  753 scratch      stripe      1
  716 CA---  754 scratch      stripe      1
  759 D--T-  807 scratch      stripe      0
  760 CA---  808 scratch      stripe      1
```

The column headings are defined as:

conf	Number configuration ID
state	Five-character code representing a configuration's state as follows (left-to-right): 1. Goal: `C` = Create; `D` = Destroy 2. Setup: `A` = Actualized; `-` = non-actualized 3. Condition: `F` = Fuse blown (an error exists); `-` = fuse intact 4. Status: `T` = Transitioning; `-` = inert 5. Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks
inst	Numeric instance ID
type	Configuration type - `scratch`, `cache`, or `swap`
access_type	Access mode - `stripe` or `private`
activs	Number of activations to which a configuration belongs

EXAMPLE: `dwstat registrations`

```
$ dwstat registrations
  reg state sess conf wait
  648 CA---  832  715 true
  649 CA---  833  716 true
  674 CA---  904  760 true
```

The column headings are defined as:

reg Numeric registration ID

state Five-character code representing a registration's state as follows (left-to-right):

1. Goal: `C` = Create; `D` = Destroy
2. Setup: `A` = Actualized; `-` = non-actualized
3. Condition: `F` = Fuse blown (an error exists); `-` = fuse intact
4. Status: `T` = Transitioning; `-` = inert
5. Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks

sess Numeric session ID

conf Numeric configuration ID

wait If true, this session is waiting for associated configuration to finish asynchronous activities

EXAMPLE: dwstat activations

```
$ dwstat activations
activ state sess conf nodes      mount
  622 CA---  832  715    20  /tmp/tst1
  623 CA---  833  716     1  /tmp/tst2
  648 CA---  904  760     1  /tmp/tst3
  650 CA---  906  762     1  /tmp/tst4
```

The column headings are defined as:

activ Numeric activation ID

state Five-character code representing a activation's state as follows (left-to-right):

1. Goal: `C` = Create; `D` = Destroy
2. Setup: `A` = Actualized; `-` = non-actualized
3. Condition: `F` = Fuse blown (an error exists); `-` = fuse intact
4. Status: `T` = Transitioning; `-` = inert
5. Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks

sess Numeric session ID

conf Numeric configuration ID

nodes Number of nodes on which an activation is present

mount Mount point for the activation

EXAMPLE: dwstat fragments

```
$ dwstat fragments
 frag state inst capacity     node
  780  CA--  753   128GiB nid00066
  781  CA--  754   128GiB nid00069
  842  D---  807   128GiB nid00022
  843  CA--  808   128GiB nid00065
```

The column headings are defined as:

frag Numeric fragment ID

state Four-character code representing a fragment's state as follows (left-to-right):

1. Goal: `C` = Create; `D` = Destroy
2. Setup: `A` = Actualized; `-` = non-actualized
3. Status: `T` = Transitioning; `-` = inert
4. Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks

inst Numeric instance ID

capacity Total capacity of a fragment

node Hostname of node on which a fragment is located

EXAMPLE: `dwstat namespaces`

```
$ dwstat namespaces
   ns state conf frag span
  758  CA--  715  780    1
  759  CA--  716  781    1
  818  CA--  760  843    1
```

The column headings are defined as:

ns Numeric namespace ID

state Four-character code representing a namespace's state as follows (left-to-right):

1. Goal: `C` = Create; `D` = Destroy
2. Setup: `A` = Actualized; `-` = non-actualized
3. Status: `T` = Transitioning; `-` = inert
4. Spectrum: `M` = Mixed (tasks remaining); `-` = no outstanding tasks

conf Numeric configuration ID

frag Numeric fragment ID

span Number of fragments contained within a namespace

EXAMPLE: `dwstat nodes`

```
$ dwstat nodes
    node  pool online drain  gran   capacity insts activs
nid00022 space   true false  8MiB    3.64TiB     7      0
nid00065 space   true false 16MiB 1023.98GiB     7      0
nid00066 space   true false 16MiB 1023.98GiB     7      0
nid00069 space   true false 16MiB 1023.98GiB     7      0
nid00070 space   true false 16MiB 1023.98GiB     6      0
nid00004     -   true false     0           0     0      3
```

The column headings are defined as:

node Node hostname

pool	Name of pool to which node is assigned
online	The node is available
drain	`true` = resource is draining
gran	Node granularity
capacity	Total capacity of a node
insts	Number of instances on a node
activs	Number of activations on a node

dwcli(8)

NAME

`dwcli` - Command line interface for DataWarp

SYNOPSIS

`dwcli` [*common_options*] [*ACTION RESOURCE* [*resource_attributes*]]

DESCRIPTION

The `dwcli` command provides a command line interface to act upon DataWarp resources. This is primarily an administration command, although a user can initiate some actions using it. With full WLM support, a user does not have a need for this command.

IMPORTANT: The `dws` module must be loaded to use this command.

```
$ module load dws
```

COMMON OPTIONS

`dwcli` accepts the following common options:

`--debug`
: Enable debug mode

`-h | --help`
: Display usage information for the command, actions, and resources:

- `dwcli -h`
- `dwcli` *`action`* -h
- `dwcli` *`action resource`* -h

`-j | --json`
: Display debug output as json if applicable (not valid with `--debug`)

`-r` *`ROLE`*
: Request a role outside the user's level

`-s | --short`
: Display abbreviated `create` output

`-v | --version`
: Display `dwcli` version information

ACTIONS

The following actions are available:

`create` Create resource

Valid for: `activation`, `configuration`, `instance`, `pool`, and `session`.

`ls` Display information about a resource

Valid for: `activations`, `configurations`, `instances`, `fragments`, `namespaces`, `nodes`, `pools`, `registrations`, and `sessions`.

`rm` Remove a resource

Valid for: `activation`, `configuration`, `instance`, `pool`, `registration`, and `session`.

`stage` Stage files and directories in or out

Valid for options: `in`, `out`, `query`, and `terminate`.

`update` Update the attributes of a resource

Valid for: `activation`, `configuration`, `instance`, `node`, `registration`, and `session`.

RESOURCES

`dwcli` accepts the following resources:

- **activation**

  A DataWarp activation is an object that represents an available instance configuration on a set of nodes. The activation resource has the following attributes:

  `--configuration` *`CONFIGURATION`*

  Numeric configuration ID for activation

  `--hosts` *`CLIENT_NODES`*

  Hostnames on which the referenced datawarp instance configuration may be activated. If not defined, the hostnames associated with *`SESSION`* are used.

  `--id` *`ID`*

  Numeric activation ID

  `--mount` *`MOUNT`*

  Client mount directory for scratch configurations

  `--replace-fuse`

  Directs DWS to replace the activation's fuse and retry activating it

  `--session` *`SESSION`*

  Numeric ID of session with which the datawarp activation is associated

- **configuration**

  A DataWarp configuration represents a specific way in which a DataWarp instance will be used. The configuration resource has the following attributes:

  `--access-type` *`ACCESS_TYPE`*

  Type of access, either `stripe` or `private`

  `--group` *`GROUP_ID`*

  Numeric group ID for the root directory of the storage

  `-i | --id` *`ID`*

  Numeric configuration ID

--instance *INSTANCE*

Numeric ID of instance in which this configuration exists

--max-files-created *MAX_FILES_CREATED*

Maximum number of files allowed to be created in a single configuration namespace

--max-file-size *MAX_FILE_SIZE*

Maximum file size, in bytes, for any file in the configuration namespace

--replace-fuse

Directs DWS to replace the configuration's fuse and retry configuration tasks

--root-permissions *ROOT_PERMISSIONS*

File system permissions used for the root directory of the storage for type scratch, in octal format (e.g., 0777)

--type *TYPE*

Type of configuration; currently only `scratch` is valid

- **fragment**

  A DataWarp fragment is a subset of managed space found on a DataWarp node. The fragment resource has no attributes available for this command.

- **instance**

  A DataWarp instance is a collection of DataWarp fragments, where no two fragments in a DataWarp instance exist on the same node. DataWarp instances are essentially unusable raw space until at least one DataWarp instance configuration is created, specifying how the space is to be used and accessed. A DataWarp instance may not be created unless a DataWarp session is supplied at creation time. The instance resource has the following attributes:

  --capacity *size*

  Instance capacity in bytes

  --expiration *epoch*

  Expiration time in Unix, or epoch, time

  -i | --id *ID*

  Numeric instance ID

  --label *LABEL*

  Instance label name

  --optimization *OPTIMIZATION*

  Requested optimization strategy; options are `bandwidth`, `interference`, and `wear`. Specifying `bandwidth` optimization results in the DWS picking as many server nodes as possible while satisfying the capacity request. Specifying `interference` optimization results in the DWS picking as few server nodes as possible when satisfying the capacity request. Specifying `wear` optimization results in the DWS picking server nodes primarily on the basis of the health of the SSDs on the server nodes. Both `bandwidth` and `interference` make use of SSD health data as a second-level optimization.

  --pool *pname*

  Name of pool with which the instance is associated

  --private

  Controls the visibility of the instance being created; private is visible only to administrators and the user listed in *SESSION*

--public

Controls the visibility of the instance being created; public is visible to all users. Persistent datawarp instances, which are meant to be shared by multiple users, are required to be public.

--replace-fuse

Directs DWS to replace the instance's fuse and retry instance tasks

--session *SESSION*

Numeric ID of session with which the instance is associated

--write-window-length *WW_LENGTH*

Write window duration in seconds; used with `--write-window-multiplier`

--write-window-multiplier *WW_MULTIPLIER*

Used with `--write-window-length`, for each fragment comprising the instance, the size of the fragment is multiplied by *WW_MULTIPLIER* and the user is allowed to write that much data to the fragment in a moving window of the *WW_LENGTH*. When the limit is exceeded, the `scratch_limit_action` specified in `dwsd.yaml` is performed. This can aid in the detection of anomalous usage of a datawarp instance.

- **node**

A DataWarp node can host DataWarp capacity (server node), have DataWarp configurations activated on it (client node), or both. The node resource has the following attributes; they are only valid with `update`:

--drain

Set `drain=true`; do not use for future instance requests

-n | --name *NAME*

Hostname of node

--no-drain

Set `drain=false`; node is available for requests

--pool *POOL*

Name of pool to which this node belongs

--rm-pool

Disassociate the node from a pool.

- **pool**

A DataWarp pool represents an aggregate DataWarp capacity. The pool resource has the following attributes:

--granularity *GRANULARITY*

Pool allocation granularity in bytes

-n | --name *NAME*

Pool name

- **registration**

A DataWarp registration represents a known use of a configuration by a session. The registration resource has the following attributes:

-i | --id *ID*

Numeric registration ID

--no-wait

Set `wait=false`; do not wait for associated configuration to finish asynchronous activities such as waiting for all staged out data to finish staging out to the PFS

--replace-fuse

Directs DWS to replace the registration's fuse and begin retrying registration tasks

--wait

Set `wait=true`; wait for associated configuration to finish asynchronous activities

- **session**

  A DataWarp session is an object used to map events between a client context (e.g., a WLM batch job) and a DataWarp service context. It establishes node authorization rights for activation purposes, and actions performed through the session are undone when the session is removed. The session resource has the following attributes:

  --creator *CREATOR*

  Name of session creator

  --expiration *EXPIRATION*

  Expiration time in Unix, or epoch, time. If `0`, the session never expires.

  --hosts *CLIENT_NODE* **[***CLIENT_NODE***...]**

  List of hostnames to which the session is authorized access

  -i | --id *ID*

  Numeric session ID

  --owner *OWNER*

  Userid of session owner

  --replace-fuse

  Directs DWS to replace the session's fuse and retry session tasks

  --token *TOKEN*

  Session label

STAGE OPTIONS

The `stage` action stages files/directories and accepts the following options:

- **in**: stage a file or directory from a PFS into DataWarp. The following arguments are accepted:
  -b | --backing-path *BACKING_PATH*

  Path of file/directory to stage into the DataWarp file system

  -c | --configuration *CONFIGURATION_ID*

  Numeric configuration ID

  -d | --dir *DIRNAME*

  Name of directory to stage into the DataWarp file system

  -f | --file *FILENAME*

  Name of file to stage into the DataWarp file system

  -s | --session *SESSION*

  Numeric session ID

- **list**: provides a recursive listing of all files with stage attributes for a staging session/configuration. The following arguments are accepted:
  -c | --configuration *CONFIGURATION_ID*

Numeric configuration ID

`-s | --session` *SESSION*

Numeric session ID

- **out**: stage a file or directory out of the DataWarp file system to a PFS. The following arguments are accepted:
  `-b | --backing-path` *BACKING_PATH*

  PFS path to where file/directory is staged out

  `-c | --configuration` *CONFIGURATION_ID*

  Numeric configuration ID

  `-d | --dir` *DIRNAME*

  Name of directory to stage out to the PFS

  `-f | --file` *FILENAME*

  Name of file to stage out to the PFS

  `-s | --session` *SESSION*

  Numeric session ID
- **query**: query staging status for a file or directory. The following arguments are accepted:
  `-c | --configuration` *CONFIGURATION_ID*

  Numeric configuration ID

  `-d | --dir` *DIRNAME*

  Name of a DataWarp directory to query (optional)

  `-f | --file` *FILENAME*

  Name of a DataWarp file to query (optional)

  `-s | --session` *SESSION*

  Numeric session ID
- **terminate**: terminate a current stage operation. The following arguments are accepted:
  `-c | --configuration` *CONFIGURATION_ID*

  Numeric configuration ID

  `-d | --dir` *DIRNAME*

  Name of directory for which staging is terminated

  `-f | --file` *FILENAME*

  Name of file for which staging is terminated

  `-s | --session` *SESSION*

  Numeric session ID

EXAMPLE: Create a pool

Only an administrator can execute this command.

```
smw# dwcli create pool --name example-pool --granularity 16777216
created pool name example-pool
```

EXAMPLE: Assign a node to the pool

Only an administrator can execute this command.

```
smw# dwcli update node --name example-node --pool example-pool
```

EXAMPLE: Create a session

Only an administrator can execute this command.

```
$ dwcli create session --expiration 4000000000 --creator $(id -un) --token example-
session --owner $(id -u) --hosts example-node
created session id 10
```

EXAMPLE: Create an instance

Only an administrator can execute this command.

```
$ dwcli create instance --expiration 4000000000 --public --session 10 --pool
example-poolname --capacity 1099511627776 --label example-instance --optimization
bandwidth
created instance id 8
```

EXAMPLE: Create a configuration

```
$ dwcli create configuration --type scratch --access-type stripe --root-
permissions 0755 --instance 8 --group 513
created configuration id 7
```

EXAMPLE: Create an activation

```
$ create activation --mount /some/pfs/mount/directory --configuration 7 --session
10
created activation id 7
```

EXAMPLE: Set a registration to `--no-wait`

Directs DWS to not wait for associated configurations to finish asynchronous activities such as waiting for all staged out data to finish staging out to the PFS. Note that no output after this command indicates success.

```
$ dwcli update registration --id 1 --no-wait
$
```

EXAMPLE: Remove a pool

Only an administrator can execute this command.

```
$ dwstat pools
  pool units quantity    free  gran
canary bytes  3.98GiB 3.97GiB 16MiB
$ dwcli rm pool --name canary
$ dwstat pools
no pools
```

EXAMPLE: Remove a session

Only an administrator can execute this command.

```
$ dwstat sessions
sess state token creator owner             created expiration nodes
   1 CA---    ok    test 12345 2015-09-18T16:31:24    expired     1

$ dwcli rm session --id 1
sess state token creator owner             created expiration nodes
```

```
   1 D----    ok    test 12345 2015-09-18T16:31:24    expired     0

After some time...
$ dwstat sessions
no sessions
```

EXAMPLE: Fuse replacement

```
$ dwstat instances
inst state sess bytes nodes             created expiration intact           label
public confs
   1 D-F-M    1 16MiB     1 2015-09-18T17:47:57    expired  false canary-
instance   true     1
$ dwcli update instance --replace-fuse --id 1
$ dwstat instances
inst state sess bytes nodes             created expiration intact           label
public confs
   1 D---M    1 16MiB     1 2015-09-18T17:47:57    expired  false canary-
instance   true     1
```

EXAMPLE: Stage in a directory, query immediately, then stage list

```
$ dwcli stage in --session $session --configuration $configuration --dir=/tld/. --backing-path=/tmp/
demo/
path   backing-path nss ->c ->q ->f <-c <-q <-f <-m
/tld/. -            1   3   1   -   -   -   -   -

$ dwcli stage query --session $session --configuration $configuration
path  backing-path nss ->c ->q ->f <-c <-q <-f <-m
/.    -            1   4   -   -   -   -   -   -
/tld/ -            1   4   -   -   -   -   -   -

$ dwcli stage list --session $session --configuration $configuration
path                  backing-path               nss ->c ->q ->f <-c <-q <-f <-m
/tld/filea            /tmp/demo/filea            1   1   -   -   -   -   -   -
/tld/fileb            /tmp/demo/fileb            1   1   -   -   -   -   -   -
/tld/subdir/subdirfile /tmp/demo/subdir/subdirfile 1   1   -   -   -   -   -   -
/tld/subdir/subfile    /tmp/demo/subdir/subfile    1   1   -   -   -   -   -   -
```

EXAMPLE: Stage a file in afterwards, stage list, then query

Note the difference in the `stage query` output.

```
$ dwcli stage in --session $session --configuration $configuration --file /dwfsfile --backing-path /tmp/
demo/filea
path      backing-path    nss ->c ->q ->f <-c <-q <-f <-m
/dwfsfile /tmp/demo/filea 1   1   -   -   -   -   -   -

$ dwcli stage list --session $session --configuration $configuration
path                  backing-path               nss ->c ->q ->f <-c <-q <-f <-m
/dwfsfile             /tmp/demo/filea            1   1   -   -   -   -   -   -
/tld/filea            /tmp/demo/filea            1   1   -   -   -   -   -   -
/tld/fileb            /tmp/demo/fileb            1   1   -   -   -   -   -   -
/tld/subdir/subdirfile /tmp/demo/subdir/subdirfile 1   1   -   -   -   -   -   -
/tld/subdir/subfile    /tmp/demo/subdir/subfile    1   1   -   -   -   -   -   -


$ dwcli stage query --session $session --configuration $configuration
path      backing-path    nss ->c ->q ->f <-c <-q <-f <-m
/.        -               1   5   -   -   -   -   -   -
/tld/     -               1   4   -   -   -   -   -   -
/dwfsfile /tmp/demo/filea 1   1   -   -   -   -   -   -
```

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, which are treated as comments by the WLM and passed to the DataWarp infrastructure. They provide the DataWarp Service (DWS) with information about the DataWarp resources a job requires. The DataWarp job script commands start with the characters `#DW` and include:

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

#DW jobdw - Job Script Command

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

access_mode=*mode* Specifies the mode in which this instance is accessed. Valid options are `striped`, `private`, or both.

The compute node path to the instance storage is:

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

capacity=*n* Specifies the 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.

type=scratch Specifies how the DataWarp instance will function; currently only `scratch` is supported.

#DW persistentdw - Job Script Command

NAME

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

SYNOPSIS

`#DW persistentdw` `name=`*`piname`*

DESCRIPTION

Optional command to configure access to an existing persistent DataWarp instance with the specified parameters. This command can appear multiple times in a job script.

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

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

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

- striped access mode: `$DW_PERSISTENT_STRIPED_`*`piname`*

#DW stage_in - DataWarp Job Script Command

NAME

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

SYNOPSIS

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

DESCRIPTION

Optional command to stage files from a parallel file system (PFS) into an existing DataWarp instance at job start. Missing files will cause the job to fail. This command can appear multiple times in a job script. Currently supports scratch configurations only.

The `#DW stage_in` command accepts the following options and parameters:

`destination=`*`dpath`* Specifies the path within the DataWarp instance; `destination` must always start with exactly `$DW_JOB_STRIPED`

`source=`*`spath`* Specifies the path within the PFS; it must be readable by the user.

`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 in fails.
- `list` - Source is a file containing a list of files to stage (one file per line). If a specified file is a directory, other non-regular file, or has hard links, the stage in fails.

#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
```

DESCRIPTION

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

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

destination=*dpath* Specifies the path within the PFS; it must be writable by the user.

source=*spath* Specifies the path within the DataWarp instance; `source` must always start with exactly `$DW_JOB_STRIPED`

Specifies the type of entity for staging. Options are:

- `directory` - Source is a single directory to stage, including all files and subdirectories within the directory. 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 per line). If a specified file is a directory, other non-regular file, or has hard links, the stage out fails.

DataWarp Job Script Command Examples

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

EXAMPLE: DataWarp job instance, no staging

Batch command:

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

Job script `job.sh`:

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

aprun -n 3 -N 1 my_app $DW_JOB_STRIPED/sharedfile $DW_JOB_PRIVATE/scratchfile
```

Each compute node has striped/shared access to DataWarp via `$DW_JOB_STRIPED` and access to a per-compute node scratch area via `$DW_JOB_PRIVATE`. At the end of the job, the WLM runs a series of commands to wait for data staged out by `my_app` (see *libdatawarp - the DataWarp API* on page 32) as well as to clean up any usage of the DataWarp resource.

EXAMPLE: DataWarp persistent instance

Creating persistent instances differs per WLM; some allow users to schedule persistent DataWarp instance creation, while others rely on the CLI tool. Note that using the CLI tool to create a persistent instance with a WLM that expects to manage all DataWarp capacity may cause the WLM to become confused or slow down job scheduling. For further details, see the appropriate WLM documentation.

A user or administrator creates the persistent instance with the mechanism appropriate to the site-specific WLM prior to job submission:

```
% dw_wlm_cli -f create_persistent --capacity 100TiB --caller CLI --accessmode
striped --user 12345 --type scratch --token piname
qsub -lmppwidth=3,mppnppn=1 job.sh
```

Job script `job.sh`

```
#DW persistentdw name=piname
aprun -n 3 -N 1 my_app $DW_PERSISTENT_STRIPED_piname/test1
```

Each compute node has shared access to DataWarp via `$DW_PERSISTENT_STRIPED_piname`.

EXAMPLE: Use both job and persistent instances

Creating persistent instances differs per WLM; some allow users to schedule persistent DataWarp instance creation, and others rely on the CLI tool. Note that using the CLI tool to create a persistent instance with a WLM that expects to manage all DataWarp capacity may cause the WLM to become confused or slow down job scheduling. For further details, see the appropriate WLM documentation.

A user or administrator creates the persistent instance with the mechanism appropriate to the site-specific WLM prior to job submission:

```
% dw_wlm_cli -f create_persistent --capacity 100TiB --caller CLI --accessmode
striped --user 12345 --type scratch --token piname
qsub -lmppwidth=3,mppnppn=1 job.sh
```

Job script `job.sh`

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

aprun -n 3 -N 1 my_app $DW_PERSISTENT_STRIPED_piname/test1
```

Each compute node has shared access to the persistent DataWarp instance via `$DW_JOB_PRIVATE`, shared access to the job DataWarp instance via `$DW_JOB_STRIPED`, and access to a per-compute node scratch area via `$DW_JOB_PRIVATE`.

EXAMPLE: Staging

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

Job script `job.sh`

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

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

EXAMPLE: Interactive 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 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.

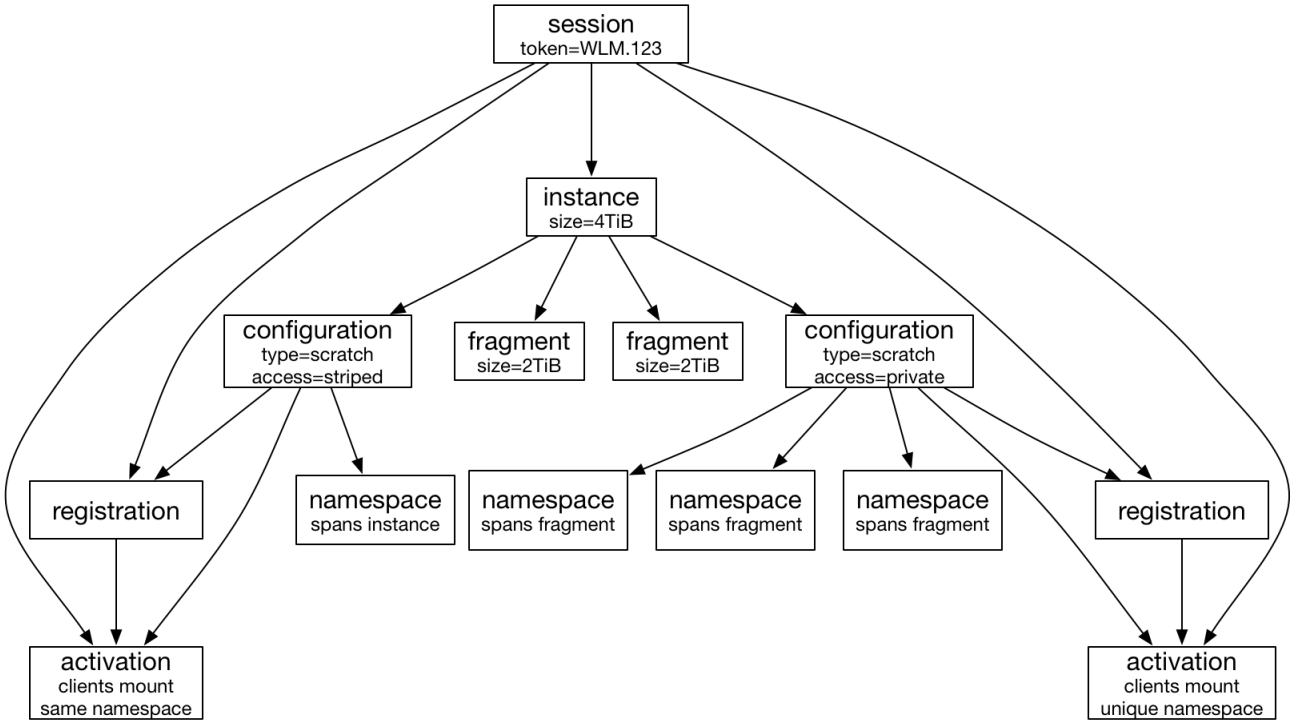
Diagrammatic View of Batch Jobs

EXAMPLE: DataWarp job instance, no staging

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

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

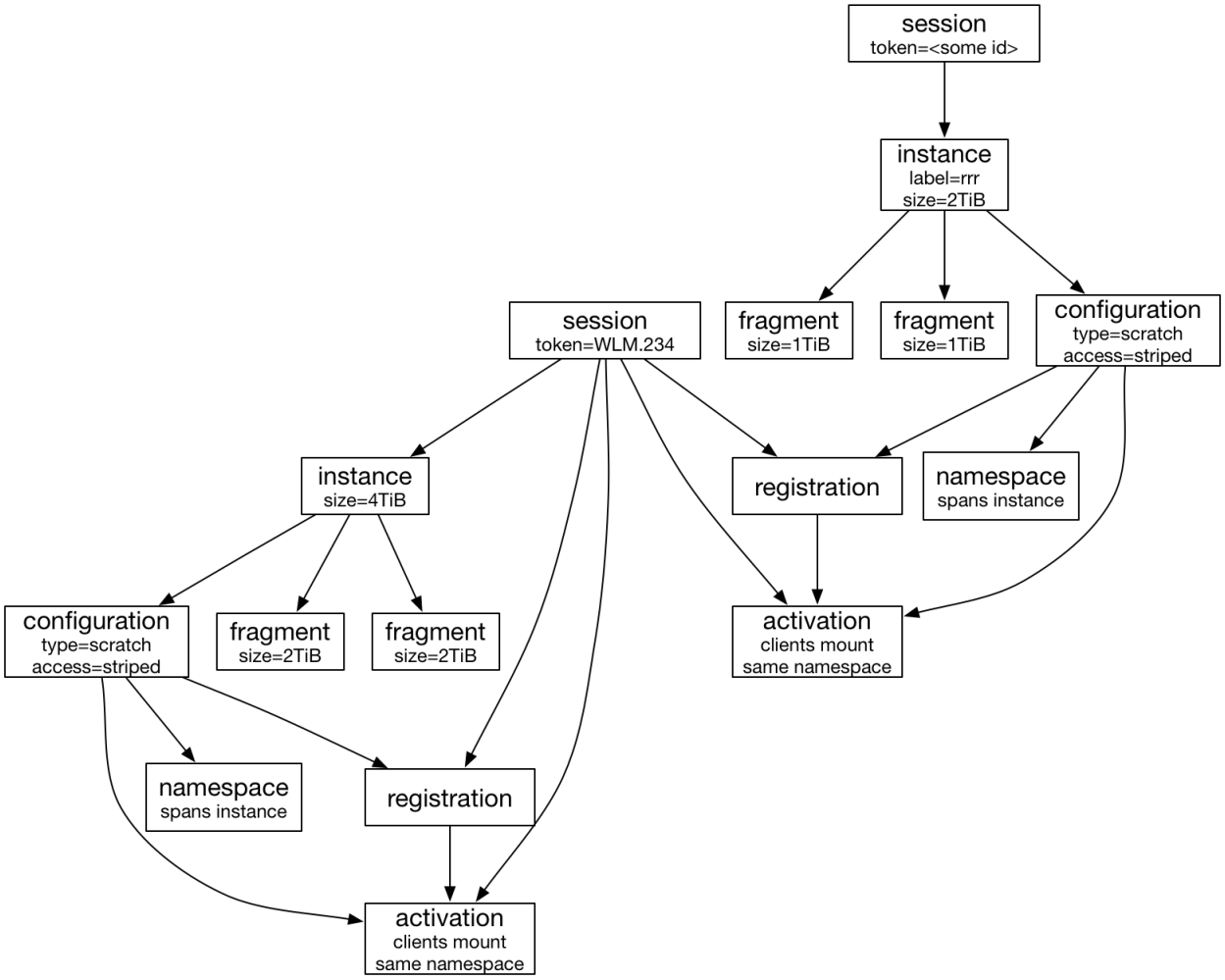
If any of the referenced boxes are removed (e.g., `dwcli rm session --id` *id*), then all boxes that it points to, recursively, are removed. In this example, the scratch stripe configuration gets one namespace and the scratch private configuration gets three namespaces, one for each compute node. The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



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
```


session
token=<some id>
instance
label=rrr
size=2TiB
session
token=WLM.234
fragment
size=1TiB
fragment
size=1TiB
configuration
type=scratch
access=striped
instance
size=4TiB
registration
namespace
spans instance
configuration
type=scratch
access=striped
fragment
size=2TiB
fragment
size=2TiB
activation
clients mount
same namespace
namespace
spans instance
registration
activation
clients mount
same namespace

libdatawarp - the DataWarp API

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

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

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

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

  ```
  > module load datawarp
  ```

dw_get_stripe_configuration

- Get stripe configuration

SYNOPSIS

```
int dw_get_stripe_configuration(int fd,
       int *stripe_size,
       int *stripe_width,
       int *starting_index)
```

DESCRIPTION

The `dw_get_stripe_configuration` function returns the current stripe configuration for a file. The *`stripe_width`* represents the maximum number of stripes the file can have, not the current number it is actually using. For a file that has no stripe configuration, `dw_get_stripe_configuration` returns the default values for the DataWarp type and access mode of the file.

PARAMETERS

`fd`	Open file descriptor of the file or directory to be queried
`stripe_size`	Pointer to an int to receive the size of a stripe.
`stripe_width`	Pointer to an int to receive the maximum number of stripes available to the file.

starting_index Pointer to an int to receive the stripe index (between `0` and *`stripe_width`*) for file offset `0`

RETURN VALUES

0 Success

<0 Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_query_directory_stage

- Query stage operations for a DataWarp directory

SYNOPSIS

```
int dw_query_directory_stage(const char *dw_directory_path,
        int *complete,
        int *pending,
        int *deferred,
        int *failed)
```

DESCRIPTION

The `dw_query_directory_stage` function queries all files within a directory and all subdirectories. Files and directories the caller does not have permission to read are skipped and not included in any count.

PARAMETERS

dw_directory_path	Path of DataWarp directory to query.
complete	The number of completed stage operations.
pending	The number of pending (active and waiting) stage operations.
deferred	The number of deferred stage operations.
failed	The number of failed stage operations.

RETURN VALUES

0	Success.
-ENOENT	Specified directory does not exist.
-EINVAL	Invalid *`dw_directory_path`* argument.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_query_file_stage

- Query stage operations for a DataWarp file

SYNOPSIS

```
int dw_query_file_stage(const char *dw_file_path,
        int *complete,
        int *pending,
        int *deferred,
        int *failed)
```

PARAMETERS

dw_file_path	Path of DataWarp file to query.
complete	The number of completed stage operations.
pending	The number of pending (active and waiting) stage operations.
deferred	The number of deferred stage operations.
failed	The number of stage operations that failed.

RETURN VALUES

0	Success.
-ENOENT	Specified file does not exist.
-EINVAL	Specified file has not had a stage operation initiated on it.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

For a single file query, only one of the returned counts will be non-zero.

dw_query_list_stage

- Query stage operations for all files in a list

SYNOPSIS

```
int dw_query_list_stage(const char **dw_list_path,
        int *complete,
        int *pending,
        int *deferred,
        int *failed)
```

DESCRIPTION

The `dw_query_list_stage` function queries stage operations for all files from *dw_list_path* that exist at the time the call is made.

PARAMETERS

dw_list_path	A NULL-terminated list of DW files to query.
complete	The number of completed stage operations.

pending	The number of pending (active and waiting) stage operations.
deferred	The number of deferred stage operations.
failed	The number of stage operations that have failed.

RETURN VALUES

0	Success.
-ENOENT	A file specified in the list does not exist.
-EINVAL	A file specified in the list has not had a stage operation initiated on it.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_set_stage_concurrency

- Set the maximum number of concurrent stage operations

SYNOPSIS

```
int dw_set_stage_concurrency(const char *dw_instance_path, unsigned int
concurrent_stages)
```

DESCRIPTION

The `dw_set_stage_concurrency` function sets the maximum number of concurrent stage operations.

PARAMETERS

dw_instance_path	Path to the DW instance for which the concurrency value is to be set. The path must exist and be readable.
concurrent_stages	The maximum number of concurrent stage operations the system can keep active. The value supplied applies to the entire instance and may be silently constrained by the DataWarp infrastructure. The concurrent stages are evenly distributed across all server nodes in the target instance. Changes to *concurrent_stages* may take some amount of time to become effective.

RETURN VALUES

0	Success
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_stage_directory_in

- Stage in all files from a PFS directory

SYNOPSIS

```
int dw_stage_directory_in(const char *dw_directory, const char *pfs_directory)
```

DESCRIPTION

The `dw_stage_directory_in` function stages all regular files (that exist, are readable and have no hard links when the call is made) from *`pfs_directory`* into *`dw_directory`*. Individual files are staged asynchronously; nested directories are staged recursively. The PFS must be mounted on the compute nodes and DataWarp service nodes using the same path.

PARAMETERS

`dw_directory`	Path to the DataWarp directory. The directory must exist and be writable.
`pfs_directory`	Path to the PFS directory. All files in this directory are staged in to *`dw_directory`*. The directory must be readable.

RETURN VALUES

0	Success.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_stage_directory_out

- Stage files to the PFS

SYNOPSIS

```
int dw_stage_directory_out(const char *dw_directory,
       const char *pfs_directory,
       enum dw_stage_type stage_type)
```

DESCRIPTION

The `dw_stage_directory_out` function stages all regular files (that exist, are readable, and have no hard links when the call is made) in *`dw_directory`*to *`pfs_directory`*. Individual files are staged asynchronously; nested directories are staged recursively. The PFS must be mounted on the compute nodes and DataWarp service nodes using the same path.

When this API returns, the stage state is persistent. When this API is used to swap the `DW_STAGE_AT_JOB_END` state of 2 files (e.g., set `DW_STAGE_AT_JOB_END` on one and `DW_REVOKE_STAGE_AT_JOB_END` on another), the order of the calls must be considered:

- If `DW_REVOKE_STAGE_AT_JOB_END` is done first, a DataWarp infrastructure failure may result in neither file being staged at job end.

- If `DW_STAGE_AT_JOB_END` is done first, a DataWarp infrastructure failure may result in both files being staged at job end.

Stage out requests that are active when an instance is deleted (when a job ends for a job instance) will continue until they either complete successfully or fail. For job instances, the job does not complete and the storage is not released until the stage out completes. Any failures of stage out requests in progress when the job exits are reported in the batch jobs `stderr` file.

PARAMETERS

dw_directory Path to the DataWarp directory; the directory must be readable.

pfs_directory Path to the PFS directory; the directory must exist and be writable.

stage_type The type of stage behavior requested:

- `DW_STAGE_IMMEDIATE`: Stage the file as soon as possible.
- `DW_STAGE_AT_JOB_END`: Defer staging the file until the job ends, either normally or abnormally.
- `DW_ACTIVATE_DEFERRED_STAGE`: Stage a directory previously staged with `DW_STAGE_AT_JOB_END` as soon as possible. The *pfs_directory* parameter is ignored.
- `DW_REVOKE_STAGE_AT_JOB_END`: Revoke a previous `DW_STAGE_AT_JOB_END` request. The *pfs_directory* parameter is ignored.

RETURN VALUES

0 Success.

<0 Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_stage_file_in

- Stage in a PFS file to DataWarp

SYNOPSIS

```
int dw_stage_file_in(const char *dw_file_path, const char *pfs_file_path)
```

DESCRIPTION

The `dw_stage_file_in` function initiates an asynchronous stage in from the PFS into the DataWarp instance. The DataWarp infrastructure initiates asynchronous stage operations on all DataWarp nodes associated with the file. When the stage is complete, `fsync()` is called for the DataWarp file. Only one stage request can be active on a file at any given time.

PARAMETERS

dw_file_path Path to the DataWarp file. If the file exists, it must be writable or an error occurs. If the file does not exist, it is created. The file is truncated before the stage in starts.

pfs_file_path Path to the PFS source file. The file must be a readable, regular file with no hard links.

RETURN VALUES

0 Success.

<0 Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_stage_file_out

- Stage out DataWarp file to PFS

SYNOPSIS

```
int dw_stage_file_out(const char *dw_file_path,
       const char *pfs_file_path,
       enum dw_stage_type stage_type)
```

DESCRIPTION

The `dw_stage_file_out` initiates an asynchronous stage out from the DataWarp instance into the PFS. The DataWarp infrastructure initiates asynchronous stage operations on all DataWarp nodes associated with the file. When the stage is complete, `fsync()` is called for the PFS file. Only one stage request can be active on a file at any given time.

When this API returns the stage state is persistent. When this API is used to swap the `DW_STAGE_AT_JOB_END` state of 2 files (e.g., set `DW_STAGE_AT_JOB_END` on one and `DW_REVOKE_STAGE_AT_JOB_END` on another), the order of the calls must be considered:

- If `DW_REVOKE_STAGE_AT_JOB_END` is done first, a DW infrastructure failure may result in neither file being staged at job end.
- If `DW_STAGE_AT_JOB_END` is done first, a DW infrastructure failure may result in both files being staged at job end.

Stage out requests that are active when an instance is deleted (when a job ends for a job instance) continue until they either complete successfully or fail. For job instances, the job does not complete and the storage is not released until the stage out completes. Any failures of stage out requests in progress when the job exits are reported in the batch job `stderr` file.

PARAMETERS

dw_file_path Path to the DataWarp file to be staged out; the file must be a readable, regular file with no hard links.

pfs_file_path Path to the PFS destination file. If the file exists, it must be writable or an error occurs. If the file does not exist, it is created. The file is truncated before the stage out starts.

stage_type The requested type of stage behavior:

- `DW_STAGE_IMMEDIATE`: Stage the file as soon as possible.
- `DW_STAGE_AT_JOB_END`: Defer staging the file until the job ends, either normally or abnormally.

- `DW_ACTIVATE_DEFERRED_STAGE`: Stage a file previously staged with DW_STAGE_AT_JOB_END as soon as possible. The *`pfs_file_path`* parameter is ignored.
- `DW_REVOKE_STAGE_AT_JOB_END`: Revoke a previous `DW_STAGE_AT_JOB_END` request; the `pfs_file_path` parameter is ignored.

RETURN VALUES

0 Success.

<0 Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_stage_list_in

- Stage in all files from a list

SYNOPSIS

```
int dw_stage_list_in(const char *dw_directory, const char **pfs_list)
```

DESCRIPTION

The `dw_stage_list_in` function stages all regular files (that exist, are readable, and have no hard links when the call is made) from *`pfs_list`* into *`dw_directory`*. Individual files are staged asynchronously.

PARAMETERS

dw_directory Path to the DataWarp directory; the directory must exist and be writable.

pfs_list A NULL-terminated array of pointers to the paths of the PFS files to be staged (directories are not supported). Each file in this list is staged in to *`dw_directory`*.

RETURN VALUES

0 Success.

<0 Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_stage_list_out

- Stage out all files in a list

SYNOPSIS

```
int dw_stage_list_out(const char **dw_list,
       const char *pfs_directory,
       enum dw_stage_type stage_type)
```

DESCRIPTION

The `dw_stage_list_out` function stages all files from *dw_list*, present at the time the call is made, into *pfs_directory*. Individual files are staged asynchronously.

When this API returns the stage state is persistent. When this API is used to swap the `DW_STAGE_AT_JOB_END` state of 2 files (e.g., set `DW_STAGE_AT_JOB_END` on one and `DW_REVOKE_STAGE_AT_JOB_END` on another), the order of the calls must be considered:

- If `DW_REVOKE_STAGE_AT_JOB_END` is done first, a DataWarp infrastructure failure may result in neither file being staged at job end.
- If `DW_STAGE_AT_JOB_END` is done first, a DW infrastructure failure may result in both files being staged at job end.

Stage out requests active when an instance is deleted (when a job ends for a job instance) continue until they either complete successfully or fail. For job instances, the job does not complete and the storage is not be released until the stage out completes. Any failures of stage out requests in progress when the job exits are reported to the batch job `stderr` file.

PARAMETERS

dw_list A NULL-terminated array of pointers to the paths of the DataWarp files to be staged (directories are not supported). Each file in this list is staged out to *pfs_directory*.

pfs_directory Path to the PFS directory; the directory must exist and be writable.

stage_type The requested type of stage behavior:

- `DW_STAGE_IMMEDIATE`: Stage the file as soon as possible.
- `DW_STAGE_AT_JOB_END`: Defer staging the file until the job ends, either normally or abnormally.
- `DW_ACTIVATE_DEFERRED_STAGE`: Stage the files in *dw_list* previously staged with `DW_STAGE_AT_JOB_END` as soon as possible. The *pfs_directory* parameter is ignored.
- `DW_REVOKE_STAGE_AT_JOB_END`: Revoke a previous `DW_STAGE_AT_JOB_END` request; the `pfs_file_path` parameter is ignored.

RETURN VALUES

0 Success

<0 Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_terminate_directory_stage

- Terminate stage operations

SYNOPSIS

```
int dw_terminate_directory_stage(const char *dw_directory_path)
```

DESCRIPTION

The `dw_terminate_directory_stage` fuction terminates one or more in-progress or waiting stage operations. If the stage is in progress, the amount of data written by the stage is undefined.

PARAMETERS

dw_directory_path	Path to the DataWarp directory of files for which staging is terminated.

RETURN VALUES

0	Success.
-ENOENT	Specified directory does not exist.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_terminate_file_stage

- Terminate a stage operation

SYNOPSIS

```
int dw_terminate_file_stage(const char *dw_file_path)
```

DESCRIPTION

The `dw_terminate_file_stage` function terminates an in-progress or waiting stage operation. If the stage is in progress, the amount of data written by the stage is undefined.

PARAMETERS

dw_file_path	Path of the DataWarp file for which staging is terminated.

RETURN VALUES

0	Success.
-ENOENT	Specified file does not exist.
-EINVAL	Specified file has not had a stage operation initiated on it.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_terminate_list_stage

- Terminate stage operations

SYNOPSIS

```
int dw_terminate_list_stage(const char **dw_list)
```

DESCRIPTION

The `dw_terminate_list_stage` function terminates one or more in-progress or waiting stage operations. If the stage is in progress, the amount of data written by the stage is undefined.

PARAMETERS

dw_list Pointer to a NULL-terminated list of DataWarp files for which staging is terminated.

RETURN VALUES

0	Success.
-ENOENT	A specified file does not exist.
-EINVAL	Specified file has not had a stage operation initiated on it.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_wait_directory_stage

- Wait for stage operations to complete

SYNOPSIS

```
int dw_wait_directory_stage(const char *dw_directory_path)
```

DESCRIPTION

The `dw_wait_directory_stage` function waits for one or all stage operations to complete. The calling process is blocked until the staging is complete. This request is not interruptible by a signal.

PARAMETERS

dw_directory_path Path to the DataWarp directory containing files for which staging must complete before the job continues.

RETURN VALUES

0	Success.
-ENOENT	Specified directory does not exist.
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_wait_file_stage

- Wait for stage operation to complete

SYNOPSIS

```
int dw_wait_file_stage(const char *dw_file_path)
```

DESCRIPTION

The `dw_wait_file_stage` function waits for a stage operation to complete for the target file. The calling process is blocked until staging is complete. This request is not interruptible by a signal.

PARAMETERS

dw_file_path Path to DataWarp file for which staging must complete before the job continues.

RETURN VALUES

0	Success.
-ENOENT	Specified file does not exist.
-EINVAL	Specified file has not had a stage operation initiated on it
<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

dw_wait_list_stage

- Wait for stage operations to complete

SYNOPSIS

```
int dw_wait_list_stage(const char **dw_list)
```

DESCRIPTION

The `dw_wait_list_stage` function waits for one or all stage operations to complete. The calling process is blocked until staging is complete. This request is not interruptible by a signal.

PARAMETERS

dw_list Path to NULL-terminated list of DataWarp files for which staging must complete before the job continues.

RETURN VALUES

0	Success.
-ENOENT	A specified file does not exist.
-EINVAL	Specified file has not had a stage operation initiated on it.

<0	Error; the negative integer value represents the error (as defined in Linux `errno.h`).

Failed Stage Identification

- Identify failed stages

SYNOPSIS

```
int dw_open_failed_stage (const char *dw_instance_path, dw_failed_stage_t **handle)
```

```
int dw_read_failed_stage (dw_failed_stage_t *handle, char *path,
       int path_size,
       int *ret_errno)
```

```
int dw_close_failed_stage (dw_failed_stage_t *handle)
```

DESCRIPTION

These functions are used in combination to identify failed stages.

PARAMETERS

dw_instance_path	DataWarp directory path from which to begin file tree walk
path	Pointer to memory to receive the DataWarp path name of a failed stage
path_size	Maximum size of the path to be returned to the path parameter
ret_errno	Errno reason that the stage failed
handle	Value returned from a prior `open_failed_stage`

RETURN VALUES

open_failed_stage	▪ ==0: Success, pointer to newly created dw_failed_stage_t *handle* for use in `read_failed_stage` and `close_failed_stage` returned in *handle* argument ▪ <0: Error; the negative integer value represents the error (as defined in Linux `errno.h`).
read_failed_stage	▪ >0: *path* and *ret_errno* returned successfully. Return value is length of returned path. ▪ ==0: Completed path walk, no more failed stages. ▪ <0: Error; the negative integer value represents the error (as defined in Linux `errno.h`).
close_failed_stage	▪ ==0: Success, *handle* has been cleaned up.

- <0: Error; the negative integer value represents the error (as defined in Linux errno.h).

read_failed_stage returns the DataWarp path of a stage operation that has failed. It should be called multiple times to discover all failed stage operations. A file remains a failed stage until the file is either unlinked, the stage is terminated or the stage is restarted. To restart a failed stage scan, the *handle* must be closed and reopened (i.e., there is no equivalent of lseek).

If the list of failed stages changes asynchronously after an open_failed_stage is started those changes may not be visible to that instance of open_failed_stage. Changes made to a failed stage already seen by an instance of open_failed_stage (for example terminating or restarting a stage or unlinking the file) will not affect that instance.

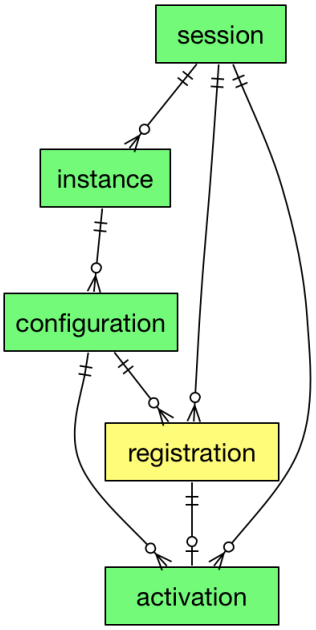
EXAMPLE

```
dw_open_failed_stage("dw_instance_path", &hdl);
rval = dw_read_failed_stage(hdl, &buf, 1024, &errval);
while (rval > 0) {
   printf("Stage failed on: %s, errno: %d\n", buf, errval);
   rval = dw_read_failed_stage(hdl, &buf, 1024, &errval);
}
dw_close_failed_stage(hdl);
```

Terminology

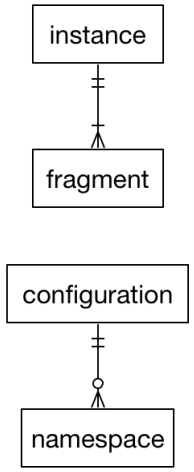
The following diagram shows the relationship between the majority of the DataWarp Service terminology using Crow's foot notation. A session can have 0 or more instances, and an instance must belong to only one session. An instance can have 0 or more configurations, but a configuration must belong to 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 4. 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 5. Instance/Fragment ↔ Configuration/Namespace Relationship



Instance	A specific subset of the storage space comprised of DataWarp fragments, where no two fragments exist on the same node. An instance is essentially raw space until there exists at least one DataWarp instance configuration that specifies how the space is to be used and accessed.
DataWarp Service	The DataWarp Service (DWS) manages access and configuration of DataWarp instances in response to requests from a workload manager (WLM) or a user.
Fragment	A piece of an instance as it exists on a DataWarp service node
Job Instance	A DataWarp instance whose lifetime matches that of a batch job and is only accessible to the batch job because the `public` attribute is not set.
Namespace	A piece of a scratch configuration; think of it as a folder on a file system.
Node	A DataWarp service node (with SSDs) or a compute node (without SSDs). Nodes with space are server nodes; nodes without space are client nodes.
Persistent Instance	A DataWarp instance whose lifetime matches that of possibly multiple batch jobs and may be accessed by multiple user simultaneously because the `public` attribute is set.
Pool	Groups server nodes together so that requests for capacity (instance requests) refer to a pool rather than a bunch of nodes. Each pool has an overall quantity (maximum configured space), a granularity of allocation, and a unit type. The units are either bytes or nodes (currently only bytes are supported). Nodes that host storage capacity belong to at most one pool.
Registration	A known usage of a configuration by a session.
Server Node	An IO service blade that contains two SSDs and has network connectivity to the PFS.
Session	An intagible object (i.e., not visible to the application, job, or user) used to track interactions with the DWS; typically maps to a batch job.

Prefixes for Binary and Decimal Multiples

Multiples of bytes						
SI decimal prefixes				IEC binary prefixes		
Name	Symbol	Standard SI	Binary Usage	Name	Symbol	Value
kilobyte	kB	10^3	2^{10}	kibibyte	KiB	2^{10}
megabyte	MB	10^6	220	mebibyte	MiB	2^{20}
gigabyte	GB	10^9	2^{30}	gibibyte	GiB	2^{30}
terabyte	TB	10^{12}	2^{40}	tebibyte	TiB	2^{40}
petabyte	PB	10^{15}	2^{50}	pebibyte	PiB	2^{50}
exabyte	EB	10^{18}	2^{60}	exbibyte	EiB	2^{60}
zettabyte	ZB	10^{21}	2^{70}	zebibyte	ZiB	2^{70}
yottabyte	YB	10^{24}	2^{80}	yobibyte	YiB	2^{80}

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