TIBCO DataSynapse GridServer® Manager

PDriver Tutorial

Version 7.1.0
July 2022

Document Updated: September 2022

Contents

Using PDriver Tools

This guide serves as a tutorial for PDriver, the parametric Job Driver. The following tutorial describes PDriver and explains its use.

This section explains the basics behind using each of the PDriver tools, including `pdriver`, `bsub`, `bcoll`, `bstatus`, and `bcancel`. These commands are the basis of using PDriver, and are used in further sections to run and control PDS scripts.

PDriver tools are included in the GridServer SDK. For more information about installation, see the *TIBCO GridServer® Installation*TIBCO GridServer® Installation.

Product Overview

PDriver, or the Parametric Job Driver, is a Driver that can execute command-line programs as a parallel processing service using the GridServer environment. This enables you to take a single program, run it on several Engines, and return the results to a central location, without writing a program in Java or C++.

PDriver achieves parallelism by running the same program on Engines several times with different parameters. A script is used to define how these parameters change. For example, a distributed search mechanism using the `grep` command can conduct a brute-force search of a network-attached file system, with each task in the Service being given a different directory or piece of the file system to search.

One way PDriver scripts can achieve parallelism is to iteratively change the value of variables that are passed to successive tasks as parameters. A script can step through a range of numbers and use each value as a parameter for each task that is created. Or, a variable can be defined containing a list of parameters.

PDriver uses its scripting language, called *PDS*, to define how jobs occur. These scripts can also be used to set options for a PDriver Service, such as remote logging and exit code checking.

Before You Begin

This guide assumes that you already have a Manager running and know the hostname, username, and password. If this isn't true, see *TIBCO GridServer® Installation* or contact your administrator.

You must also install the GridServer SDK, which includes PDriver and code examples. See *TIBCO GridServer® Installation*.

Running PDriver

There are multiple ways to start PDriver and run a script, depending on if you want to do so synchronously or asynchronously. We'll briefly explain each method, and then show how to run some sample code included with the GridServer SDK.

To start, open a command shell and change directories to the root level of the GridServer SDK, then change to the `pdriver` directory.

Running Synchronously

The simplest way to start PDriver is to run it synchronously. This runs through each block of the PDS script, waiting for each step to complete. This is the easiest way to run a test script, provided you don't mind waiting for the results.

For example, here is a very simple PDS script, let us call it `simplejob.pds`:

```
job simplejob

task 10
    log "hello, world"
end
end job
```

This PDS script contains a single `job` block, and nothing more. It creates ten tasks and each one writes a hello message to the Engine log.

Run this script by typing the following code:

```
pdriver simplejob.pds
```

This launches PDriver, sends the ten tasks to the grid, and waits for the results to return. The Driver log messages are printed on your console window, and you can see as each task is submitted to and received from the Engines running on your grid. The Driver logs are also captured in a timestamped file created in a `log` directory; this directory is created in the directory PDriver is invoked in.

This quick example doesn't do much, except for writing to a log. But a more complex script gives each Engine an executable to run, with different inputs for each task. More advanced scripting can possibly move data files or initialize databases before jobs or tasks are run, and results can be collected after each job or task.

For now, you can see the results of `simplejob` in the Engine log files. To do this:

1. In the Administration Tool, go to **Grid Components > Engines** > **Engine Admin**.
2. Select an Engine from the list.
3. Click the **Actions** link, and select **Log Files**. This opens a window containing a list of links to the Engine logs on that Engine.
4. Click a link to view the most recent log, and you'll see the "hello, world" message written by each task of the simple job.

Running Asynchronously

While it might be simple to run PDriver synchronously, waiting for the results to return might be impractical in a production scenario, especially if each task takes time to complete. An alternative is to run the PDS script asynchronously, using the `bsub` flag. This runs the entire script except for post-job processing. Instead of waiting for tasks to complete, it immediately returns a batch ID, which can later be used to fetch results.

Let's modify the above `simplejob.pds` to add a `postjob` block with a hypothetical command. Here, it is another simple `log` command, but in a real job, it might be a command that copies, collates, or otherwise post-processes task results into a final result. Let us call this script `simplejob2.pds`:

```
job simplejob

task 10
    log "hello, world"
end
postjob
    log "job processed."
```

```
end
end job
```

Run this script by typing the following command:

```
pdriver -bsub simplejob2.pds
```

In the Driver logs displayed on the screen, there are lines that look similar to the following lines:

```
03/02 14:54:16:265 Info: Batch job has been submitted
03/02 14:54:16:265 Info: Collect outputs using batch id:
61924497789921878
```

The batch ID is important; use this to retrieve your results. In a production scenario, you can extract this ID from the Driver log file with grep or some similar approach.

After the job has been submitted, it runs on its own, and the results are queued. You can retrieve the results when you are ready by typing the following command:

```
pdriver -bcoll 61924497789921878 simplejob2.pds
```

Running Commands

Aside from running PDS scripts to iteratively run several instances of an executable with different inputs, it's also possible to run PDriver to run a single command on the Grid, and then collect the results later. This ca nbe used to run a one-time command, or to use another scripting language to run a multi-task job.

The `bsub` command can be used to run a command on a single Engine in the Grid. For example, if you have Windows Engines on your Grid, you can type the following command (UNIX users must replace the `dir` with `ls`.):

```
bsub dir
```

This submits a single-task job to an Engine, and returns the batch ID in the Driver logs.

The `bcoll` command is used to collect the results after using `bsub`. To collect the results, you simply need to give `bcoll` the batch ID. For example:

```
bcoll 149463216266848451
```

The results are fetched into a new directory with the same name as the batch ID. The directory contains a `bsub.err` file with the stderr output of the job, and a `bsub.out` file containing the stdout output of the job. In this case, the `bsub.out` file contains a directory listing of the Engine that ran the job.

Other PDriver Commands

In addition to `pdriver` and the `bsub` and `bcoll` commands, there are two other utilities you can use when running PDriver jobs.

First, the command `bstatus` returns information about job, batch, and Engine status. For example, type the following command:

```
bstatus -jobs
```

This displays the name, job IDs, tasks, priority, and status of all of the PDriver jobs recently submitted. You can also use the options `batches` and `engines` to display information about recently submitted batches and available Engines.

Another common task is stopping a submitted job. This is a two part task: removing pending jobs, and killing running jobs.

To remove a pending batch job, use the `bcancel` command, along with a batch ID. The batch ID is the same one returned when you initially ran `pdriver -bsub` or `bsub`. If you don't have the batch ID, you can look at the results of `bstatus -batches` to determine it.

The `bcancel` command removes jobs that are still waiting in the queue for execution, but it can't stop a job that has already started running and that now exists as one or more tasks on your grid.

To stop a running job, you can use another `bstatus` option called `canceljob`. But instead of the batch ID, you need to determine the Service ID of the job. This can be found by using `bstatus -jobs`, or by looking at the **Services > Services > Service Session Admin** page in the Administration Tool. To cancel a job, type the following command, where *serviceid* is the Service ID of the job:

```
bstatus -canceljob serviceid
```

Summary

- The `pdriver` command can run a PDS script synchronously.
- To run a PDS script asynchronously, use `pdriver -bsub`.
- Retrieve PDS script results with `pdriver -bcoll`.
- `bsub` and `bcoll` can be used to submit a single command and recall its results from the grid.
- Status on batches, jobs, and Engines can be displayed with `bstatus`.
- To kill a pending batch, use `bcancel` and a batch ID; to kill a running batch, user `bstatus -canceljob` and a Service ID.

PDS Structure and Variables

PDriver uses its scripting language, called *PDS*, to define how jobs occur. These scripts can also be used to set options for a PDriver Service, such as remote logging and exit code checking. This section describes the various parts of a typical PDS script and how to write a basic PDS script.

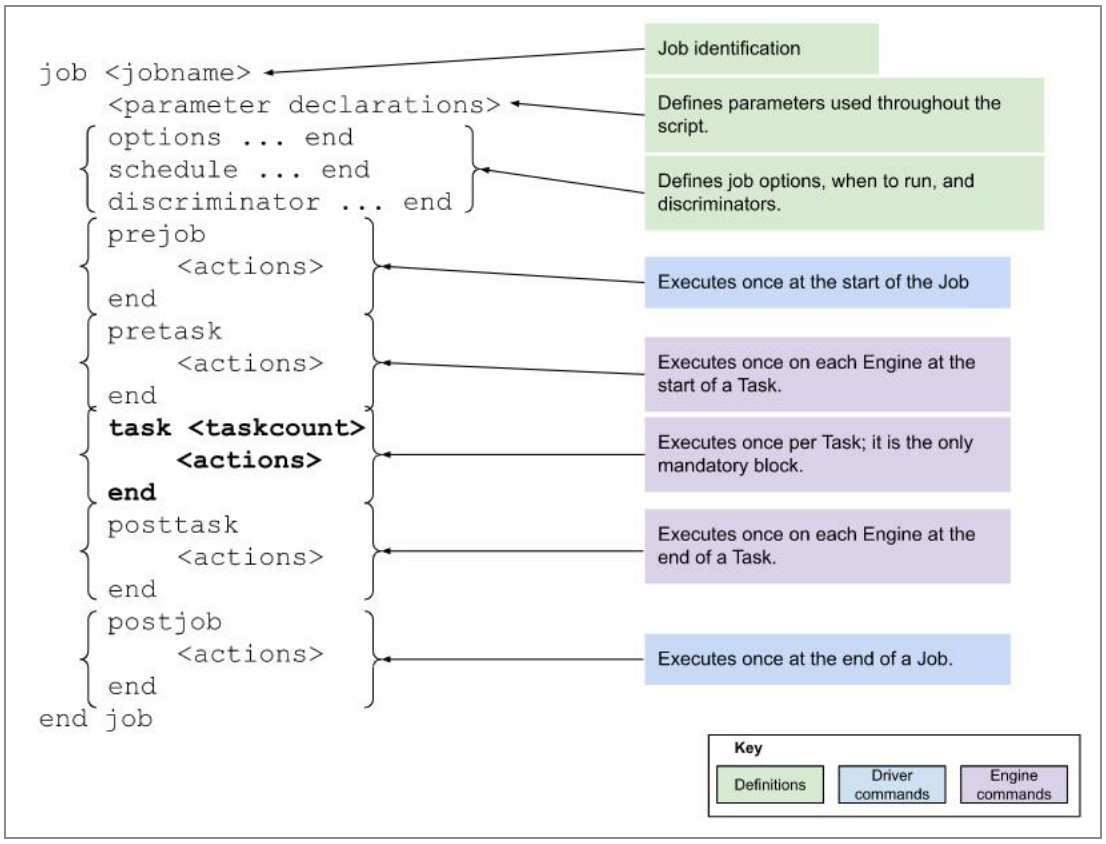
For a complete reference to the PDS scripting language, see the *TIBCO GridServer® Developer's Guide*.

PDS Structure

A PDS script typically coordinates three parts of a PDriver job:

1. Provide a pre-processing step to distribute inputs.
2. Provide a processing step to be done on Engines in parallel.
3. Provide a post-processing step to consolidate outputs.

To accomplish this, a PDS script is structured into different *blocks*, which are run at different points in the job's lifecycle. Each block can contain statements (or actions) or other specific options. This section explains what blocks are used in a basic script, and what the other blocks are for.



Structure of a PDS file

The Basic Script

All of the blocks in a PDS script are optional except two. Each PDS script requires the following blocks:

1. A basic PDS script consists of a `job` block. This begins with the line `job` *jobname*, where *jobname* is the name of your PDS job, and ends with `end job`. More advanced scripts can contain multiple `job` blocks which are run serially by default.
2. The other required block is the `task` block which is in the `job` block, and begins with `task` *taskcount* and ends with `end`.

Our `simplejob.pds` script from the previous section, contains these two required blocks:

```
#From the last section
job simplejob

task 10
    log "hello, world"
```

```
end
end job
```

A few other points to mention about PDS scripts. First, PDS is case-sensitive. Also, in PDS, whitespace is not significant. In fact, you can include multiple statements on one line, or even write an entire PDS script on a single line, although it's easier to read with a statement per line. And any text after a '#'sign on a line iss ignored, and can be used as a comment, as shown above.

Note that each of the remaining blocks we discuss typically begin with the name of the block and end simply with an `end`.

The Options Block

One of the additional blocks you can define in a PDS script is the `options` block. Within it, you can select set job options and descriptions, job priorities, and other settings.

```
options
    onerror ( fail | retry | ignore )
    maxFailedTaskRetries = val (default: 3)
    enableBlacklisting = true|false>
    jobPriority = val (default: 5)
    autoCancelMode = "always" | "never" | "libloadfailure"
    jobOption key val
    jobDescription key val
end
```

The Schedule Block

When submitting a job asynchronously with `bsub`, you can declare a start time using the `schedule` block, and also send an email when a Service is complete. This block has no effect if the job is submitted synchronously.

The `schedule` block can contain a `type` property, which defines what type of schedule is used. The possible values are relative, for defining a Service that starts a certain number of minutes after submission, and absolute, for a Service that starts at a defined time.

After inserting a type property, you must insert a time property to indicate the time when the Service must start. For relative services, use the `minuteDelay` property, set to an

integer. For absolute properties, use the `startTime` property, set to a string in the `"mm/dd/yy hh:mm AM|PM"` format.

Allowed schedule declarations are:

Relative

```
schedule
    type = relative
    minuteDelay = val
end
```

Absolute

```
schedule
    type = absolute
    startTime = "mm/dd/yy hh:mm AM|PM"
end
```

With either type, declaring `email="string"` in the `schedule` block sends an email to the address given in the string when the Service is complete.

The Discriminator Block

Discrimination is a feature of GridServer that allows you to selectively use Engines based on their properties. They enable you to specify if a job or a task can be taken by certain Engines. Normally, discriminators are attached to Services using the GridServer API. However, with PDriver, you can specify job-level or task-level discriminators in your PDS script.

In the `discriminator` block of a PDS script, you can declare discriminators, which consist of a property name, a comparator, and a value:

```
property-name  ==|!=|<|>|<=|>=  param-value
```

For example, putting the following code in a `discriminator` block discriminates against Engines with a value of 350 or less in the `cpuMHz` Engine property:

```
cpuMHz > 350
```

Declaring a discriminator creates a job-level discriminator by default. You can also create a discriminator that requires conditions to be met before the discriminator is applied to a task. This is accomplished by using an `affects` clause in the `discriminator` block.

Use of the `affects` clause specifies conditions that must be met for the discriminator to be applied to a particular task. For example, the following discriminator block shows how to apply the same discriminator as above, but only for the first five tasks:

```
discriminator
    affects
        $DSTASKID <= 5
      end
    cpuMHz > 350
end
```

Multiple discriminator blocks with `affects` clauses can be used.

The property name refers to Engine properties. A predefined set of properties are assigned to all Engines by default. You can assign additional properties to Engines by selecting **Edit/View Properties** from the Actions list in **Grid Components > Engines > Daemon Admin** in the Administration Tool.

Prejob and Pretask Blocks

PDriver Jobs and tasks might require preprocessing. For example, you might need to prepare files, move them to an Engine, start a database, or do other preparation before the `task` block begins. To provide this functionality, two blocks are available to run before the task block: `prejob` and `pretask`. The two blocks start with either `prejob` or `pretask` and end with an `end`, and they can contain any number of statements. Also, both blocks of code must be located before the `task` block in your PDS script.

One difference between these two blocks is that `prejob` runs once, at the start of the job, and `pretask` runs once per Engine, prior to the start of the first task to run on that Engine. Also, the `prejob` block code executes on the Driver (that is, the machine on which you are running PDriver), and the `pretask` block code executes on the Engine.

```
prejob
    statements
end
```

```
pretask
     statements
end
task taskcount
     statements
end
```

Posttask and postjob blocks

After a PDriver task or job runs, you might want to run code to do some postprocessing. For example, you can use `posttask` blocks to clean up the state on the Engine. You can use the `postjob` blocks to move results to other locations. Like the `pretask` and `prejob` blocks, simply surround your code with a `posttask` or `postjob` keyword at the beginning, and an end keyword at the end. Locate both blocks of code after the task block in your PDS script.

Code in the `posttask` block runs on each Engine after the job completes, while code in the `postjob` block runs once on the Driver. One important difference in the `posttask` and `postjob` blocks is when they are executed. The `posttask` block happens after all of the tasks are completed. There's no guarantee that they happen immediately after the tasks are done, but they finish before the job ends. If you are running your job synchronously, the `postjob` block runs when the job finishes. And if you used `pdriver -bsub` to submit a job, the `postjob` block runs when you collect the results.

Variables and Parameters

There are several items implemented in PDS to make it easy to manipulate variables and use them as parameters. This section explains the basics.

The Basics

PDS has two primitive types: strings and floating-point numbers. You don't need to declare variables, and they can change types. Variables are referenced by preceding them with a dollar sign. Values are converted to the appropriate type depending on context. For instance, a string is converted to a number when it appears in an arithmetic expression. The grammar forbids certain combinations of expressions to catch common mistakes.

For example, if you add a string to a number, the string is converted to a number.

```
a = 7
b = "1.5"
c = $a + $b                 # value is 8.5
d = 7 + "1.5"               # disallowed
```

You can also do variable substitution within quotes. Use braces around a variable to separate it from adjacent text. For example:

```
a = "New"
b = "$a York"                # b equals "New York"
c = "${a}ark"                # c equals "Newark"
```

About Scope

You can assign variables within the `job` block or within any other block, and they are local to that block. For example, if you assign a variable in the job block but outside and before any other block, you can use it in the `prejob`, `postjob`, `pretask`, or `posttask` blocks. Inside a block, a variable can be assigned as global variable, which is visible within other related blocks, with the following syntax:

```
    global a = 5.2
```

Summary

- A PDS script typically coordinates three parts of a PDriver job: provide a pre-processing step to distribute inputs; provide a processing step to be done on Engines in parallel; and provide a post-processing step to consolidate outputs.
- A PDS script is structured into different blocks, which are run at different points in the job's lifecycle.
- A basic PDS script consists of a `job` block.
- The other required block is `task` block.

PDS Statements

Statements are used within a PDS script to execute commands, manipulate files, log messages, and perform other tasks that you might need to do during a job. They can also be used to add more advanced control to a PDS script, with statements like `if`, `for`, and `depends`.

This section describes common statements used in PDS scripts.

Built-in statements

Built-in statements are OS-independent commands that can be used inside any block in a PDS script. There are a few exceptions, however. You cannot use `onerror`, and `throw` in every block. For more information, see *TIBCO GridServer® Developer's Guide*.

Arguments to all statements except for `onerror` must be enclosed in quotes. Quoted arguments might contain line breaks only if they are escaped by backslashes.

Command Execution

To execute a command on a local machine within a PDS script, you can use the `execute` statement. A subshell is spawned for the command, unless specified otherwise with the shell command. The subshell has the path available on the Engine or Driver. The execute statement also enables you to set the stdin, stdout, and stderr used by the command. For example, the following script is a simple PDS script that uses `execute`:

```
job simplecommand
task 1
    execute stdin="input.txt" stdout="output.txt"
    stderr="error.txt"
        "sort"
end
end job
```

This script creates one task, which takes the contents of `input.txt`, sends them to the `sort` command on an Engine, collects the results into the file `output.txt`, and collects any

errors in `error.txt`. All three files are located on the Engine in the Engine's installation directory.

You can also use the "backtick" syntax (such as `‘command‘`) to do the same thing as `execute`, but the input, output, error, and shell cannot be set.

The `shell` statement can be used to specify the shell to be used for commands run by the `execute` command. The default is `/bin/bash` for UNIX and `cmd.exe` for Windows. This can be set to `none`, whereupon no shell is spawned.

You can also change the behavior of a script based on the return code of a command run with the `execute` statement by using the `onerror` statement. For more information about the `onerror` statement, see the *TIBCO GridServer® Developer's Guide*.

File Manipulation

Several built-in statements enable you to manipulate files and directories on the Driver or Engines. These can be used to move input and output files to and from the Engine.

The `copy` statement copies a file. Typically, the `prejob` block is used to copy input files from the Driver machine to the staging directory on the Manager, which can be referenced as `$DSSTAGEDIR`. In the `task` or `pretask` block, one can then copy that file from the staging directory to the Engine's local filesystem. Output files can be copied in the reverse direction.

You can remove files from the Driver or Engine using the `delete` statement. The wildcard `*` is supported.

Similarly, the `mkdir` and `rmdir` statements create or remove directories. To use `rmdir`, the directory must be empty.

The file manipulation statements are not OS-dependent and pathnames are automatically translated to work on the appropriate platform. For example, `mkdir "sample/log1"` creates `sample\log1` on Windows systems.

The following script uses files for standard input, output, and errors. It essentially copies everything from the input file to the output file.

```
job simplefile
prejob
    copy "hello.txt" $DSSTAGEDIR
end
task 1
```

```
    copy "$DSSTAGEDIR/hello.txt" $DSWORKDIR
    execute stdin="$DSWORKDIR/hello.txt" stdout="$DSWORKDIR/output.txt"
stderr="error.txt"
        "sort"
    copy "$DSWORKDIR/output.txt" $DSSTAGEDIR
end
end job
```

Other Built-in Statements

Several more built-in statements are available. The following table contains statements that might be useful in your PDS scripts.

Statement	Description
`log "`*log-message*`"`	Writes a message to the Manager log (in prejob or postjob blocks) or the Engine log (in pretask, task, or posttask blocks.)
`env (variable)`	Returns the value of the specified environment variable. If the variable is a string literal, it must be enclosed in quotes. An empty string is returned for nonexistent variables
`throw "`*message*`"`	Causes the task to fail, and throws an exception. The message is displayed on the Driver and written into the Driver log. This cannot be used in the posttask block.

Conditional Statements

Several conditional statements can be used within blocks to control the flow of a PDS script.

The If Statement

The syntax of the basic if statement is

```
if expression comparisonOperator expression then
    statements
end
```

PDS also supports `elsif` and `else` clauses. The `if` statement can be used inside a block (`prejob`, `pretask`, `task`, `posttask`, `postjob`), to conditionally execute statements. One typical use is to execute different commands depending on the Engine's operating system:

```
if $DSOS == "win32" then
    cmd = "dir"
else
    cmd = "ls"
end
execute "$cmd"
```

An `if` statement can also appear at the top level of a PDS script, to include or exclude global assignments or entire blocks. An example is

```
if $1 == "alwaysCancel" then
    options
        autoCancelMode "always"
    end
end
```

The `if` statement is not legal in the `options`, `schedule`, or `discriminator` blocks.

The For Statement

Although explicit looping over an array is not often needed, it can be achieved with the `for` statement:

```
for variable in expression
    statements
end
```

The expression must evaluate to an array. The variable takes on successive elements of the array on each iteration of the loop. Assignment to the loop variable is legal, but has an effect only for the remainder of that iteration. It does not alter the array.

The Include Statement

You can include another PDS script within a PDS script by using the `include` statement. For example:

```
include "filename"
```

This includes a PDS script contained in `filename`. If the filename is relative, it is interpreted as relative to the working directory from which you run PDriver. The PDS file must contain a full job, and can be used within any block.

The Depends Statement

Multiple `job` blocks can be included in a single PDS file, and by default, they run sequentially. It is also possible to define jobs that run other jobs that require the completion or failure of the current job using the `depends` statement. `depends` is used within a job block, and defines if another job after the current job in the PDS file must run either on the completion or failure of the current job. For example, the following code within a job runs `firstjob` if the current job succeeds, `secondjob` if it fails, and `thirdjob` in either case:

```
depends
    firstjob succeeds
    secondjob fails
    thirdjob succeeds or fails
end
```

You can also run the `pdriver` command with the `-parallel` flag to run multiple jobs in one PDS file in parallel.

Summary

- Built-in statements are OS-independent commands that can be used inside almost any block in a PDS script.
- Conditional statements can be used within blocks to control the flow of a PDS script.

Arrays

Arrays are fundamental to achieving parametric parallelism in PDS, because an array variable is implicitly indexed in the task block.

Construction

Arrays of primitive values can be constructed in several ways. A literal value is written as follows:

```
a = [1, 2, 3, 4, 5]
```

Arrays can also be constructed by autorange expressions. The expression

```
begin n end m step k
```

constructs an array starting with *n* and proceeding in increments of *k* until *m* is reached. More precisely, it constructs

```
[n, n+k, n+2k, ..., n+rk]
```

where *r* is the largest integer such that *n+rk* <= *m*.

The expression

```
begin n count c step k
```

constructs an array of *c* elements beginning with *n* and proceeding in increments of *k*, that is,

```
[n, n+k, n+2k, ..., n+(c-1)k].
```

For example,

```
    begin 10 end 15 step 2
```

and

```
    begin 10 count 3 step 2
```

both construct the array:

```
    [10, 12, 14].
```

The third way to construct an array is to use the `split` function, which divides a string into array elements at whitespace. Quoted elements keep embedded whitespace and strip the quotes upon placement into the array. For example, on UNIX machines:

```
    split(‘ls‘)
```

returns an array of the files in the current directory.

Indexing

In most contexts, when a variable containing an array is referenced, the first element is returned. However, in the task block, the element corresponding to the ID of the currently running task is returned. (If the task ID exceeds the array size, the last element of the array is returned.) This feature makes it easy to write the most common kinds of distributed computations. For example, we can set up an array of values and run a command on each one in parallel with the following PDS script:

```
args = begin 100 end 200 step 5
task sizeof($args)
    execute "doit $args"
end
```

The `$args` in the execute statement expands to 100 for the first task, 105 for the second, and so on.

Explicit array indexing is not supported: the only ways to obtain an array element other than the first are implicit indexing in the task block, and the `for` statement. Even in those

contexts, assignment to the variable holding the array changes the entire variable value, not the current element.

Other features

An array that includes both string and numeric values is legal. Arrays of arrays are not allowed.

The number of elements in an array can be determined by the `sizeof` function, as shown in the argument to the `task` block in the above example.

A PDriver Example

This section shows an example PDS script that calculates pi by using a distributed Monte Carlo calculation.

The Pi.pds Example

This example works by running an executable that can be built from `[Gridserver SDK dir]\examples\pdriver`. Build and deploy the executable using the supplied script.

A single script, `pi.pds`, runs several tasks, each with a different seed and multiple iterations.

The beginning of `pi.pds` starts the job block by defining the number of tasks and an output directory for the tasks.

```
# pi.pds
#
# performs a distributed monte carlo pi calculation.
#
job pijob
tasks = 50
outputdir = "pijob.example.out"
seed = begin 200 count $tasks step 3
iterations = 500000
```

Next, a task block must be written to run the executable. However, the task must run one of two different binaries, depending on the operating system of the Engine. A conditional is used to check the status of the `$DSOS` variable. If it is win32, the `PdriverPiCalc.exe` binary is run, otherwise, `pdriverPicalc` is run from the `$DSOS` resources directory. The program takes two parameters: the seed, and the number of iterations. The standard output and standard error from the program is redirected to the work directory. After each task is completed, the output and error output is copied back to the staging directory.

```
task $tasks
    onerror retry
    shell "none"
    if ( $DSOS == "win32" ) then
```

```
        execute stdout="$DSWORKDIR/pijob.$DSTASKID.out"
                stderr="$DSWORKDIR/error.$DSTASKID"
                "${gridLibraryDirWin}\commands\PdriverPiCalc.exe $seed
$iterations"
    else
        execute stdout="$DSWORKDIR/pijob.$DSTASKID.out"
                stderr="$DSWORKDIR/error.$DSTASKID"
                "${gridLibraryDirUnix}/$DSOS/commands/PdriverPiCalc
$seed $iterations"
    end
    copy "$DSWORKDIR/pijob.$DSTASKID.out" "$DSSTAGEDIR"
    copy "$DSWORKDIR/error.$DSTASKID" "$DSSTAGEDIR"
end
```

After the task block has run 50 times, the postjob block is run. It processes the output of each task with a `pi-average-win32.bat` or `pi-average-unix.sh` script.

```
postjob
    onerror ignore
    mkdir "$outputdir"
    copy "$DSSTAGEDIR/pijob.*" "$outputdir"
    if ( $DSOS == "win32" ) then
        execute stdout="PIVALUE.OUT"
                """%DS_SDK_DIR%\examples\pdriver\pi-average-win32.bat""
$outputdir"
    else
        execute stdout="PIVALUE.OUT"
                "$$DS_SDK_DIR/examples/pdriver/pi-average-unix.sh
$outputdir $tasks"
    end
end
end job
```

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

Documentation for TIBCO GridServer® is available on the TIBCO GridServer® Product Documentation page.

The following documents for this product can be found in the TIBCO Documentation site:

- TIBCO GridServer® Release Notes
- TIBCO GridServer® Installation
- TIBCO GridServer® Introducing TIBCO GridServer®
- TIBCO GridServer® Administration
- TIBCO GridServer® Developer's Guide
- TIBCO GridServer® Upgrade
- TIBCO GridServer® Security
- TIBCO GridServer® COM Integration Tutorial
- TIBCO GridServer® PDriver Tutorial
- TIBCO GridServer® Speedlink
- TIBCO GridServer® Service-Oriented Integration Tutorial

How to Contact TIBCO Support

Get an overview of TIBCO Support. You can contact TIBCO Support in the following ways:

- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support website.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to TIBCO Support website. If you do not have a user name, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices