

# Frontier<sup>®</sup> Rapids<sup>™</sup> User Guide

## Introduction

Frontier Rapids is an environment for running native applications on the Frontier Enterprise Computing Platform. By native applications, we mean applications that were not originally developed to run on the Frontier platform. These applications can be 3<sup>rd</sup> party applications (e.g., statistical or bioinformatics programs) or proprietary in-house applications. In general, native applications are applications developed and compiled to run on specific operating system platforms (e.g., Windows or Linux), but native applications can also be applications developed in scripting languages, interpreted languages, and high-level languages like Matlab, R, and Perl. Frontier Rapids allows any of these native application types to run on the Frontier platform without modification.

Frontier Rapids provides the ability to run a large number of instances of an unmodified native application as a single distributed job on Frontier. For many classes of applications, this can result in dramatically faster execution times. Using Frontier Rapids with existing applications can enable you to solve problems that could not be solved when running those applications in conventional computing environments.

## Installation

### *Installation Requirements*

There is a Frontier Rapids installer available for both Windows (including Windows 2000, Windows XP, Windows Server 2003, Windows Server 2008, Windows Vista, and Windows 7) and Generic Unix (includes UNIX, Linux, and MAC OS X). The host must also have an installation of the Java Runtime Environment (JRE) 5.0 or later.

To run Rapids jobs, you will also need a Frontier client account. You can sign up for an account on Parabon's online grid at [www.parabon.com](http://www.parabon.com).

For help in setting up your account, send e-mail to [support@parabon.com](mailto:support@parabon.com).

### *Installation and First Use*

To install Frontier Rapids, you need only run the installer and follow the prompts.

The first time you run Frontier Rapids, your installation will be configured. You will be asked for your Frontier account credentials at this time. The configuration will also install the sample Rapids templates (see below).

If you need to subsequently change the Rapids configuration, you can invoke rapids with the Q or C option (/Q or /C on Windows; -q or -c on UNIX/Linux platforms). The Q option does a quick configuration. The C option enables advanced configuration.

## Frontier Concepts

The Frontier platform is made up of three main components, each of which plays a significant role in performing computational work on Frontier:

*The Client Application.* Executed from a single computer, a *client application* is a domain specific application configured to execute compute-intensive jobs via communication with a Frontier server. This communication occurs within the context of a session, during which jobs can be launched, monitored, and terminated.

*The Frontier Compute Engine.* The Frontier compute engine is a desktop application that utilizes the spare computational power of an Internet-connected machine to process small units of computational work called *tasks* during idle time. Individuals who install and run the Frontier compute engine to contribute their computers' spare power to a Frontier grid are called *providers*.

*The Frontier Server.* This is the central hub of the Frontier platform, which communicates with both the client application and multiple Frontier compute engines. It coordinates the scheduling and distribution of tasks; maintains records identifying all provider nodes, client sessions, and tasks; and ensures the platform's consistency and reliability.

Computational work to be performed on Frontier is grouped into a single, relatively isolated unit called a *job*. Within a job, work is divided into an arbitrary set of individual *tasks*, with each task being executed independently on a single node. Tasks can reference *elements*, which are a mechanism used to efficiently transport relatively large chunks of binary data required to execute a task. Elements are sent from a client application to the server and directed to computational nodes as required before the execution of a task is initiated.

Tasks return results to the client application in the form of a *task status* reports. These reports include run mode, results or exceptions, and progress, as well as other pieces of information that the Frontier compute engine and server may include such as computational work performed. The final task status report contains the final task results.

Frontier applications generally operate in one of two stages: *launching* and *listening*. Both stages can occur within a single instance of an application, two or more separate invocations of the application, or even via two or more completely separate applications. When launching a job, tasks are created and sent to the server for processing. Listening involves gathering results and status updates or removing tasks from Frontier. Launching and listening can intermingle in a single session or occur over the course of several sessions with the server.

## The Rapids Application

Frontier Rapids is a Frontier application that enables users to run native applications on a Frontier Grid. The user creates a Rapids *template* to tell Rapids how to run a particular native application on Frontier. Once the template is created, the user uses the Rapids application to launch, monitor, and control jobs based on that template.

The Rapids application consists of two logical components: the Rapids command line interface (CLI) and the Rapids listener. The Rapids CLI is used to create, launch, monitor, and control jobs. The Rapids listener is a supporting process that handles job launching and listening (see the discussion of launching and listening in the previous section). When a job is launched from the Rapids CLI, a Rapids listener is started for that job and runs in background. The listener is

independent of the Rapids CLI and will continue to run until the job completes or the user explicitly stops the listener. The user can stop the listener without impacting the job running on the Frontier Grid. The user need only re-start the listener for that job at a future time to retrieve the job results. The user can have multiple Rapids jobs running at one time. In this case, there will be multiple Rapids listeners running simultaneously.

## Rapids Jobs

A Rapids *job* consists of a set of one or more *tasks*. A task is simply a *command-line* that is executed on a Frontier node. When a user launches a Rapids job, Rapids creates the corresponding Frontier job and associated tasks on the Frontier server. The Frontier server then manages their execution on Frontier nodes. A task can have one or more *elements* that are automatically copied to the task's *working directory* on the Frontier node. These elements can be any type of file (data, executables, scripts, libraries, etc). The elements are copied before the command line is executed, so the command line is free to reference them. The standard output and standard error of the command-line is returned to the user. The user can also request that specific output files created by the task be returned as well.

Rapids supports three types of elements: *job scope elements*, *task scope elements*, and *client scope elements*. Job elements are defined at the job level and are sent to all tasks in that job. Job elements are cached on the compute engine so they don't have to be downloaded for each task. Task elements are task-specific (each task gets its own copy of the element). Client elements are defined once for a client and can be referenced from many jobs (they must be uploaded to the server independent of any job using the Rapids *upload* command). Client elements are cached on the compute engine, so they don't have to be downloaded for each job/task. Client elements are particularly useful when a client's jobs use very large elements that take significant time to download. Client scope elements are deleted from the server if they have not been referenced by a job for 30 days.

Client scope elements can be very useful for relatively heavyweight native applications. One way of using them is to archive an entire application installation directory and make that archive a client scope element. Archives can be created as a tar/gzip file on UNIX/Linux or as a self-extracting zip/tar file on Windows. The command line can then run a script that unzips/untars the file into the application directory. Since the element is client scope, it generally only gets sent to the compute node once (rather than once per task or job). For managed jobs, a better alternative to heavyweight application distribution is to use the `REQUIRED_EXTENSIONS` property (see Appendix A).

There are two types of Rapids jobs: *managed* and *unmanaged*. The tasks of managed Rapids jobs run in virtualized runtime environment (e.g., within a Linux virtual machine). There are several advantages to managed jobs. Because managed jobs run in a virtualized environment, the executables that constitute those jobs are not limited by the capabilities of the underlying platform. As long as a virtual machine (VM) is available for job's target platform, the job's tasks can run on any grid node by deploying a copy of the required VM. Parabon supports a variety of standard VM types and customers can supply their own VM if a standard VM is not available to meet their application requirements. A second advantage of managed jobs is that they are confined to a virtual environment with no direct access to the grid node's resources (specific access is dictated by a configurable policy). The fact that managed jobs are securely confined makes it possible to run them in environments where unmanaged jobs would represent an unacceptable security risk.

Unmanaged Rapids jobs run directly on Frontier nodes without any intermediate virtualization layer. The Frontier scheduler ensures that native tasks get scheduled on the appropriate platforms. Rapids enables developers to create unmanaged jobs that run on multiple platforms (i.e., by allowing the developer to specify native executables for the different platforms). These multi-platform jobs enable unmanaged Rapids jobs to run across heterogeneous sets of nodes. For security reasons, only managed jobs are permitted on the Parabon Computation Grid. Customers deploying their own Frontier Enterprise grid or those employing a Virtual Private Grid (VPG) where tasks run only on their own computers, may choose to run unmanaged jobs at their discretion.

## Rapids Template Structure

Rapids jobs are created from a user-created job *template*. The template provides the Rapids application with all the information required to run a native job on Frontier. A Rapids template defines the command line to be used to start tasks, task parameters, required elements, the environment in which tasks should be run, where the results are to be placed, and other options that control the execution of the job.

Rapids Templates are created in the user's templates directory. By default, the templates directory is named *templates*, and is created in Rapids home directory. Under Windows, the Rapids home directory is created in the user's profile folder (%USERPROFILE%\FrontierRapids). Under UNIX/Linux, the Rapids home directory is created in the user's home directory (~\FrontierRapids). The default template directory can be changed in the Rapids configuration (see *Installation and First Use* above).

Each subdirectory in the templates directory is a job template. The name of the directory is the template name. A template directory can contain a *job.properties* file, a *common* directory, and a *datasets* directory. Only the job.properties file is required. In addition, when a job is run, the results will be placed by in the template's *resultsets* directory. These are described in detail below.

### Job Properties File

The job.properties file defines a set of properties that describe the characteristics of the job. A property assignment is a key-value pair of the form KEY = VALUE. A property value that ends with a "\ " is continued on the next line. Any entry that starts with a "# " or a "! " is a comment and is ignored by Rapids.

The only required properties for a managed job are the RUNTIME and COMMAND\_LINE properties. The RUNTIME property specifies the runtime environment that should be used to run the job's tasks. The COMMAND\_LINE property specifies the command line that is used to execute the task. For example, the following is the properties file from the sample template *noelements.managed*:

```
RUNTIME = .*centos.*
COMMAND_LINE = echo "Hello World!"
```

This is a simple task that runs under a Centos VM and simply outputs the result: "Hello World!".

The only required properties for an unmanaged job are PLATFORMS and COMMAND\_LINE properties. The PLATFORMS property specifies the platforms on which the jobs tasks should be scheduled (e.g., WINDOWS, LINUX, MACOSX). The COMMAND\_LINE property specifies the command line that should be run. For example, the following is the properties file from the sample template *noelements.unmanaged*:

```
PLATFORMS = LINUX; MACOSX
COMMAND_LINE = echo "Hello World!"
```

In some cases, when using unmanaged tasks, you will need to use different command lines for different platforms. In this case, you can include a platform specific command line where required. For example, the following properties file is from the sample template *bundle.unmanaged*:

```
PLATFORMS = LINUX; WIN; MACOSX
COMMAND_LINE = sh startup.cmd
COMMAND_LINE.WIN = cmd.exe /c startup.cmd
```

In this case, Linux and Mac use the default COMMAND\_LINE, but Windows needs a different command line.

In a real application, the COMMAND\_LINE parameter generally specifies an application or script to be launched and it is generally launched using a shell (e.g., sh and cmd.exe above). Typically, this is a simple script that sets up the environment for the job to run and then runs the main executable for the job. These scripts or applications would be staged on the compute engine as an element. Furthermore, the application or script could access other programs, libraries, or data staged as additional elements.

The return code from the task's command line is used to determine whether the task has succeeded or failed. If the script returns zero, the task is considered successful. If the script returns non-zero, the task is considered failed. Normally, a failed task will exit and the failure code/message will be written to the task log (which can be viewed with the Rapids log command). Such a task will not be rescheduled on another node.

The complete set of properties settable in the job.properties file is described in Appendix A.

## **Common Elements Directory**

The *common* directory is optional part of the template that contains elements that are common to all instances of the job (i.e., the part of the job that doesn't vary with the data set). It is common to place the executable components of the job here so they don't have to be repeated in each data set. The common directory has a *job\_elements* subdirectory that contains job scope elements that will be common to all instance of the job. These files are automatically copied to every tasks working directory. The element's file name on the compute engine will be the same as the name used in the *job\_elements* directory. These elements are copied to the working directory *before* the command specified in the COMMAND\_LINE property is invoked. This means that these files can be safely referenced from that command. Typically, a main executable is passed as a common element. Also,

if there are input files that are shared by all tasks in all instances of the job, they will be passed as common elements as well.

If you are defining a *managed* Rapids job, then any common element you include in your template must be appropriate for the RUNTIME that you specify in your jobs.properties file (e.g., if you specify a Linux runtime, you wouldn't want to pass in a Windows executable).

Similarly, if you are defining an *unmanaged* Rapids job that is targeted at a single platform (i.e., only one platform is specified using the PLATFORMS property), then any common element you include in your template must be appropriate for the platform that you specify.

If you are defining an *unmanaged* Rapids job that is targeted at *multiple* platforms (i.e., multiple platforms are specified using the PLATFORMS property), then you *may* need to use *bundle elements*. A bundle element is a single logical element that can take different forms for different platforms. You need to use a bundle element in cases where you want to send out a different version of an element to different platforms (e.g., a native executable compiled for different operating systems/architectures, or a script implemented in PowerShell for Windows, but Borne Shell for Linux). You can mix regular and bundle elements in your template. If a single version of an element can be used across platforms, then these files should be passed as normal elements. If a different version of the element is needed on different platforms, then these files should be passed as bundle elements.

Creating a bundle element is easy. Instead of creating a file in the *job\_elements* directory for the element, you create a directory instead. The name of the directory is the name that will be used for the element on the compute engine. The directory should contain one sub-directory for each platform you want to support (e.g., WINDOWS, LINUX, UNIX, MACOSX). The platform subdirectory contains the version of the element for that specific platform. The file in the platform subdirectory may or may not have the same name as the native element directory name. When Rapids copies this file to the compute node, it will be given the name of the top-level bundle element directory name. When you run your job, Frontier will ensure that the appropriate versions of the bundle elements get sent to the appropriate nodes. See the sample template *bundle.managed* for an example.

Every file in the common/job\_elements directory is considered an element. If you have certain meta files that you want rapids to ignore (e.g., CVS or .svn) you can add these to the ".rapids.exclude" file in the Rapids home directory. All files listed in the ".rapids.exclude" file are ignored when loading templates.

## ***Datasets Directory***

The *datasets* directory is an optional directory that contains configuration information that varies between runs of a job. If a template includes data sets, then you would specify a data set name when you create an instance of the job from the template. It is the dataset construct that enables you to run multiple instances of a job with different data. Each directory in the datasets directory is a separate dataset. The name of this directory is the dataset name.

The dataset can contain job scope elements, task scope elements, and/or task parameters. Job scope elements are contained in the *job\_elements* subdirectory and are specified in exactly the same way as the job scope elements in the common directory (see above). You might use a job scope element in a dataset when the element is shared by all tasks in an instance of the job, but different job instances use different versions of the element. See the sample templates *dsjobelement.managed* and *dsjobelement.unmanaged* for examples of the use of job scope elements in a dataset.

A dataset can also have a *task\_elements* directory that contains task scope elements. A task element is an element that is specific to a particular task. For example, if you want each task to have its own version of a data file, you would make that file a task scope element. There will be one subdirectory in the *task\_elements* directory for each task scope element. The contents of this directory will be the individual task instances. The naming convention for these files is:

```
<element-name><task-index>[.<element-ext>]
```

The element-name and element-extension come from the name you used for the element in the *task\_elements* directory (e.g., the task element directory name). The extension is optional. For example, if you want to create a task scope element named “data.dat” for a three-task job, you would create a directory named *data.dat* in the *task\_elements* directory, and within this directory, you would create three files named “data0.dat”, “data1.dat”, and “data2.dat”. See the sample templates *dstaskelement.managed* and *dstaskelement.unmanaged* for examples of the use of task scope elements.

It is possible to create element bundles for job scope and task scope elements specified in a dataset in exactly the same way that they are created for the job scope elements in the common directory. See sample template *dsjobbundle\_unmanaged* for an example of the use of dataset job scope bundle elements. See sample template *dstaskbundle.unmanaged* for an example of the use of dataset task scope bundle elements.

A datasets directory may also contain a *parameters.txt* file. If this file exists, it contains parameters that are to be passed to the task (one line per task). Each line represents a set of parameters that will be added to the tasks *COMMAND\_LINE*. A line that ends with a “\” is continued on the next line. Any entry that starts with a “#” or a “!” is a comment and is ignored by Rapids. Parameters are separated by white space. Parameters containing spaces can be quoted. For example, if the *COMMAND\_LINE* property is:

```
COMMAND_LINE = sh myscript.sh
```

And the *parameters.txt* file contains the following:

```
1 alpha "a b"  
2 beta "c d"  
3 gamma "d e"
```

The job will have three tasks invoked as follows:

```
sh myscript.sh 1 alpha "a b"
```

```
sh myscript.sh 2 beta "c d"  
sh myscript.sh 3 gamma "d e"
```

If your data set has both task scope elements and parameters, the cardinality of the two must match. That is, the number of task input files must match the number of lines in the parameters.txt file. See the sample templates param.managed and param.unmanaged for an example of the use of task parameters.

## **Result Sets Directory**

The *resultsets* directory is used to store job output. Each directory in the resultsets directory contains a single result set (i.e., the output files from a single run of a job created from that template). The result sets in the resultsets directory are created automatically when the job is run. The name of the directory is the name of the result set. When a job is run via the Rapids CLI the user specifies a result set name. If the user specified the OUTPUT\_FILES property in the job.properties file, then when the a task runs, the specified output files will be returned to the Rapids listener and will be written to the result set. For each output file named in the OUTPUT\_FILES property, there will be a set of files in the result set directory corresponding to the tasks in the job. The naming convention for these files is:

<output-file><task-index>[.<ext>]

The extension is optional. For example, a three-task job with an output file named "output.dat" will have three output files in its resultset directory called "output0.dat", "output1.dat", and "output2.dat".

There are several other files that are always written to the resultsets directory. The standard output/standard error for each task is written to the file stdout<task-index>.txt. The listener log is written to the file log.txt. Both the listener log and task log files are also viewable using the Rapids log command. There is also a file called jobid.txt is written into the job's resultsets directory. This file contains the corresponding job's job ID. Finally, the file command\_line.txt contains the original rapids newjob command line used to create the job.

## **Client Elements Directory**

The user's templates directory can also contain a special directory named *client\_elements*. This directory contains the user's client scope elements. This directory is at the top level in the templates directory, and not in a specific templates directory, because client scope elements are independent of any job. The elements in the client\_elements directory can be regular or bundle elements and are specified in exactly the same way as the elements in the common/job\_elements directory. Client elements must be uploaded to the server using the Rapids *upload* command before they are used.

## **Using the Rapids Command Line Interface**

The Rapids command line interface is used to create, launch, monitor, and control jobs. The Rapids CLI is started from an appropriate start script (rapids.sh on UNIX/Linux and rapids.bat on Windows). The application can be used in interactive mode or command mode. If launched without an argument, the application is in interactive mode. If a command is provided as an argument to the Rapids application, the command is immediately executed and Rapids exits when the command complete.

The Rapids application currently supports the following commands:

newjob	- create a new job
launch	- launch a job and start its listener
listen	- restart the listener for a job
stop	- stop a job listener
kill	- kill a job or task
status	- return status information for a job
delete	- remove the state information for a completed job
remove	- remove a job from the Frontier Server (advanced)
log	- display a job or task log
list	- list jobs
help	- display a list of available commands
upload	- uploads a client scope element to the server
update	- adds new tasks to an existing job
quit	- exit Rapids

A detailed description of each of these commands is available using the `rapids help` command (the help files are in the `docs/commands` directory).

The following is a sample rapids session:

```
$ rapids.sh
rapids> newjob param.managed ds2 rs2
Created Job 1.
rapids> list
0 - newjob param.unmanaged ds1 rs2
1 - newjob param.unmanaged ds2 rs2
rapids> launch 1
Launching Job 1.
rapids> status 1
Listener is running.
0      1.000000 COMPLETE
1      0.250000 RUNNING
2      1.000000 COMPLETE
rapids> log 1
<displays listener log for job 1>
rapids> log 1 0
<displays task log for job 1 task 0>
rapids> delete 1
$
```

In this example, the `newjob` command is used to create a new job from the template `param.managed` using dataset `ds2`. The result set will be named `rs2`. When a job is created, it is automatically assigned a job number. In this example, the job number is 1. The `list` command is used to view the currently defined jobs. After the job is created, it still needs to be launched. The `launch` command is used to launch the job. As mentioned above, the job listener is started in background, so the launch command returns immediately. One implication of this background listener model is that the user can use one instance of Rapids to run multiple jobs in parallel and follow their progress as they run.

As a job runs, it saves its results to the specified resultsets directory (i.e., rs2). The *status* and *log* command are used to follow the progress of a job as it runs. The *status* command is used to get a snapshot of the job's status. The *log* command is used to view the job listener log and can also be used with the job number and task number to view a task log. After a job has completed, the job can be deleted using the *delete* command. The *delete* command will not delete the corresponding resultset. The *quit* command exits the Rapids CLI.

Exiting the Rapids CLI does not affect running jobs. The listener will keep running in background and the status/logs can be viewed the next time Rapids is started. If for some reason the listener stops, for example, because the user issues a *stop* command or shuts down their machine, the job will continue to run on the Frontier Grid. To get the results of the job at a later time, the user need only restart the listener using the *listen* command and the job results up to that time will be retrieved. If for some reason the user wants to kill a job, they can do that with the *kill* command.

## Rapids Template Samples

This table summarizes the samples included in the Rapids distribution (see the templates directory). The templates that end with *managed* are for managed jobs and the templates that end with *unmanaged* are for unmanaged jobs.

Sample Name	Description
bundle.unmanaged	This template demonstrates the use of platform-specific command lines and common job scope element bundles.
clientelement.managed	This template demonstrates the use of client scope elements.
clientelement.unmanaged	This template demonstrates the use of client scope elements.
dsjobbundle.unmanaged	This template demonstrates the use of platform-specific command lines and the use of job scope element bundles in a data set.
dsjobelement.managed	This template demonstrates the use of job scope elements in a data set.
dsjobelement.unmanaged	This template demonstrates the use of job scope elements in a data set.
dstaskbundle.unmanaged	This template demonstrates the use of task scope element bundles in a data set.
dstaskelement.managed	This template demonstrates the use of task scope elements in a dataset.
dstaskelement.unmanaged	This template demonstrates the use of task scope elements in a dataset.
element.managed	This template demonstrates common job scope elements.
element.unmanaged	This template demonstrates common job scope elements.
env.managed	This template demonstrates the use of the ENV property to set the task environment.
interim.managed	This template demonstrates the use of the runtime-request script to return interim task results.
netaccess.managed	This template demonstrates the use of the

	REQUIRE_NETWORK_ACCESS property to enable a task to access the network.
noelements.managed	This is a template with no elements and no parameters. It is the simplest managed template.
noelements.unmanaged	This is a template with no elements and no parameters. It is the simplest unmanaged template.
output.managed	This template demonstrates the use of the OUPUT_FILES property to specify output files to be returned.
output.unmanaged	This template demonstrates the use of the OUPUT_FILES property to specify output files to be returned.
param.managed	This template demonstrates the use of task parameters.
param.unmanaged	This template demonstrates the use of task parameters.
progress.managed	This template demonstrates the use of the runtime-request script to return periodic progress updates.

### ***Developing and Testing New Template***

Before attempting to develop your own templates, you should run the samples. This is the best way to become familiar with Rapids. The following is a logical progression through the templates:

1. noelements.\*
2. element.\*
3. output.\*
4. param.\*
5. dstaskelement.\*
6. clientelement.\*

The remaining templates demonstrate advanced features and can be reviewed as needed.

Once you have successfully run the samples, you can create a template for native jobs you want to run on Frontier. It is recommended that you start with a sample template and adapt it to your needs. When debugging your native job, be sure to use the log command to look at the job and task logs. If something goes wrong, these will usually give you enough information to debug your problem.

If you need assistance, please contact [support@parabon.com](mailto:support@parabon.com).

## Appendix A – Frontier Rapids Job Properties

The following table summarizes the properties that can be used in the job.properties file in alphabetical order. The Job Type column indicates whether the property applies to managed jobs (M), unmanaged jobs (U), or both (M, U).

Property Name	Required Property	Job Type	Description
CHECKPOINT_TASK	N	U	The execution of a task on a Frontier node can be interrupted for a variety of reasons. Interrupted tasks can be handled in two ways, they can be restarted from the beginning, or they can be restarted from where they were when they were interrupted. In order to do the latter, the native executable needs to support checkpointing. In this context, checkpointing means that the executable writes its state out to the task directory periodically, and if restarted, it will read in that state and pick up where it left off. If your native executable supports checkpointing, you should set the checkpoint flag parameter in the Rapids configuration to true. If this flag is false, then your task will not checkpoint correctly. The default value of this parameter is false. <i>Note that checkpointing is not necessary in managed jobs.</i>
CLIENT_ELEMENTS	N	M, U	Some jobs need to reference client scope elements. The CLIENT_ELEMENTS property contains a “;” separated list of the client scope elements required by the job. The list entries reference client scope elements that appear in the templates/client_elements directory.
COMMAND_LINE	Y	M, U	The base command line that all tasks will execute. The arguments are delimited by white space. If an argument contains white space, it

			should be quoted. The actual task command line will be this plus any task-specific parameters.
COMMAND_LINE.PLATFORM	N	U	This is a platform specific command line. All platforms must have a command line. If a platform-specific command line is not available, the generic COMMAND_LINE command line is used instead.
DEBUG_TASK	N	U	If set to true, all runtime exceptions are returned to the application. The Frontier Service is designed to insulate the developer from problems with particular Engines (e.g., disk full). For this reason, when the runtime throws certain exceptions, the server will attempt to schedule the same task on a different engine. Sometimes this behavior can make debugging a job more difficult, so setting this flag to true will turn off this rescheduling behavior. This setting should NOT be used for production runs, as it may cause tasks to fail due to transient Engine issues.
DISK_QUOTA	N	U	This setting limits the total amount of disk space that can be used by the task (in bytes). If the task's disk usage exceeds this amount, the task will be terminated.
ENV	N	M	The property sets the environment for the COMMAND_LINE. Entries have the form "NAME=VALUE". If the value contains spaces, the entire NAME=VALUE pair should be quoted.
EXCEPTION_FILE	N	U	This is the file name for the exception file to be generated by a task application. Normally, if a task fails on one Frontier node, it can be expected to fail on all nodes (e.g., because of a problem in the task

			<p>itself). There are however, cases where you might want a failed task to be rescheduled on another node (e.g., because of a local resource problem). This can be accomplished by specifying an <code>EXCEPTION_FILE</code>. The native application should then write a file of this name to the task's working directory if the application encounters a node-specific error condition. If an exception file is specified and the task's command script exits with a non-zero value, the task will be automatically rescheduled on a different Frontier node where it may find the required resources. Note that if a task fails on too many nodes, Frontier will automatically abort the task. This prevents a scheduling loop for malformed tasks.</p>
<code>INTERIM_RESULT_FREQUENCY</code>	N	M, U	<p>This setting controls how frequently interim results are returned (in seconds). If set, the compute engine will take a snapshot of the output files and return to the rapids listener on the specified schedule. The listener will save them to the job resultset. Note however, the files may be in an inconsistent state since the snapshot may occur while the task is being written.</p> <p>There is an alternative mechanism available for managed jobs. For a safer snapshot, the task's main program can periodically call the "runtime-request interim-result [progress-value]" command via the shell or equivalent system() call to request an interim result. This process will block until the interim result is complete to ensure</p>

			<p>consistent data.</p> <p>Note: the compute engine honors a system wide maximum status interval which may override the supplied value.</p>
OUTPUT_FILES	N	M, U	<p>The output files that are to be returned the task instances. Multiple output file names should be ";" delimited.</p>
PROGRESS_REPORT_FREQUENCY	N	M	<p>Sets the progress report interval (in seconds). If set, the compute engine will report a progress value (a number between 0 and 1) at the specified interval. Alternatively, the task's main program can periodically call the "runtime-request progress" command via the shell or equivalent system() call to request an immediate progress report. Note: the compute engine honors a system wide maximum status interval which may override the supplied value.</p>
REQUIRE_NETWORK_ACCESS	N	M	<p>Determines whether the job requires network access. The value should be "true" or "false". Any other value is treated as false. Default is false.</p> <p>The client account also requires a server-side authorization to access the network.</p>
REQUIRED_EXTENSIONS	N	M	<p>An extension is a specially tagged client-scope or global scope element that is pre-deployed on the engine runtime during the provisioning phase. This property specifies the list of extensions to be deployed on the engine. Multiple extension names should be ";" delimited.</p>

			The use of extensions can significantly improve the performance of heavyweight native applications. Please contact <a href="mailto:support@parabon.com">support@parabon.com</a> to setup extensions for your job.
RUNTIME	Y	M	The runtime to be used. Contact support for the current set of supported runtimes.