What's New in Qube 6.6

Smart Share

SmartShare is a feature where the system automatically expands job instances to fill up idle, or "surplus", worker job slots.

The expansion is automatically balanced among jobs, so that each job gets an equal number of surplus job slots.
Click here to expand...

> New in Qube 6.6-0

Overview

SmartShare is a feature in which the system automatically expands job instances to fill up idle, or "surplus", worker job slots. The expansion is automatically balanced among jobs, so that each job gets an equal number of surplus job slots.

Jobs have a 'max_cpus' value which puts an upper limit on how wide a single job can expand. This value can be supplied via a site-wide default or specified on a job-by-job basis.

SmartShare from a user's perspective

When SmartShare is enabled, jobs will automatically start using it. Note that SmartShare is only only applicable to agenda-based jobs.

It is possible to control how much a job automatically expands, on a job-by-job basis, by setting a new job attribute called "max_cpus" at submission. This value sets an upper limit to the number of instances to which the job will expand. The max_cpus parameter is only visible in the submission UI when in Expert Mode. If an end-user does nothing, their job will expand as widely as the system administrator has defined (see Configuring SmartShare, below). The administrator can specify a default and a maximum value for `max_cpus`.

Jobs will expand to their max_cpus value if there are enough empty slots on the farm. As other users begin to submit jobs as well, a user's job will shrink to allow for a more "balanced" usage of the farm between jobs. No job will shrink below its "instance count" (or "CPUs") value. For example, a job submitted from the command line "qbsub" utility as

```
$ qbsub -priority 5 -cpus 10 -max_cpus 50 -range 1-100 sleep 60
```

ends up with 10 primary instances as specified by the "cpus" option at priority 5, and may expand up to 50 instances if surplus job slots are available. In other words, as many as 40 secondary instances will be dispatched, at a lower priority.

Settings

- Setting `max_cpus` to "*" (asterisk) or a negative number, such as "-1", means "no limit", which will allow the job to expand to the `supervisor_max_cpus_limit` setting. This is configured by the site administrator.
- Setting `max_cpus` to 0 disables SmartShare for the job.
- If `max_cpus` is unspecified at submission, the system will apply the default value, which is `supervisor_default_max_cpus`. This is configured by the site administrator.
- The `max_cpus` value may also be modified after submission, in the GUI, with the qbmodify command, or via the API.

Priorities

- The instances that were automatically expanded by SmartShare ("secondary instances") run at low priority, and are the first subject to pre-emption in order to run primary instances from other jobs. Secondary instances may also be pre-empted in order to run secondary instances from other jobs, to balance the number of secondary instances among all jobs.
- The method of pre-emption can either be "aggressive", meaning the instance is immediately terminated, or "passive", meaning the instance is allowed to finish processing the current agenda item before terminating, and is configured by the site administrator (see `supervisor_smart_share_preempt_policy` below).
- The default smart share pre-emption mode is "aggressive"; secondary instances will be stopped immediately in order to accomplish the balancing of jobs as quickly as possible

Configuring SmartShare

> SmartShare is automatically enabled for new 6.6+ installs and upgrades alike.

To opt out, the site administrator can set the **supervisor_smart_share_mode** **and** **supervisor_smart_share_preempt_policy** in qb.conf on the Supervisor machine:

```
supervisor_smart_share_mode = none

supervisor_smart_share_preempt_policy = disabled
```

- **supervisor_smart_share_mode**
  - Can be either “jobs” or “none”.
  - default: "jobs"
  - Setting it to “jobs” will make the system try to balance the number of the automatically-expanded “secondary" instances on a job-to-job basis
  - Setting it to “none” disables SmartShare.
- **supervisor_smart_share_preempt_policy**
  - method of preemption for secondary instances
  - aggressive, passive, disabled
  - default: aggressive
- **supervisor_default_max_cpus**
  - used to set the site-wide default value of “max_cpus” if not given at submission
  - default: -1
  - Note: “*” can be used in the qb.conf file too, instead of -1
- **supervisor_max_cpus_limit**
  - largest value of “max_cpus” that can be used on any job
  - default 100
  - Note: if any job specifies a value greater than this, the system will silently cap it to this value

Flight-Checks (Preflights and Postflights)

Flight checks, AKA preflights and postflights, are programs and/or scripts that are run on the worker just before and after the actual job or agenda starts.

The exit codes of flight checks determine the processing and exit status of their respective job instances or agendas.

Click here to expand...

Overview

Flight checks, AKA preflights and postflights, new in Qube 6.6, are programs and/or scripts that are run on the worker just before and after the actual job or agenda starts.

The exit codes of flight checks determine the processing and exit status of their respective job instances or agendas.

System-wide vs Job-specific

System-wide flight checks are installed by the site administrator into predefined locations on the worker systems, and are run for every job. By default, system-wide flight checks are installed in one of four subdirectories under **$QBDIR/flightCheck** on the worker. More precisely:

- $QBDIR/flightCheck/instance/pre (instance-level preflights)
- $QBDIR/flightCheck/instance/post (instance-level postflights)
- $QBDIR/flightCheck/agenda/pre (agenda-level preflights)
- $QBDIR/flightCheck/agenda/post (agenda-level postflights)

where **$QBDIR** is the system's Qube install location, which is, by default, "/usr/local/pfx/qube" on Linux, "/Applications/pfx/qube" on Mac OS X, and "C:\Program Files\Pfx\Qube" on Windows.

The worker parameter **worker_flight_check_path** may be set in qb.conf (or qbwrk.conf), to override the default "$QBDIR/flightCheck"

Any file with a ".txt" extension in those subdirectories are ignored, but every other file will be attempted to run at their respective timings. **It is expected that those files are all proper executables** for the platforms on which they are intended to run-- **otherwise, they will fail and thus jobs will fail**.

Individual Jobs may also specify their own flight checks at submission. For example, the "qbsub" command now supports options such as "-preflights", "-postflights", "-agenda_postflights" and "-agenda_postflights" that may be used to specify a **comma-separated list of full-paths to flight check executables** on a per-submission basis. See the "qbsub" documentation for details. The Qube API's job submit routine also may be used to specify flight checks when submitting jobs. These API job attribute names are "preflights" and "postflights" for instance-level, and "agenda_preflights" and "agenda_postflights" for agenda-level flight checks.

As with the system-wide flight checks (see: Universal Callbacks), these job-specific ones also need to be proper executables for the target platform.

Multiple flight checks may be installed or specified for a single job. Job-specific flight checks are run before the system-wide ones. Note that if any flight check program fails (i.e., returns non-zero), all subsequent flight checks for that job instance or agenda are not run.

The postflight checks are still run if an instance or agenda-item fails.

Instance-level vs Agenda-level flight checks and the effect of their exit codes

Instance-level preflights and postflights are run on the worker just before and after job instances are processed, respectively. Similarly, agenda-level preflights and postflights are run just before and after agendas are processed, respectively.

A non-zero exit code returned from a flight check will abort the processing of any additional flight checks for the respective job instance or agenda, and also affect their exit status. More specifically:

- If an instance-level preflight exits non-zero, any additional preflights for the job instance are skipped, the execution of the job instance is canceled, and the job instance is reported to the supervisor as "failed".
- If an agenda-level preflight program exits non-zero, any addtional preflights for the agenda are skipped, the agenda will be unprocessed, and reported as "failed". The job instance will move onto processing the next agenda.
- If an instance-level postflight exits non-zero, any additional postflights for the job instance are skipped, and the job instance is reported to be "failed".
- If an agenda-level postflight exits non-zero, any additional postflights for the agenda are skipped, and the agenda is reported as "failed". The job instance will move onto processing the next agenda.

Universal Callbacks vs. FlightChecks

At first glance, universalCallbacks and the Job Pre- and Post-FlightChecks appear similar, but they have an important difference:

- **Universal callbacks** are run by and on the **supervisor** host.
- **Flight checks** are only run on the **worker** hosts.

Tips

When writing postflight scripts for **cmdline** and **cmdrange** jobs, you can query the environment variable **QB_SYSTEM_EXIT_CODE** to find out the numerical exit code of the last command that was run.

Also, in postflight scripts, access the status of the last-processed frame or instance via the **QB_FRAME_STATUS** and **QB_INSTANCE_STATUS** environment variables, respectively. These will be set to strings such as "complete" and "failed".

Universal Callbacks

Universal callbacks are operations which are automatically attached to every single job.

These are installed by a site administrator and are run by the supervisor host.
Click here to expand...

Overview

Universal callbacks are operations which are automatically attached to every job that is submitted. These are installed by a site administrator either on the Supervisor's local disk or on a file system which is accessible by the supervisor service.

Setting Up Universal Callbacks

To set up Universal Callbacks, the site administrator needs to make a directory (default $QBDIR/callback) and create a text-based configuration file, called "callbacks.conf" in that directory. Further, in the same directory, files containing the implementation (aka "code") of each Universal Callback must also be installed.

The callbacks.conf file serves as a map that tells the system which callback code should be triggered to run on what events.

The "callbacks.conf" file

The callbacks.conf file is a text file, much like qb.conf, containing one or more lines with a "key = value" pair, associating each implementation file to a trigger event. The syntax is:

```
# lines starting with a hash mark are comments
filename = trigger
```

The **filename** points to a file in the same directory that implements the callback code. Note that Universal Callbacks support Python, Perl, and Qube callbacks, and the filename must have the extension .py, .pl, or .qcb, respectively.

The **trigger** specifies the triggering event that activates the callback, described in details at Triggers

Here's an example:

```
#
# callbacks.conf
#
# syntax of this file is :
# filename = triggers
#
logFailuresToDB.py = failed-job-self
mail-status.qcb = done-job-self
checkWork.pl = done-work-self-*
submitted.py = submit-job-self
```

In this example, there are presumably 4 implementation files in the callback directory, logFailuresToDB.py, mail-status.qcb, submitted.py, and checkWork.pl, that have the implementation code in them.

Use Popen.subprocess in callbacks
If you ever need to run an external script in a callback, we recommend the use of `Popen.subprocess()` to run the external script inside the callback. This returns immediately and allows the callback to continue running, rather than blocking and waiting for the external script to complete; otherwise the supervisor process is tied up for the duration of the external script's execution.

Do **not** use os.system() to run the external script, as this call will block until the external script exits. When a large number of callbacks tie up supervisor processes at the same time, your supervisor performance will suffer.

Never use sys.exit() in a callback
Do not call `sys.exit()` at the end of the callback code, this kills the calling supervisor process.

submitted.py

```
#!/usr/bin/env python

import sys
import qb
import traceback

fh = open('/tmp/univeral_callback_test', 'a')
try:
    # ================================================
    #  === NOTE: ===
    #  the qb.jobinfo() in callbacks is not the
    #  same as the one in the external python API
    # ================================================
    job = qb.jobinfo("-id", qb.jobid())[0]
    fh.write('submitted %(id)s: %(name)s\n' % job)
except:
    fh.write(traceback.format_exc())
fh.close()
```

Universal Callbacks vs. FlightChecks

At first glance, universalCallbacks and the Job Pre- and Post-FlightChecks appear similar, but they have an important difference:

- **Universal callbacks** are run by and on the **supervisor** host.
- **Flight checks** are only run on the **worker** hosts.

qb.conf Parameters

supervisor_universal_callback_path

- Path to the directory where Universal Callbacks (the callbacks.conf file and the implementation files) are found
- May be a comma-separated list to specify multiple locations
- default: $QBDIR/callback

Windows MySQL upgraded from 5.1 to 5.5.37

MySQL 5.5 running on Windows offers a significant performance increase over previous MySQL versions; prior to 5.5, MySQL used a port of the linux threading model on Windows. In MySQL 5.5, Oracle switched MySQL over to the native Windows threading, resulting in greatly improved performance.

The version of MySQL automatically installed on a Windows supervisor has been upgraded from 5.1 to 5.5.

If you are upgrading an supervisor from a version prior to 6.6-0, your supervisor's MySQL and all the tables will be automatically upgraded from 5.1 to 5.5.

Since the database tables are upgraded during a supervisor upgrade, it is not recommended to downgrade a supervisor back to Qube! 6.5 or earlier once that supervisor has been upgraded to Qube! 6.6.

Ubuntu support added

Qube! is now supported on Ubuntu 12.0.4-LTS

Python-based "Load Once" jobs now supported on Windows

Python-based "LoadOnce" jobs are currently limited to Nuke and Houdini, but this will allow for easier integration of applications which provide a Python API or prompt mode.

Auto-Wrangling enabled by default

Auto-Wrangling has been enabled by default in Qube! 6.6. If you're upgrading and do not have the supervisor_language_flags value explictly set and do not wish to enable auto-wrangling, ensure that you specify an explicit value for this parameter.